Section 5: Predicting Future Stock Behavior

In this lesson, you will try to predict future stock behavior.

We'll cover the following

There are two main techniques used to analyze stock behavior.

1. **Fundamental analysis**: This mostly deals with the intrinsic value of companies based on the various changes in their financials on a regular basis.
2. **Technical analysis**: This provides results based on the historical data of a company's stock.

Fundamental analysis is beyond the scope of this course, and the information required for it is also not easily accessible.

Until now, our focus has been on technical analysis, as we calculated various results from the historical data of the companies. However, these techniques won't help us predict the random and irregular behavior of stocks.

Random walk theory

Many analysts believe that the stock market prices follow the **random walk theory**. This theory states the following:

> ***The stock market may take random, irregular, and unpredictable paths in determining the prices of stocks. It also assumes that past trends are useless in predicting future prices and that the future price only***

> ***depends on the current price of the stock.***

Detailed information about this theory can be found here. This theory rejects both fundamental and technical analysis techniques with rational arguments.

We will also use the *random walk theory* to determine the future behavior of stocks. The *Monte Carlo simulations* will be deployed to assess the results using this theory.

MonteCarlo simulations

This method uses randomness to solve problems. It converts the randomness in the variables into probability distributions. It then generates a range of future price values in a normal distribution instead of just one value. More detailed information about this can be found here.

The following is the formula for MonteCarlo simulations:

$$S_{t+1} = S_t * e^{Drift+Rv}$$

$$Drift = AveragedDailyReturn - DailyReturnVariance/2$$

$$Rv = STD * NORMSINV(RAND())$$

Here, S_{t+1} is the *future price* of a stock. S_t is the current price of a stock. **e** is the universal constant. *Drift* and *Rv* are represented by their formulas. The *Drift* component represents the direction of stock, whether it'll go up or down. The *Rv* is our random variable; which either pushes the stock price up or down. More information on the `NORMSINV()` function can be found here.

Implementation

Now, let's see *monte-carlo simulations* in action by predicting future stock price values of the **Systems Ltd** company. Our final range of predictions will be in a multidimensional `NumPy` array that will be plotted to get the predicted range of future prices.

Before computing the results from the formula, some extra variables need to be calculated. We learned how to calculate daily returns for our stocks in the previous lesson. For this exercise, *logarithmic* returns will be calculated instead of daily returns, as they provide more concise information.

Calculating drift

Here we show how drift is calculated using logarithmic daily returns.

```
# Importing the packages
import numpy as np
import pandas as pd



# Preprocessing steps as before
sys = pd.read_csv('Year_2018/SYS.csv')
sys['Time'] = pd.to_datetime(sys.Time)
sys = sys.set_index('Time')

daily_returns = sys['Close'].pct_change() # Calculating daily returns

# Calculating log returns from daily returns
log_returns = np.log(1 + daily_returns)

avg = log_returns.mean() # Calculating average of log returns
var = log_returns.var() # Calculating variance

drift = avg - (var / 2.0) # Calculating drift

drift = np.array(drift) # Convert to array

print("The calculated Drift is:", drift)
```

After the preprocessing, the daily returns are calculated using the `pct_change()` method as discussed before.

On **line 15**, the *log* returns are calculated using the `np.log()` function of the `NumPy` package. The log of every value in the *daily return* `Series` is calculated after adding **1** to it and another `Series` is returned.

On **lines 17 & 18**, the respective *average* and *variance* of the *log-returns* are calculated.

On **line 20**, the drift component is calculated according to the above-described formula.

On **line 22**, the *drift* is converted to a `NumPy` array to simplify further calculations with arrays.

