Order and Inventory Management System for Small Online Businesses Utilizing Personal Messaging for Order Collection

Jessie Cruz, Anna Perez, Julia Rivera

What is your project and what does it aim to do?

In this project, we utilized our knowledge on Python and management to create an efficient and data-driven program that would help online small businesses improve their operations by automating their order and inventory management systems.

Who is the project for and why?

This project aims to help increase the efficiency of online small businesses. We know that there has been a significant increase in the number of small businesses online over the last couple of years, and we want to be able to help enhance their internal operations. We've noticed from personal experience that small businesses usually struggle to survive or grow because they have limited manpower but hefty manual work, resulting in inefficiency. Moreover, they are unaware of how to expand their operations given the slow transition to onsite. These problems could easily be dealt with given data and programming resources.

Thankfully, with this program, small businesses can easily manage their orders and stocks by compiling their order messages into a CSV file and running it through the program.

For this project, we used data by craftsby.bb on Instagram to illustrate the business problem. This is an online small business based in NCR that sells crocheted flower bouquets and targets couples. Like other small businesses, they prefer receiving orders via Instagram DM for convenience and a more personal experience. However, this means that information isn't automatically sorted and is formatted differently. They manage their orders, count their stocks and compute their finances manually, which can be time-consuming and prone to error. We collected data on their current order-taking processes and orders.

Process-flow or expected user journey [what does your user do (or can do) given your project?]

In this program, there are two parts. The first part is the Order Management System, and the second part is the Inventory Management System. They are separate programs yet interconnected systems.

Note: Order program and inventory program can be run at any time necessary, however, it is most ideal that these programs are run DAILY (or at least when there are new orders). For order management, this is to ensure orders are organized immediately for ease and for ability to update inventory data, and to ensure personalized messages are immediately generated and ready to be sent as well. For inventory management, this is to ensure that the inventory is constantly updated. However, unrelated to orders incoming, you can run the inventory program at any convenient time so you can go over the products, add stocks, and view inventory (process duplicate condition added to allow for going over of new updates files with the same file name ***file name for data per month should be the same).

Order Management

I. For the first part of the program, the input includes data in a CSV file of all the compiled order messages of customers. These messages follow the outline that is sent by the Instagram business to customers planning to purchase. Ideally, the business (user) runs the program at the end of each day to sort the orders. update their inventory and generate the messages to be sent to their customers. The output is two data frames that are converted to two CSV files. The first includes the sorted personal information of the customer, their order information cleaned and counted, and the total price of their order (per order). This file can be used on its own for better order management, or can also be later used for the inventory management (after renaming). The second will include a personalized automated message for the customer with their name, total, delivery date, and mode of delivery, in a dataframe along with their username, username link, and customers name for convenience of the business.

Input: In this case, the CSV file imported in the program was entitled originalorders.csv and contained only one column of the compiled order messages following the outline sent by craftsby.bb to customers planning to purchase.

```
import pandas as pd
import re
```

```
originalorders = pd.read_csv("originalorders.csv")
originalorders

# import one-column file
```

```
entrycount = len(originalorders)
entrycount
```

The len function was utilized to count the total number of entries

```
originalorderslist = originalorders['MESSAGE'].tolist()
originalorderslist[:5]

# transform message column into a list
# just to check output
```

```
originalorderslist_split = []

for i in originalorderslist:
    k = i.split("\n")
    originalorderslist_split.append(k)

originalorderslist_split[:3]

# split message into strings including individual line items
# just to check output
```

```
usernamelist = []
namelist = []
contactlist = []
addresslist = []
datelist = []
shippinglist = []

for i in originalorderslist_split:
    usernamelist.append(i[0])
    namelist.append(i[1])
    contactlist.append(i[2])
    addresslist.append(i[3])
    datelist.append(i[4])
    shippinglist.append(i[5])

# creation of lists that will be added as columns to the final dataframe
```

The message column was turned into a list, then each order was split by the newlines and into individual line items. After, the data from the split original orders were put into lists which would later become columns in the final dataframe of organized orders.

```
usernamelinklist = []

for i in usernamelist:
    link = "https://www.instagram.com/" + i
    usernamelinklist.append(link)

usernamelinklist[:3]

# creation of list that will be added as a column to the final dataframe
# so the business owner can easily access the customer's profile and send them a message
# just to check output
```

In addition to already retrieved data, we created a list of Instagram links with the corresponding usernames for the convenience of the business.

```
orderlist = []

for i in originalorderslist_split:
    orderlist.append(i[6:])

orderlist[:3]

# no data cleaning yet
# creation of list that will be added as a column to the final dataframe
# just to check output
```

Another list was creator that would also be a column later on, which was the order list. Data from split up original order list was placed into this list (only from 7th row onwards), but this data is not yet cleaned.

```
orderlist_not_unique = []

for i in orderlist:
    i_length = len(i)
    for j in range(i_length):
        k = (i[j]).title()
        k = re.sub("Order: ","",k)
        k = re.sub("Letters","Letter",k)
        k = re.sub("Solo \(","Solo Bouquet (",k)
        k = re.sub("r,","rs,",k)
        k = re.sub("\s*-"," ",k)
        k = re.sub("\s\s*"," ",k)
        k = re.sub("\)\s",")",k)

# python functions and regex to clean data
# quantities are not removed yet

        orderlist_not_unique.append(k)
        i[j] = k

# i[j] = k is for cleaning the original string

orderlist_unique = list(set(orderlist_not_unique))

# creating a list of unique values including quantities

orderlist_unique_nonumbers = []

for i in orderlist_unique:
    if i == "":
        continue
    else:
        first_space_index = i.find(" ")
        orderlist_unique_nonumbers.append(i[(first_space_index + 1):])

# quantities are removed at this stage

orderlist_unique_nonumbers = list(set(orderlist_unique_nonumbers))

for i in orderlist_unique_nonumbers:
    print(i)

print()
print(len(orderlist_unique_nonumbers))

# data cleaning
# to see the unique values
# to check if the number of unique values = the number of SKUs (13)
```

For the cleaning up of orders, we looked for common differences in the formatting of the orders sent by customers. Used Regex to fix capitalization, removed certain words, correct spelling, etc. The data is now cleaned, but the numbers are still there, so the numbers were removed.

Just to make sure everything was cleaned, the number of unique values was also shown in this part which was correct and equal to the number of products being sold.

```
sm_list = []
sl_list = []
rm_list = []
rl_list = []
dm_list = []
dl_list = []
lm_list = []
ll_list = []
tm_list = []
tl_list = []
ew_list = []
el_list = []
dd_list = []

# intermediary list that will not be added as a column to the final dataframe
# but will be helpful in computing the total quantities and price per order
```

Creation of intermediary lists for computing the prices and total quantities.

```
for i in orderlist:
    i_length = len(i)
    for j in range(i_length):
        if "Solo Bouquet (Sunflowers, Mini)" in i[j]:
            sm_list.append(int(i[j][:i[j].find(" ")]))
        else:
            sm_list.append(0)
        if "Solo Bouquet (Sunflowers, Large)" in i[j]:
            sl_list.append(int(i[j][:i[j].find(" ")]))
        else:
            sl_list.append(0)
        if "Solo Bouquet (Roses, Mini)" in i[j]:
            rm_list.append(int(i[j][:i[j].find(" ")]))
        else:
            rm_list.append(0)
        if "Solo Bouquet (Roses, Large)" in i[j]:
            rl_list.append(int(i[j][:i[j].find(" ")]))
        else:
            rl_list.append(0)
        if "Solo Bouquet (Daisies, Mini)" in i[j]:
            dm_list.append(int(i[j][:i[j].find(" ")]))
        else:
            dm_list.append(0)
        if "Solo Bouquet (Daisies, Large)" in i[j]:
            dl_list.append(int(i[j][:i[j].find(" ")]))
        else:
            dl_list.append(0)
        if "Solo Bouquet (Lavenders, Mini)" in i[j]:
            lm_list.append(int(i[j][:i[j].find(" ")]))
        else:
            lm_list.append(0)
        if "Solo Bouquet (Lavenders, Large)" in i[j]:
            ll_list.append(int(i[j][:i[j].find(" ")]))
        else:
            ll_list.append(0)
        if "Solo Bouquet (Tulips, Mini)" in i[j]:
            tm_list.append(int(i[j][:i[j].find(" ")]))
        else:
            tm_list.append(0)
        if "Solo Bouquet (Tulips, Large)" in i[j]:
            tl_list.append(int(i[j][:i[j].find(" ")]))
        else:
            tl_list.append(0)
        if "Extra Wrapping" in i[j]:
            ew_list.append(int(i[j][:i[j].find(" ")]))
        else:
            ew_list.append(0)
        if "Extra Leaves" in i[j]:
```

```
        else:
            el_list.append(0)
        if "Dedication Letter" in i[j]:
            dd_list.append(int(i[j][:i[j].find(" ")]))
        else:
            dd_list.append(0)

# to count quantities by line item
```

Took the number from each line item to count the quantities by line item.

```
sm_list_final = []
sl_list_final = []
rm_list_final = []
rl_list_final = []
dm_list_final = []
dl_list_final = []
lm_list_final = []
ll_list_final = []
tm_list_final = []
tl_list_final = []
ew_list_final = []
el_list_final = []
dd_list_final = []

orderlistitems = []

for i in orderlist:
    orderlistitems.append(len(i))

# goal is for these lists to have the same length as orderlist (which is equal to entrycount)
# to count the number of line items per order
# creation of lists that will be added as columns to the final dataframe
```

Creation of new lists which represented the different products being sold and how many per order, which were columns to be added to the data frame later on

```
orderlistpairs = []

start = 0
for i in orderlistitems:
    orderlistpairs.append([start,start+i])
    start += i

orderlistpairs[:5]

# to get the start and end indices of each order (i.e. first order refers to the first two rows/line items of sm_list)
```

Creation of order list pairs which will show where each order starts and ends based on the line items.

```
def sumrange(listbylineitem, x, y, listbyorder):
    sum = 0
    for i in range(x, y, 1):
        sum += listbylineitem[i]
    listbyorder.append(sum)

for i in range(entrycount):
    sumrange(sm_list, orderlistpairs[i][0], orderlistpairs[i][1], sm_list_final)
    sumrange(sl_list, orderlistpairs[i][0], orderlistpairs[i][1], sl_list_final)
    sumrange(rm_list, orderlistpairs[i][0], orderlistpairs[i][1], rm_list_final)
    sumrange(rl_list, orderlistpairs[i][0], orderlistpairs[i][1], rl_list_final)
    sumrange(dm_list, orderlistpairs[i][0], orderlistpairs[i][1], dm_list_final)
    sumrange(dl_list, orderlistpairs[i][0], orderlistpairs[i][1], dl_list_final)
    sumrange(lm_list, orderlistpairs[i][0], orderlistpairs[i][1], lm_list_final)
    sumrange(ll_list, orderlistpairs[i][0], orderlistpairs[i][1], ll_list_final)
    sumrange(tm_list, orderlistpairs[i][0], orderlistpairs[i][1], tm_list_final)
    sumrange(tl_list, orderlistpairs[i][0], orderlistpairs[i][1], tl_list_final)
    sumrange(ew_list, orderlistpairs[i][0], orderlistpairs[i][1], ew_list_final)
    sumrange(el_list, orderlistpairs[i][0], orderlistpairs[i][1], el_list_final)
    sumrange(dd_list, orderlistpairs[i][0], orderlistpairs[i][1], dd_list_final)

# adding the total quantity of each SKU per order to each individual SKU list
# ex. for the first order, the values that will be appended to the 13 lists are: 0,0,1,0,0,0,0,0,0,0,0,1,0
# these values will be the first row of the dataframe but also the first value in each list
```

