Table of Contents

[1 Introduction 3](#_Toc4096538)

[2 Project description 4](#_Toc4096539)

[3 Used methods and objects 6](#_Toc4096540)

[4 Code description and data types 8](#_Toc4096541)

[5 Testing 14](#_Toc4096542)

[5.1 White box testing 14](#_Toc4096543)

[5.2 Black box testing 15](#_Toc4096544)

[6 References 16](#_Toc4096545)

# Introduction

This document is a final stage report for the project “Stock Market Predictor”, whose aim is to give an overview about the various stages of development of the project and some basic information about the technologies and theories used to forecast and analyse data sets.

The project was developed using the Python programming language, an easily readable language, with a user-friendly syntax well suited to scientific computing, data analysis and artificial intelligence. It has an extensive library of built-in functions, giving the possibility for additional libraries to be installed and imported as needed, facilitating coding. The Python interpreter used was Spyder: a powerful scientific environment specifically designed for data analysis.

The main objective of the project was to create a system capable of generating a prediction as accurate as possible of stock price drops and jumps to allow a potential investor to determine the most profitable moment to buy or sell shares. However, it was challenging to get an accurate prediction, since a multitude of random combined factors influence supply and demand, resulting in market fluctuation.

Over the years, with the development of machine learning, numerous techniques have been developed to make stock market predictions. One of the most known are the ANN, artificial neural networks, systems inspired to the human brain that try to replicate methods and functionalities of a real human brain.

Although an ANN represents by far the most accurate method available for stock market predictions, this project will focus on trying to overcome the predictive challenge using an autoregressive integrated moving average model known as ARIMA. It is also known as the Box-Jenkins method, introduced by Box and Jenkins in 1970 and has been proved to be able to efficiently generate short-term forecasts.

ARIMA uses historical data to make a future prediction of stock prices over the desired timeframe. A graphical user interface (GUI) was created to enable the user to easily communicate with the software.

# Project description

A share of stock represents a fractional ownership of a traded company. Share prices are contingent on company’s performance and the contextual environment in which it operates. At their core, stock prices are a representation of investors’ confidence. Today, trading is a largely computer-aided endeavour; however, it is human-driven events and biases that underpin the rise and fall of stocks. The inherent unpredictability of human behaviours makes forecasting stock trends with certainty extremely difficult.

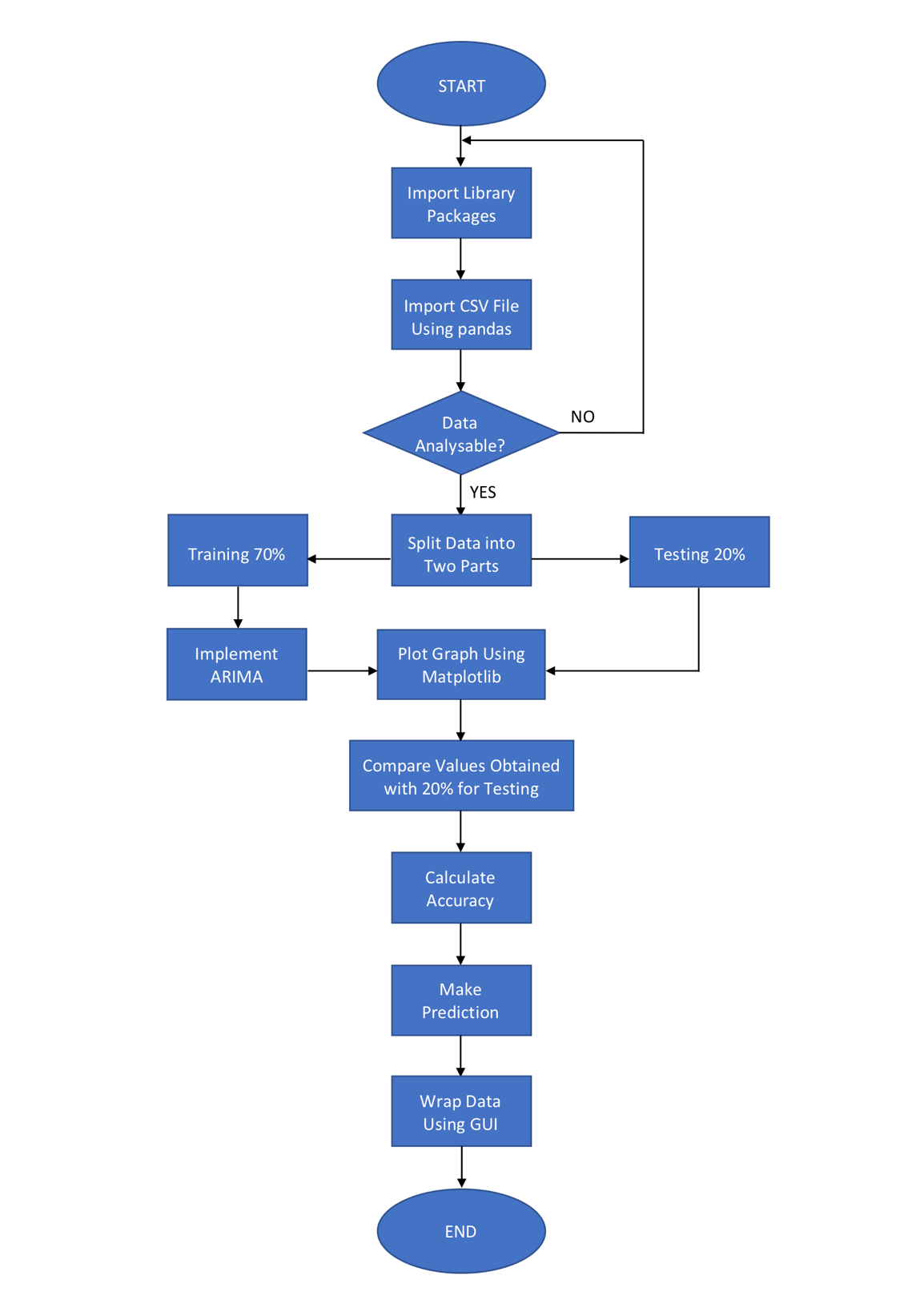
For each company, various types of stock prices are made available online: the opening price, which represents the value of shares at the opening bell of the stock market; the high and low which represent respectively the highest and lowest prices at which the stock was traded for the day, and the close, the price at the closing bell. The project focused its analysis on the closing prices, considered by investor the most significant as they represent the most up-to-date valuation of the day.

