Libraries used in python-:

- ➢ **yfinance** is a Python library that **provides functionality for fetching financial data from Yahoo Finance**. It allows users to retrieve various financial data, including stock prices, historical market data, company information, and more, directly from Yahoo Finance's website.
- ➢ **pandas** is an open-source Python library **for data manipulation and analysis**. It provides easy-to-use data structures and data analysis tools for working with structured data, making it an essential tool in data science, data analysis, and machine learning.
- ➢ The **pmdarima** (Pyramid ARIMA) is a Python library that provides tools for time series analysis and forecasting. It's specifically **designed to simplify the process of working with ARIMA** (AutoRegressive Integrated Moving Average) models and other time series forecasting techniques.One of the primary features of pmdarima is its ability to automatically select the optimal ARIMA model parameters

The **auto_arima** function within the pmdarima library is a powerful tool for automatically selecting the optimal ARIMA (AutoRegressive Integrated Moving Average) model parameters for a given time series dataset.

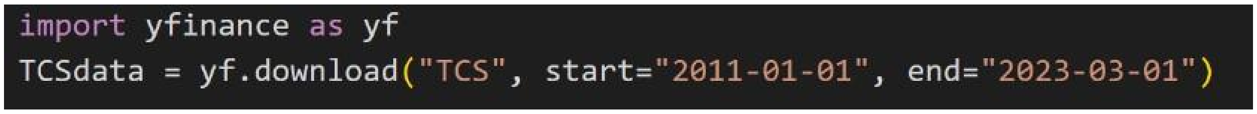
- ➢ **statsmodels.tsa** is a part of the StatsModels library,focuses on time series analysis and provides **tools for working with time series data**, including various models like ARIMA.
- ➢ **NumPy(Numerical Python)** is a fundamental and widely used Python library for numerical and mathematical operations. It **provides support for working with large, multi-dimensional arrays** and matrices, as well as a collection of mathematical functions to operate on these arrays.
- ➢ **matplotlib.pyplot** is a part of the widely used Python library Matplotlib and is **used for creating data visualizations, particularly for producing 2D plots and charts**. You can create a wide range of high-quality plots and graphs to convey your data effectively.
- ➢ The **sklearn.preprocessing** library is a part of the scikit-learn (Sklearn) machine learning library in Python. **It provides a wide range of tools and functions for data preprocessing and feature engineering**, which are essential steps in the machine learning pipeline. **MinMaxScaler** is a data preprocessing technique provided by the sklearn.preprocessing library for scaling and transforming numerical data. It is a popular method for feature scaling, specifically designed to rescale data to a specific range, typically between 0 and 1. It ensures that differently scaled data now have similar influence on machine learning algorithms that are sensitive to the scale of input features by rescaling them.
- ➢ **TensorFlow library's Keras API is a popular framework for building and training neural networks.**

1. **Sequential** is a fundamental Keras model for building neural networks. It allows you to create a model by stacking layers one after the other in a linear fashion.
2. **Dense** is a type of layer in Keras, also known as a fully connected or feedforward layer. It's used for defining the hidden layers of a neural network. Each neuron in a Dense layer is connected to every neuron in the previous and subsequent layers.
3. **EarlyStopping** is a callback in Keras used during the training of a neural network. It monitors a specified metric, such as validation loss, and stops training early if the monitored metric stops improving. This is helpful for preventing overfitting and optimizing training time.

➢ **The sklearn.metrics library in scikit-learn (Sklearn) provides a wide range of functions and classes for evaluating the performance of machine learning models.** Thesy help to measure how well a model's predictions align with the actual data and help assess various aspects of model performance, including classification, regression, clustering, and more.

Code sections and their explanations-:

1. We begin with downloading the historical data for the stock we considered which is ***"TCS.NS"*** here.



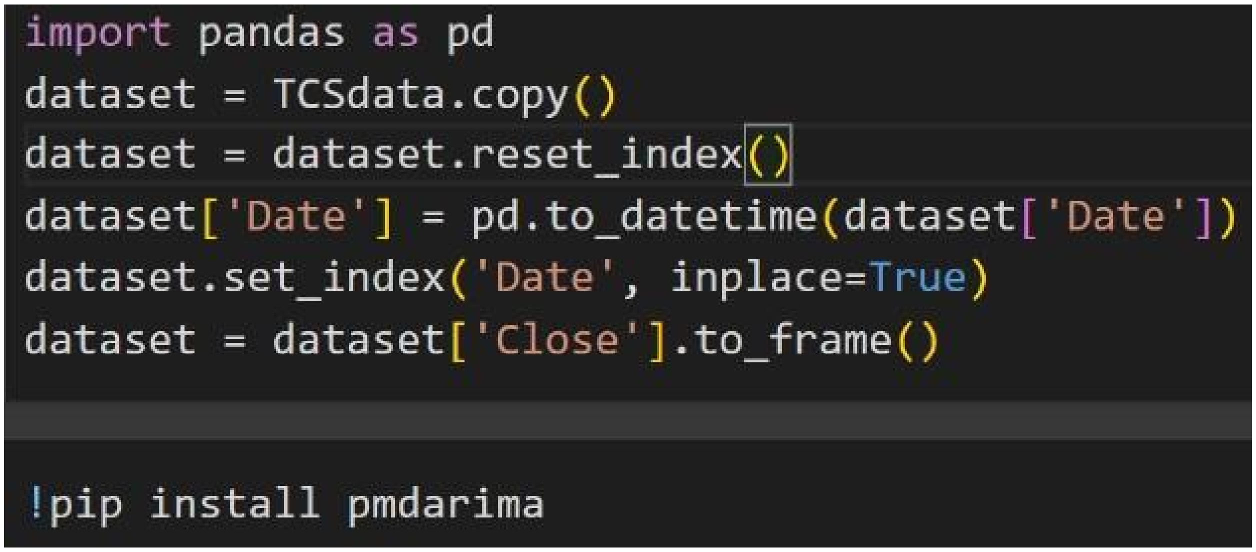
```
import yfinance as yf
TCSdata = yf.download("TCS", start="2011-01-01", end="2023-03-01")
```

2. Preprocessing of the data: This preprocessing is often done to prepare data for further analysis or modeling, especially in time series data analysis.

i. The code takes an existing DataFrame (gs), makes a copy of it (dataset), resets the index, converts the 'Date' column to datetime objects, sets the 'Date' column as the new index, and then extracts only the 'Close' column, creating a new DataFrame with just the 'Close' data.

ii. The **reset_index() function** moves the current index (usually the date in this case) back into a regular column, and a new default integer index is created.' Date' column is the new index for the DataFrame using the **set_index() function** .inplace=True enables us to change it in the DataFrame itself, rather than creating a new DataFrame.

iii. pd.to_datetime(dataset['Date']) converts the values in the 'Date' column of the DataFrame to datetime objects using the **pd.to_datetime() function**. This is done to enable date-based operations and filtering.



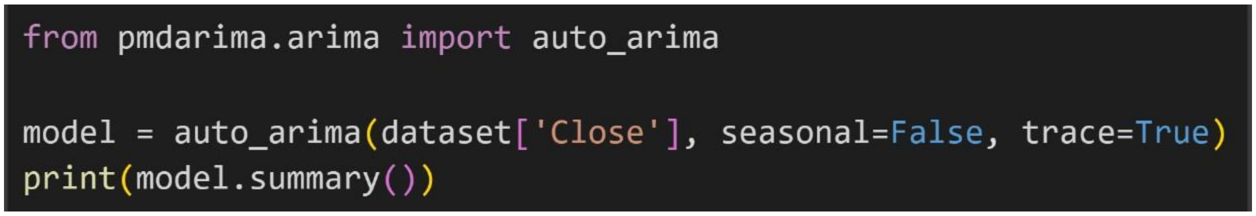
```
import pandas as pd
dataset = TCSdata.copy()
dataset = dataset.reset_index()
dataset['Date'] = pd.to_datetime(dataset['Date'])
dataset.set_index('Date', inplace=True)
dataset = dataset['Close'].to_frame()


!pip install pmdarima
```

__3.__ **These lines of code automatically select the optimal parameters for an ARIMA time series model**. Here, dataset['Close'] is the time series data for which you want to fit an ARIMA model.

__**The parameters:**__

i.seasonal=False indicates that you're not considering a seasonal component

ii.trace=True prints the detailed information about the ARIMA model selection process, including the candidate models considered and their respective evaluation metrics.