Per order, we counted each product and placed it into their own SKU list.

```
prices = {"SM":550,"SL":950,"RM":500,"RL":800,"DM":500,"DL":800,"LM":500,"LL":800,"TM":530,"TL":750,"EW":50,"EL"\
         :80,"DD":30}

# dictionary to calculate the total order price
```

```
totallist = []

for i in range(entrycount):
    total = (sm_list_final[i]*prices["SM"]) + (sl_list_final[i]*prices["SL"]) + (rm_list_final[i]*prices["RM"])\
    + rl_list_final[i]*prices["RL"] + (dm_list_final[i]*prices["DM"]) + (dl_list_final[i]*prices["DL"])\
    + lm_list_final[i]*prices["LM"] + (ll_list_final[i]*prices["LL"]) + (tm_list_final[i]*prices["TM"])\
    + tl_list_final[i]*prices["TL"] + (ew_list_final[i]*prices["EW"]) + (el_list_final[i]*prices["EL"])\
    + dd_list_final[i]*prices["DD"]
    totallist.append(total)

# sum product but python version
# creation of list that will be added as a column to the final dataframe
```

Dictionary with each product and their prices. Then, we see a final list for total amount to be paid by the customer which will also become a column in final dataframe).

```
for i in range(len(orderlist)):
    orderlist[i] = "\n".join(orderlist[i])

# prepare for final dataframe; return orders to original format as a singular string separated by newlines
```

```
finalorders = pd.DataFrame({"username":usernamelist,"username link":usernamelinklist,"name":namelist,"contact":\
                            contactlist,"address":addresslist,"date":datelist,"shipping":shippinglist,\
                           "orders":orderlist,\
                            "SM":sm_list_final,"SL":sl_list_final,"RM":rm_list_final,"RL":rl_list_final,\
                            "DM":dm_list_final,"DL":dl_list_final,"LM":lm_list_final,"LL":ll_list_final,\
                            "TM":tm_list_final,"TL":tl_list_final,\
                            "EW":ew_list_final,"EL":el_list_final,"DD":dd_list_final,\
                            "total":totallist, "IF PROCESSED": ''})

# create dictionary of column names and lists to be converted to the final dataframe

finalorders.head()

# inspect first five rows of the data with the head method
```

The order list was previously split into individual items, so now, they are returned to original format where they are separated by newlines. Then, we use pandas to create the data frame.

```
finalorders.to_csv("MONTH YEAR - DATA FOR CROCHET SHOP - FlowerOrders.csv")

# export dataframe as csv file
# syntax for reference: [nameofdataframe].to_csv("[filename]")
```

The dataframe is exported into a csv file called “MONTH YEAR - DATA FOR CROCHET SHOP - FlowerOrders.csv

```
messagelist = []

for i in range(len(namelist)):
    message = "Hey there "+str(namelist[i])+"! " + "Your order comes to a total of "+str(totallist[i])+" pesos. "+"It will be ready for "+str(shippinglist[i])+" on "+str(datelist[i])+"."
    messagelist.append(message)

messagelist[:3]

# creation of list that will be added as a column to the final dataframe
# so the business owner can easily copy the customer's personalized message and send it to them
# just to check output
```

For the automated personalized messages for each customer, a message list was created which will be added as a column in the additional final dataframe.

Using data collected from the organized orders dataframe, the output is the message “Hey there name! Your order comes to a total of total pesos. It will be ready for mode of

delivery on delivery date." This will be dependent on each customer and their given information (name, total amount, mode of delivery and delivery date).

```
finalmessages = pd.DataFrame({"username":usernamelist,"username link":usernamelinklist,"name":namelist,\
                              "message":messagelist})

# create dictionary of column names and lists to be converted to the final dataframe

finalmessages.head()

# inspect first five rows of the data with the head method
```

```
finalmessages.to_csv("MONTH YEAR - MESSAGES FOR CROCHET SHOP - FlowerOrders.csv")

# export dataframe as csv file
# syntax for reference: [nameofdataframe].to_csv("[filename]")
```

This is added to a new dataframe along with the username, username link, customers name and the automated message. Then, it is exported to a csv file called "MONTH YEAR - MESSAGES FOR CROCHET SHOP - FlowerOrders.csv" for quick messaging of customers.

Output: Two CSV files. In this case, the first CSV file outputted by the program was entitled "MONTH YEAR - DATA FOR CROCHET SHOP - FlowerOrders.csv." and the second was entitled "MONTH YEAR - MESSAGES FOR CROCHET SHOP - FlowerOrders.csv.".

Inventory Management

II. For the second part of the program, the input is the CSV file that resulted from the order management section. This CSV file includes the sorted and cleaned data of each order.

Input: In this case, the CSV file was entitled “MONTH YEAR - DATA FOR CROCHET SHOP - FlowerOrders.csv.” This naming format should be followed when using other files as input. If you already have an automated orders file and choose to use only the inventory management system and not the order management system, the formats should still be followed.

a. Install PyCharm (The following files must be created as they work/interact with one another.)

- FILE: main.py

This is the file that should be run to get the inventory system up and working.

from windows import * will come from file windows.py(to be discussed later)

This contains the main menu display of interactive buttons as shown on the left. It brings us to windows defined in the windows.py class functions part. The codes such as the grid.() help manage the layout of the inventory system.

```
from windows import *

root = Tk()
# MAIN MENU
#to create interactive buttons
add_product = Button(root, width=15, bd=3,
                     text="List of Products", font=times_new_roman,
                     bg="pink",
                     command=ListOfProduct)
add_stock = Button(root, width=15, bd=3,
                         text="Add Stock", font=times_new_roman,
                         bg="pink",
                         command=AddStock)
sell_product = Button(root, width=15, bd=3,
                      text="Sales", font=times_new_roman,
                      bg="pink",
                      command=Sales)
view_inventory = Button(root, width=15, bd=3,
                        text="View Inventory", font=times_new_roman,
                        bg="pink",
                        command=Inventory)
```

```
#displays the four buttons to access certain windows once run
add_product.grid()
add_stock.grid()
sell_product.grid()
view_inventory.grid()

root.title("CROCHET SHOP INVENTORY")
root.resizable(False, False)
#centering position
root.eval('tk::PlaceWindow . center')
root.mainloop()
```

- FILE: global_variable.py

  This file simply stores “functions” or “defined aliases” for database.py to use and call

```
#minimum is the starting index to be displayed
#maximum is the index number after the last index to be displayed
#simply, indexes 0-9
minimum = 0
maximum = 10

#data stores data from csv file and database
data = []

#searchdata stores filtered data depending on what is searched
search_data = []

#for util defError function
input_error = False
error_msg = ''

#to check if user searched something or not
search = False
```

- FILE: database.py

  This file connects the DB Browser for SQLite database entitled Crochet Inventory.db created there(see “Install DB Browser for SQLite” in the docs outline)

```python
import sqlite3

import global_variable as gv

#connect sqlite database to crochet inventory db(created from sqlite)
def conn():
    return sqlite3.connect('CROCHET INVENTORY.db')
```

  Several def functions are created containing queries that interact with the database to insert, select/extract and update

```python
#create query - insert product variable that was passed into inventory table and product table
def add_product(product):
    c = conn()
    cur = c.cursor()
    cur.execute("INSERT or IGNORE INTO product VALUES(null, ?, ?)", product)
    cur.execute("INSERT or IGNORE INTO inventory VALUES(null, 0, ?, ?)", product)
    if cur.rowcount:
        c.commit()
    c.close()

#create query - getdata is using select query to get/extract data from the crochet inventory database
def get_data(table_name_, column_name_='*', filter_='None', filter_value_='None', fetch=''):
    c = conn()
    cur = c.cursor()
    cur.execute(f"SELECT {column_name_} FROM {table_name_}") if filter_ == 'None' \
        else cur.execute(f"SELECT {column_name_} FROM {table_name_} WHERE {filter_} = '{filter_value_}'")
    if fetch == 'all' or filter_ == 'None':
        gv.data = cur.fetchall()
    else:
        gv.data = cur.fetchone()
    c.commit()
    c.close()
    return gv.data

#create query - update query for edit button
def update(table_name_, set_value, filter_, filter_value_):
    c = conn()
    cur = c.cursor()
    cur.execute(f"UPDATE {table_name_} SET {set_value} WHERE {filter_} = {filter_value_}") if filter_value_ == 'id' \
        else cur.execute(f"UPDATE {table_name_} SET {set_value} WHERE {filter_} = '{filter_value_}'")
    c.commit()
    c.close()
```

```
#create query - insert query to insert data into sales table
def sell(month, year):
    c = conn()
    cur = c.cursor()
    for order in gv.data:
        cur.execute(f"INSERT INTO '{month} {year} sales' VALUES(null, ?, ?, ?, ?, ?, ?)", order)
        c.commit()
    c.close()
```

This certain def function when called on will create tables in the corresponding SQLite Database. It will store data from csv files imported in the inventory system, or simply, data found in the inventory system, specifically, the “windows” or “tabs” of main menu, namely AddStock(same as Purchase History) and Sales.

```
#create query - creates purchase and sales table for crochet inventory db
#automatically creates a new table for each month inside the db
def create_table(month, year, window):
    c = conn()
    cur = c.cursor()
    if window == f'PurchaseHistory':
        cur.execute(f"""CREATE TABLE IF NOT EXISTS "{month} {year} purchase" (
                        "id"    INTEGER NOT NULL,
                        "date"  varchar(10) NOT NULL,
                        "QTY"   INTEGER NOT NULL,
                        "product_name"  varchar(50) NOT NULL,
                        PRIMARY KEY("id" AUTOINCREMENT)
                    )""")
    if window == 'Sales':
        cur.execute(f"""CREATE TABLE IF NOT EXISTS "{month} {year} sales" (
                        "id"    INTEGER NOT NULL,
                        "date"  varchar(10) NOT NULL,
                        "customer_name" varchar(20) NOT NULL,
                        "product_name"  INTEGER NOT NULL,
                        "QTY"   INTEGER NOT NULL,
                        "price" INTEGER NOT NULL,
                        "amount"    INTEGER NOT NULL,
                        PRIMARY KEY("id" AUTOINCREMENT)
                    )""")
    c.commit()
    c.close()
```

- FILE: fonts.py

  This file simply contains data for certain font formats that can be called for convenience. It removes the need to type in font styles and font size.

```
#store font formats for shortcut

times_new_roman = ('Times', 15, 'bold')
verdana = ('verdana', 8, 'bold')
Sans_button = ('Sans-serif ', 7)
```

- FILE: csv.handler.py

  This file contains codes that would help open and read imported csv files, run it, process it, and update it. It contains codes that manipulates the data such as the price

```
import csv
import re
#asterisk imports everything
from database import *

#function for reading csv file to import the data from crochet shop
def read_csv(filename):$
    gv.data.clear()
    tempList = list()
    #to open file, encoding is to allow for reading symbols
    with open(f'{filename}', 'r', encoding='utf-8') as file:
        reader = csv.DictReader(file)

        # checks each row in csv file to see if it has been processed alrdy, if processed alrdy, inventory system wont read it anymore to avoid duplicates
        for row in reader:
            if row['IF PROCESSED'] != 'PROCESSED':
                total_qty = 0
                for order in str(row['orders']).split('\n'):
                    (qty, product_name) = int(re.findall('[0-9]+', order)[0]), re.split('[0-9]+ ', order)[1]

                    total_qty += qty
```

```
#price is put into string format, remove peso sign, convert into float format and removed decimals
if str(product_name).lower() == "Solo Bouquet (Sunflowers, Mini)".lower():
    product_name = 'Sunflowers Mini Bouquet'
    price = 550
elif str(product_name).lower() == "Solo Bouquet (Sunflowers, Large)".lower():
    product_name = 'Sunflowers Large Bouquet'
    price = 950
elif str(product_name).lower() == "Solo Bouquet (Roses, Mini)".lower():
    product_name = 'Roses Mini Bouquet'
    price = 500
elif str(product_name).lower() == "Solo Bouquet (Roses, Large)".lower():
    product_name = 'Roses Large Bouquet'
    price = 800
elif str(product_name).lower() == "Solo Bouquet (Daisies, Mini)".lower():
    product_name = 'Daisies Mini Bouquet'
    price = 500
elif str(product_name).lower() == "Solo Bouquet (Daisies, Large)".lower():
    product_name = 'Daisies Large Bouquet'
    price = 800
elif str(product_name).lower() == "Solo Bouquet (Lavenders, Mini)".lower():
    product_name = 'Lavender Mini Bouquet'
    price = 500
elif str(product_name).lower() == "Solo Bouquet (Lavenders, Large)".lower():
    product_name = 'Lavender Large Bouquet'
    price = 800
elif str(product_name).lower() == "Solo Bouquet (Tulips, Mini)".lower():
    product_name = 'Tulips Mini Bouquet'
    price = 530
elif str(product_name).lower() == "Solo Bouquet (Tulips, Large)".lower():
    product_name = 'Tulips Large Bouquet'
```

```
elif str(product_name).lower() == "Extra Layer of Wrapping".lower() or str(
        product_name).lower() == "Extra Wrapping".lower():
    product_name = 'Extra Layer of Wrapping'
    price = 50
elif str(product_name).lower() == "Extra Leaves".lower():
    product_name = 'Extra Leaves'
    price = 80
elif str(product_name).lower() == "Dedication Letter".lower():
    product_name = 'Dedication Letter'
    price = 30
```

The codes in this section call on functions to extract stock quantity that is filtered or searched in the entry textboxes of the windows.

