Final Project: The Misbehaviour of Markets:

Project Description:

1. Write a python program(s) to download end-of-day data last 25 years the major global stock market indices from Google Finance, Yahoo Finance, Quandl, CityFALCON, or another similar source.
2. It is a common assumption in quantitative finance that stock returns follow a normal distribution whereas prices follow a lognormal distribution for all these indices check how closely price movements followed a log-normal distribution.
3. Verify whether returns from these broad market indices followed a normal distribution?
4. For each of the above two parameters (price movements and stock returns) come up with specific statistical measures that clearly identify the degree of deviation from the ideal distributions. Graphically represent the degree of correspondence.
5. One of the most notable hypothesis about stock market behaviour is the "Efficient market hypothesis" which also internally assume that market price follows a random-walk process. Assuming that Stock Index prices follow a geometric Brownian motion and hence index returns were normally distributed with about 20% historical volatility, write a program sub-module to calculate the probability of an event like the 1987 stock market crash happening? Explain in simple terms what the results imply.
6. What does "fat tail" mean? Plot the distribution of price movements for the downloaded indices (in separate subplot panes of a graph) and identify fat tail locations if any.
7. It is often claimed that fractals and multi-fractals generate a more realistic picture of market risks than log-normal distribution. Considering last 10 year daily price movements of NASDAQ, write a program to check whether fractal geometrics could have better predicted stock market movements than log-normal distribution assumption. Explain your findings with suitable graphs.

NOTE:

Program may require another run, if it errors out on the first. This is a known bug in Fix Yahoo Finance, and hence, it returns empty DataFrames that error out the program. The program is fully functional, and if it errors out in the beginning, please give it another run.

Python Libraries used:

The following libraries have been used. They can be installed either using Anaconda (https://www.continuum.io/downloads) or using the Unix command *pip* as follows:

```
>> pip install “Library Name”
```

```
import logging  # Logging class for logging in the case of an error, makes
debugging easier
import sys  # For gracefully notifying whether the script has ended or not
from pandas_datareader import data as pdr  # The pandas Data Module used for
fetching data from a Data Source
import pandas as pd  # For calculating coefficient of correlation
import numpy as np  # For getting log values from prices
import warnings  # For removing Deprecation Warning w.r.t. Yahoo Finance Fix
import matplotlib.pyplot as plt
import scipy.stats
import statsmodels.graphics.gofplots as sm
import math
```

Program Description and inference:

The program beings with the “if __name__ == ‘main’” block, which initializes the logger and executes the “main()” function, that contains all the logic for solving the problem set in a sequential manner

```
if __name__ == '__main__':
    # Initialize Logger
    logging.basicConfig(format='%(asctime)s %(message)s: ')
    logging.info('Application Started')
    exit_code = main()
    logging.info('Application Ended')
    sys.exit(exit_code)
```

Within the main function, before any of the functions are executed, we do a “Step 0: Sanitization”. In this step, we fix the Yahoo Finance API, because of changes done to Yahoo Finance, where we cannot use it directly. For this, we use the fix_yahoo_finance library, which is initialized according to the instructions specified at (https://github.com/ranaroussi/fix-yahoo-finance) within the function “yahoo_finance_bridge()”. After initializing we go to the next step, that is given below which solves the following:

1. ***Write a python program(s) to download end-of-day data last 25 years the major global stock market indices from Google Finance, Yahoo Finance, Quandl, CityFALCON, or another similar source.***

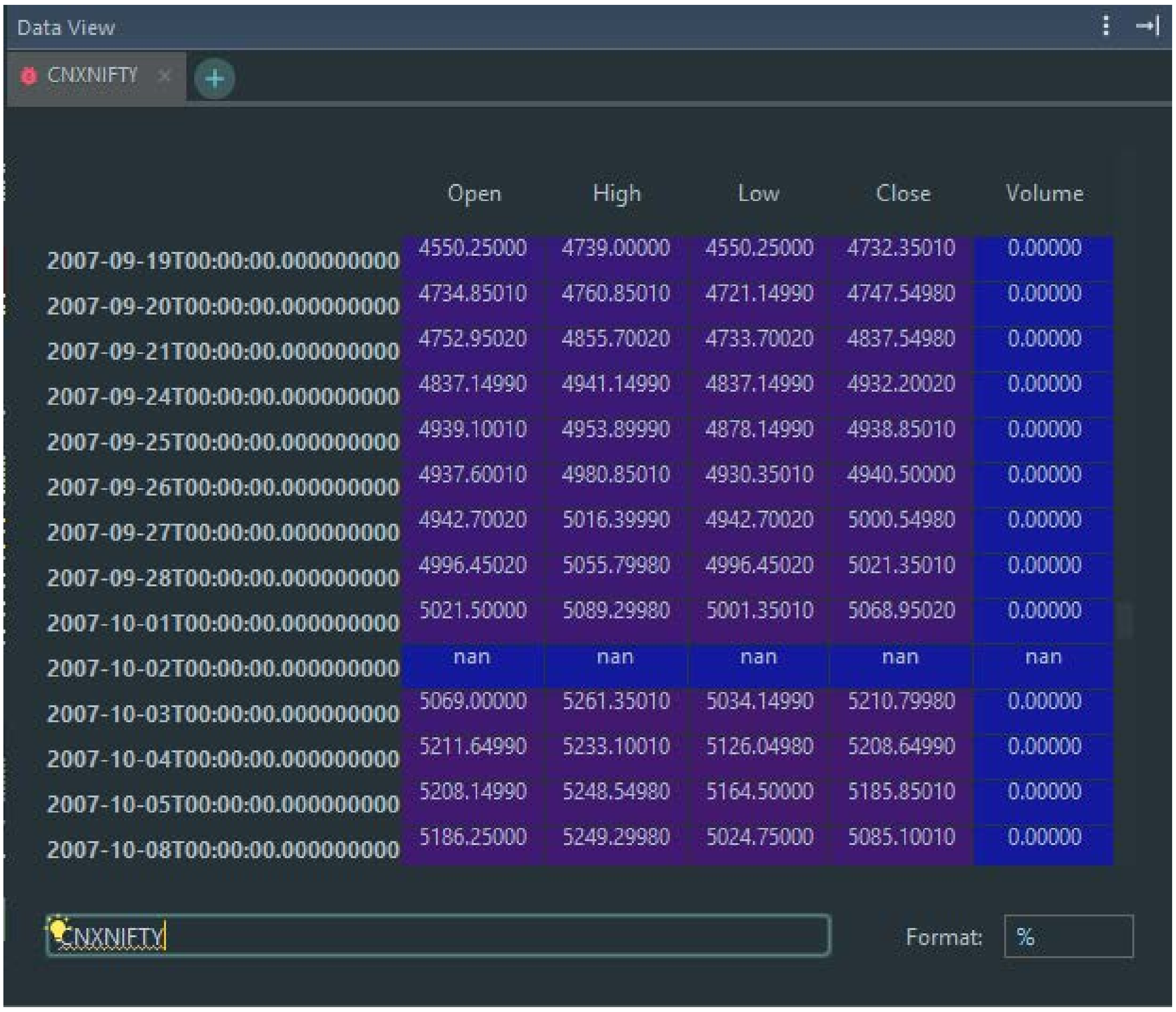
   Yahoo finance was chosen to get the data for the last 25 years. Although I have chosen all of the indices specified, an index from each region (AEJ, EMEA, AM) should have sufficed for the study, but then again, it’s a conscious call of whether you want more supporting data, or if you want the program to run on a daily basis in quick succession. In our use-case, it’s just a single time run to do some analytics.

The call to Yahoo Finance is done with a single call, which returns us a MultiIndex DataFrame as specified by the optional parameters. The API call is well documented at (https://github.com/ranaroussi/fix-yahoo-finance) and hence I won't be going in detail explaining every aspect of the API.