Time series are generally analysed for two purposes: to understand and model the random mechanism that results into an observed series or, to forecast the future values of a time series. Forecasts are based mainly on the time series history and, when possible, on other related factors. Different approaches may be taken depending on which subject data is related, such as biology, business or ecology.

One of the most used forecasting models for both stationary and non-stationary time series is the ARIMA model. ARIMA was used to model the historical data and get a future prediction of prices for the desired timeframe.

The flowchart in Figure 1 shows a top-level schematic of the program and summarizes the main steps of the stock market predictor. After importing the necessary libraries, the data is uploaded in a CSV format and is split into two parts, one for training the model (70 %) and one for testing future predictions (30 %).

Critical is the choice of the quantity of data used to train ARIMA. If the model is overwhelmed with data, this will result in an overfitting effect; conversely, insufficient data leads to an underfitting effect. This can be controlled by adjusting the parameters (p, d, q), as needed, to get the right amount of data needed for training and hence the most accurate prediction. The implementation of ARIMA results in the stock price forecast. The error between the predictions made for the timeline regarding the 30% test data and the corresponding historical data itself, is calculated using the mean squared method. Once ARIMA is trained, and its accuracy tested, the user can input the prediction length desired. The actual prediction can be then calculated and plotted with the historical test data and prediction using Matplotlib, a 2D Python plotting library, and displayed on the graphical user interface.



Testing 30 %

Compare Values Obtained with 30 % for Testing

**Figure 1**: Flowchart for Stock Market Predictor

# Used methods and objects

In order be able to import libraries onto the Python environment, the PIP3 package management system was installed. The first library that had to be imported was Statsmodels, that provides functions for statistic model estimates and data explorations and contains the ARIMA model. Pandas, a fundamental package used for scientific computing, easily allowed data import in a CSV format for recall when required, since it can be used as a container for dataset. [1]

To import the CSV file the function “read\_csv()” was used, specifying the name and location of the document. To test the first prototype, data regarding the Apple.Inc stock market was downloaded from <https://www.nasdaq.com/symbol/aapl/historical>. The program is able to analyse any stock market data; therefore, this process was automated, allowing the user to insert any CSV file. An Excel format as well as .tsv files could have been used since the data structure is compatible with pandas. [2] The Sklearn library was also used as it contains a series of tools useful for statistics, optimisation, integration, regression algorithms and enables the mean squared error between the real trend and the prediction to be calculated.

This project utilised a top-down approach that made the code, easy to break down into smaller parts called functions or methods, rather than writing the program in a large chunk of instructions. Code decomposition is a process that makes each function’s meaning clearer and hence the code more readable, manageable and easy to test. The project was decomposed in three main parts:

* get data
* process calculations
* display results

To do so, different functions containing smaller parts of code were defined. This allowed functions to be individually tested and executed when needed by calling them using the command buttons placed in the GUI.

The calculation process focused on the definition of the main function *ArimaModel()* containing 3 important subtasks. Firstly, using the Pandas library, the data necessary to execute the program was imported and split into test and train. The second subtask constitutes the core of the project: the implementation of the ARIMA model, where the model is trained with historical data, allowing it to make a prediction of the future points in the series.