Here, the code to update the inventory and subtract the quantity read from the imported csv file based on its product name. Consequently, it calls on the update function found in database.py.

A temporary list to store the data for the table of the Sales ‘Window” where it also appends the elements to get the list of elements needed. The templist is aliased as gv.data to be used later on.

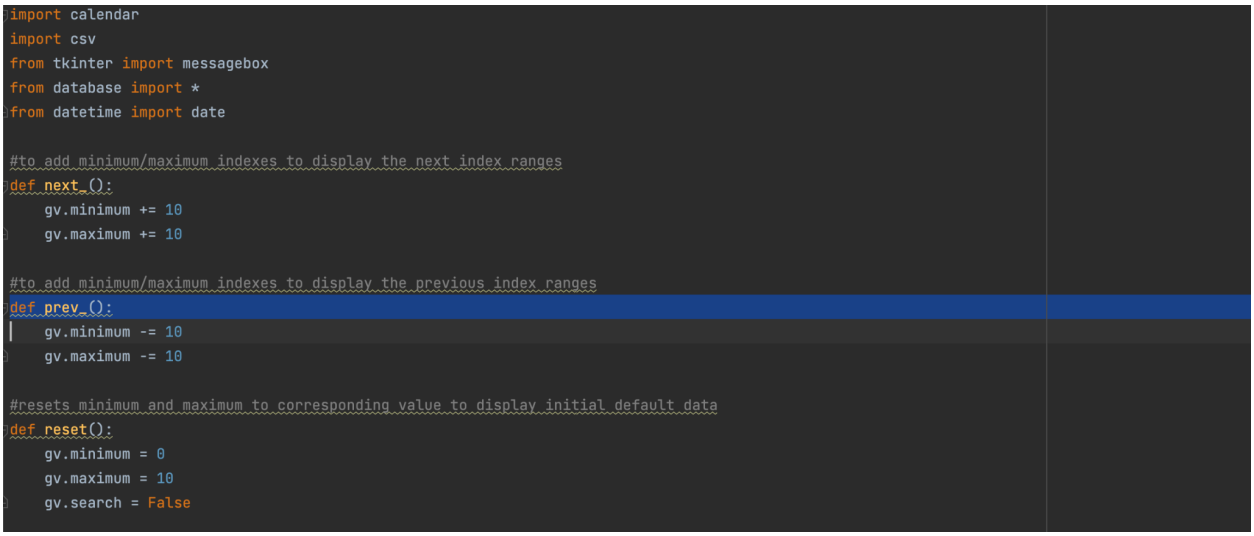
```
    # there is a comma to turn it into tuple --- getdata function gets stock quantity of the product name that is filtered or searched
    (stock,) = get_data(table_name_='inventory', column_name_='stock',
                        filter_='product_name', filter_value_=product_name)

    # if processed, inventory will be subtracted and updated based on order quantity value
    stock -= qty
    # value of stock will be placed here
    stock_value = f"stock = {stock}"

    # calls the update function in database.py
    update(table_name_='inventory', set_value=stock_value,
           filter_='product_name', filter_value_=product_name)

    # templist temporarily stores the data for sales windows table
    tempList.append((row['date'], row['name'], product_name, qty, price, row['total']))
tempList.append((row['date'], "Total", '', total_qty, '', row['total']))

# templist is assigned to gvdata
gv.data = tempList
```

This section contains the function to write and update csv file, avoiding orders that have been already processed. For example, the month is July. On the first week, the user imports a July csv file and the next week, an updated July csv file is imported. This function helps avoid orders that have been processed already in the system.

```
#to update csv if processed alrdy or not to avoid duplicates
def write_csv(filename):
    tempList = list()
    with open(f'{filename}', 'r', encoding='utf-8') as read_file:
        reader = csv.reader(read_file)
        for row in reader:
            tempList.append(row)

    with open(f'{filename}', 'w', newline='', encoding='utf-8') as write_file:
        writer = csv.writer(write_file)
        writer.writerow(tempList[0])

        for i in range(1, len(tempList)):
            if tempList[i][23] != "PROCESSED":
                tempList[i][23] = "PROCESSED"
            writer.writerow(tempList[i])
```

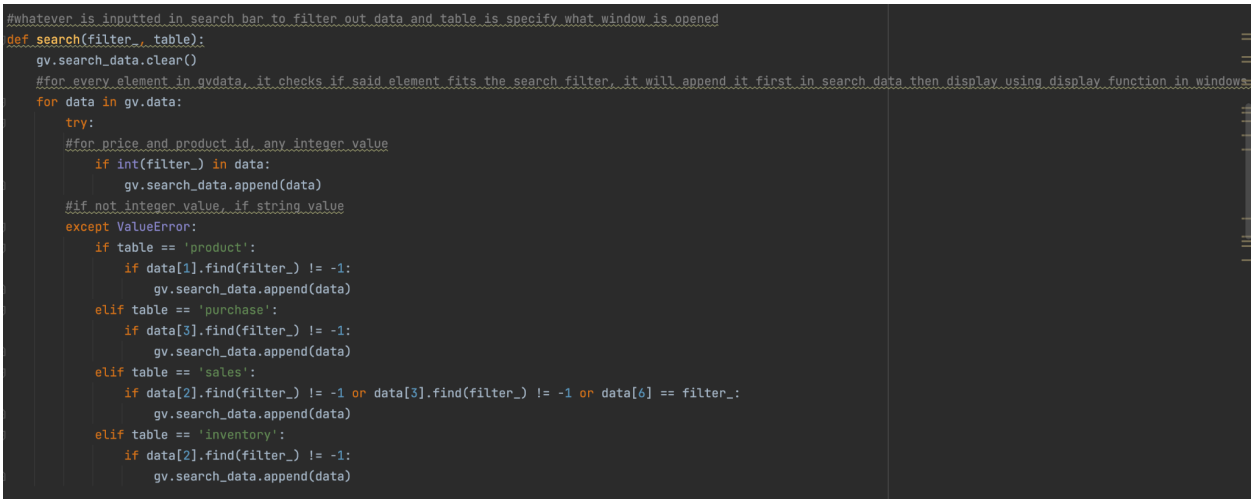
- FILE: utils.py

  This file contains codes for the interactive buttons in the inventory system. It determines which range of indexes will be displayed since the table can only display indexes 0-9 then 10-19 and so on and so forth. Basically, there are only 10 rows in each table.



```
import calendar
import csv
from tkinter import messagebox
from database import *
from datetime import date

#to add minimum/maximum indexes to display the next index ranges
def next_():
    gv.minimum += 10
    gv.maximum += 10

#to add minimum/maximum indexes to display the previous index ranges
def prev_():
    gv.minimum -= 10
    gv.maximum -= 10

#resets minimum and maximum to corresponding value to display initial default data
def reset():
    gv.minimum = 0
    gv.maximum = 10
    gv.search = False
```

This section contains codes to help extract data that was specifically searched for or filtered in the textbox and appends the data using a for loop. The try and except is simply for searches since the input might be integer or non-integer value.



```
#whatever is inputted in search bar to filter out data and table is specify what window is opened
def search(filter_, table):
    gv.search_data.clear()
    #for every element in gvdata, it checks if said element fits the search filter, it will append it first in search data then display using display function in windows
    for data in gv.data:
        try:
        #for price and product id, any integer value
            if int(filter_) in data:
                gv.search_data.append(data)
        #if not integer value, if string value
        except ValueError:
            if table == 'product':
                if data[1].find(filter_) != -1:
                    gv.search_data.append(data)
            elif table == 'purchase':
                if data[3].find(filter_) != -1:
                    gv.search_data.append(data)
            elif table == 'sales':
                if data[2].find(filter_) != -1 or data[3].find(filter_) != -1 or data[6] == filter_:
                    gv.search_data.append(data)
            elif table == 'inventory':
                if data[2].find(filter_) != -1:
                    gv.search_data.append(data)
```

This section contains codes that when input gives out an Error, a pop-up box returns a message on what the error is. All possible mistakes were thought of to cater to every error input in the textboxes. The last line of code is simply a def function for current date.

```
#to check which errors are inputted in the input bars and if mali, it will return a message what the error is
#assume all possible mistakes and give out error messages
def get_error(input_, window_, command_):
    gv.input_error = False
    gv.error_msg = ''
    #to check which window/tab opened
    if window_ == 'ListOfProduct':
        if command_ == 'add':
            for i in range(2):
                if input_[i] == '' or input_[i] == 0:
                    gv.input_error = True
                    if i == 0:
                        gv.error_msg += 'Please input product name\n'
                    elif i == 1:
                        gv.error_msg += 'Please input valid price\n'
                elif input_[0] != '':
                    product_name = get_data(table_name_='product', column_name_='product_name',
                                            filter_='product_name', filter_value_=f'{input_[0]}')
                    if product_name is not None:
                        gv.input_error = True
                        gv.error_msg = 'This Product Already Exist'
                        break
        elif command_ == 'edit' or command_ == 'delete':
            if input_ is None:
                gv.input_error = True
                if command_ == 'edit':
                    gv.error_msg = 'Cannot find the product name in database'
                else:
                    gv.error_msg = 'This product name doesnt exist'
            elif input_[0] == '':
                gv.input_error = True
                if command_ == 'edit':
                    gv.error_msg = 'Please input product name to edit'
                else:
                    gv.error_msg = 'Please input product name to delete'
    elif window_ == 'PurchaseHistory':
        if command_ == 'add':
            if input_ == 0:
                gv.input_error = True
                gv.error_msg += 'Please input valid QTY\n'

        elif command_ == 'edit' or command_ == 'delete':
            if input_ == 0 or input_ is None:
                gv.input_error = True
                if input_ == 0:
                    gv.error_msg += 'Please input valid ID\n'
                elif input_ is None:
                    gv.error_msg += 'ID not found\n'
        elif command_ == 'set_date':
            try:
                get_data(table_name_=input_)
            except sqlite3.OperationalError:
                gv.input_error = True
                gv.error_msg += 'No Data Found\n'
    elif window_ == 'Sales':
        if command_ == 'set_date':
            try:
                get_data(table_name_=input_)
            except sqlite3.OperationalError:
                gv.input_error = True
                gv.error_msg += 'No Data Found\n'
        elif command_ == 'import':
            try:
                with open(f'{input_}', 'r', encoding='utf-8') as file:
                    csv.reader(file)
            except FileNotFoundError:
                gv.input_error = True
                gv.error_msg += 'File Not Found\n'

    if gv.input_error is True:
        messagebox.showwarning(message=gv.error_msg)

#to get current date
def get_date():
    year, month, day = str(date.today()).split('-')
    return calendar.month_name[int(month)], int(day), int(year)
```

- FILE: windows.py

  This file contains all the codes to set up the inventory system, its windows and its buttons and the layout/style/positioning.