KOSPI and CNX NIFTY, both these indices do not have 25 year old data. Therefore, I had two choices here:

a. Backfill them: There is also the fear of skewing your results if you backfill too much, or use varying data sources.
b. Interpolate holes, while dropping those that cannot be interpolated: I decided to drop NaN using the dropna() function in pandas for those rows that do not have data. But there is a catch here. Sometimes there could be a legitimate data point that could be interpolated. For example, the screenshot given below:

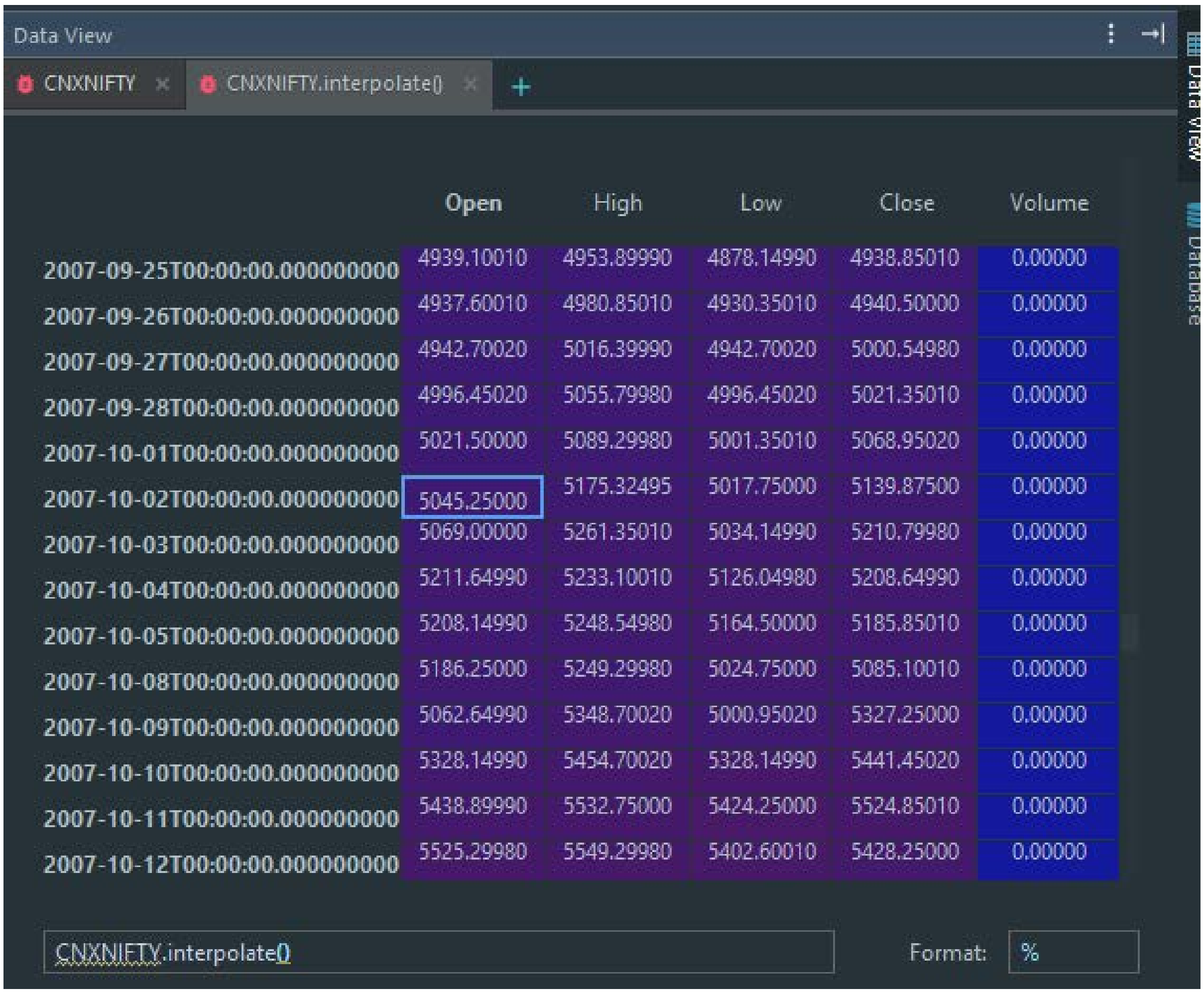
```
data = pdr.get_data_yahoo(['^DJI', '^GSPC', '^IXIC', '^GDAXI', '^FTSE', '^HSI',
'^KS11', '^NSEI'],
                          start="1991-10-01", end="2017-10-01", as_panel=False,
group_by='index',
                          auto_adjust=True)
# Assign them to variables,
# Interpolate holes, using a simple Linear Interpolation,
# and drop NaN rows that could not be interpolated, since backfilling could skew
results
DJI = data['^DJI'].interpolate().dropna()
SP500 = data['^GSPC'].interpolate().dropna()
NASDAQ = data['^IXIC'].interpolate().dropna()
DAX = data['^GDAXI'].interpolate().dropna()
FTSE = data['^FTSE'].interpolate().dropna()
HSI = data['^HSI'].interpolate().dropna()
KOSPI = data['^KS11'].interpolate().dropna()
CNXNIFTY = data['^NSEI'].interpolate().dropna()
```



	Open	High	Low	Close	Volume
2007-09-19T00:00:00.000000000	4550.25000	4739.00000	4550.25000	4732.35010	0.00000
2007-09-20T00:00:00.000000000	4734.85010	4760.85010	4721.14990	4747.54980	0.00000
2007-09-21T00:00:00.000000000	4752.95020	4855.70020	4733.70020	4837.54980	0.00000
2007-09-24T00:00:00.000000000	4837.14990	4941.14990	4837.14990	4932.20020	0.00000
2007-09-25T00:00:00.000000000	4939.10010	4953.89990	4878.14990	4938.85010	0.00000
2007-09-26T00:00:00.000000000	4937.60010	4980.85010	4930.35010	4940.50000	0.00000
2007-09-27T00:00:00.000000000	4942.70020	5016.39990	4942.70020	5000.54980	0.00000
2007-09-28T00:00:00.000000000	4996.45020	5055.79980	4996.45020	5021.35010	0.00000
2007-10-01T00:00:00.000000000	5021.50000	5089.29980	5001.35010	5068.95020	0.00000
2007-10-02T00:00:00.000000000	nan	nan	nan	nan	nan
2007-10-03T00:00:00.000000000	5069.00000	5261.35010	5034.14990	5210.79980	0.00000
2007-10-04T00:00:00.000000000	5211.64990	5233.10010	5126.04980	5208.64990	0.00000
2007-10-05T00:00:00.000000000	5208.14990	5248.54980	5164.50000	5185.85010	0.00000
2007-10-08T00:00:00.000000000	5186.25000	5249.29980	5024.75000	5085.10010	0.00000

CNXNIFTY

As we can see, we can use the interpolate() function (the default is linear interpolation) to derive the value of this missing data point which is in between two datapoints. The interpolation gives us the following results:



	Open	High	Low	Close	Volume
2007-09-25T00:00:00.000000000	4939.10010	4953.89990	4878.14990	4938.85010	0.00000
2007-09-26T00:00:00.000000000	4937.60010	4980.85010	4930.35010	4940.50000	0.00000
2007-09-27T00:00:00.000000000	4942.70020	5016.39990	4942.70020	5000.54980	0.00000
2007-09-28T00:00:00.000000000	4996.45020	5055.79980	4996.45020	5021.35010	0.00000
2007-10-01T00:00:00.000000000	5021.50000	5089.29980	5001.35010	5068.95020	0.00000
2007-10-02T00:00:00.000000000	5045.25000	5175.32495	5017.75000	5139.87500	0.00000
2007-10-03T00:00:00.000000000	5069.00000	5261.35010	5034.14990	5210.79980	0.00000
2007-10-04T00:00:00.000000000	5211.64990	5233.10010	5126.04980	5208.64990	0.00000
2007-10-05T00:00:00.000000000	5208.14990	5248.54980	5164.50000	5185.85010	0.00000
2007-10-08T00:00:00.000000000	5186.25000	5249.29980	5024.75000	5085.10010	0.00000
2007-10-09T00:00:00.000000000	5062.64990	5348.70020	5000.95020	5327.25000	0.00000
2007-10-10T00:00:00.000000000	5328.14990	5454.70020	5328.14990	5441.45020	0.00000
2007-10-11T00:00:00.000000000	5438.89990	5532.75000	5424.25000	5524.85010	0.00000
2007-10-12T00:00:00.000000000	5525.29980	5549.29980	5402.60010	5428.25000	0.00000

CNXNIFTY.interpolate()   Format: %

As for those data points that could not be interpolated (because they lie from the initial data point, that is, year 1991, all the way up to the first legitimate value, which was somewhere in 2003) I ended up dropping those values. An interpolation of that would mean that we have way too many data points that have been derived which can skew the results, and hence our distribution will not be the desired distribution which we are testing because a large set of data is skewing it, which will lead it to become a 'fat-tailed' distribution (more on that later). Therefore, I took the conscious decision of dropping large contiguous NaN values, and interpolating the 'holes' in between for all the market indices.