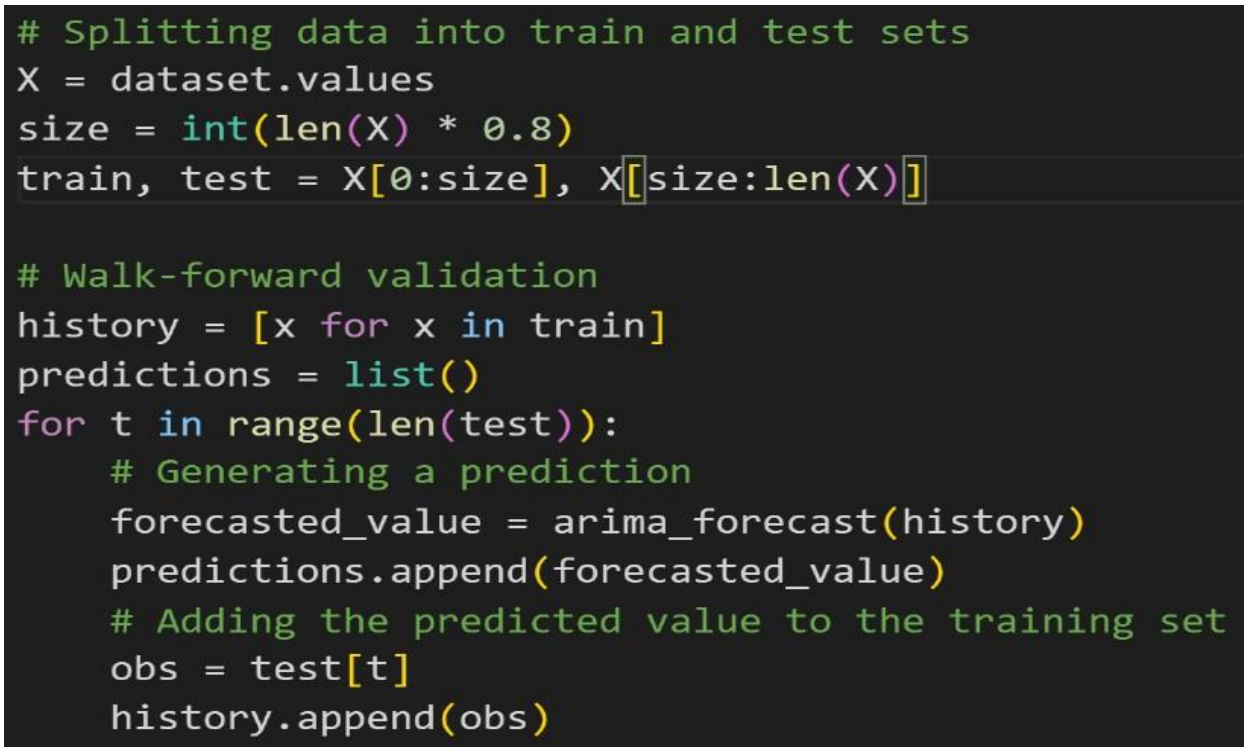
```
from pmdarima.arima import auto_arima

model = auto_arima(dataset['Close'], seasonal=False, trace=True)
print(model.summary())
```

After auto_arima completes its search for the best ARIMA model, it returns an ARIMA model object with the selected parameters. **The model.summary() method is used to print a summary of the selected ARIMA model, which includes information about the model's order (p, d, q), AIC (Akaike Information Criterion) value, BIC (Bayesian Information Criterion) value, and other model statistics.**

__4.__ These lines of code demonstrate how to perform time series forecasting using an ARIMA model in Python.

__**i.Defining the Function:**__ It takes a history of observed values as input and creates an ARIMA model with an order of (0,1,0), indicating a differencing order of 1 to make the time series stationary (if it's not already). Then the model is fitted to the history data. Next, A one-step-ahead forecast is made using the model_fit.forecast() method and the forecasted value is returned.

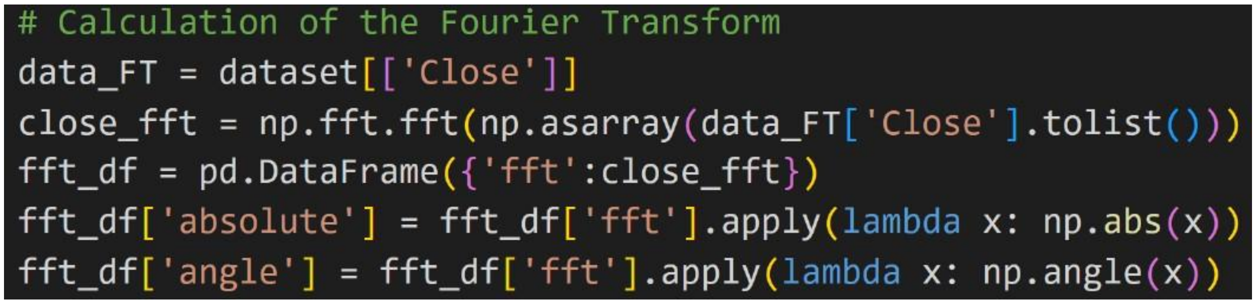
ii. Splitting data into train and test sets: X is the dataset to be split into training and test sets.size determines the split point data till the size point is used to train the set and the data post the size point is used to test the model.
iii. Walk-forward validation: The model is updated at each time step with the latest observed data and used to make forecasts. The performance of the model can be evaluated by comparing the predicted values (predictions) with the actual test data. History is initialized with the training data. **The code then iterates through the test set making one-step-ahead forecasts and appending the same to the predictions list. The observed value from the test set (obs) is added to the history so that the model can make the next forecast based on the updated history.**



```
# Splitting data into train and test sets
X = dataset.values
size = int(len(X) * 0.8)
train, test = X[0:size], X[size:len(X)]

# Walk-forward validation
history = [x for x in train]
predictions = list()
for t in range(len(test)):
    # Generating a prediction
    forecasted_value = arima_forecast(history)
    predictions.append(forecasted_value)
    # Adding the predicted value to the training set
    obs = test[t]
    history.append(obs)
```

