Purpose:
The goal of MultiItemML.py was to take files created in the data generator that were separated into item specific csv files and perform the machine learning process determined as being a good balance of efficiency and execution time. The process moves between DB and ML working directories to load the csv files and perform the machine learning respectively. The csv data is indexed by the date of order, sin and cos function representing yearly item level fluctuations are created by mapping an arbitrary "Seconds" value to each field, performing two formulas that produce the sin and cos values as outputs, and adding those outputs as additional columns to the DataFrame object. The data then goes through a function to covert from a DataFrame object to a numpy array. Next, the data is separated into training, validation, and testing data and all that data is preprocessed. A gated recurrent unit model and a model checkpoint are created then the model is compiled and fitted with the training, validation, and test data. After the lengthy execution time of the fitting process, a plot of the predicted values being compared to the reserved actual values of the test data is saved in the ML directory. The SingleItemML.py is one iteration for one item of the same process so that we could perform a less time intensive version for testing purposes.

Libraries used:

```
import tensorflow as tf
from tensorflow import keras
import os
import csv
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.metrics import RootMeanSquaredError
from tensorflow.keras.optimizers import Adam
```

```
from sklearn.metrics import mean_squared_error as mse
import time
```

- Tensorflow – a machine learning and artificial intelligence library
- Keras - an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.
- OS – allows for the use of operating system dependent functionality.
- CSV – library for csv file manipulation.
- PANDAS – a data analysis tool. Used for handling DataFrame objects in this program.
- NUMPY - adds support for multi-dimensional arrays and high-level mathematical functions.
- MatPlotLib – a python library for creating mathematical plots and other graphing tools.

- Seeborn - Seaborn is a Python data visualization library based on matplotlib. Used previously during the model testing phase of the project.
- Keras sublibraries models, layers, callbacks, losses, metrics optimizers – necessary sublibraries for the modeling process. Sequential is the model type, layers are the basic building blocks of neural networks, ModelCheckpoint is the process that stores the previous best case so the model can improve upon itself during future iterations, MeanSquaredError and RootMeanSquaredError are metrics being used to compare model iterations, and Adam is the learning optimizer used for the model.
- MSE – is the mean squared error metric but used as a return value of the plot predictions function.
- Time – A system time library used to track program execution time.

Functions:

```
# Function to plot predictions vs actuals for a model
def plot_predictions1(model, X, y, cp, start=0, end=100):
    predictions = model.predict(X).flatten()
    df = pd.DataFrame(data={'Predictions':predictions, 'Actuals':y})
    plt.plot(df['Predictions'][start:end])
    plt.plot(df['Actuals'][start:end])
    #plt.ylabel("Quantity")
    #plt.xlabel("Index of Reserved Test Data")
    plt.legend(['Predictions', 'Actuals'])
    #plt.savefig("figure.jpg")
    #plt.title("Results: " + str(csv_filename[i]))
    ax = df.plot.line()
    #ax.text(50,45, "RMSE: " + str(cp.best), ha='center')
    ax.set_title("Results: " + str(csv_filename[i]) + "\nRMSE: " + str(cp.best))
    ax.set_ylabel("Quantity")
    ax.set_xlabel("Index of Reserved Test Data")
    ax.figure.savefig(str(csv_filename[i]) + '.pdf')
    return df, mse(y, predictions)
```

Plot predictions - takes the model object, X test values, y test values, and the model checkpoint object as inputs. The output of the function is a pdf file saved to the ML directory that contains a graph with the predicted quantities of an item (Predictions) compared against the reserved test values (Actuals). The commented-out fields were previously used in a Google Colab version of the program that displayed the plot in the output of the cell but saving to a separate file required a different approach. The plot is labels with a title that contains the csv filename and the root mean squared error metric. The y-axis is labeled ‘Quantity’ and the x-axis is labeled ‘Index of Reserved Test Data’.

```
def df_to_X_y2(df, window_size=7):
    df_as_np = df.to_numpy()
    X = []
    y = []
    for i in range(len(df_as_np)-window_size):
        row = [r for r in df_as_np[i:i+window_size]]
        X.append(row)
        label = df_as_np[i+window_size][0]
        y.append(label)
    return np.array(X), np.array(y)
```

df_to_X_y2 - coverts a DataFrame object into a numpy array so the model can take the array as inputs. Window size is set to 7, meaning that 7 previous input values will be used to predict an output.

```
def preprocess(X):
    X[:, :, 0] = (X[:, :, 0] - qnt_training_mean) / qnt_training_std
    return X
```

preprocess- handles some minor preprocessing of the input numpy array so that the quantity, sin, and cos inputs don't affect the individual predicted outputs. The only concern of the program is the quantity output but all three outputs as they are needed for further predictions based on those outputs.