The last part consisted in the method accuracy estimation using mean squared error and the plot of the predictions using matplotlib on an xy-Cartesian coordinate system. The mean squared error is derived from the difference between the actual data point, , and it is estimated value, , and can be calculated as in (1).

(1)

ARIMA is the most popular model used for forecasting a time series, both stationary and non-stationary.

A stationary series is a series in which statistical properties such as the mean are constant over time whereas a nonstationary series is one whose statistical properties variate over time.

If the series is non-stationary, it has to be converted into a stationary series by differentiation before being analysed. This is so because if a series is considered to be constantly increasing over time, the sample mean will grow with the size of the sample, and ARIMA will underestimate the mean for future periods resulting in an inaccurate prediction.

There are three main parameters in ARIMA, (p, d, q) that determine the accuracy of the output values. The term *p* represents the auto-regressive part of the model and it analyzes the likelihood that the function has to keep increasing or not, based on the past values. It represents the number of autoregressive terms. The term *d* is the integrated part of the model that determines, through integration, the number of points that have to be subtracted from the average trend because they are statistical outliers. It subtracts an observation from an observation at the previous step in order to make the series stationary. The term *q* is the moving normal part of the model, that is used to calculate the error graphically, comparing our prediction with the remaining historical data-set for testing.

In the ARIMA process, given a series of values of past stock prices, the future value results as a combination of past values Y and past errors ε at time t as in (2),

(2)

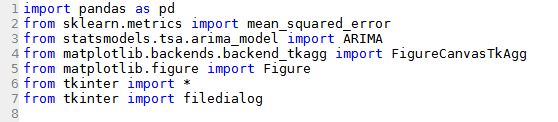
where p and q are the ARIMA parameters and φ and θ are coefficients, and where is (3) the difference between the actual value of the series and the forecast value. [3]

(3)

Before running the model, data was imported by the user using the browse button to trigger the *fileDialog()* function*.* The objective of this function was to upload the data file chosen by the user, allowing the addressed file to be into the function *ArimaModel().*

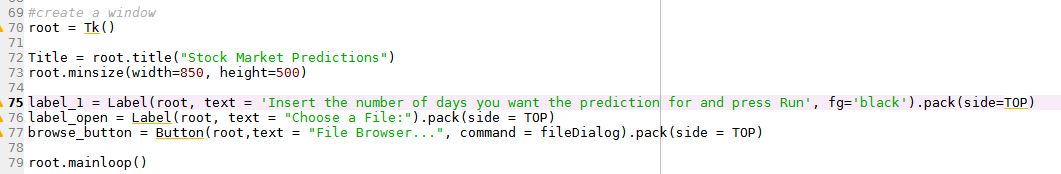
# Code description and data types

The dataset used in this simulation describes the stock market prices of Apple.Inc from 01/11/2018 to 02/01/2019. Libraries were imported as shown in Listing 1, after downloading the PIP3 package. The libraries necessary for this project were pandas imported as pd, the mean squared error function imported from the sklearn library and ARIMA imported from statsmodels. FigureCanvasTk and Figure were both imported from the matplotlib library as they were necessary for graph plotting. Tkinter on the other hand is Python’s standard GUI package and was imported and utilized to create the user interface.



**Listing 1**: libraries imported

To have a better understanding on how the code works, the code descriptions will follow the same order of steps that the code would follow when run, examining different fragments singularly. The code in Listing 2 shows that to initialize Tkinter, a Tk root widget was created.



**Listing 2**: Create GUI using tkinter

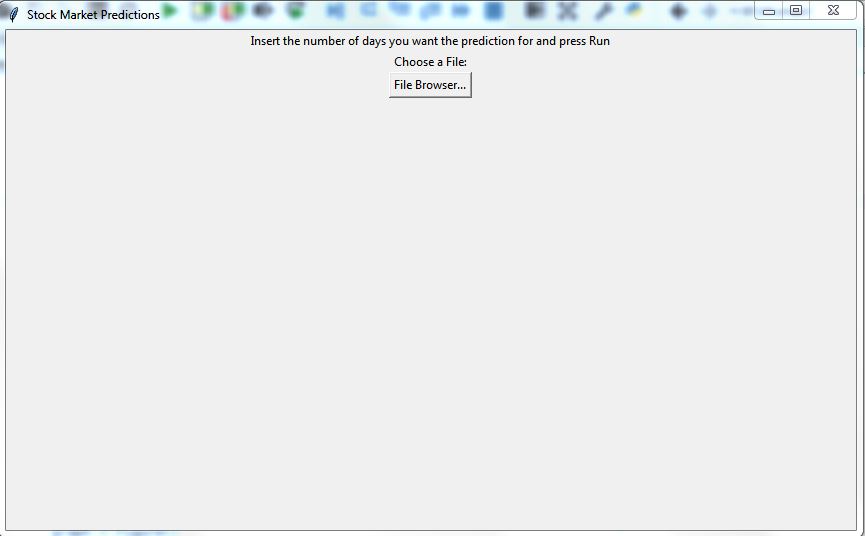
This represents an ordinary window, a container to populate with widgets, to which was assigned a title displayed in the title bar resulting in the empty window. A minimum size was also assigned, preventing minimizing but still allowing to maximize since the resizable is enabled by default.