5. A plot that visualizes the predictions made by the ARIMA model against the actual values from the test set is created and displayed. Here in,
i. dpi parameter sets the dots per inch for the plot, which affects its resolution.
ii.dataset.iloc[size:,:].index represents the date index for the test set.
6. Calculating and Plotting the Fourier Transform:
First, we assign the data of 'Close' column to a new DataFrame.Then we use the **np.fft.fft(a NumPy function) for performing the Fast Fourier Transform (FFT).**In this:
i.data_FT['Close'].tolist() converts the 'Close' column into a list of values.
ii.np.asarray() converts the list into a NumPy array.
iii. The np.fft.fft function is then applied to the array, which calculates the FFT of the 'Close' price series and stores it in the *'close_fft'* array.

Next, A new data frame named *'fft_df'* is created to store the FFT results, and the 'fft' column in *'fft_df'* is populated with the *'close_fft'* array, which contains the FFT values.
In the next line of code, A new column 'absolute' is added to the *'fft_df'* DataFrame, and a lambda function that calculates the **absolute value (np.abs)** of each FFT component and stores it in the 'absolute' column is applied to each element in the 'fft' column using the **.apply() function**. Similarly, a lambda function that calculates the phase angle (np.angle) of each FFT component is applied to the column and a new column 'angle' is created to store the values.



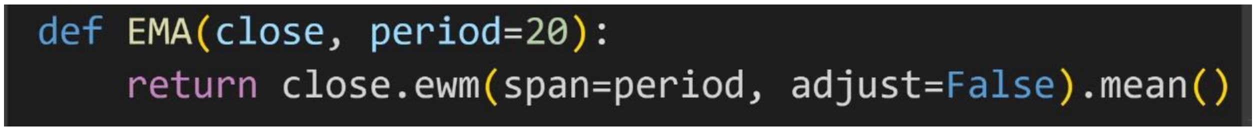
```
# Calculation of the Fourier Transform
data_FT = dataset[['Close']]
close_fft = np.fft.fft(np.asarray(data_FT['Close'].tolist()))
fft_df = pd.DataFrame({'fft':close_fft})
fft_df['absolute'] = fft_df['fft'].apply(lambda x: np.abs(x))
fft_df['angle'] = fft_df['fft'].apply(lambda x: np.angle(x))
```

Our *'fft_df'* DataFrame finally contains the FFT values, their absolute values, and their phase angles. These components can be useful for analyzing the frequency domain characteristics of the original time series data, which can provide insights into periodic patterns or dominant frequencies in the stock price data.

<u>7. Plot to visualize the stock prices and their Fourier transforms-:</u>

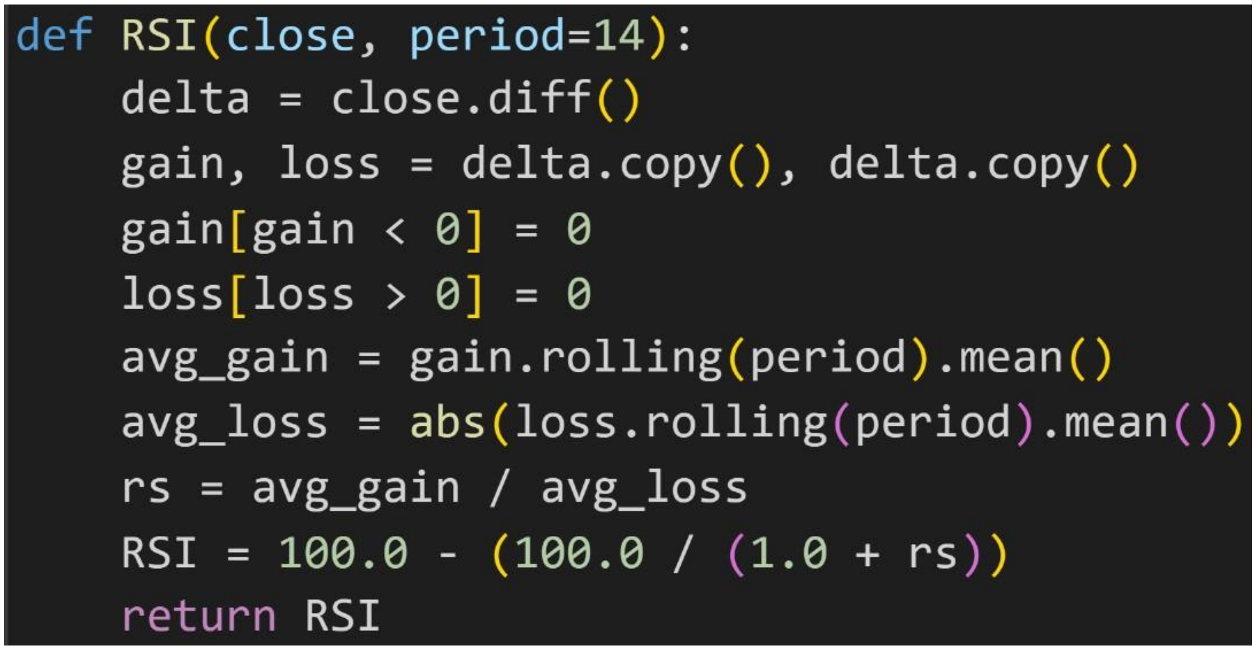
i. The original stock prices are plotted first ***(np.asarray(data_FT['Close'].tolist()))*** with the label 'Real.'
ii. Then, a loop is used to iterate through different numbers of components (3, 6, and 9) for the Fourier transform. For each iteration, a copy of the original ***FFT (fft_list_m10)*** is modified by zeroing out some of its components to retain only a specified number of frequency components.
iii. Then the ***inverse FFT (np.fft.ifft)*** is applied to the modified FFT to obtain the reconstructed signal with the selected number of frequency components.
iv. Each reconstructed signal is plotted with a label indicating the number of components used in the Fourier transform.
The plot shows the original stock prices and how they change when different numbers of frequency components are considered in the Fourier transform. This helps us visualize how the signal can be represented with different levels of detail in the frequency domain.
<u>8.</u> We use several technical indicators to enhance our predictions and help in investment decisions. This piece of code defines the technical indicators we opt for later use.