2. ***It is a common assumption in quantitative finance that stock returns follow a normal distribution whereas prices follow a lognormal distribution for all these indices check how closely price movements followed a log-normal distribution.***

This statement was quite open-ended. What does "prices" here mean? Does it mean the "Price Action" otherwise known as "Price Movement"? For my analysis I have assumed the Close prices and not used the Price Movements. To clarify, here's why:

A price movement is given by the following formula:
>> Today (Adj. Close) – Previous (Adj. Close)

This can, however, give us negative values. Therefore, I could not use this (price movement) to test whether the given distribution is log-normal, because log-normal distribution has two important characteristics:

- It has a lower bound of zero.
- The distribution is skewed to the right, i.e., it has a long right tail.

A lognormal distribution is commonly used to describe distributions of financial assets such as share prices. A lognormal distribution is more suitable for this purpose because asset prices cannot be negative.

To test for normality the following steps can be performed:

a. First we need to define a null hypothesis and an alternative hypothesis. For our case, the null hypothesis will be 'The prices follow a log-normal distribution', while the alternative hypothesis will be 'The prices do not follow a log-normal distribution'

On the basis of H0 and H1, we can take the log of the prices and check whether this follows a normal distribution or not. This way, if it does, we can be certain that h0 holds true, provided our p-value is higher than alpha, which in our case would be 0.05. Else, we accept the alternative hypothesis, which is, H1. Therefore, a normal distribution test on the logarithmic values of the prices should give us whether the prices follow a log-normal distribution or not, demonstrated below

```
p=sp.stats.mstats.normaltest(a.apply(lambda x:np.log(x)), axis=0).pvalue
if p<0.01:
   print 'distribution is not log-normal'
```

The other way is to use Kolmogorov-Smirnov test to find out whether the 'Price Movements' follow a Log-Normal distribution or not. Kolgomorov-Smirnov test is used as a test of goodness of fit. It compares the cumulative distribution function for a variable with a specified distribution. The null hypothesis assumes no difference between the observed and theoretical distribution and the value of test statistic 'D' is calculated as

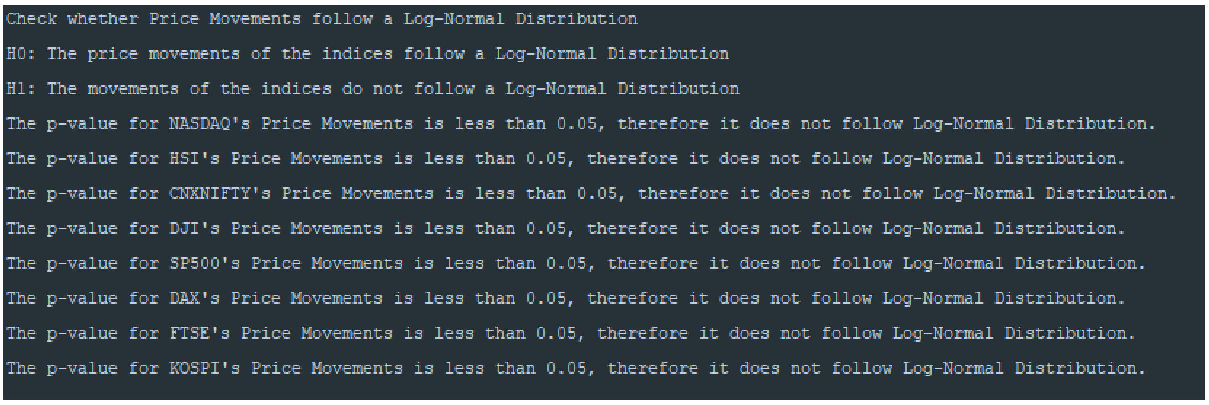
$$D=\text{Maximum}|F_o(X)-F_r(X)|$$

Where −

- Fo(X)Fo(X) = Observed cumulative frequency distribution of a random sample of n observations.
- and Fo(X)=knFo(X)=kn = (No.of observations ≤ X)/(Total no.of observations).
- Fr(X)Fr(X) = The theoretical frequency distribution.

The critical value of DD is found from the K-S table values for one sample test.

For our given cases, we have the following output:



```
Check whether Price Movements follow a Log-Normal Distribution
H0: The price movements of the indices follow a Log-Normal Distribution
H1: The movements of the indices do not follow a Log-Normal Distribution
The p-value for NASDAQ's Price Movements is less than 0.05, therefore it does not follow Log-Normal Distribution.
The p-value for HSI's Price Movements is less than 0.05, therefore it does not follow Log-Normal Distribution.
The p-value for CNXNIFTY's Price Movements is less than 0.05, therefore it does not follow Log-Normal Distribution.
The p-value for DJI's Price Movements is less than 0.05, therefore it does not follow Log-Normal Distribution.
The p-value for SP500's Price Movements is less than 0.05, therefore it does not follow Log-Normal Distribution.
The p-value for DAX's Price Movements is less than 0.05, therefore it does not follow Log-Normal Distribution.
The p-value for FTSE's Price Movements is less than 0.05, therefore it does not follow Log-Normal Distribution.
The p-value for KOSPI's Price Movements is less than 0.05, therefore it does not follow Log-Normal Distribution.
```

The following for-loop takes in each of the close prices and compares them with ‘lognorm’. If we get a p-value less than alpha, we accept the alternative hypothesis.

```
def lognormal_check_markets(dataframe_dictionary):
    """
    This function takes in markets and does a check to find out whether they the
price movements follow
    a Log Normal Distribution

    :param: dataframe_dictionary: Consist of a Dictionary Mapping of Market Name to
Maket Data DataFrame
    :type: dict

    :return: dict
    """
    print('\n\nCheck whether Price Movements follow a Log-Normal Distribution')
    print('H0: The price movements of the indices follow a Log-Normal
Distribution')
    print('H1: The movements of the indices do not follow a Log-Normal
Distribution\n')

    for key, value in dataframe_dictionary.items():
        under_test = scipy.stats.kstest(value['Close'], "lognorm",
scipy.stats.lognorm.fit(value['Close']))
        # Populate Return Dictionary
        if under_test.pvalue < 0.05:
            print('The p-value for ' + key + '\'s Price Movements is less than
0.05, '
                                             'therefore it does not follow Log-
Normal Distribution.')
        else:
            print(key + '\'s Price Movements follow a Log-Normal Distribution')
```

Hence, **stock prices do not follow a log-normal distribution.**

The other part of this question asks us to check how closely it followed *log-normal distribution*. For this, I’ll take the log of the data, and then apply a normality test on it. If it passes the normality test, this means that the data is log-normally distributed. In other words, I can take the log of the prices, and check its skewness, kurtosis and/or deviation from normal distribution. This way, I’ll have the have the deviation of the existing distribution from log-normal distribution. To achieve that, I use np.log(close_price) to get log of the values. Then I calculate the t-statistic to show the deviation, along with skewness and kurtosis. This along with descriptive statistics such as mean and median depict how far off the logarithmic values are to normal distribution, which in turn means how far off the values are to the logarithmic distribution.