**IN GENERAL: For each class, a window is created with certain dimensions. A table is created as well with columns. Number of columns and column header names will differ per window. All “classes” also have a def display function which essentially displays the table created specific to that “class” or “window”. All classes also have an init function as a constructor for every instance that the “class” or “window” has been called. There are def add, delete, import, edit , and set date functions as well as shown below. Data is appended depending on what is looped and called. Database and inventory are updated according to what is input in the entry/text boxes for making changes and importing files. FIRST WINDOW LIST OF PRODUCTS:

```
from tkinter import *
from tkinter import ttk

import csv_handler
from fonts import *
from utils import *
from csv_handler import *

# ListOfProduct class is where you define the (List of Products) Window Layout and functionalities through the use of constructor and methods.
class ListOfProduct:
    #  __init__ function is the constructor of ListOfProduct class which is called everytime the instance of the class has been made.
    # it contains the layout of the List Of Products Window and the commands (user defined function) to be called for each specific button.
    def __init__(self):
        reset()
        self.window = Toplevel()
        self.window.geometry("589x600")
        Label(self.window, width=28, text="List of Products", font=times_new_roman, bg="pink").grid(row=0, columnspan=3)

        self.column_name1 = Entry(self.window, width=4, bg="light grey")
        self.column_name1.insert(0, "ID")
        self.column_name1.grid(row=1, column=0)

        self.column_name2 = Entry(self.window, width=50, bg="light grey")
        self.column_name2.insert(0, "PRODUCT NAME")
        self.column_name2.grid(row=1, column=1)

        self.column_name3 = Entry(self.window, width=9, bg="light grey")
        self.column_name3.insert(0, "PRICE")
        self.column_name3.grid(row=1, column=2)
        plus = 100

        self.prev_list = Button(self.window, text="< Prev", font=Sans_button, command=lambda: self.display('prev'))
        self.next_list = Button(self.window, text="Next >", font=Sans_button, command=lambda: self.display('next'))
        self.prev_list.place(x=54.5, y=250 + plus)
        self.next_list.place(x=104.5, y=250 + plus)

        self.search_box = Entry(self.window, width=20)
        self.search_box.place(x=180.5, y=250 + plus)

        self.search_button = Button(self.window, text="SEARCH", font=Sans_button, width=7,
                                    command=lambda: self.display('search'))
        self.search_button.place(x=389.5, y=250 + plus)

        self.clear_button = Button(self.window, text="CLEAR", font=Sans_button, width=7,
                                   command=lambda: self.display('clear'))
        self.clear_button.place(x=389.5, y=280 + plus)

        Label(self.window, text="Product Name", bg="pink", font=verdana).place(x=44.5, y=310 + plus)
        self.product_name = Entry(self.window, width=50)
        self.product_name.place(x=44.5, y=330 + plus)

        self.price_default = IntVar(self.window)
        Label(self.window, text="Price", bg="pink", font=verdana).place(x=44.5, y=360 + plus)
        self.price = Spinbox(self.window, from_=0, to=1000, width=9, textvariable=self.price_default)
        self.price.place(x=89.5, y=360 + plus)
```

```
        self.add_button = Button(self.window, text="ADD", font=Sans_button, width=7, command=self.add)
        self.add_button.place(x=289.5, y=360 + plus)

        self.edit_button = Button(self.window, text="EDIT", font=Sans_button, width=7, command=self.edit)
        self.edit_button.place(x=289.5, y=390 + plus)

        self.delete_button = Button(self.window, text="DELETE", font=Sans_button, width=7, command=self.delete)
        self.delete_button.place(x=289.5, y=420 + plus)

        self.display()

        self.window.resizable(False, False)
        self.window.configure(bg="pink")
        self.window.mainloop()
```

For def display function:

The following conditions handle which command is passed as an argument to the display function. If the command is 'next' it means that the user pressed the next button and means that the next_ function needs to be called to increase the data range to be displayed. If the command is 'prev' it means that the user pressed the prev button and means that the prev_ functions needs to be called to decrease the data range to be displayed if and only if the minimum range is greater than. If the command is 'search' it means that the user pressed the search button and means that the search function needs to be called to filter the data based on the given filter and to set the bool search to true. If the command is 'clear' it means that the user pressed the clear button and means that the search filter needs to be cleared and bool search needs to be set to false.

```
    # display function is used to display the data from the database to the application
    def display(self, command=''):
        get_data(table_name_='product')

        # the following conditions handles which command is passed as an argument to the display function
        # if the command is 'next' it means that the user pressed the next button and means that the next_ function needs to be called to increase the data range to be d
        # if the command is 'prev' it means that the user pressed the prev button and means that the prev_ functions needs to be called to decrease the data range to be
        # if the command is 'search' it means that the user pressed the search button and means that the search function needs to be called to filter the data based on t
        # if the command is 'clear' it means that the user pressed the clear button and means that the search filter needs to be cleared and bool search need to be set t
        if command == 'next':
            # the following conditions is responsible for checking if the size of the data extracted from the database is greater than or equal to the current range
            # if the size of data extracted is valid the next_ function is called
            # if the size of search_data extracted is valid the next_ function is called
            if not gv.search and len(gv.data) - 1 >= gv.maximum:
                next_()
            if gv.search and len(gv.search_data) - 1 >= gv.maximum:
                next_()
        elif command == 'prev' and gv.minimum > 0:
            prev_()
        elif command == 'search':
            reset()
            # the following condition is responsible for checking if the user inputted some filter after clicking the search button
            # if the user inputted a non empty filter then the following codes will be executed
            if self.search_box.get() != '':
                search(self.search_box.get(), 'product')
                gv.search = True
        elif command == 'clear':
            self.search_box.delete(0, 'end')
            gv.search = False
```

```
    # if there is no search filter, all data from gv.data is appended to display data
    # if search is true, data appended will be dependent on search filter
    # if range of appended data is less than the maximum, other rows will be kept empty
    display_data = []
    if len(gv.data) != 0:
        if gv.search is True:
            if len(gv.search_data) != 0:
                for i in range(gv.minimum, gv.maximum):
                    display_data.append(gv.search_data[i])
                    if i == len(gv.search_data) - 1:
                        for j in range(gv.maximum - len(display_data)):
                            display_data.append(('', '', ''))
                        break
        else:
            for i in range(gv.minimum, gv.maximum):
                display_data.append(gv.data[i])
                if i == len(gv.data) - 1:
                    for j in range(gv.maximum - len(display_data)):
                        display_data.append(('', '', ''))
                    break

    # displays the data from display_data list
    for i in range(2, 12):
        color = "pink" if i % 2 == 0 else "light grey"

        (product_id, product_name, price) = display_data[i - 2] if len(display_data) != 0 else ('', '', '')

        column_1 = Entry(self.window, width=4, bg=color)
        column_1.grid(row=i, column=0)
```

For def input delete function:

The code simply deletes what was typed in the textbox, assuming that it's found in that window’s table display.

For def add function:

The function is for the add button and used to add product in database. If there is no error, it calls the add_product function in database.py and inventory system is updated.

```
        column_1.insert(0, product_id)

        column_2 = Entry(self.window, width=50, bg=color)
        column_2.grid(row=i, column=1)
        column_2.insert(0, product_name)

        column_3 = Entry(self.window, width=9, bg=color)
        column_3.grid(row=i, column=2)
        column_3.insert(0, price)

# input_delete function is used to delete what was typed in textbox
def input_delete(self):
    self.product_name.delete(0, 'end')
    self.price_default.set(0)

# add function is specifically for add button and is used to add product in database
# if no error(calling add_product function in database.py)
# after input data is added in database, inventory system is also updated
def add(self):
    product = (self.product_name.get(), int(self.price.get()))
    get_error(input_=product, window_='ListOfProduct', command_='add')

    if gv.input_error is False:
        self.input_delete()
        add_product(product)
        self.display()
```

For def edit function:

The following function is specifically for the edit button and is used to edit the product in the database. It can correct spelling and price. Data will be extracted from database if no error(calling get_data function in database.py), based on which product the user wants to edit and will allow the data to be editable in the inventory system. After input data is edited in database, inventory system is also updated.

```
    # edit function is specifically for edit button and is used to edit the product in database
    # data will be extracted from database if no error(calling get_data function in database.py), based from which product the user wants to edit and will allow the da
    # can correct spelling and price
    # after input data is edited in database, inventory system is also updated
    def edit(self):
        prev_data = tuple()
        get_error(input_=(self.product_name.get(),), window_='ListOfProduct', command_='edit')
        if gv.input_error is False:
            prev_data = get_data(table_name_='product', column_name_='product_name, price',
                                 filter_='product_name', filter_value_=self.product_name.get())
            get_error(input_=prev_data, window_='ListOfProduct', command_='edit')

        if gv.input_error is False:
            self.add_button.config(state=DISABLED)
            self.edit_button.config(state=DISABLED)
            self.delete_button.config(state=DISABLED)

            self.price_default.set(prev_data[1])

            save_button = Button(self.window, text="SAVE", font=Sans_button, width=7, command=lambda: save_edit())
            save_button.place(x=289.5, y=450)

        # save_edit function is used to save and update the edited data back into the database if no error(calling update function in database.py)
        def save_edit():
            self.add_button.config(state=NORMAL)
            self.edit_button.config(state=NORMAL)
            self.delete_button.config(state=NORMAL)
            product_name, price = (self.product_name.get(), int(self.price.get()))


            product_value = f"product_name = '{product_name}', price = {price}"
            update(table_name_='product', set_value=product_value,
                   filter_='product_name', filter_value_=prev_data[0])

            save_button.destroy()

            self.input_delete()
            self.display()
```

For def delete function:

The delete function is specifically for delete button and is used to delete product in database. Data that will be deleted will be extracted from the database if no error(calling get_data function in database.py). Ddata will be deleted from the database if no error(calling delete function in database.py), based from which product the user wants to delete. After input data is deleted in database, inventory system is also updated

```
    # delete function is specifically for delete button and is used to delete product in database
    # data that will be deleted will be extracted from the database if no error(calling get_data function in database.py)
    # data will be deleted from the database if no error(calling delete function in database.py), based from which product the user wants to delete
    # after input data is deleted in database, inventory system is also updated
    def delete(self):
        prev_data = tuple()
        get_error(input_=(self.product_name.get(),), window_='ListOfProduct', command_='delete')

        if gv.input_error is False:
            prev_data = get_data(table_name_='product', column_name_='product_name',
                                 filter_='product_name', filter_value_=self.product_name.get())
            print(prev_data)
            get_error(input_=prev_data, window_='ListOfProduct', command_='delete')

        if gv.input_error is False:
            delete(table_name_='product', filter_='product_name', filter_value_=prev_data[0])
            delete(table_name_='inventory', filter_='product_name', filter_value_=prev_data[0])
            self.input_delete()
            self.display()
```

SECOND WINDOW (ADD STOCK):