Machine Learning loop:

```
start = time.time()
# Loading csv filenames into an array
ml_dir = os.getcwd()
os.chdir('..\\..\\documentation\\DB\\individual_csv')
db_dir = os.getcwd()
csv_filename = sorted(str(item) for item in os.listdir('.'))

for i in range(len(csv_filename)):
    os.chdir(db_dir)
    csv_path = (str(csv_filename[i]))
    df = pd.read_csv(csv_path)
    os.chdir(ml_dir)

    # Index to datetime
    df.index = pd.to_datetime(df['Day'])

    qnt = df['Quantity']
    qnt.plot()
```

Before the start of the loop the location of the ML directory and DB directory containing the individual item csv files are saved so that we can easily go back and forth between directories. The filenames of the csv files are then stored in an array. Using the length of the filename array, the program runs the entire machine learning process on each individual csv file. In the loop, the working directory is changed to the DB directory to load the csv contents into a pandas DataFrame object and the working directory is then changed to the ML directory for the rest of the loop. The DataFrame object is indexed by the date that the item was ordered and the Quantity field is placed in a new DataFrame object qnt.

```
# Mapping seconds (of arbitrary value) based on the timestamp
qnt_df = pd.DataFrame({'Quantity':qnt})
qnt_df['Seconds'] = qnt_df.index.map(pd.Timestamp.timestamp)
#qnt_df

day = 60*60*24
year = 365.2425*day

qnt_df['Year sin'] = np.sin(qnt_df['Seconds'] * (2 * np.pi / year))
qnt_df['Year cos'] = np.cos(qnt_df['Seconds'] * (2 * np.pi / year))
qnt_df.head()

qnt_df = qnt_df.drop('Seconds', axis=1)
qnt_df.head()
```

To create sin and cos functions based on the yearly trends of ordered quantity a separate DataFrame qnt_df was created and indexed by mapped ‘Seconds’ values. The value of the ‘Seconds’ is arbitrary and doesn’t actually represent any real world value—it is just used to facilitate the creation of the sin and cos functions. A day variable is created based on the seconds, minutes, and hours within a day and a year variable is created based on the average number of days in a year and the day variable. The sin and cos function outputs are then added to the DataFrame. They will be used as additional inputs in the model to help more accurately predict yearly trends.

```
X2, y2 = df_to_X_y2(qnt_df)
X2.shape, y2.shape

# Added size of the train/val/test range to generalize based on number of items
q_80 = int(len(qnt_df)*.80)
q_90 = int(len(qnt_df)*.90)

X2_train, y2_train = X2[:q_80], y2[:q_80]
X2_val, y2_val = X2[q_80:q_90], y2[q_80:q_90]
X2_test, y2_test = X2[q_90:], y2[q_90:]
X2_train.shape, y2_train.shape, X2_val.shape, y2_val.shape, X2_test.shape, y2_test.shape

qnt_training_mean = np.mean(X2_train[:, :, 0])
qnt_training_std = np.std(X2_train[:, :, 0])
```

Using the df_to_X_y2 function, the qnt_df DataFrame object is converted to X2 and y2 numpy arrays. The numpy arrays are then separated into training, validation, and testing numpy arrays. The qnt_training_mean and qnt_training_std are created to be used in the preprocessing function.

```
preprocess(X2_train)
preprocess(X2_val)
preprocess(X2_test)

model3 = Sequential()
model3.add(InputLayer((7, 3)))
# Return sequences return the hidden state output for each input time step so
# the model is effectively passing through 2 GRU instances
model3.add(GRU(32,return_sequences=True))
model3.add(GRU(64))
model3.add(Dense(128, 'relu'))
model3.add(Dense(64, 'relu'))
model3.add(Dense(32, 'relu'))
model3.add(Dense(1, 'linear'))

model3.summary()

cp3 = ModelCheckpoint('model_' + csv_filename[i] + '/', monitor='val_root_mean_squared_error', save_best_only=True)
model3.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=0.0001), metrics=[RootMeanSquaredError()])

model3.fit(X2_train, y2_train, validation_data=(X2_val, y2_val), epochs=1000, callbacks=[cp3])
```

The separated training, validation, and test data gets preprocessed using the function previously described. A sequential model is created, an input layer is created with a window size of 7 and 3 outputs, two gated recurrent unit (GRU) layers are used which signifies the type of recurrent neural network that is being used, and multiple dense layers are used to change the topology of the neural network (this is considered a hyperparameter that can be adjusted to improve/degrade performance of the model). A model checkpoint ‘cp3’ is created to store the best saved version of the model so that the model improves over iterations of training. The checkpoints are saved to new folders using the csv filename so that the program will continue to improve the training of the same models if the entire program is reran

from the beginning. The model is compiled using the MeanSquaredError as a loss metric, a learning rate of 0.0001 which dictates how fast the model is trained with a faster training usually resulting in a loss of accuracy, and RootMeanSquaredError as a metric measuring the quality of the model. The majority of the execution time of the program is in the model fitting process which trains the model based on reserved training data, validates the model using reserved validation data, and saves the best version of the model to cp3. The model is trained of 1000 'epochs' or iterations.

```
plot_predictions1(model3, X2_test, y2_test, cp3)
end = time.time()
print("The program has been running for :" + str(((end-start)/60)) + " minutes." )
```

The final portion of the program calls the plot_prediction function to save the plot of the item predictions versus actuals. A execution time for the program was added since the training process for all of the models that are created causes the program to run for several hours.