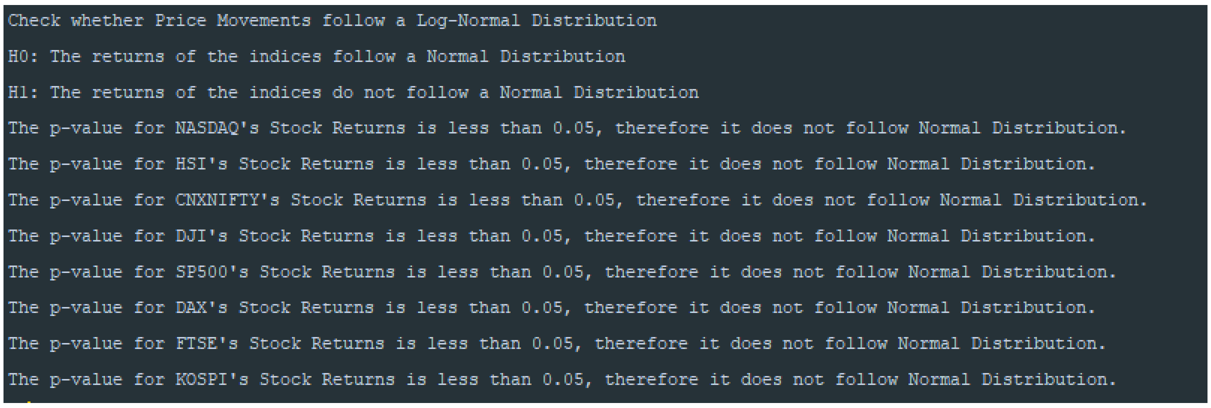
```python
def lognormal_check_deviation(data_dict):
    """
    This function takes a dictionary containing Markets DataFrame, and does a
np.log() on the closing
    prices. Then it gives skewness, kurtosis, mean, median std, min and max of the
values
    :param data_dict: The dictionary containing the DataFrames
    :return: dict
    """
    # Iterate through every market
    for key, value in data_dict.items():
        # Take the log of close prices
        close_price = np.log(value['Close'])
        print('\n===== Calculating Statistic for Prices for %s =====' % key)
        skew = scipy.stats.skewtest(close_price)
        print('Skew for %s is : %f' % (key, skew.statistic))
        kurt = scipy.stats.kurtosistest(close_price)
        print('Kurtosis for %s is : %f' % (key, kurt.statistic))
        t_test = scipy.stats.ttest_1samp(close_price, 0)
        print('T-Statistic for %s is : %f' % (key, t_test.statistic))
        desc = close_price.describe()
        print('Mean for %s is : %f' % (key, desc.iloc[1]))
        print('Median for %s is : %f' % (key, desc.iloc[5]))
        print('Standard Deviation for %s is : %f' % (key, desc.iloc[2]))
        print('Min for %s is : %f' % (key, desc.iloc[3]))
        print('Max for %s is : %f' % (key, desc.iloc[7]))
        print('========================================\n')
```

3. ***Verify whether returns from these broad market indices followed a normal distribution?***

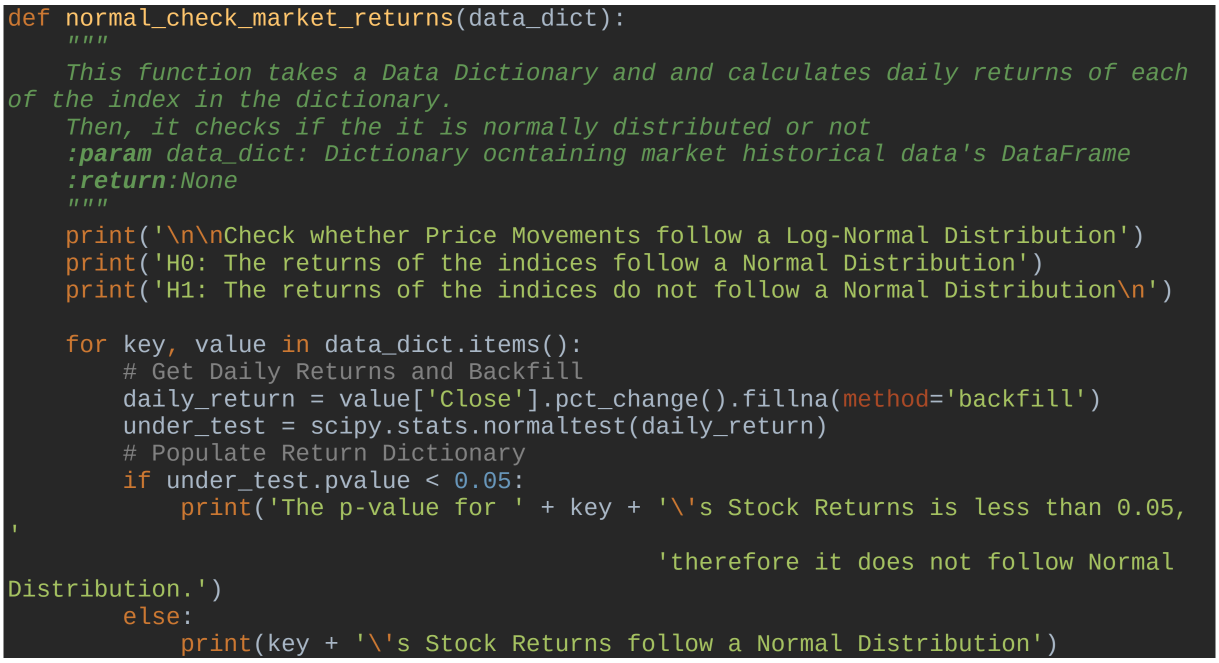
First, we need to calculate the returns on each of the indices. For this, we use the pct_change() method in pandas with a period of 1 to get daily returns.

!! –INSERT CODE HERE

Then, we check whether this data is normally distributed or not, using scipy's normtest method, where null hypothesis or H0 states whether the distribution comes from a normal distribution while the alternative hypothesis H1 states whether the distribution does not come from Normal Distribution. This way, if our pvalue is less than alpha, which in our case is 0.05, then we reject H0 and accept H1. Else, we accept H0. Therefore, we have the following output:


```
Check whether Price Movements follow a Log-Normal Distribution
H0: The returns of the indices follow a Normal Distribution
H1: The returns of the indices do not follow a Normal Distribution
The p-value for NASDAQ's Stock Returns is less than 0.05, therefore it does not follow Normal Distribution.
The p-value for HSI's Stock Returns is less than 0.05, therefore it does not follow Normal Distribution.
The p-value for CNXNIFTY's Stock Returns is less than 0.05, therefore it does not follow Normal Distribution.
The p-value for DJI's Stock Returns is less than 0.05, therefore it does not follow Normal Distribution.
The p-value for SP500's Stock Returns is less than 0.05, therefore it does not follow Normal Distribution.
The p-value for DAX's Stock Returns is less than 0.05, therefore it does not follow Normal Distribution.
The p-value for FTSE's Stock Returns is less than 0.05, therefore it does not follow Normal Distribution.
The p-value for KOSPI's Stock Returns is less than 0.05, therefore it does not follow Normal Distribution.
```

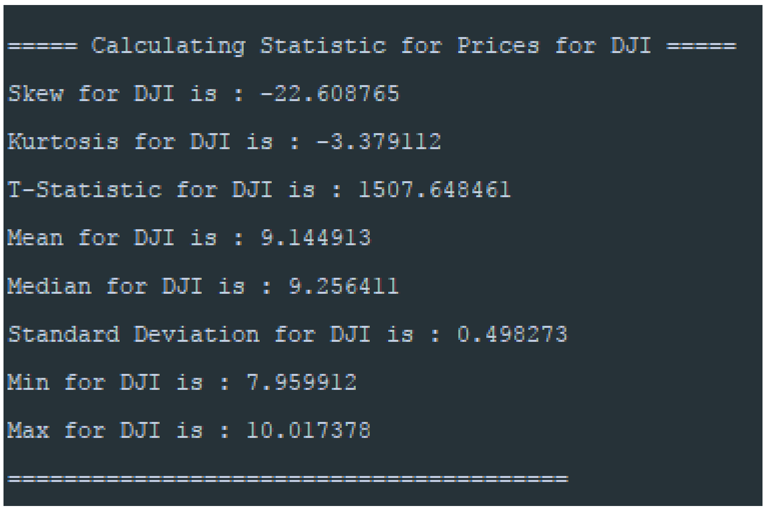
For the following code:


```
def normal_check_market_returns(data_dict):
    """
    This function takes a Data Dictionary and and calculates daily returns of each
of the index in the dictionary.
    Then, it checks if the it is normally distributed or not
    :param data_dict: Dictionary ocntaining market historical data's DataFrame
    :return:None
    """
    print('\n\nCheck whether Price Movements follow a Log-Normal Distribution')
    print('H0: The returns of the indices follow a Normal Distribution')
    print('H1: The returns of the indices do not follow a Normal Distribution\n')

    for key, value in data_dict.items():
        # Get Daily Returns and Backfill
        daily_return = value['Close'].pct_change().fillna(method='backfill')
        under_test = scipy.stats.normaltest(daily_return)
        # Populate Return Dictionary
        if under_test.pvalue < 0.05:
            print('The p-value for ' + key + '\'s Stock Returns is less than 0.05,
'
                                                  'therefore it does not follow Normal
Distribution.')
        else:
            print(key + '\'s Stock Returns follow a Normal Distribution')
```