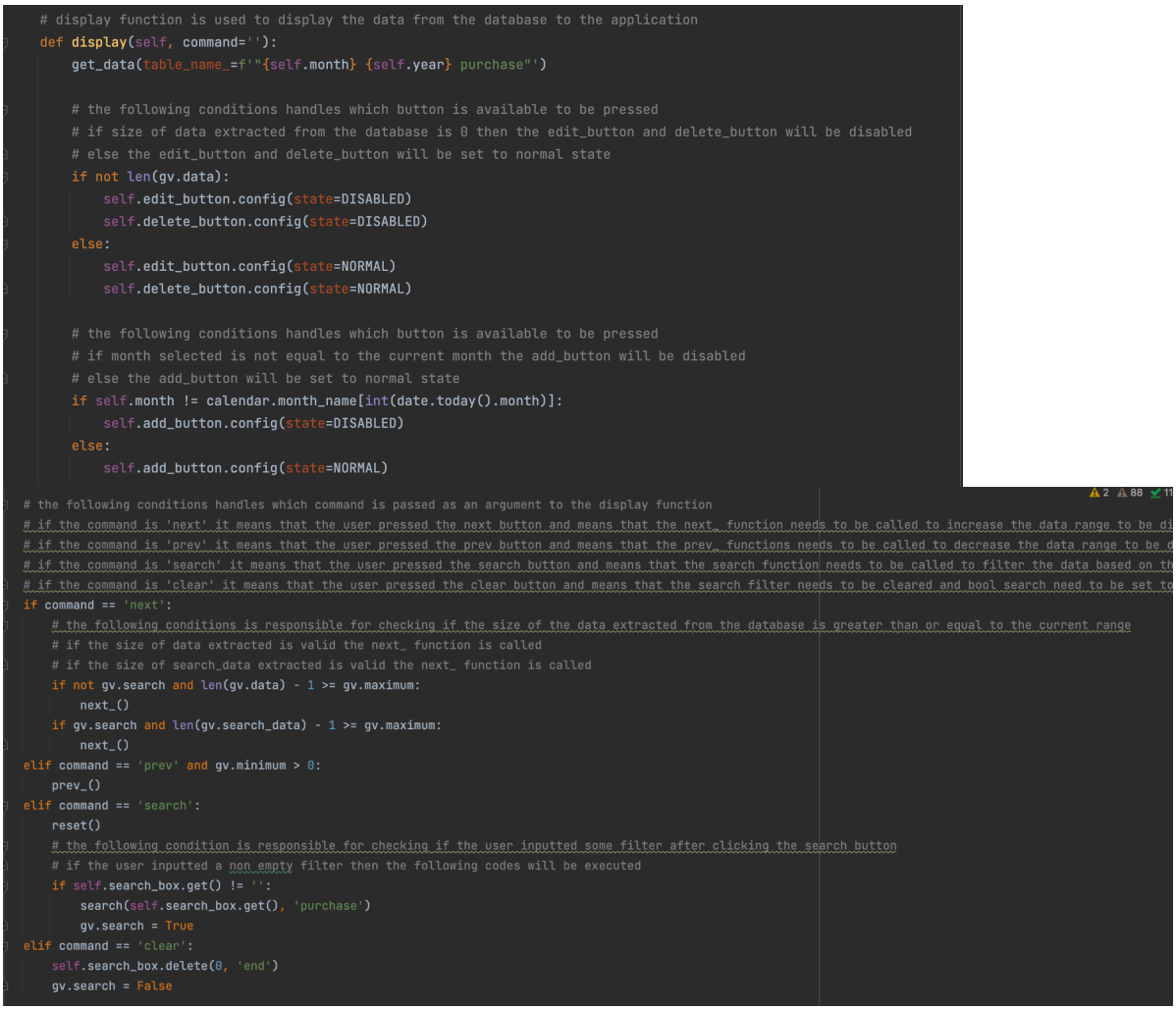
LAYOUT/COMMANDSFUNCTIONALITY SAME AS PREVIOUS WINDOW WITH SLIGHT MODIFICATIONS

```
# AddStock class is where you define the (Add Stock) Window Layout and functionalities through the use of constructor and methods.
class AddStock:
    # __init__ function is the constructor of Add Stock class which is called everytime the instance of the class has been made.
    # it contains the layout of the Add Stock Window and the commands (user defined function) to be called for each specific button.
    def __init__(self):
        reset()
        self.window = Toplevel()
        self.window.geometry('630x700')

        self.month, self.day, self.year = get_date()
        print(self.month, self.day, self.year)
        create_table(self.month, self.year, 'PurchaseHistory')

        self.month_label = StringVar(self.window)
        self.month_label.set(f"{self.month} Stock")
        Label(self.window, textvariable=self.month_label,
              font=times_new_roman, bg="pink").grid(row=0, columnspan=5)

        self.column_name1 = Entry(self.window, width=4, bg="light grey")
        self.column_name1.insert(0, "ID")
        self.column_name1.grid(row=1, column=0)

        self.column_name2 = Entry(self.window, width=10, bg="light grey")
        self.column_name2.insert(0, "DATE")
        self.column_name2.grid(row=1, column=1)

        self.column_name3 = Entry(self.window, width=5, bg="light grey")
        self.column_name3.insert(0, "QTY")
```

```
        self.column_name3.grid(row=1, column=2)

        self.column_name4 = Entry(self.window, width=50, bg="light grey")
        self.column_name4.insert(0, "PRODUCT NAME")
        self.column_name4.grid(row=1, column=3)

        plus = 100

        self.prev_list = Button(self.window, text="< Prev", font=Sans_button, command=lambda: self.display('prev'))
        self.next_list = Button(self.window, text="Next >", font=Sans_button, command=lambda: self.display('next'))
        self.prev_list.place(x=82.25, y=250 +plus)
        self.next_list.place(x=132.25, y=250 +plus)

        self.search_box = Entry(self.window, width=20)
        self.search_box.place(x=230, y=250 +plus)

        self.search_button = Button(self.window, text="SEARCH", font=Sans_button, width=7,
                                    command=lambda: self.display('search'))
        self.search_button.place(x=460, y=250 +plus)

        self.clear_button = Button(self.window, text="CLEAR", font=Sans_button, width=7,
                                   command=lambda: self.display('clear'))
        self.clear_button.place(x=460, y=280 +plus)

        Label(self.window, text="Product Name", bg="pink", font=verdana).place(x=14.5, y=310 +plus)
        self.product_name = ttk.Combobox(self.window, width=50, textvariable=StringVar())
        product_list = ['No Product Yet']
        for i in get_data(table_name_='product', column_name_='product_name'):
            product_list.append(i[0])
            if 'No Product Yet' in product_list:
                product_list.remove('No Product Yet')
        self.product_name['values'] = product_list
        self.product_name.place(x=14.5, y=330 +plus)
        self.product_name.current(0)

        self.qty_default = IntVar(self.window)
        Label(self.window, text="QTY", bg="pink", font=verdana).place(x=500.5, y=310 +plus)
        self.qty = Spinbox(self.window, from_=0, to=1000, width=8, textvariable=self.qty_default)
        self.qty.place(x=500.5, y=330 +plus)

        self.id_default = IntVar(self.window)
        Label(self.window, text="ID", bg="pink", font=verdana).place(x=14.5, y=360 +plus)
        self.purchase_id = Spinbox(self.window, from_=0, to=1000, width=8, textvariable=self.id_default)
        self.purchase_id.place(x=44.5, y=360 +plus)

        self.add_button = Button(self.window, text="ADD", font=Sans_button, width=7, command=self.add)
        self.add_button.place(x=360, y=360 +plus)

        self.edit_button = Button(self.window, text="EDIT", font=Sans_button, width=7, command=self.edit)
        self.edit_button.place(x=360, y=390 +plus)

        self.delete_button = Button(self.window, text="DELETE", font=Sans_button, width=7, command=self.delete)
        self.delete_button.place(x=360, y=420 +plus)

        self.month_cb = ttk.Combobox(self.window, width=11, textvariable=StringVar())
        month_list = []
        for i in calendar.month_name:
            if '' in month_list:
                month_list.remove('')
            month_list.append(i)
        self.month_cb['values'] = month_list
        self.month_cb.place(x=14.5, y=420 +plus)
        self.month_cb.current(0)

        self.year_default = IntVar(self.window)
        self.year_sp = Spinbox(self.window, width=5, from_=2000, to_=3000, textvariable=self.year_default)
        self.year_sp.place(x=135.5, y=420 +plus)
        self.year_default.set(self.year)

        self.date_button = Button(self.window, text="GO", font=Sans_button, width=3, command=self.set_date)
        self.date_button.place(x=230.5, y=420 +plus)

        if 'No Product Yet' in product_list:
            self.add_button.config(state=DISABLED)

        self.display()

        self.window.resizable(False, False)
        self.window.configure(bg="pink")
        self.window.mainloop()
```

For def display function:

The following conditions handle which command is passed as an argument to the display function. If the command is 'next' it means that the user pressed the next button and means that the next_ function needs to be called to increase the data range to be displayed. If the command is 'prev' it means that the user pressed the prev button and means that the prev_ functions needs to be called to decrease the data range to be displayed if and only if the minimum range is greater than. If the command is 'search' it means that the user pressed the search button and means that the search function needs to be called to filter the data based on the given filter and to set the bool search to true. If the command is 'clear' it means that the user pressed the clear button and means that the search filter needs to be cleared and bool search needs to be set to false.



```
    # display function is used to display the data from the database to the application
    def display(self, command=''):
        get_data(table_name_=f'"{self.month} {self.year} purchase"')

        # the following conditions handles which button is available to be pressed
        # if size of data extracted from the database is 0 then the edit_button and delete_button will be disabled
        # else the edit_button and delete_button will be set to normal state
        if not len(gv.data):
            self.edit_button.config(state=DISABLED)
            self.delete_button.config(state=DISABLED)
        else:
            self.edit_button.config(state=NORMAL)
            self.delete_button.config(state=NORMAL)

        # the following conditions handles which button is available to be pressed
        # if month selected is not equal to the current month the add_button will be disabled
        # else the add_button will be set to normal state
        if self.month != calendar.month_name[int(date.today().month)]:
            self.add_button.config(state=DISABLED)
        else:
            self.add_button.config(state=NORMAL)
```

```
# the following conditions handles which command is passed as an argument to the display function
# if the command is 'next' it means that the user pressed the next button and means that the next_ function needs to be called to increase the data range to be di
# if the command is 'prev' it means that the user pressed the prev button and means that the prev_ functions needs to be called to decrease the data range to be d
# if the command is 'search' it means that the user pressed the search button and means that the search function needs to be called to filter the data based on th
# if the command is 'clear' it means that the user pressed the clear button and means that the search filter needs to be cleared and bool search need to be set to
if command == 'next':
    # the following conditions is responsible for checking if the size of the data extracted from the database is greater than or equal to the current range
    # if the size of data extracted is valid the next_ function is called
    # if the size of search_data extracted is valid the next_ function is called
    if not gv.search and len(gv.data) - 1 >= gv.maximum:
        next_()
    if gv.search and len(gv.search_data) - 1 >= gv.maximum:
        next_()
elif command == 'prev' and gv.minimum > 0:
    prev_()
elif command == 'search':
    reset()
    # the following condition is responsible for checking if the user inputted some filter after clicking the search button
    # if the user inputted a non empty filter then the following codes will be executed
    if self.search_box.get() != '':
        search(self.search_box.get(), 'purchase')
        gv.search = True
elif command == 'clear':
    self.search_box.delete(0, 'end')
    gv.search = False
```

```
    # if there is no search filter, all data from gv.data is appended to display data
    # if search is true, data appended will be dependent on search filter
    # if range of appended data is less than the maximum, other rows will be kept empty
    display_data = []
    if len(gv.data) != 0:
        if gv.search is True:
            if len(gv.search_data) != 0:
                for i in range(gv.minimum, gv.maximum):
                    display_data.append(gv.search_data[i])
                    if i == len(gv.search_data) - 1:
                        for j in range(gv.maximum - len(display_data)):
                            display_data.append(('', '', '', ''))
                        break
        else:
            for i in range(gv.minimum, gv.maximum):
                display_data.append(gv.data[i])
                if i == len(gv.data) - 1:
                    for j in range(gv.maximum - len(display_data)):
                        display_data.append(('', '', '', ''))
                    break

    # displays the data from display_data list
    for i in range(2, 12):
        color = "pink" if i % 2 == 0 else "light grey"


        (purchase_id, date_purchased, qty, product_name) = \
            display_data[i - 2] if len(display_data) != 0 else ('', '', '', '')

        column_1 = Entry(self.window, width=4, bg=color)
        column_1.grid(row=i, column=0)
        column_1.insert(0, purchase_id)

        column_2 = Entry(self.window, width=10, bg=color)
        column_2.grid(row=i, column=1)
        column_2.insert(0, date_purchased)

        column_3 = Entry(self.window, width=5, bg=color)
        column_3.grid(row=i, column=2)
        column_3.insert(0, qty)

        column_4 = Entry(self.window, width=50, bg=color)
        column_4.grid(row=i, column=3)
        column_4.insert(0, product_name)
```

For def inputdelete function:

The delete function is used to delete what is typed in the textbox.

```
    # input_delete function is used to delete what was typed in textbox
    def input_delete(self):
        self.qty_default.set(0)
        self.id_default.set(0)
```

For def add function:

The function is for the add button and used to add stock in database. If there is no error, it calls the add_product function in database.py and inventory system is updated.

```
    # add function is specifically for add button and is used to add stock in database
    # if no error(calling purchase function in database.py)
    # after input data is added in database, inventory system is also updated
    def add(self):
        get_error(input_=int(self.qty.get()), window_='PurchaseHistory', command_='add')

        if gv.input_error is False:
            product = (date.today(), int(self.qty.get()), self.product_name.get())

            purchase(self.month, self.year, product)

            self.input_delete()
            self.display()
```

```python
# edit function is specifically for edit button and is used to edit the product in database
# data will be extracted from database if no error(calling get_data function in database.py), based from which product the user wants to edit and will a
# can correct quantity
# after input data is edited in database, inventory system is also updated
def edit(self):
    get_error(input_=int(self.purchase_id.get()), window_='PurchaseHistory', command_='edit')

    if gv.input_error is False:
        prev_data = get_data(table_name_=f'"{self.month} {self.year} purchase"',
                             column_name_='id, date, QTY, product_name',
                             filter_='id', filter_value_=f'{self.purchase_id.get()}')
        get_error(input_=prev_data, window_='PurchaseHistory', command_='edit')

        if gv.input_error is False:
            self.add_button.config(state=DISABLED)
            self.edit_button.config(state=DISABLED)
            self.delete_button.config(state=DISABLED)

            self.id_default.set(prev_data[0])
            self.qty_default.set(prev_data[2])
            self.product_name.set(prev_data[3])

            save_button = Button(self.window, text="SAVE", font=Sans_button, width=7, command=lambda: save_edit())
            save_button.place(x=390, y=450)

    # save_edit function is used to save and update the edited data back into the database if no error(calling update function in database.py)
    def save_edit():
        self.add_button.config(state=NORMAL)
        self.edit_button.config(state=NORMAL)
        self.delete_button.config(state=NORMAL)

        if int(self.qty.get()) != prev_data[2]:
            qty = int(self.qty.get())
            inventory_qty, rsp = get_data(table_name_='inventory', column_name_='QTY, RSP',
                                          filter_='product_name', filter_value_=f'{prev_data[3]}')
            new_amount = qty * rsp

            update(table_name_=f"'{self.month} {self.year} purchase'",
                   set_value=f"QTY = {qty}, amount = {new_amount}",
                   filter_='id', filter_value_=prev_data[0])

            purchase_total = 0
            for i, in get_data(table_name_=f"'{self.month} {self.year} purchase'", column_name_='amount',
                               filter_='date', filter_value_=prev_data[1], fetch='all'):
                purchase_total += i

            update(table_name_=f"'{self.month} {self.year} purchase_total'",
                   set_value=f"total = {purchase_total}",
                   filter_='date', filter_value_=prev_data[1])

            if qty > prev_data[2]:
                inventory_qty += (qty - prev_data[2])
            else:
                inventory_qty -= (prev_data[2] - qty)
```

```
            update(table_name_='inventory', set_value=f'QTY = {inventory_qty}',
                   filter_='product_name', filter_value_=prev_data[3])
        elif self.product_name.get() != prev_data[3]:
            product_name = self.product_name.get()
            rsp, = get_data(table_name_='inventory', column_name_='RSP',
                            filter_='product_name', filter_value_=f'{product_name}')
            new_amount = prev_data[2] * rsp

            update(table_name_=f"'{self.month} {self.year} purchase'",
                   set_value=f"product_name = '{product_name}', amount = {new_amount}",
                   filter_='id', filter_value_=prev_data[0])

            purchase_total = 0
            for i, in get_data(table_name_=f"'{self.month} {self.year} purchase'", column_name_='amount',
                               filter_='date', filter_value_=prev_data[1], fetch='all'):
                purchase_total += i

            update(table_name_=f"'{self.month} {self.year} purchase_total'",
                   set_value=f"total = {purchase_total}",
                   filter_='date', filter_value_=prev_data[1])

            inventory_qty, = get_data(table_name_='inventory', column_name_='QTY',
                                      filter_='product_name', filter_value_=f'{prev_data[3]}')
            inventory_qty -= prev_data[2]
            update(table_name_='inventory', set_value=f'QTY = {inventory_qty}',
                   filter_='product_name', filter_value_=prev_data[3])

            inventory_qty, = get_data(table_name_='inventory', column_name_='QTY',
                                      filter_='product_name', filter_value_=f'{product_name}')
            inventory_qty += prev_data[2]
            update(table_name_='inventory', set_value=f'QTY = {inventory_qty}',
                   filter_='product_name', filter_value_=product_name)

        self.input_delete()
        save_button.destroy()
        self.display()
```

For def delete function:

The delete function is specifically for the delete button and is used to delete stock in the database. Data that will be deleted will be extracted from the database if there is no error(calling get_data function in database.py). Ddata will be deleted from the database if there is no error(calling delete function in database.py), based on which product the user wants to delete. After input data is deleted in database, inventory system is also updated

```
# delete function is specifically for delete button and is used to delete product in database
# data that will be deleted will be extracted from the database if no error(calling get_data function in database.py)
# data will be deleted from the database if no error(calling delete function in database.py), based from which product the user wants to delete
# after input data is deleted in database, inventory system is also updated
def delete(self):
    get_error(input_=int(self.purchase_id.get()), window_='PurchaseHistory', command_='delete')

    if gv.input_error is False:
        prev_data = get_data(table_name_=f'"{self.month} {self.year} purchase"',
                             column_name_='id, date, QTY, product_name',
                             filter_='id', filter_value_=f'{self.purchase_id.get()}')
        get_error(input_=prev_data, window_='PurchaseHistory', command_='delete')

        if gv.input_error is False:
            delete(table_name_=f'"{self.month} {self.year} purchase"', filter_='id', filter_value_=prev_data[0])

            inventory_qty, = get_data(table_name_='inventory', column_name_='QTY',
                                      filter_='product_name', filter_value_=f'{prev_data[3]}')

            purchase_total = 0
            for i, in get_data(table_name_=f"'{self.month} {self.year} purchase'", column_name_='amount',
                               filter_='date', filter_value_=prev_data[1], fetch='all'):
                purchase_total += i

            update(table_name_=f"'{self.month} {self.year} purchase_total'",
                   set_value=f"total = {purchase_total}",
                   filter_='date', filter_value_=prev_data[1])
```

```
            inventory_qty -= prev_data[2]

            update(table_name_='inventory', set_value=f'QTY = {inventory_qty}',
                   filter_='product_name', filter_value_=prev_data[3])

            self.input_delete()
            self.display()
```

For set date function:

This allows user to use the go button and date textboxes to set the month and year and view the Stock Table on the specified date if there is data in the database on that date.

```
# set_date function is specifically for go button and is used to set the month and year which will be used to specify which addstock/purchase table you wanted to view
# if the sales table is found on the data base, the data retrieved will be shown in the inventory system
def set_date(self):
    get_error(input_=f'"{self.month_cb.get()} {self.year_sp.get()} purchase"',
              window_="PurchaseHistory", command_="set_date")
    if gv.input_error is False:
        self.month = self.month_cb.get()
        self.year = self.year_sp.get()
        self.month_label.set(f"{self.month} Purchase History")
        print(self.month, self.year)
        reset()
        self.display()
```

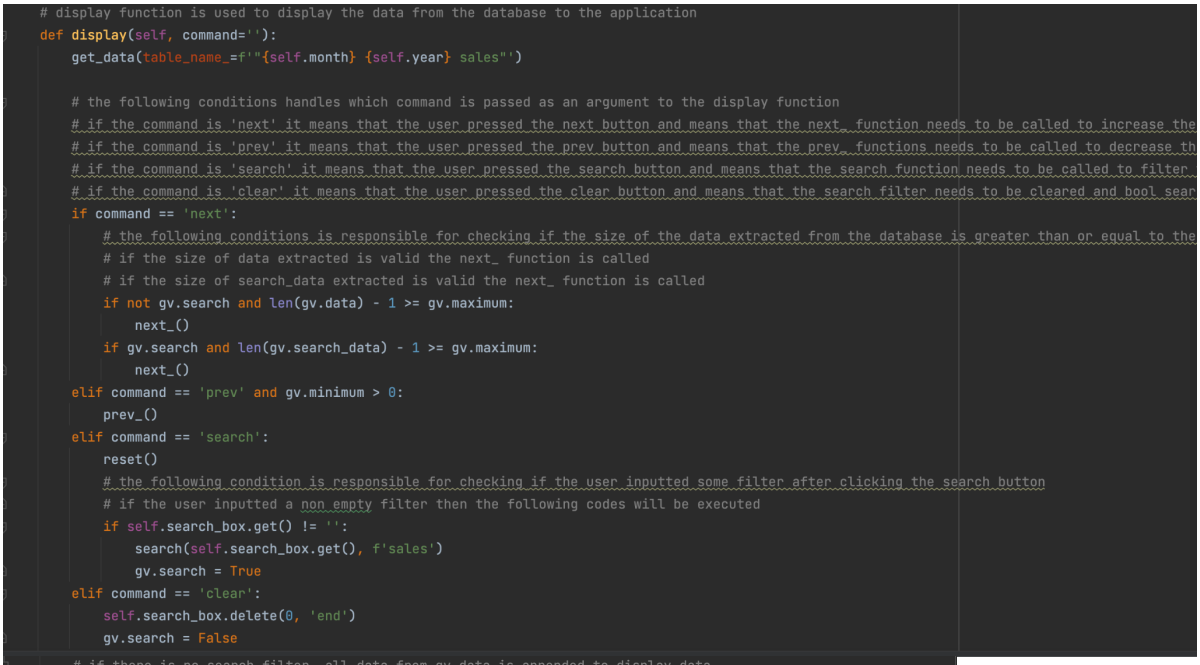
THIRD WINDOW(SALES): SAME LAYOUT/COMMANDS/FUNCTIONALITY WITH SLIGHT MODIFICATIONS

```
# Sales class is where you define the (Sales) Window Layout and functionalities through the use of constructor and methods.
class Sales:
    # __init__ function is the constructor of Sales class which is called everytime the instance of the class has been made.
    # it contains the layout of the Sales Window and the commands (user defined function) to be called for each specific button.
    def __init__(self):
        reset()
        self.window = Toplevel()
        self.window.geometry("722x500")

        self.month, self.day, self.year = get_date()
        print(self.month, self.day, self.year)
        create_table(self.month, self.year, 'Sales')

        self.month_label = StringVar(self.window)
        self.month_label.set(f"{self.month} Sales")
        Label(self.window, width=28, textvariable=self.month_label,
              font=times_new_roman, bg="pink").grid(row=0, columnspan=7)

        self.column_name1 = Entry(self.window, width=4, bg="#FF3399")
        self.column_name1.insert(0, "ID")
        self.column_name1.grid(row=1, column=0)

        self.column_name2 = Entry(self.window, width=12, bg="#FF3399")
        self.column_name2.insert(0, "DATE")
        self.column_name2.grid(row=1, column=1)

        self.column_name3 = Entry(self.window, width=20, bg="#FF3399")
        self.column_name3.insert(0, "CUSTOMER NAME")
        self.column_name3.grid(row=1, column=2)
```

```
        self.column_name4 = Entry(self.window, width=50, bg="#FF3399")
        self.column_name4.insert(0, "PRODUCT NAME")
        self.column_name4.grid(row=1, column=3)

        self.column_name5 = Entry(self.window, width=5, bg="#FF3399")
        self.column_name5.insert(0, "QTY")
        self.column_name5.grid(row=1, column=4)

        self.column_name6 = Entry(self.window, width=7, bg="#FF3399")
        self.column_name6.insert(0, "PRICE")
        self.column_name6.grid(row=1, column=5)

        self.column_name8 = Entry(self.window, width=9, bg="#FF3399")
        self.column_name8.insert(0, "AMOUNT")
        self.column_name8.grid(row=1, column=6)

        plus = 100

        self.prev_list = Button(self.window, text="< Prev", font=Sans_button, command=lambda: self.display('prev'))
        self.next_list = Button(self.window, text="Next >", font=Sans_button, command=lambda: self.display('next'))
        self.prev_list.place(x=208.5, y=250 +plus)
        self.next_list.place(x=258.5, y=250 +plus)

        self.search_box = Entry(self.window, width=20)
        self.search_box.place(x=332.375, y=250 +plus)

        self.search_button = Button(self.window, text="SEARCH", font=Sans_button, width=7,
                                    command=lambda: self.display('search'))
        self.search_button.place(x=550.375, y=250 +plus)
        self.clear_button = Button(self.window, text="CLEAR", font=Sans_button, width=7,
                                   command=lambda: self.display('clear'))
        self.clear_button.place(x=550.375, y=280 +plus)

        minus = 75

        Label(self.window, text="CSV Filename", bg="pink", font=verdana).place(x=163.375 - minus, y=310 +plus)
        self.csv_filename = Entry(self.window, width=59)
        self.csv_filename.place(x=163.375 - minus, y=330 +plus)

        self.import_button = Button(self.window, text="IMPORT", font=Sans_button, width=7,
                                    command=lambda: self.import_())
        self.import_button.place(x=630.375, y=330 +plus)

        self.id_default = IntVar(self.window)
        Label(self.window, text="ID", bg="pink", font=verdana).place(x=163.375 - 75, y=360 +plus)
        self.purchase_id = Spinbox(self.window, from_=0, to=1000, width=8, textvariable=self.id_default)
        self.purchase_id.place(x=163.375 - 45, y=360 +plus)

        self.month_cb = ttk.Combobox(self.window, width=11, textvariable=StringVar())
        month_list = []
        for i in calendar.month_name:
            if '' in month_list:
                month_list.remove('')
            month_list.append(i)
        self.month_cb['values'] = month_list
        self.month_cb.place(x=163.375 - minus, y=420 +plus)
        self.month_cb.current(0)

        self.year_default = IntVar(self.window)
        self.year_sp = Spinbox(self.window, width=5, from_=2000, to_=3000, textvariable=self.year_default)
        self.year_sp.place(x=254.375 - minus, y=420 +plus)
        self.year_default.set(self.year)

        self.date_button = Button(self.window, text="GO", font=Sans_button, width=3, command=self.set_date)
        self.date_button.place(x=308.375 - minus, y=420 +plus)

        self.display()

        self.window.resizable(False, False)
        self.window.configure(bg="pink")
        self.window.mainloop()
```