<u>i. EMA-:</u>The EMA function calculates the Exponential Moving Average of a given close price series. The period parameter specifies the number of periods to be considered for the EMA (default is 20). The ewm method with span set to the period calculates the EMA.



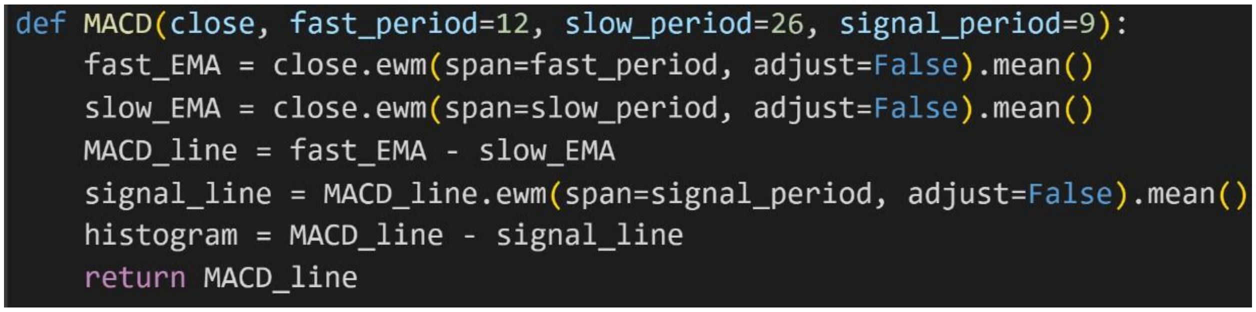
```python
def EMA(close, period=20):
    return close.ewm(span=period, adjust=False).mean()
```

ii. RSI-:The RSI function calculates the Relative Strength Index of a given close price series. The period parameter once again specifies the number of periods for which the RSI is calculated (default is 14 periods). **The function calculates the changes in closing prices, the average gain, and the average loss over the specified period to finally compute the RSI.**



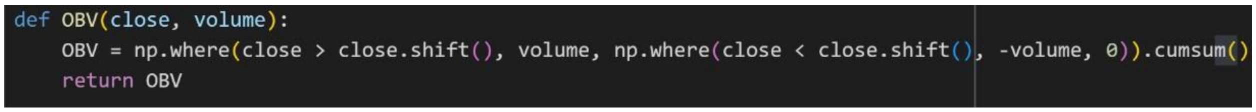
```python
def RSI(close, period=14):
    delta = close.diff()
    gain, loss = delta.copy(), delta.copy()
    gain[gain < 0] = 0
    loss[loss > 0] = 0
    avg_gain = gain.rolling(period).mean()
    avg_loss = abs(loss.rolling(period).mean())
    rs = avg_gain / avg_loss
    RSI = 100.0 - (100.0 / (1.0 + rs))
    return RSI
```

iii. MACD-:The MACD function calculates the Moving Average Convergence Divergence of a given close price series. The fast_period and slow_period parameters specify the periods for the fast and slow EMAs (default values are 12 and 26, respectively). The parameter signal_period specifies the period for the signal line (default is 9). **The function calculates the MACD line, the signal line, and the histogram based on these parameters.**



```python
def MACD(close, fast_period=12, slow_period=26, signal_period=9):
    fast_EMA = close.ewm(span=fast_period, adjust=False).mean()
    slow_EMA = close.ewm(span=slow_period, adjust=False).mean()
    MACD_line = fast_EMA - slow_EMA
    signal_line = MACD_line.ewm(span=signal_period, adjust=False).mean()
    histogram = MACD_line - signal_line
    return MACD_line
```

iv. OBV-:The obv function calculates the On-Balance Volume of a given close price series and corresponding volume series. It uses numpy where statement to determine whether the current closing price is higher, lower, or equal to the previous closing price. It then accumulates the volume changes based on these conditions.



```python
def OBV(close, volume):
    OBV = np.where(close > close.shift(), volume, np.where(close < close.shift(), -volume, 0)).cumsum()
    return OBV
```