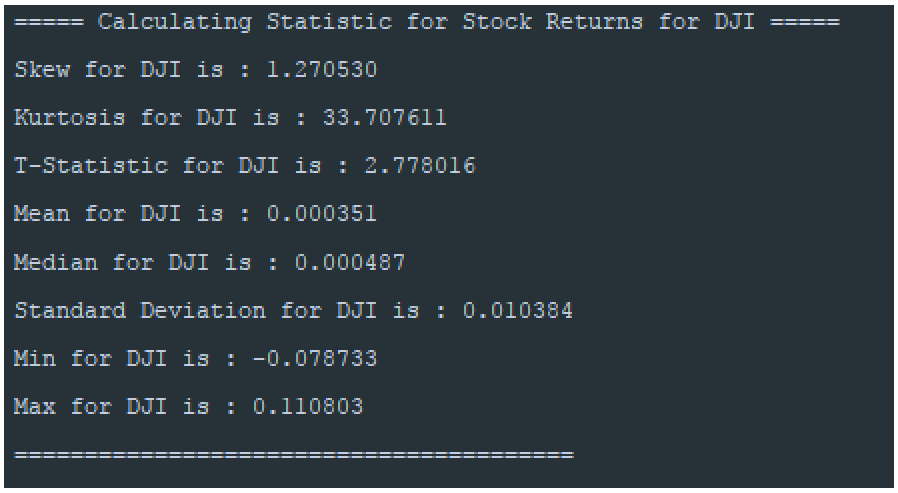
Hence, we accept our alternative hypothesis, which states that Stock Returns do not follow Normal Distribution. Hence, **stock returns do not follow a log-normal distribution.**

4. ***For each of the above two parameters (price movements and stock returns) come up with specific statistical measures that clearly identify the degree of deviation from the ideal distributions. Graphically represent the degree of correspondence***

We have already described using different statistics the deviation of prices from log normal distribution (such as skewness and kurtosis). However, for completeness sake, we'll be calling the same function, that is, lognormal_check_deviation() to print out the values again. In a similar fashion, we create a normal_check_deviation() function, that will print out statistics such as skewness and kurtosis etc. to show the deviation of **stock returns** from Normal Distribution.

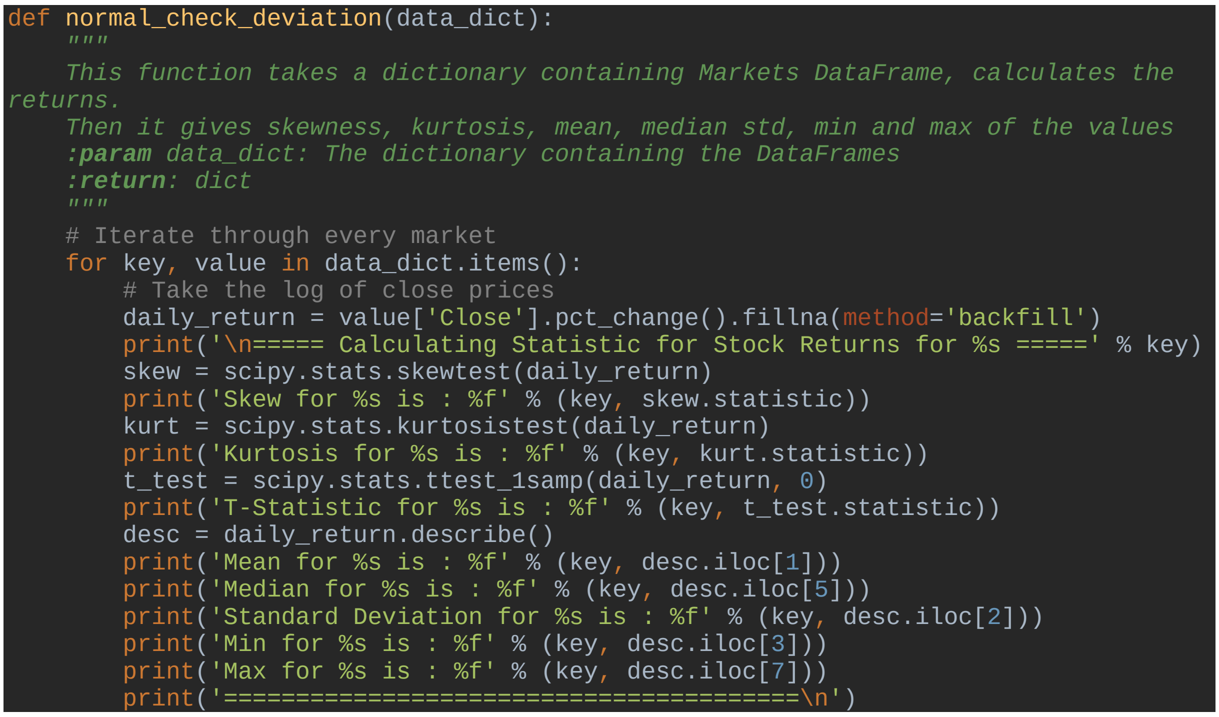


```
===== Calculating Statistic for Prices for DJI =====

Skew for DJI is : -22.608765

Kurtosis for DJI is : -3.379112

T-Statistic for DJI is : 1507.648461

Mean for DJI is : 9.144913

Median for DJI is : 9.256411

Standard Deviation for DJI is : 0.498273

Min for DJI is : 7.959912

Max for DJI is : 10.017378

========================================
```



```
===== Calculating Statistic for Stock Returns for DJI =====

Skew for DJI is : 1.270530

Kurtosis for DJI is : 33.707611

T-Statistic for DJI is : 2.778016

Mean for DJI is : 0.000351

Median for DJI is : 0.000487

Standard Deviation for DJI is : 0.010384

Min for DJI is : -0.078733

Max for DJI is : 0.110803

========================================
```

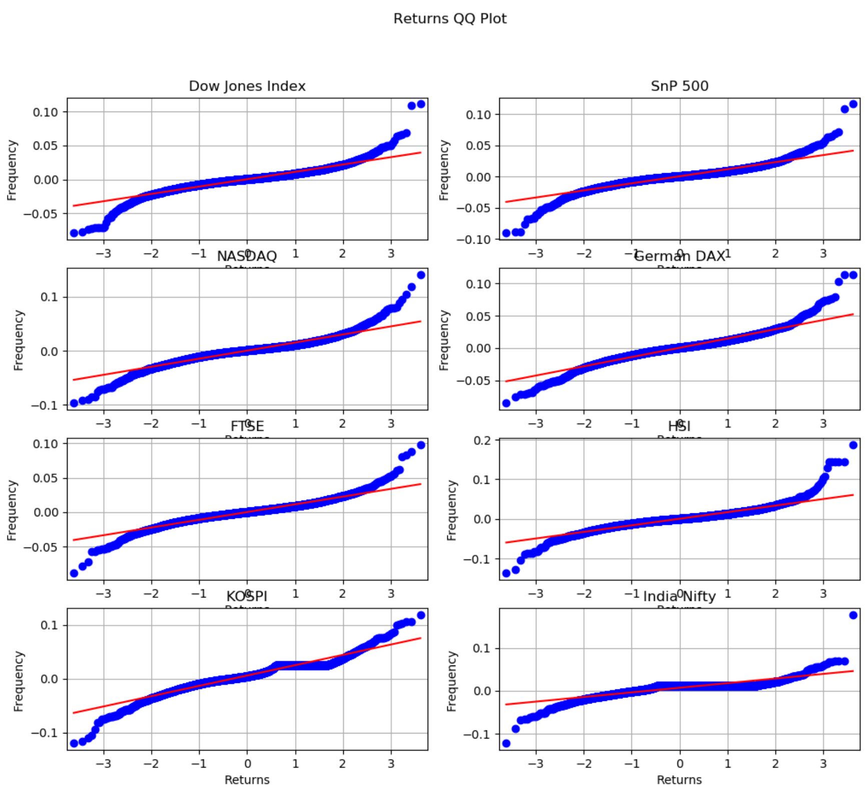
The code is as follows:

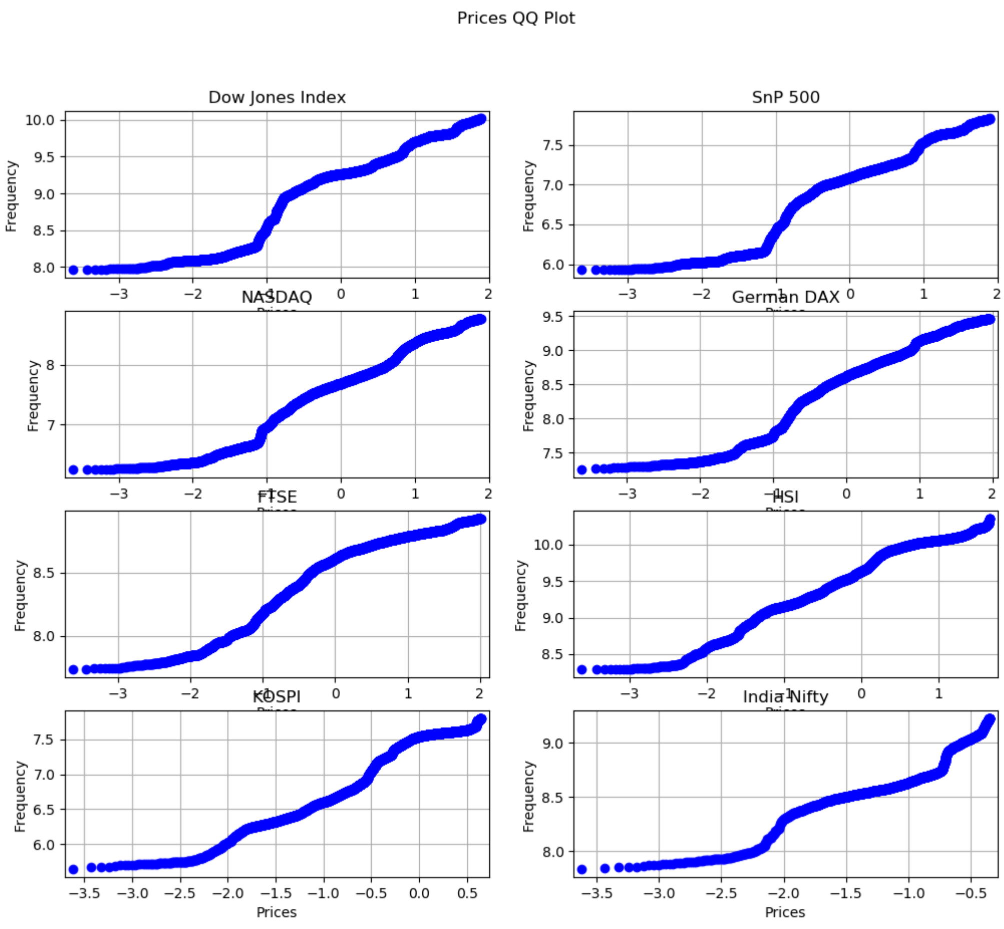


```
def normal_check_deviation(data_dict):
    """
    This function takes a dictionary containing Markets DataFrame, calculates the
returns.
    Then it gives skewness, kurtosis, mean, median std, min and max of the values
    :param data_dict: The dictionary containing the DataFrames
    :return: dict
    """
    # Iterate through every market
    for key, value in data_dict.items():
        # Take the log of close prices
        daily_return = value['Close'].pct_change().fillna(method='backfill')
        print('\n===== Calculating Statistic for Stock Returns for %s =====' % key)
        skew = scipy.stats.skewtest(daily_return)
        print('Skew for %s is : %f' % (key, skew.statistic))
        kurt = scipy.stats.kurtosistest(daily_return)
        print('Kurtosis for %s is : %f' % (key, kurt.statistic))
        t_test = scipy.stats.ttest_1samp(daily_return, 0)
        print('T-Statistic for %s is : %f' % (key, t_test.statistic))
        desc = daily_return.describe()
        print('Mean for %s is : %f' % (key, desc.iloc[1]))
        print('Median for %s is : %f' % (key, desc.iloc[5]))
        print('Standard Deviation for %s is : %f' % (key, desc.iloc[2]))
        print('Min for %s is : %f' % (key, desc.iloc[3]))
        print('Max for %s is : %f' % (key, desc.iloc[7]))
        print('========================================\n')
```

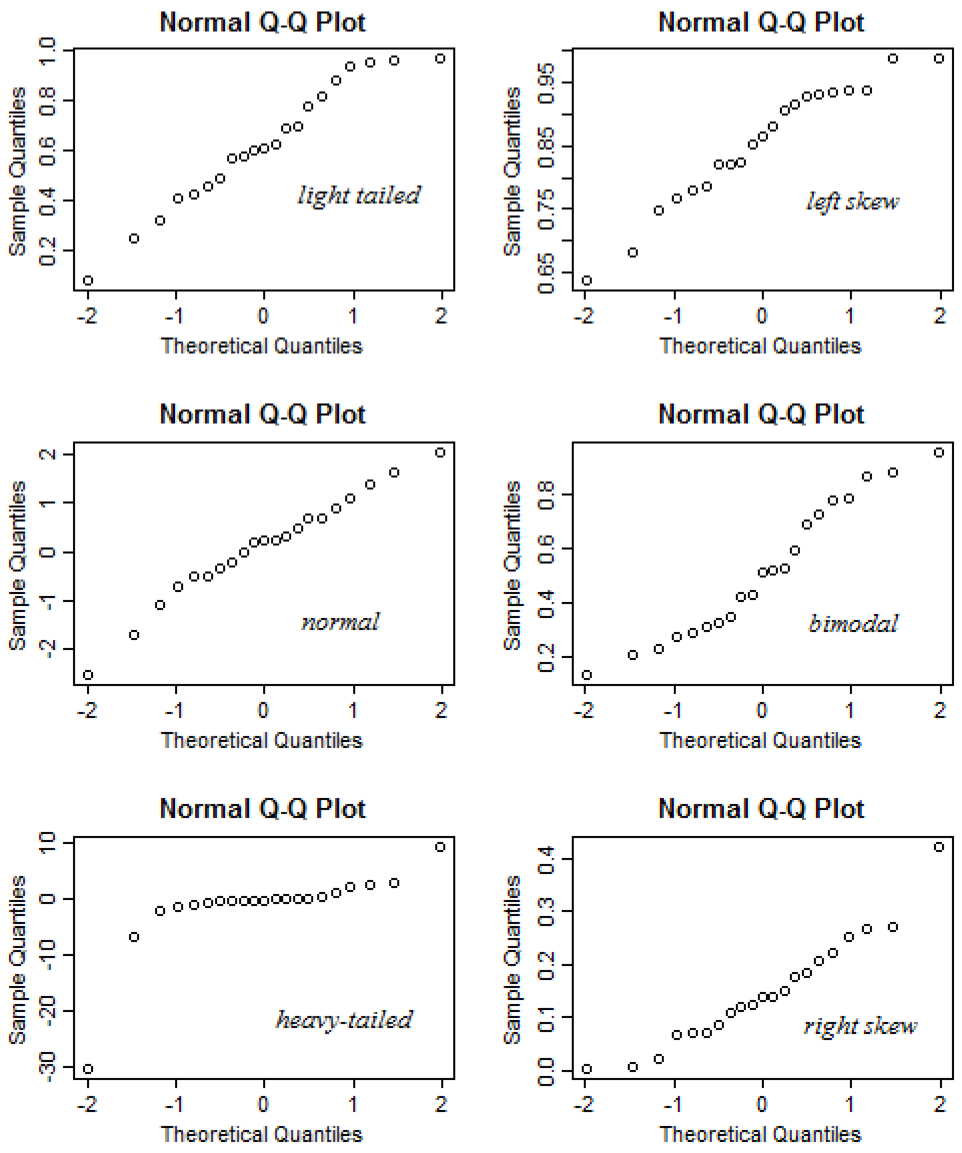
For the statistical measure, I used QQ plots. We could have used a histogram plot with an np.linspace() overlay, as shown in the book ‘Python For Finance’ by Yves which states “Comparing a frequency distribution (histogram) with a theoretical pdf is not the only way to graphically “test” for normality. So-called quantile-quantile plots (qq plots) are also well suited for this task”. Hence, I opted for QQ plots. However, before plotting QQ plots, I calculated the returns and plotted them, showcased along with a line. For the prices, I have to show the statistical measure that shows a QQ plot with respect to Log Normal (since we were assuming that prices follow log normal distribution). For this, I take log of the prices, and then use a QQ plot against Normal Distribution.