For def display function:

The following conditions handle which command is passed as an argument to the display function. If the command is 'next' it means that the user pressed the next button and means that the next_ function needs to be called to increase the data range to be displayed. If the command is 'prev', the prev_ functions needs to be called to decrease the data range to be displayed if and only if the minimum range is greater than. If the command is 'search' it means that the user pressed the search button and means that the search function needs to be called to filter the data based on the given filter and to set the bool search to true. If the command is 'clear' it means that the user pressed the clear button and means that the search filter needs to be cleared and bool search needs to be set to false.



```
    # display function is used to display the data from the database to the application
    def display(self, command=''):
        get_data(table_name_=f'"{self.month} {self.year} sales"')

        # the following conditions handles which command is passed as an argument to the display function
        # if the command is 'next' it means that the user pressed the next button and means that the next_ function needs to be called to increase the
        # if the command is 'prev' it means that the user pressed the prev button and means that the prev_ functions needs to be called to decrease th
        # if the command is 'search' it means that the user pressed the search button and means that the search function needs to be called to filter
        # if the command is 'clear' it means that the user pressed the clear button and means that the search filter needs to be cleared and bool sear
        if command == 'next':
            # the following conditions is responsible for checking if the size of the data extracted from the database is greater than or equal to the
            # if the size of data extracted is valid the next_ function is called
            # if the size of search_data extracted is valid the next_ function is called
            if not gv.search and len(gv.data) - 1 >= gv.maximum:
                next_()
            if gv.search and len(gv.search_data) - 1 >= gv.maximum:
                next_()
        elif command == 'prev' and gv.minimum > 0:
            prev_()
        elif command == 'search':
            reset()
            # the following condition is responsible for checking if the user inputted some filter after clicking the search button
            # if the user inputted a non empty filter then the following codes will be executed
            if self.search_box.get() != '':
                search(self.search_box.get(), f'sales')
                gv.search = True
        elif command == 'clear':
            self.search_box.delete(0, 'end')
            gv.search = False
        # if there is no search filter, all data from gv.data is appended to display data
        # if search is true, data appended will be dependent on search filter
        # if range of appended data is less than the maximum, other rows will be kept empty
        display_data = []
        if len(gv.data) != 0:
            if gv.search is True:
                if len(gv.search_data) != 0:
                    for i in range(gv.minimum, gv.maximum):
                        display_data.append(gv.search_data[i])
                        if i == len(gv.search_data) - 1:
                            for j in range(gv.maximum - len(display_data)):
                                display_data.append(('', '', '', '', '', '', ''))
                            break
            else:
                for i in range(gv.minimum, gv.maximum):
                    display_data.append(gv.data[i])
                    if i == len(gv.data) - 1:
                        for j in range(gv.maximum - len(display_data)):
                            display_data.append(('', '', '', '', '', '', ''))
                        break

        # displays the data from display_data list
        for i in range(2, 12):
            color = "light grey"
            if len(display_data) != 0 and display_data[i - 2][2].find('Total') != -1:
                color = "#FF99CC"

            (sales_id, date_sold, customer_name, product_name, qty, price, amount) = \
                display_data[i - 2] if len(display_data) != 0 else ('', '', '', '', '', '', '')
```

```
            if customer_name.find('Total') != -1:
                product_name = ''

            column_1 = Entry(self.window, width=4, bg=color)
            column_1.grid(row=i, column=0)
            column_1.insert(0, sales_id)

            column_2 = Entry(self.window, width=12, bg=color)
            column_2.grid(row=i, column=1)
            column_2.insert(0, date_sold)

            column_3 = Entry(self.window, width=20, bg=color)
            column_3.grid(row=i, column=2)
            column_3.insert(0, customer_name)

            column_4 = Entry(self.window, width=50, bg=color)
            column_4.grid(row=i, column=3)
            column_4.insert(0, product_name)

            column_5 = Entry(self.window, width=5, bg=color)
            column_5.grid(row=i, column=4)
            column_5.insert(0, qty)

            column_6 = Entry(self.window, width=7, bg=color)
            column_6.grid(row=i, column=5)
            column_6.insert(0, price)

            column_8 = Entry(self.window, width=9, bg=color)
            column_8.grid(row=i, column=6)
            column_8.insert(0, amount)
```

For def set date function:

This allows user to use the go button and date textboxes to set the month and year and view the Sales Table on the specified date if there is data in the database on that date.

For def import function:

This allows user to import the csv file to the system where it will be read and its data will be processed to update stock in inventory table in the next window.

```
    # set_date function is specifically for go button and is used to set the month and year which will be used to specify which sales table you wanted to view
    # if the sales table is found on the data base, the data retrieved will be shown in the inventory system
    def set_date(self):
        get_error(input_=f'"{self.month_cb.get()} {self.year_sp.get()} sales"',
                  window_="Sales", command_="set_date")
        if gv.input_error is False:
            self.month = self.month_cb.get()
            self.year = self.year_sp.get()
            self.month_label.set(f"{self.month} Sales")
            print(self.month, self.year)
            reset()
            self.display()

    # import_ function is used to import the csv file to the system to be read, the data from csv file is then processed to update the stock in the inventory table
    def import_(self):
        filename = self.csv_filename.get()
        get_error(input_=f'{filename}', window_="Sales", command_="import")
        if gv.input_error is False:
            (month, year) = str(filename).split(' - ')[0].split(' ')
            create_table(month, year, 'Sales')
            csv_handler.read_csv(filename)
            sell(month, year)
            csv_handler.write_csv(filename=filename)
            self.display()
```

FOURTH WINDOW(VIEW INVENTORY): SAME LAYOUT/COMMANDS/FUNCTIONALITY WITH SLIGHT MODIFICATIONS

```
# Inventory class is where you define the (Inventory) Window Layout and functionalities through the use of constructor and methods.
class Inventory:
    # __init__ function is the constructor of Inventory class which is called everytime the instance of the class has been made.
    # it contains the layout of the Inventory Window and the commands (user defined function) to be called for each specific button.
    def __init__(self):
        reset()
        self.window = Toplevel()
        self.window.geometry("548x500")
        Label(self.window, width=28, text="INVENTORY", font=times_new_roman, bg="pink").grid(row=0, columnspan=4)

        self.column_name1 = Entry(self.window, width=4, bg="light grey")
        self.column_name1.insert(0, "ID")
        self.column_name1.grid(row=1, column=0)

        self.column_name3 = Entry(self.window, width=9, bg="light grey")
        self.column_name3.insert(0, "STOCK")
        self.column_name3.grid(row=1, column=1)

        self.column_name2 = Entry(self.window, width=50, bg="light grey")
        self.column_name2.insert(0, "PRODUCT NAME")
        self.column_name2.grid(row=1, column=2)

        self.column_name3 = Entry(self.window, width=9, bg="light grey")
        self.column_name3.insert(0, "PRICE")
        self.column_name3.grid(row=1, column=3)

        plus = 100

        self.prev_list = Button(self.window, text="< Prev", font=Sans_button, command=lambda: self.display('prev'))
        self.next_list = Button(self.window, text="Next >", font=Sans_button, command=lambda: self.display('next'))
        self.prev_list.place(x=84, y=250 +plus)
        self.next_list.place(x=134, y=250 +plus)

        self.search_box = Entry(self.window, width=20)
        self.search_box.place(x=219, y=250 +plus)

        self.search_button = Button(self.window, text="SEARCH", font=Sans_button, width=7,
                                    command=lambda: self.display('search'))
        self.search_button.place(x=419, y=250 +plus)

        self.clear_button = Button(self.window, text="CLEAR", font=Sans_button, width=7,
                                   command=lambda: self.display('clear'))
        self.clear_button.place(x=419, y=280 +plus)

        self.display()

        self.window.resizable(False, False)
        self.window.configure(bg="pink")
        self.window.mainloop()
```

For def display function:

The following conditions handle which command is passed as an argument to the display function. If the command is 'next' it means that the user pressed the next button and means that the next_ function needs to be called to increase the data range to be displayed. If the command is 'prev', the prev_ functions needs to be called to decrease the data range to be displayed if and only if the minimum range is greater than. If the command is 'search' it means that the user pressed the search button and means that the search function needs to be called to filter the data based on the given filter and to set the bool search to true. If the command is 'clear' it means that the user pressed the clear button and means that the search filter needs to be cleared and bool search needs to be set to false.

```
    # display function is used to display the data from the database to the application
    def display(self, command=''):
        get_data(table_name_='inventory')

        # the following conditions handles which command is passed as an argument to the display function
        # if the command is 'next' it means that the user pressed the next button and means that the next_ function needs to be called to increase the data
        # if the command is 'prev' it means that the user pressed the prev button and means that the prev_ functions needs to be called to decrease the dat
        # if the command is 'search' it means that the user pressed the search button and means that the search function needs to be called to filter the d
        # if the command is 'clear' it means that the user pressed the clear button and means that the search filter needs to be cleared and bool search ne
        if command == 'next':
            # the following conditions is responsible for checking if the size of the data extracted from the database is greater than or equal to the curr
            # if the size of data extracted is valid the next_ function is called
            # if the size of search_data extracted is valid the next_ function is called
            if not gv.search and len(gv.data) - 1 >= gv.maximum:
                next_()
            if gv.search and len(gv.search_data) - 1 >= gv.maximum:
                next_()
        elif command == 'prev' and gv.minimum > 0:
            prev_()
        elif command == 'search':
            reset()
            # the following condition is responsible for checking if the user inputted some filter after clicking the search button
            # if the user inputted a non empty filter then the following codes will be executed
            if self.search_box.get() != '':
                search(self.search_box.get(), 'inventory')
                gv.search = True
        elif command == 'clear':
            self.search_box.delete(0, 'end')
            gv.search = False

        # if there is no search filter, all data from gv.data is appended to display data
        # if search is true, data appended will be dependent on search filter
        # if range of appended data is less than the maximum, other rows will be kept empty
        display_data = []
        if len(gv.data) != 0:
            if gv.search is True:
                if len(gv.search_data) != 0:
                    for i in range(gv.minimum, gv.maximum):
                        display_data.append(gv.search_data[i])
                        if i == len(gv.search_data) - 1:
                            for j in range(gv.maximum - len(display_data)):
                                display_data.append(('', '', '', ''))
                            break
            else:
                for i in range(gv.minimum, gv.maximum):
                    display_data.append(gv.data[i])
                    if i == len(gv.data) - 1:
                        for j in range(gv.maximum - len(display_data)):
                            display_data.append(('', '', '', ''))
                        break

        # displays the data from display_data list
        for i in range(2, 12):
            color = "pink" if i % 2 == 0 else "light grey"

            (product_id, stock, product_name, price) = display_data[i - 2] if len(display_data) != 0 else ('', '', '', '')
```

```
        column_1 = Entry(self.window, width=4, bg=color)
        column_1.grid(row=i, column=0)
        column_1.insert(0, product_id)

        stock_color = "#E92A5C" if str(stock).find('-') != -1 else color

        column_2 = Entry(self.window, width=9, bg=stock_color)
        column_2.grid(row=i, column=1)
        column_2.insert(0, stock)

        column_3 = Entry(self.window, width=50, bg=color)
        column_3.grid(row=i, column=2)
        column_3.insert(0, product_name)

        column_4 = Entry(self.window, width=9, bg=color)
        column_4.grid(row=i, column=3)
        column_4.insert(0, price)
```

b. Install DB Browser for SQLite
   - Create New Database entitled “CROCHET INVENTORY”
   - Create 2 tables
     a) Product