A label was created, to show the instructions for the user on the GUI. Calling the *.pack* method tells the object to size itself to fit the content and make itself visible. The attribute *.pack(side=TOP*) was also used to assign the object a specific position at the top of the window. A second label was created using the same procedure to display the text *“Choose a File:”*.

A similar process was used for the button, with the only difference being that the button needs more arguments. After specifying the text “*File Browser…”* to indicate its usage, a command has to be also stated to be run when the button is clicked. The button click results in the execution of the defined function *fileDialog().*

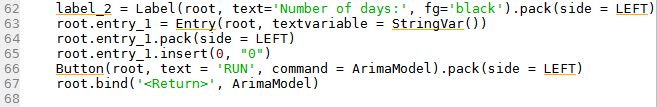
The roots object comes with a method called *mainloop()* that keeps the program in the loop and hence the window open, until it is closed by the user.

Executing this fragment of code results in the window in Figure 1.



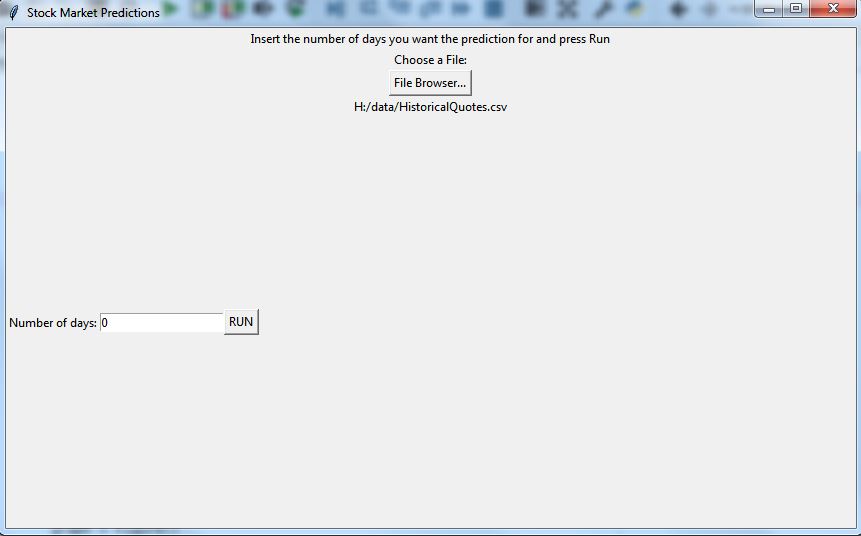
**Figure 1**: Window resulting from code in Listing 2

When the function *fileDialog()* shown in Listing 3 is called, the function *filedialog.askopenfilename*, requires the selection of an existing file.



**Listing 3**: *fileDialog ()* function

The initial directory was used by default *initialdir=”/”,* specifying the title but not including the extension of the file. A label was then created to display the directory of the file chosen. Once the file is selected, a new label with text *“Number of days:”*, a text entry field with a default text of *0* and a button with text “*RUN*” appear. The number of days desired for the prediction are considered as a string. Once the user clicks on the RUN button, being the relative command *ArimaModel*, it executes the function *ArimaModel().* This would result in the window shown in Figure 2.