Calculating Rv

In this step, the random values are defined in accordance with the *random walk theory*.

```
# Importing the packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import norm

# Preprocessing steps as before
sys = pd.read_csv('Year_2018/SYS.csv')
sys['Time'] = pd.to_datetime(sys.Time)
sys = sys.set_index('Time')

daily_returns = sys['Close'].pct_change() # Calculating daily returns
log_returns = np.log(1 + daily_returns) # Calculating log returns from daily returns

avg = log_returns.mean() # Calculating average of log returns
var = log_returns.var() # Calculating variance
drift = avg - (var / 2.0) # Calculating drift
drift = np.array(drift) # Convert to array


pred_price_overDays = 60 # Number of days
pred_count = 10 # Range of prediction

std = log_returns.std() # Calculating STD
std = np.array(std) # Convert to array

x = np.random.rand(pred_price_overDays, pred_count) # get random multidimensional array

Rv = std * norm.ppf(x) # Calculating Rv

print("The required Rv array is:\n", Rv)
```

On **line 21**, a variable `pred_price_overDays` is defined; it determines the number of days for which we need to forecast the price.

On **line 22**, a variable `pred_count` is defined; it determines how many predictions we need.

On **lines 24 & 25**, the standard deviation is calculated and converted to a `NumPy` array object to simplify further calculations with arrays.

On **line 27**, the random values are generated using the `rand` function of `NumPy`. As two parameters are given to the function, a multidimensional array is generated. An array with **sixty** rows and **ten** columns is produced as described by the two variables. This array acts as a random input, as mentioned above.

On **line 29**, the function `norm.ppf()` of the `scipy` python package takes the inverse

of the normal distribution. As mentioned in the above formula

NORMSINV(RAND()), the random numbers are already calculated, and the `ppf()` function performs the inverse normal distribution function on those random values. In simple terms, this function takes the values generated by `rand()` and converts them to distances measured from their mean. The documentation is available here. After its calculation, the array is multiplied by the *standard deviation* to get the *Rv* value.

As all the components of the original formula, i.e., $S_{t+1} = S_t * e^{Drift+Rv}$ are calculated, the final future prices can finally be generated.

Summing it up

Now, only the current stock price and the **e** value needs to be fetched and calculated, and the corresponding future prices get generated.

```
e_value = np.exp(drift + Rv) # Calculating the E value

current_price = sys['Close'].iloc[-1] # Selecting last price of the year

new_prices = np.zeros_like(e_value) # create array to store the results

new_prices[0] = current_price

print(new_prices)
```

On **line 1**, the `exp()` function of `NumPy` is used. It takes *e* to the power of any number assigned in the parameters. The `drift` and `Rv` values are added, and then the `exp()` function is evaluated. This returns a *60 x 10* array with **60** as the number of days for which the price is forecasted and **10** as the number of predictions.