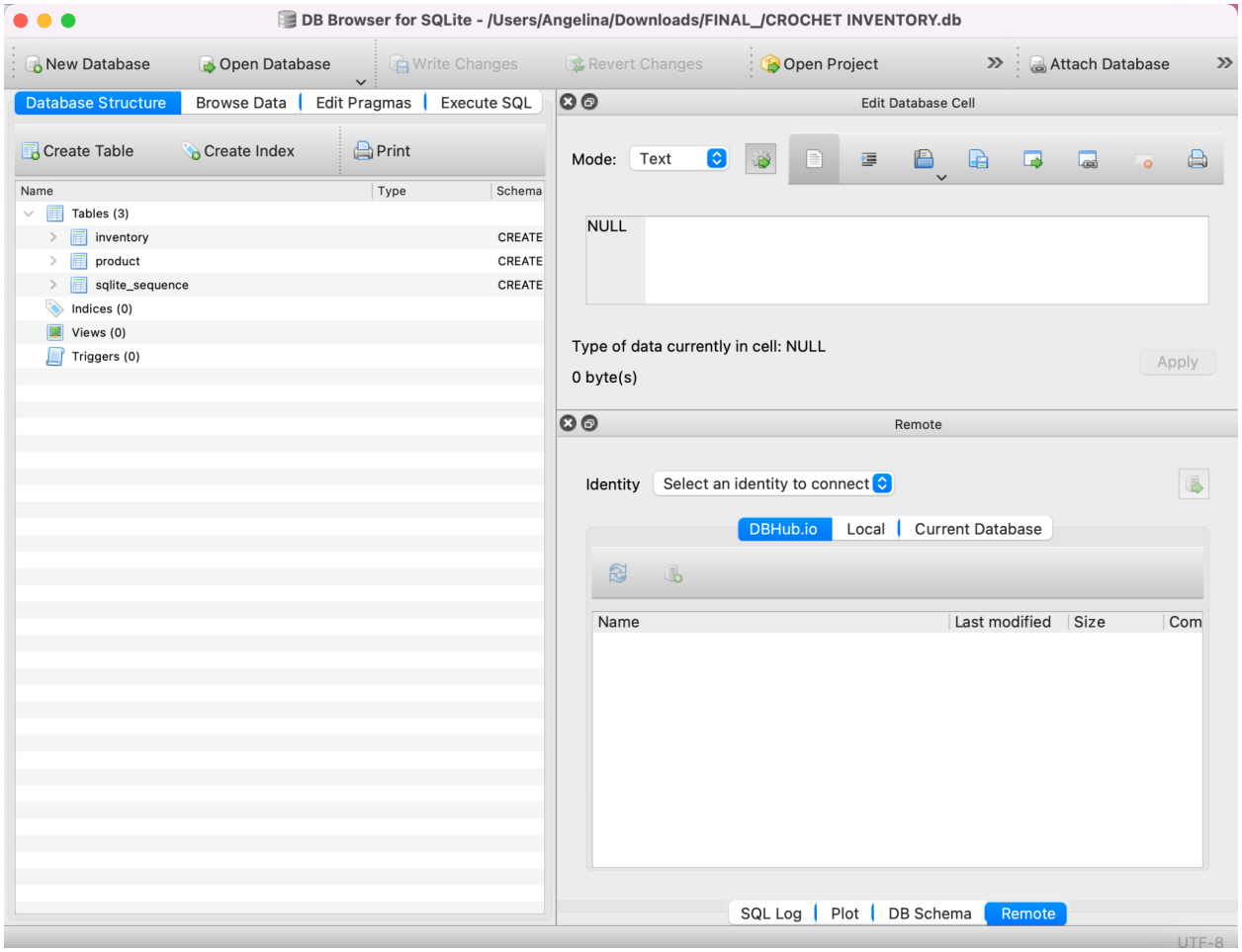
```
CREATE TABLE "product" (
        "id"    INTEGER NOT NULL,
        "product_name"      varchar(50) NOT NULL UNIQUE,
        "price"         INTEGER NOT NULL,
        PRIMARY KEY("id" AUTOINCREMENT)
)
```

     b) Inventory

```
CREATE TABLE "inventory" (
        "id"    INTEGER NOT NULL,
```

```
"stock"          INTEGER NOT NULL,
"product_name"   varchar(50) NOT NULL UNIQUE,
"price"          INTEGER NOT NULL,
PRIMARY KEY("id" AUTOINCREMENT)
)
```

This database will store data imported in the inventory system, as well as store the tables created/defined in database.py. This should be imported in the Pycharm directory where all the files listed above are stored. Sqlite_sequence automatically comes with it. When a csv file is run in the inventory system, there are queries in the py file that will automatically create tables for purchase and sales in this same data base.



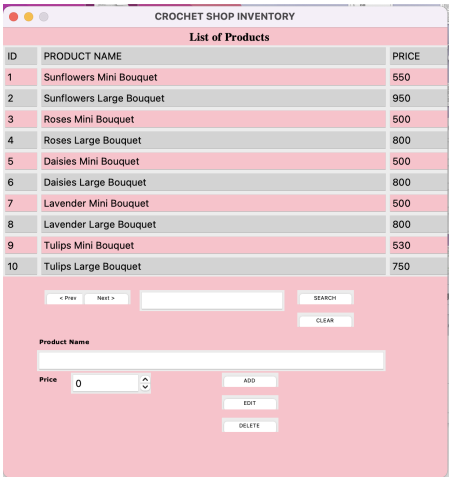
Output:

*****Inputs are case sensitive. Make sure there are no extra spaces in inputs as well.

1. Run main.py

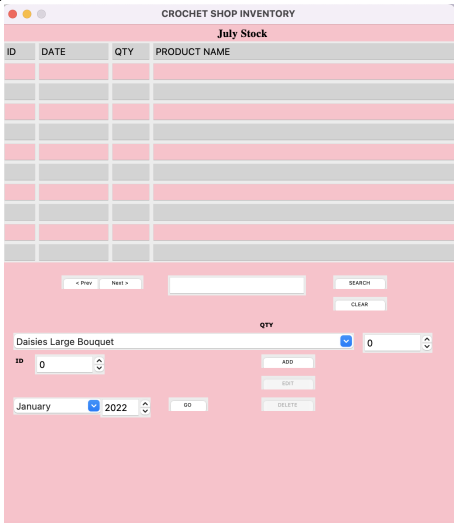
2. List of Products

   All products are displayed here along with price. In the first textbox, one can input “Sunflowers” and “Sunflowers Mini Bouquet” and “Sunflowers Large Bouquet” will appear. Since only 10 rows are shown, one can use the Prev and Next Buttons to see if there are next rows/pages in the table. In the second textbox, one can add a new product and place a price on it as well. If a mistake was made, the edit and delete button can be used.



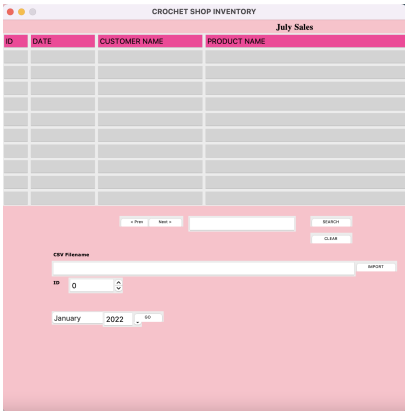
3. Add Stock

   This window is where stock is added. Edit and Delete buttons are disabled as stock can only be added. **Subtraction of stock is dependent on csv file. One can also view stock list on previous months and years. When viewing this, the current month will be automatically displayed as the header as one cannot add stock in previous or future months.



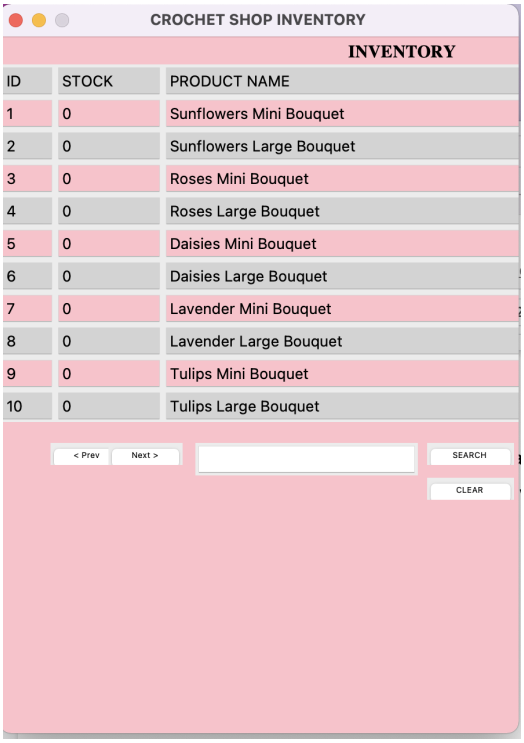
4. Sales

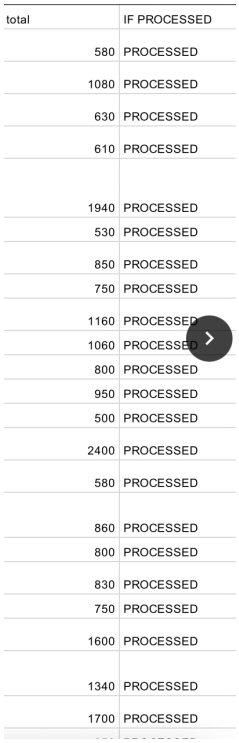
   This is where CSV files are imported. CSV file should be imported in the same directory as the py files and db file of PyCharm. The file name should be inputted exactly as it is in the textbox. Once import button is clicked, all data from the csv file will appear onto the table. Stock Quantity/Inventory will automatically be updated. **However, if the Inventory/Stock window is open when csv file is loaded in this window, updates will not reflect. It is better to keep open one window at a time. Sales in previous months ca also be viewed.



5. View Inventory

   This window is just a display of the inventory list. You can filter out the stock of product/s you want by inputting it in the search bar.





total	IF PROCESSED
580	PROCESSED
1080	PROCESSED
630	PROCESSED
610	PROCESSED
1940	PROCESSED
530	PROCESSED
850	PROCESSED
750	PROCESSED
1160	PROCESSED
1060	PROCESSED
800	PROCESSED
950	PROCESSED
500	PROCESSED
2400	PROCESSED
580	PROCESSED
860	PROCESSED
800	PROCESSED
830	PROCESSED
750	PROCESSED
1600	PROCESSED
1340	PROCESSED
1700	PROCESSED

6. New CSV Output

   Once the file has been read, a new updated CSV file will be put into the same directory. The only difference is that the IF PROCESSED columns now say processed. (If the new CSV file was under July and user would like to add more data for July, this new CSV should be used to add the new data of orders to AVOID PROCESSING DUPLICATES but STILL SEE/VIEW PREVIOUS DATA OF JULY)

Output: One application with several features. Additionally, one CSV file that prevents duplicate processing of orders.(This CSV file simply encodes “Processed” in the IF PROCESSED column of the CSV file so that when new data is added into that file, the system will read the data but not “process” or “subtract” it from inventory. In simpler terms, this is to avoid processing the same data a second time or moreif it has already went through the inventory system.)