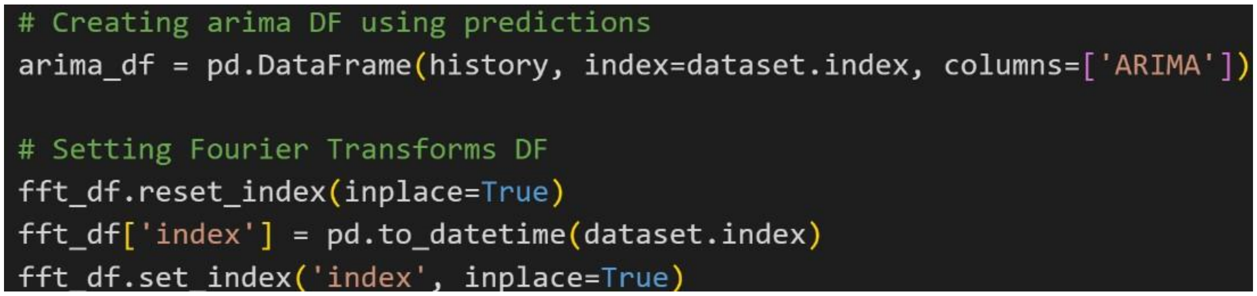
9. Applying the technical indicators to the dataset:

We add the 20-period, 50-period, and 100-period EMAs, RSI, MACD and OBV of the closing prices from the TCSdata DataFrame to the dataset DataFrame.

10.

i. First, we create a DataFrame named *arima_df* containing the ARIMA model predictions stored in the history list. The index is set to match the index of the dataset DataFrame, and the column is labeled as 'ARIMA.'

ii. First we reset the index of the *fft_df* DataFrame, convert the 'index' column to datetime, and then set the index back to the datetime index.



```python
# Creating arima DF using predictions
arima_df = pd.DataFrame(history, index=dataset.index, columns=['ARIMA'])

# Setting Fourier Transforms DF
fft_df.reset_index(inplace=True)
fft_df['index'] = pd.to_datetime(dataset.index)
fft_df.set_index('index', inplace=True)
```

iii. Next we create 2 new DataFrames, *fft_df_real* and *fft_df_imag* to separate the real and imaginary parts of the Fourier transforms and set their respective indices.

iv. A DataFrame named *"technical_indicators_df"* is created by selecting specific columns from the dataset DataFrame. Namely, EMA_20, EMA_50, EMA_100, RSI, MACD, OBV, and the 'Close' price.

v. The **pd.concat function** is used to concatenate (merge) various DataFrames along the columns (axis=1). This function creates a single DataFrame *"merged_df"* containing ARIMA predictions, real and imaginary parts of the Fourier transforms, and the selected technical indicators.

vi. Any rows containing missing data (NaN) are dropped using **dropna()** to ensure a clean dataset.

This *merged_df* DataFrame combines the ARIMA predictions, Fourier transforms, and the selected technical indicators into a unified dataset, which can be used for further analysis, modeling, or visualization of the financial time series data.

11.Training and testing our mode:
In this part of code we divide the merged_df DataFrame into training and testing subsets
→ The first line of code calculates the size of the training set as 80% of the total length of the *merged_df* DataFrame.
→train_df and test_df are created by using DataFrame slicing. *train_df* contains the first 80% of the rows in merged_df, and test_df contains the remaining 20%.