**Figure 2**: Resulting window

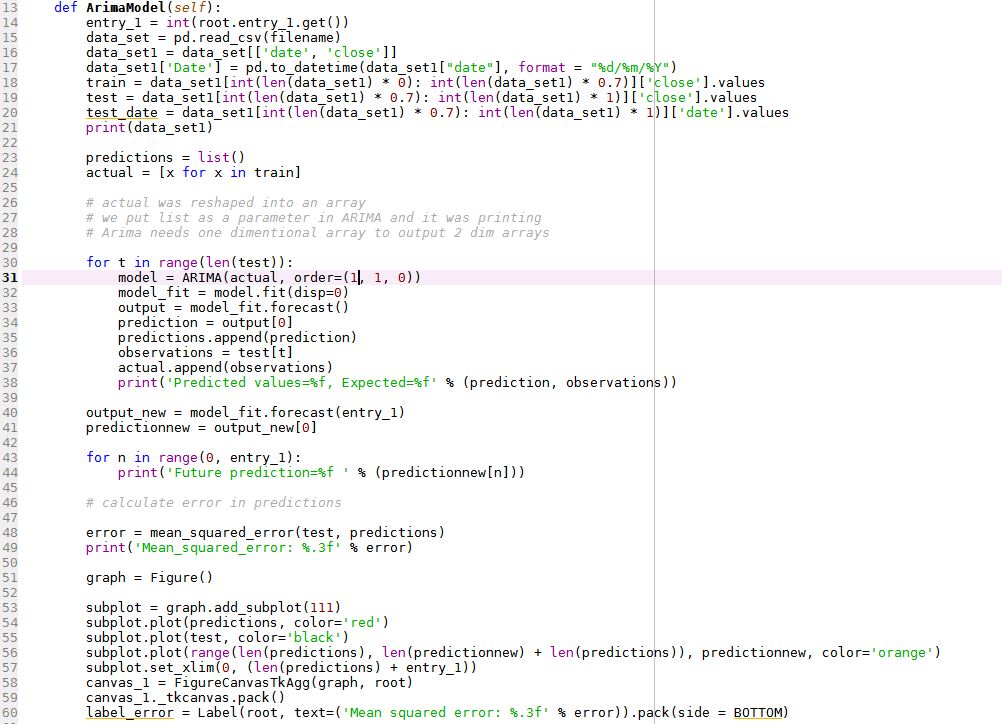
In the fragment of code referring to the function *ArimaModel(),* after importing and splitting the dataset, the ARIMA module is trained and the prediction made, calculating the error and plotting the predictions on the cartesian axes.

To clearly examine the *ArimaModel()* function, it was divided into smaller fragments showing the output of each part.

As shown in the code in Listing 4, being the variable *entry\_1* the number of days the user wants the prediction for, it has to be converted into an integer before being used since it is considered a string.

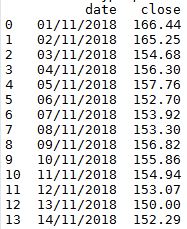
The pandas function .*read\_csv* was then used, specifying the file path as the directory of the file chosen by the user. There are two possible data structure that can be use when importing data with pandas: *Series*, that correspond to one dimensional data structure or *ndarray* capable of storing the values and index them, and *DataFrame*, a multidimensional data structure with indexed rows and named columns. In this case was used the *DataFrame* structure, specifying the name of the column to import, automatically indexed from pandas. For the *date* set, the pandas function *pd.to\_datetime* was used in order to specify the format wanted for the dates *%d/%m/%Y*.

The *data\_set1* close values were then split into train and test, where train holds the values from 0 to 0.7 of the length of the dataset, hence 70%, and test the remaining 30 % from 0.7 to 1.



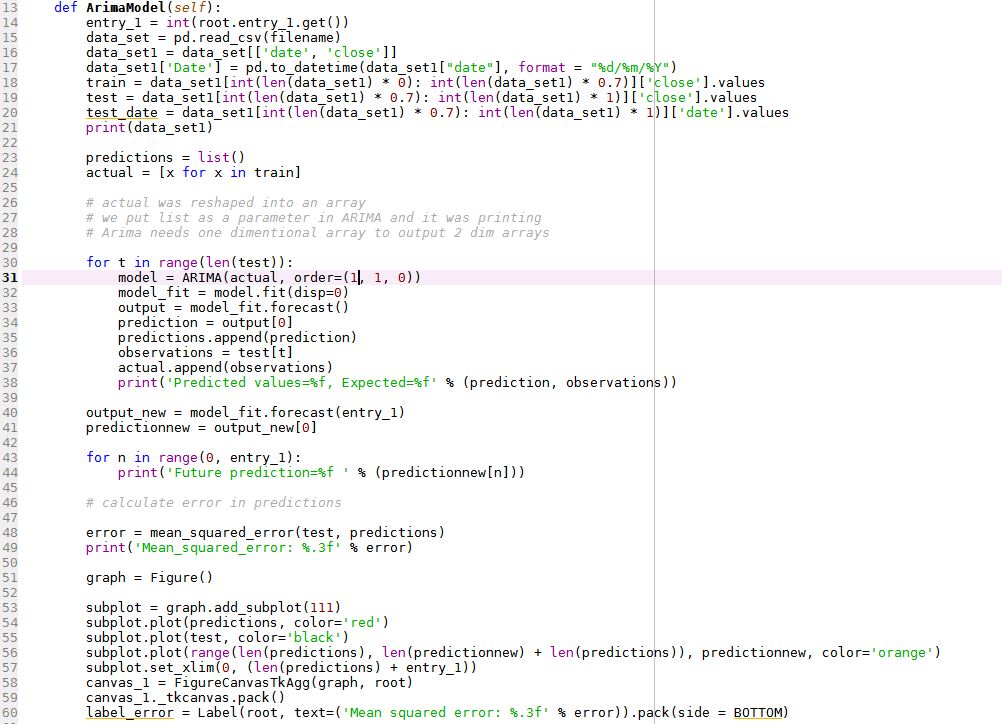
**Listing 4**: Importing and splitting data

Printing *data\_set1* would result in the list of dates and closing market prices as in Figure 3.