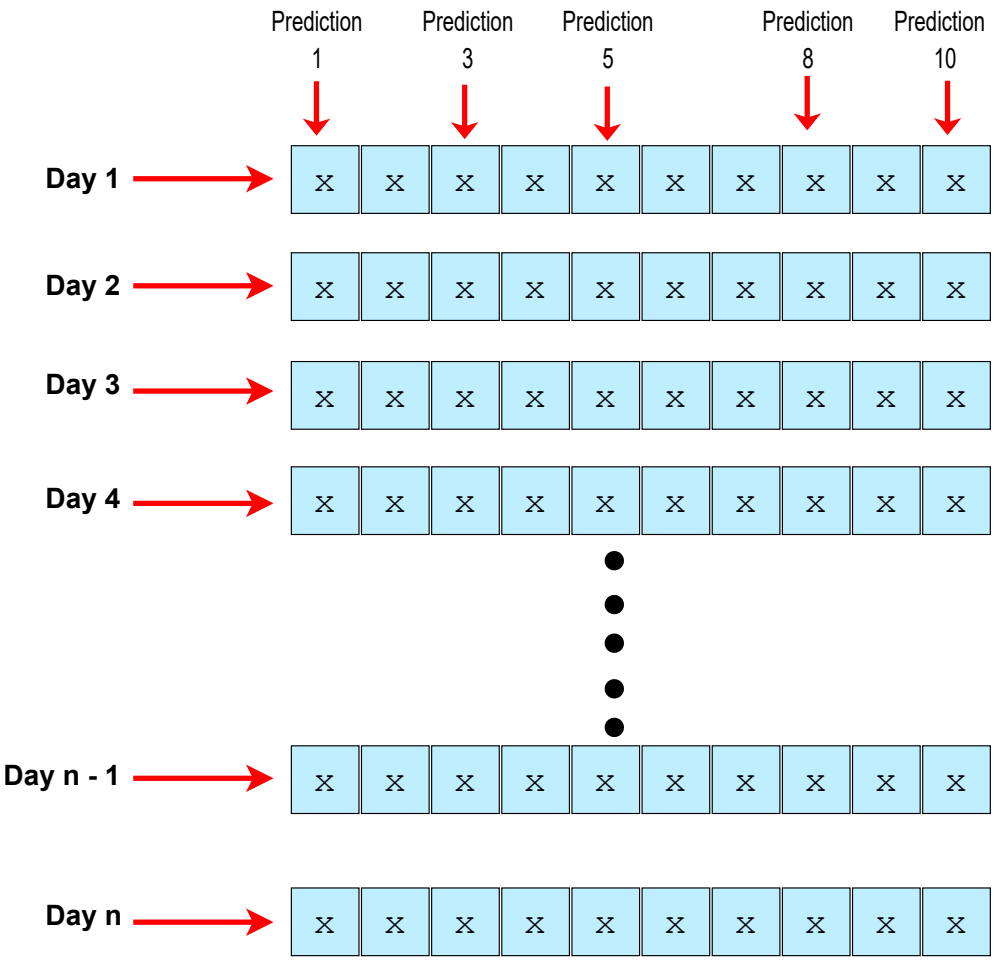
On **line 3**, the current price is fetched from the `DataFrame`. As the `DataFrame` is indexed on `Time`, it is sorted, and the last closing price is the most recent and current price of the stock.

On **line 5**, a new array `new_prices` with the exact shape as the `e_value` array is declared to store the new predicted prices. The `zeros_like` function declares this array and instantiates it with zeroes.

Now, each row value of the `new_prices` array is a different day containing the new

predicted price, and each column of the `new_prices` array is a new prediction for all the days. This concept is shown in the illustration below.

On **line** 7, the first row of the `new_prices` array is assigned the current stock value. Now, the new prices can generate in **ten** paths according to the `pred_count` variable with the same current starting price.