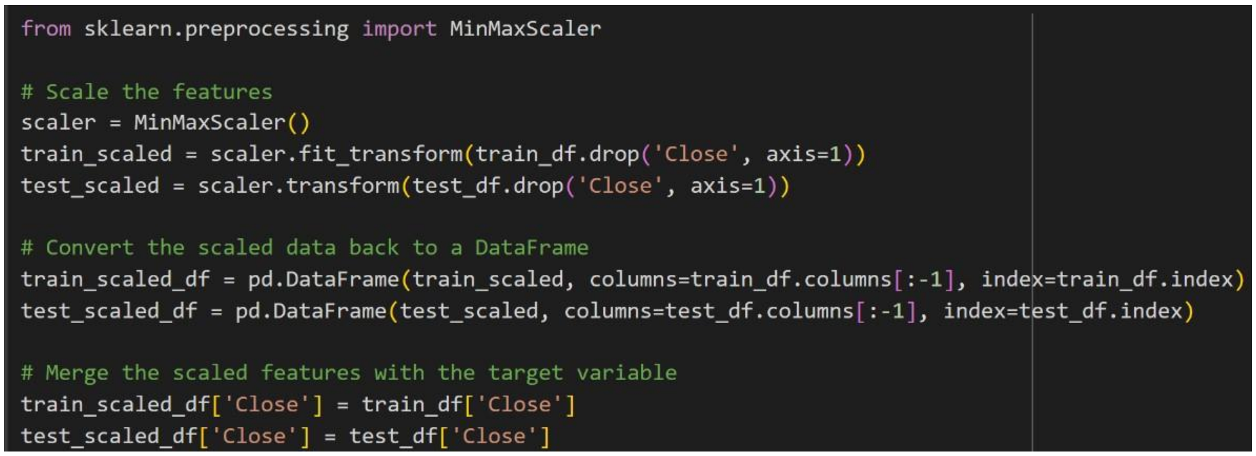
12.Data processing:

- We create a MinMaxScaler object.
- train_scaled is obtained by fitting the scaler to the training data and transforming it. This scales the features in the training set.
- test_scaled is obtained by using the same scaler to transform the testing data

→DataFrames *train_scaled_df* and *test_scaled_df* are created to hold the scaled data, with the same columns and indices as the original DataFrames.The 'Close' column from train_df and test_df is added to the scaled DataFrames. This combines the scaled features with the target variable.

- X_train contains the features from the training set (all columns except for the 'Close' column) and X_test contains the features from the testing set.
- y_train contains the target variable from the training set (the 'Close' column) and the y_test contains the target variable from the testing set.

→These steps prepare the data with the features scaled to a consistent range (0 to 1) and separated into input features (X) and target labels (y) for both the training and testing sets.



```
from sklearn.preprocessing import MinMaxScaler

# Scale the features
scaler = MinMaxScaler()
train_scaled = scaler.fit_transform(train_df.drop('Close', axis=1))
test_scaled = scaler.transform(test_df.drop('Close', axis=1))

# Convert the scaled data back to a DataFrame
train_scaled_df = pd.DataFrame(train_scaled, columns=train_df.columns[:-1], index=train_df.index)
test_scaled_df = pd.DataFrame(test_scaled, columns=test_df.columns[:-1], index=test_df.index)

# Merge the scaled features with the target variable
train_scaled_df['Close'] = train_df['Close']
test_scaled_df['Close'] = test_df['Close']
```

13. Creating and training a neural network regression model using TensorFlow/Keras:
i.We begin with defining neural networks:
A sequential model is created, representing a feedforward neural network.

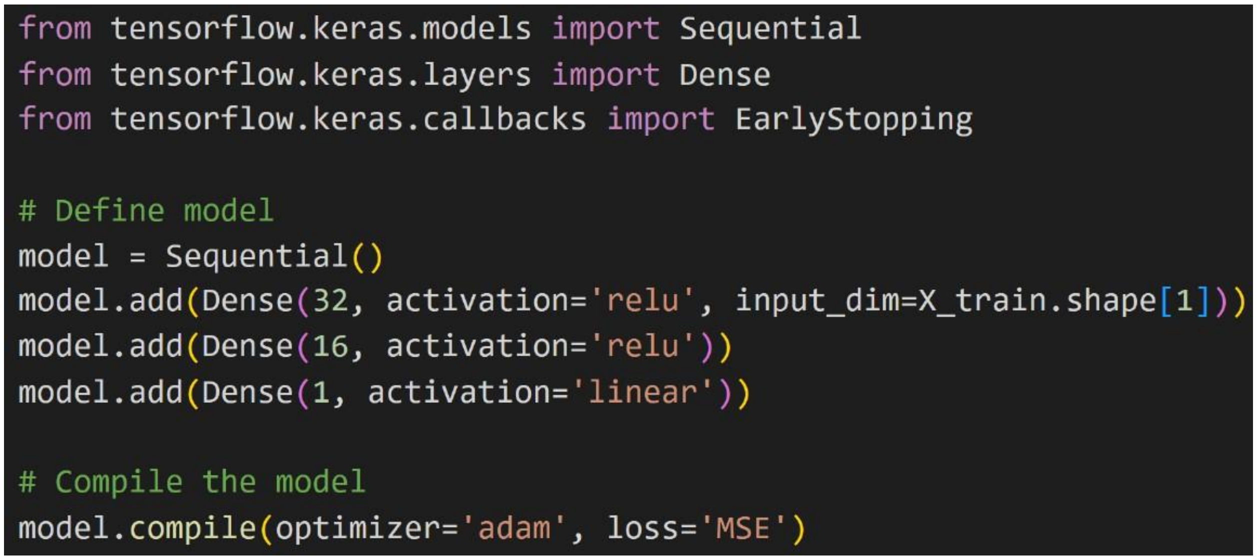
The 1st layer has 32 units with a **ReLU (Rectified Linear Unit) activation function** and input dimension defined based on the number of features in the training data and the 2nd layer has 16 units with ReLU activation. The final output layer has 1 unit with a linear activation function for regression tasks.
ii. The model is compiled with the Adam optimizer and the MSE loss function.
iii. An early stopping callback is defined to monitor the **validation loss (val_loss)**. If the validation loss doesn't improve for the patience number of epochs (20 in this case), training will stop early.