**Figure 3**: data imported from pandas

The code in Listing 5 shows the implementation of ARIMA.



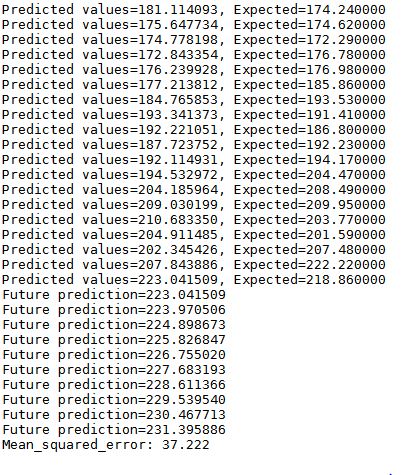
**Listing 5:** implementing the ARIMA model

In order for ARIMA to be trained, the variable train was reshaped into an array as in line ().

Also, a list was created named *predictions* to store the predictions values computed by the model. A for loop was used to iterate the model for the entire length of *test* passing the values one by one. The model was trained using the *actual* variable, corresponding to the reshaped *train*, fitting an ARIMA (1,1,0). This sets the autoregression value to 1, uses a difference of order 1 to make the series stationary and a moving average model of 0. Fitting the model, the displacement is set to 0 (turned off), since it would give by default a lot of debug information calculated during the analysis. The variable *prediction* corresponds to each forecasted value for the corresponding one in *test*, and was stored into the *predictions* list by using .*append.*

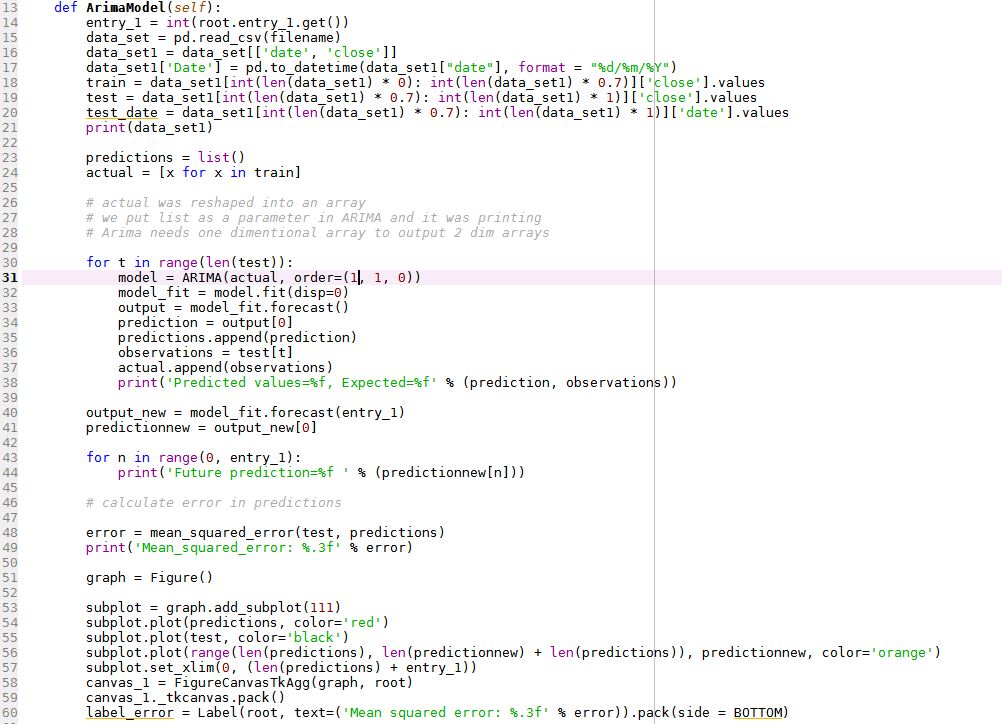
On the other hand, *observations* was used to append the corresponding value of *test* and printed in line with the *prediction* to compare them.

Being the model trained, a new prediction was made for the number of days entered by the user and stored in the variable *entry\_1*. The error was also calculated between the *predictions* made over the length of *test* and *test* itself using the MSE method as in (1). The calculated values were displayed as in Figure 4.