Observe the following figures given below for returns and prices respectively:


Returns QQ Plot
Dow Jones Index
SnP 500
NASDAQ
German DAX
FTSE
HSI
KOSPI
India Nifty
Frequency
Returns


Prices QQ Plot
Dow Jones Index
SnP 500
NASDAQ
German DAX
FTSE
HSI
KOSPI
India Nifty
Frequency
Prices

As one can observe, the plots are not totally on the Red line, neither do they follow a 45 degree angle, hence they cannot be normal. They seem to have a tail (more on that later) and from the following diagram below, you are certain that they cannot be normally distributed (or log of the prices is not normally distributed, there prices are not log normally distributed)



5. ***One of the most notable hypothesis about stock market behaviour is the "Efficient market hypothesis" which also internally assume that market price follows a random-walk process. Assuming that Stock Index prices follow a geometric Brownian motion and hence index returns were normally distributed with about 20% historical volatility, write a program sub-module to calculate the probability of an event like the 1987 stock market crash happening? Explain in simple terms what the results imply.***

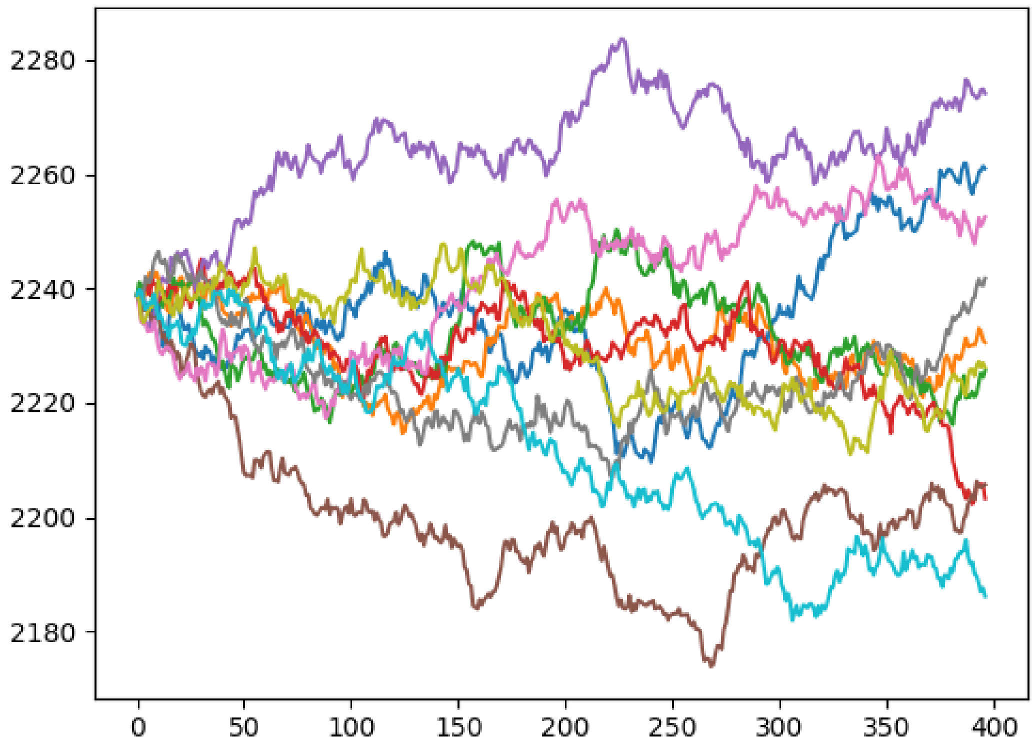
We use a Geometric Brownian Motion (GBM) because in the Standard Brownian Motion (SBM) stock prices are simulated considering only the normal distribution and time, and it does not take into account the expected return as well as the volatility of the stock. First, we need to generate data using GBM. For this, we take only 1 stock market index, that is the

SNP500, and run it over 10 simulations. The reason this is done so that the program executes faster. This acts more of a 'Proof of Concept' and hence can be replicated for other market indices as well. This is purposefully avoided because we do not want to spend a lot of time running the program.

I start at the final day of the data to be starting point this assumes that when running GBM it will give daily next year forecast, and two most important implication for the unalignment of each line will be the volatility and random (random walk) represent in the code.

At first, I pass in a single SnP500 year. I calculate the yearly return and use it as the expected return. The volatility happens to be 20%, as stated in the question, which is my sigma, that is, sigma = 0.20, I then simulate over 400 steps, where I do a T/252.0 in order to get intraday values. After the simulation has run, I store the values in a variable. I then calculate the daily returns using pct_change() for the simulated values, that is, around 397 steps. None of the steps give a return of less than -0.203 which was the largest single day crash in 1987, henceforth the probability of such a crash occurring is 0.

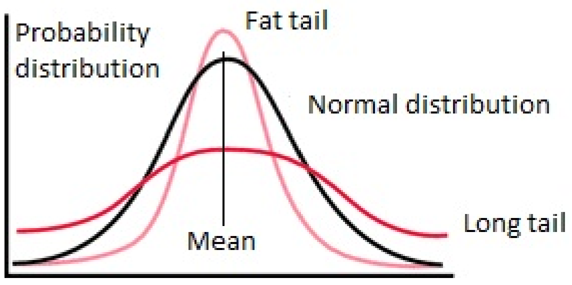
If I compare each step with the initial price, I do come to know that in some simulations, over the 400 odd steps of simulation, the return falls to around 50%. Henceforth, I decided to take the percentage change between each step rather than comparing it with the seed value.



```python
def market_crash_probability(snp_data):
    """
    This function calculates the probability of a stock market crash using GBM
    :param sp_data: the snp500 close price data
    :return: None
    """
    # Calculate Yearly Returns
    snp_yearly = snp_data['Close'].pct_change(259).dropna()[0]
    # start at the final day of the data to be starting point this assumes that
when running GBM it will give daily next year forecast
    snp0 = math.log(snp_data['Close'][-1])
    m = snp_yearly  # Expected Return
    sigma = 0.20  # Anuallized Volatility
    T = 1 / 252.0  # Maturity Date in years
    dt = .00001
    N = 10
    steps = round(T / dt)  # number of steps
    snp = np.zeros([int(N), int(steps)], dtype=float)
    x = range(0, int(steps), 1)
    storage = []  # For storing simulated values, to calc. probability
    for j in range(0, N, 1):
        snp[j, 0] = snp0
        for i in x[:-1]:
            snp[j, i + 1] = snp[j, i] + (m - 0.5 * pow(sigma, 2)) * dt + sigma *
scipy.sqrt(
                dt) * np.random.standard_normal()
        storage.append(np.exp(snp)[j])
        plt.plot(x, np.exp(snp)[j])
    plt.xlabel('Steps')
    plt.ylabel('SnP500 Prices')
    plt.show()

    # For calculating probability
    # Get the returns for each day from simulated set, minus
    storage = pd.DataFrame(storage)
    storage_returns = storage.T.pct_change()
    # Where returns are less than a 20.3% drop, which was a single day crash in
19887
    prob = storage_returns < -0.203
    probability_of_crash = prob.sum()
    for index, row in probability_of_crash.iteritems():
        print('The probability of a 1987 crash of SnP500, where a single intraday
crash was 20.3%, '
              'using GBM with the ' + str(index) + ' simulation is: ' + str(row))
```

6. ***What does "fat tail" mean? Plot the distribution of price movements for the downloaded indices (in separate subplot panes of a graph) and identify fat tail locations if any.***

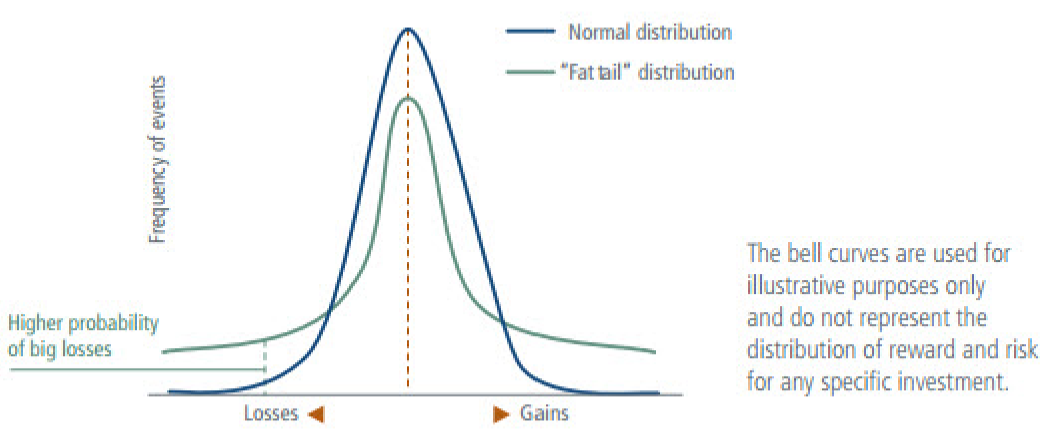
   By definition, a fat tail is a probability distribution which predicts movements of three or more standard deviations more frequently than a normal distribution, that is, fat tails are a statistical phenomenon exhibiting large leptokurtosis (*The condition of a probability density curve to have fatter tails and a higher peak at the mean than the normal distribution).*