- ***verbose=1*** indicates that messages will be printed when early stopping is triggered.
- ***mode='min'*** implies that the goal is to minimize the validation loss.

iv. The model is trained using the training data (X_train and y_train) for up to 1000 epochs, with a batch size of 32.

- The validation_data parameter is provided to monitor the model's performance on the testing data (X_test and y_test).
- The callbacks parameter includes the early stopping callback defined earlier.
- ***shuffle=False*** indicates that the data won't be shuffled during training, which is generally useful when dealing with time series data.

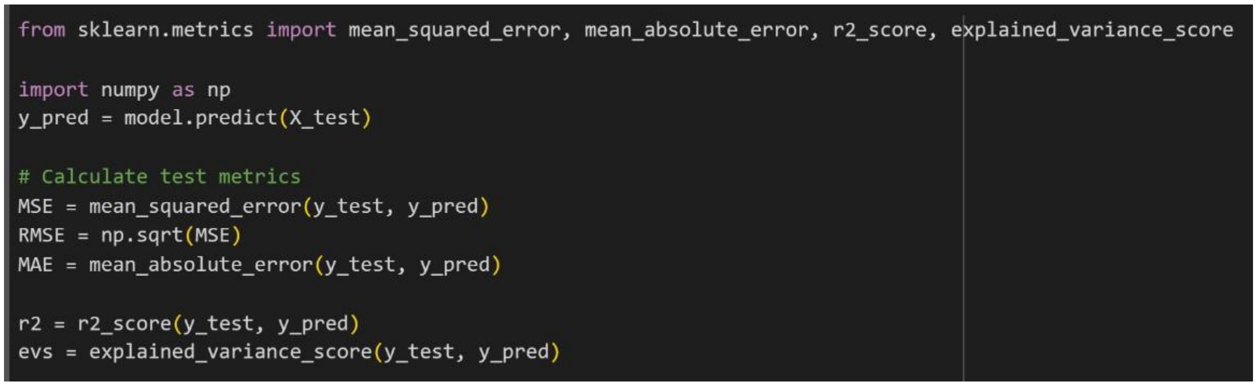
v. The training history is stored in the history variable, which can be used to analyze the model's performance.



```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping

# Define model
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=X_train.shape[1]))
model.add(Dense(16, activation='relu'))
model.add(Dense(1, activation='linear'))

# Compile the model
model.compile(optimizer='adam', loss='MSE')
```

<u>14.</u> Assessing the performance of the neural network regression model by calculating various regression evaluation metrics.Namely, MSE, RMSE, MAE, R2, EVS, MAPE and MPE.
i.MSE (Mean Squared Error) and RMSE (Root Mean Squared Error): It measures the average squared difference between the actual values (y_test) and the predicted values

(y_pred). It quantifies how well the model's predictions align with the actual data.RMSE is the square root of the MSE and provides a more interpretable metric in the same units as the target variable.
ii. MAE (Mean Absolute Error): It calculates the average absolute difference between the actual values and the predicted values. It's less sensitive to outliers compared to MSE.
Iii. R2 (R-squared score) (or) The Coefficient of Determination: It measures the proportion of the variance in the dependent variable that is predictable from the independent variables. It ranges from 0 (poor fit) to 1 (perfect fit).
iv. EVS (Explained Variance Score): It quantifies the proportion of the variance in the dependent variable that is explained by the model. It ranges from 0 to 1 (perfect fit).
v. MPE (Mean Percentage Error) and MAPE (Mean Absolute Percentage Error): Both of them measure the average % difference between the actual values and the predicted values. While MPE can be positive or negative, depending on the direction of the errors. MAPE is absolute and provides insight into the scale of prediction errors.



```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score, explained_variance_score

import numpy as np
y_pred = model.predict(X_test)

# Calculate test metrics
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)
MAE = mean_absolute_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)
evs = explained_variance_score(y_test, y_pred)
```

The print statements display the values of each metric to assess the model's performance.

<u>15. Plotting the final Predictions:</u>
It is a plot to visualize the final predictions made by the regression model compared to the actual values.
i.We plot the actual values (y_test) against the index of the testing data (test_scaled_df.index) (labelled as ‘Real’).
ii. Also the predicted values (y_pred) against the same index are plotted, and the line color is set to red (labelled 'Predicted').