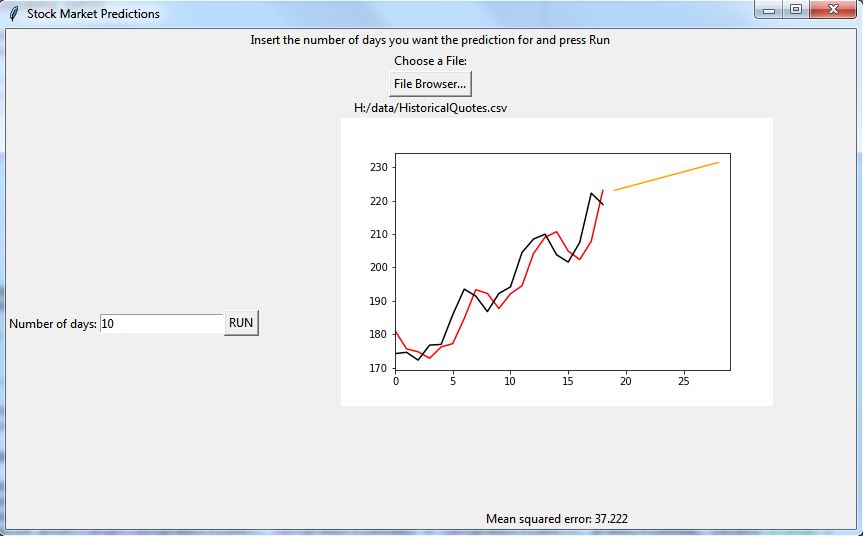
**Figure 4**: predicted values and error

As in Listing 6, to display the values a *Figure()* was created in order to show the different subplots. When creating the subplot using *.add\_subplot(111)* the three parameters (1,1,1) used stand for creating only one subplot with grid 1 x 1, corresponding to the cartesian coordinates.



**Listing 6**: displaying plots on the window

As shown in the graph in Figure 12, the test historical values, predictions and the forecasted values were plotted respectively in black, red and orange. As in line() in order for the forecasted values *predictionnew* to be correctly plotted, they had to be shifted along the x-axes for a length equal to le length of *test.* This allowed to the graph to resize itself depending on the length of the prediction wanted by the user. The graph was finally put into a canvas, a Tkinter widget that allows to display the graph plotted onto the GUI.