Predicted prices array

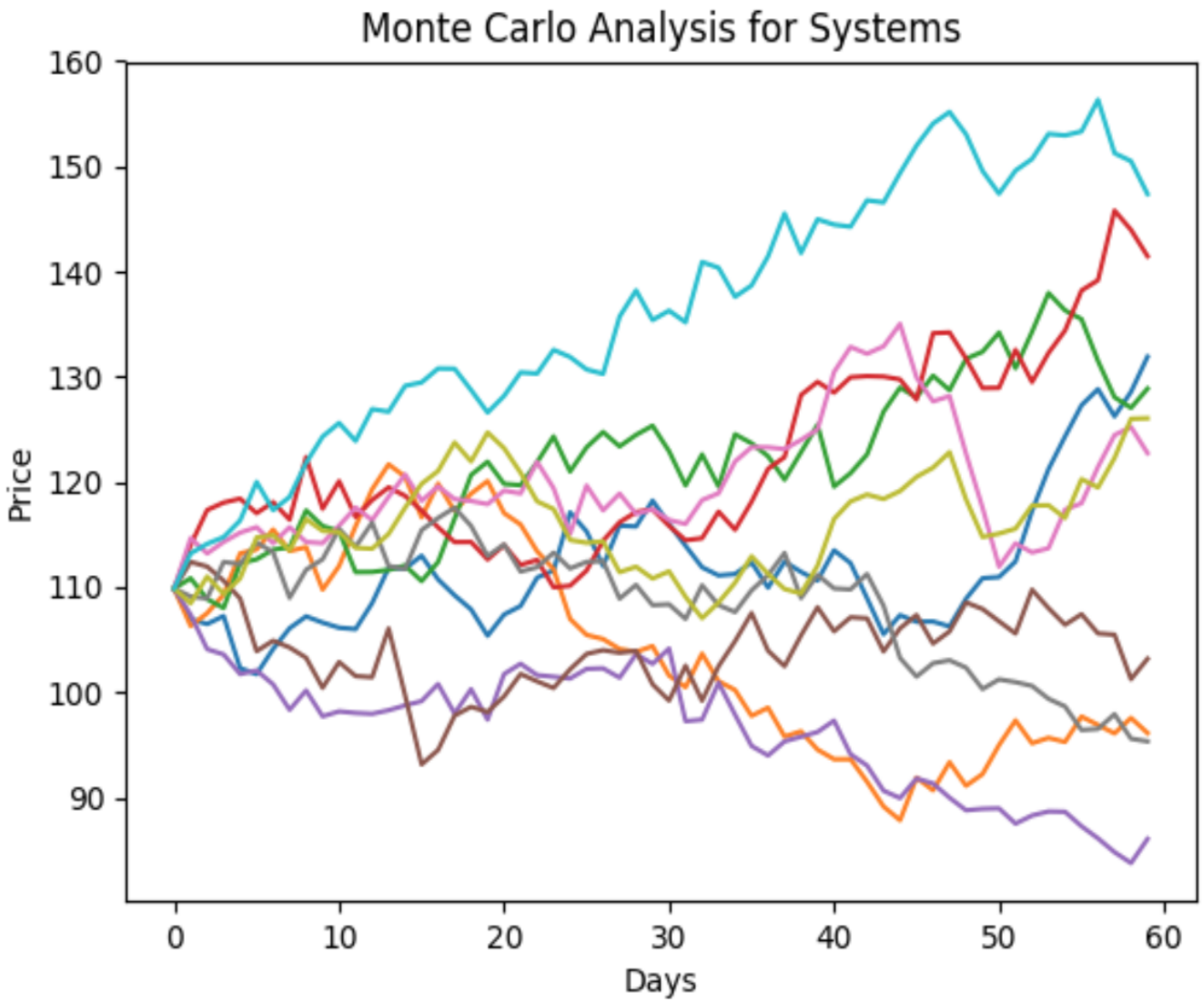
The above figure displays how the predicted prices are stored after calculation. Now, let's loop over the number of days and predict the price for each one.

```
for i in range(1, pred_price_overDays): # Loop over all the days to find their prices
    new_prices[i] = new_prices[i - 1] * e_value[i] # Calculating the future price with formula

print("The Minimum Predicted Price:", new_prices[pred_price_overDays-1].min()) # Get minimum price
print("The Maximum Predicted Price:", new_prices[pred_price_overDays-1].max()) # Get maximum price

plt.xlabel('Days') # Assign name to x-axis
plt.ylabel('Price') # Assign name to y-axis
plt.title('Monte Carlo Analysis for Systems') # Assign name to the plot
plt.plot(new_prices)# plot the figure

print("\nThe price array:\n", new_prices)
```



A plot like above, is generated from the output. It shows that from the current price, the stock can take *ten* different routes to predict the prices.

If you click the right arrow in the output, the minimum and maximum prices of the last day from all routes are shown with the whole price array with the predicted prices. According to the *monte-carlo method*, the original price should be close to the mean of each path that the stock takes.

On **lines 1 & 2**, the program loops over the total number of days defined and applies the initial formula. The new price is equal to the product of the price of the previous day and the **e** value calculated for that day. The previous day's price is set in the above code snippet.

On **lines 4 & 5**, the maximum and minimum prices for the last day are shown.

> The plot will be different every time the program is run. So you might get a different plot at different prices. The program can also be executed multiple times to compare the *mean* obtained in each run to figure out a more concise future price.

This marks the end of the stock market analysis project.