**Figure 5**: resulting graph

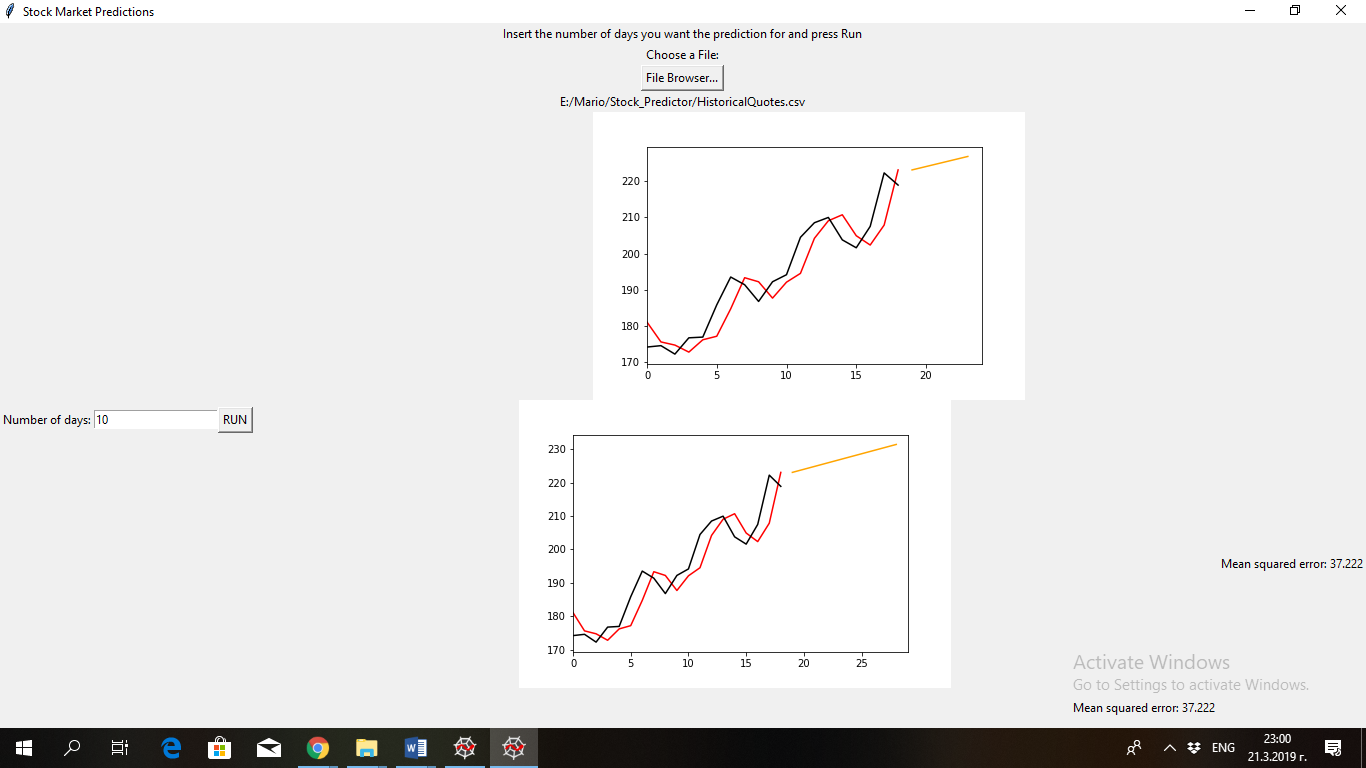
# Testing

Software testing is an essential stage of successful software development. Thorough testing is necessary if the software is to be published in an appropriate state. This can be a long and arduous process, especially for complex code. Developers cannot anticipate every strain the end-user will put upon the software and so must use a variety of testing methods to account for this. Basic testing procedure splits the task into white box and black box testing. White box testing looks at the internal code which comprises the software. Testing constraints are designed so as to check that the software responds as intended by the written code. This must be done by someone with sufficient expertise to understand the code’s functionality. In black box testing, the tester is unaware of the innerworkings of the software. They perceive the functionality and purpose of the software but not the underlying code. Hence, this emulates the conditions that the end-user will place upon the software. Unlike white box testing, this need not be carried out by experienced programmers.

## White box testing

One of the advantages of white box testing is that allows testing to be commenced at an earlier stage, even if the GUI is not yet available. In contrast to black box testing, having a better understanding of what the code is doing gives the possibility to cover more paths and test each function singularly. This testing part was commenced at a very early stage, to ensure that each function worked properly before proceeding with the code development. Throughout the process one of the problems during testing was caused by inconsistent indentation, that indicates when statements are parts of the body of a function. Particular attention was paid when declaring a variable, that allow to store information using up a portion of memory. Being the program large, to minimize the memory used to store variables, they were only declared when needed.

When testing the final code, it was noticed that if a second prediction is required by the user, this would be plotted and shown below the first prediction as in Figure 6.



**Figure 6**: Double plot on GUI

This problem could be solved by updating the old prediction and plotting the new graph in the same position of the first plot. Predicted values calculated by ARIMA could be also added on the GUI to allow the user to have a better understanding of how the prices are going to increase or decrease.

In general, every part of the code works properly without any confusion or delays.

Problems could occur in situations where the “*File browse…”* button is pressed with no following input of a file. Even if a number is chosen there would be no prediction because there is no actual data imported. A warning should be displayed to notice the user that A file must be imported: “*Error: to make a prediction import a .csv file with historical data first”*. Further improvements could have been made for the buttons on the GUI allowing to use the button Enter on the keyboard instead of clicking.

One of most important aspects of every program is it visual representation, so because of using tkinter which is simple GUI library it is difficult to achieve great and vivid design. Therefore, some of the future program updates will consist of new GUI package which is going to enable developers to succeed in making extremely appreciating product for every user.

To improve the functionality and accuracy of the program more forecasting algorithms could be added which are going to cooperate together in analyzing the data and making more accurate predictions. More information about the process of predicting will be presented to the user so more options to engage himself deeply in the world of stock predictions.

## Black box testing

The program was tested by a person not aware of its internal structure, but informed of its functionality, following and documenting every step.

Once the tester started the program, a line of instructions appeared on the screen. Since an historical data set was not provided, one was downloaded by the user from the London Stock exchange website. Being the headings of the columns in the file not compatible with the ones requested by the program, an error appeared.

When importing the file using pandas the program is asked to look for the columns with title “close” and “date”. Since Python is case sensitive, being the columns named in the downloaded file “Date” and “Close” they were not found. Therefore, another dataset was downloaded and accepted by the program that asked the user the number of days desired for the prediction. After entering the number and pressing on the run button, the program computed the future stock prices and displayed the graph on the GUI after about 10 seconds.

This leads to the conclusion that a clearer list of instructions should be provided when the program is executed, to avoid misunderstandings and malfunctioning issues with the user.

A legend could have been included stating the meaning of each trend line.

# References

[1] <https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/>

[last access 19/03/2019]

[2] <https://fxdata.cloud/tutorials/a-guide-for-time-series-forecasting-with-arima-in-python-3>

[last access 19/03/2019]

[3] Jonathan D. Cryer, Kung-Sik Chan, “Time Series Analysis with Applications in R”, Springer, New York, USA, 2008 (page 3-23, 79-98)