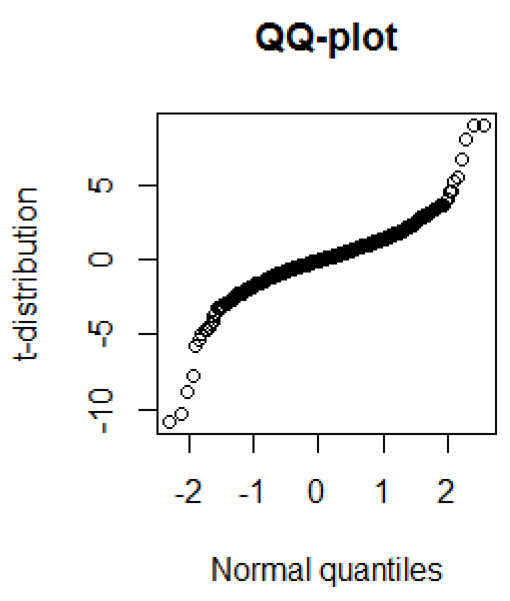
Which means that if plot a distribution of the values, we will get a fat tail v/s normal distribution picturized as follows:

Figure 1: Market tails may be "fatter" than normal

Tail events are very rare in a normal curve, but market tails are in fact "fatter," or more frequent, than many people realize.

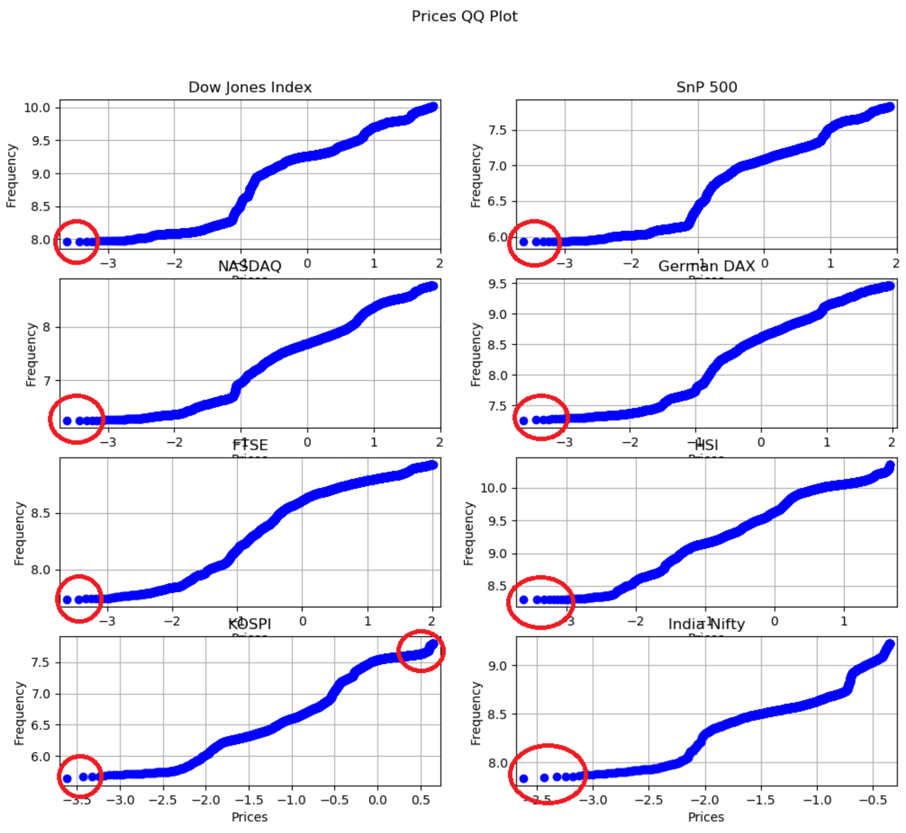


We can also observe Fat Tailed distribution using QQ plots, rather than histograms. For example:



In the given plot, the 'tails' deviate from a straight 45 degree line. Therefore, if we were to image the distribution line passing through the centre (That is, through the thick dots), the hollow ones depict the 'fat tails' of the distribution. This way, we can use QQ plots to identify 'fat tails'. I could not reduce the alpha programmatically (had I used PP plots then statsmodel's API gives you the ability to do so). Hence, I have manually gone and circled the

Fat Tail locations for price movements.



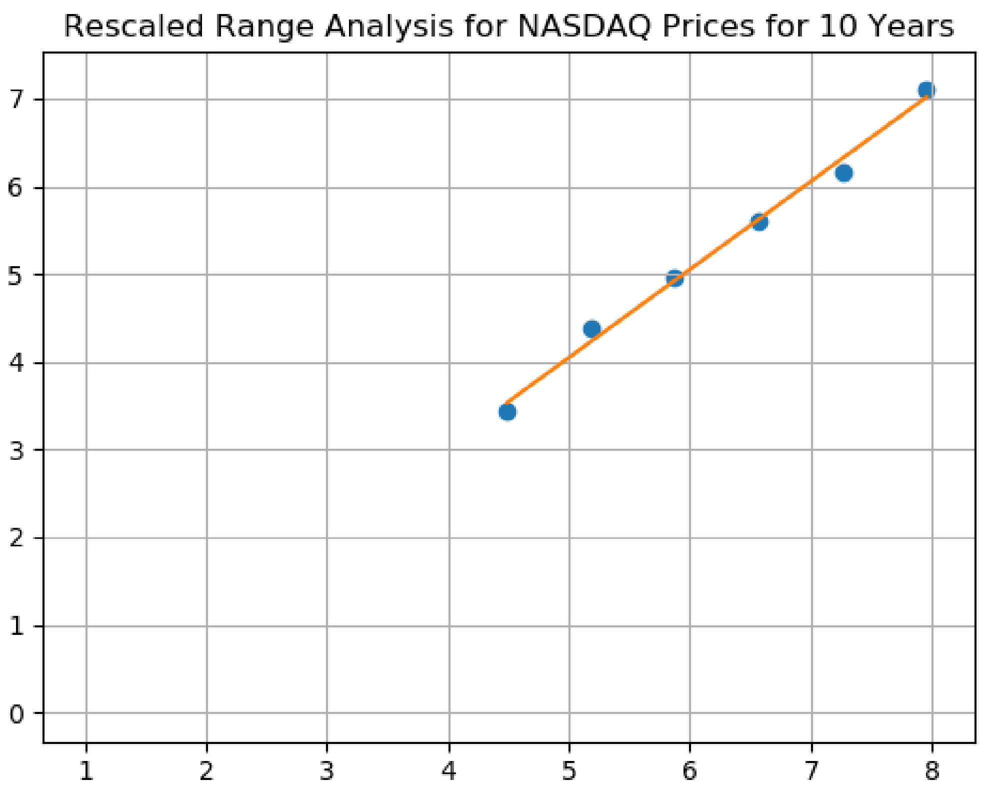
7. ***It is often claimed that fractals and multi-fractals generate a more realistic picture of market risks than log-normal distribution. Considering last 10 year daily price movements of NASDAQ, write a program to check whether fractal geometrics could have better predicted stock market movements than log-normal distribution assumption. Explain your findings with suitable graphs.***

The Hurst Exponent is used for checking if the time series are predictable or not. According to our findings, we had made an assumption that prices are log-normally distributed. However, this was proven false by performing the steps 1-4 in the Final Project. This means that if we get a trend behaviour or a mean reverting behaviour from our Hurst Exponent, we can say that multi-fractals generate a more realistic picture of market risk than log-normal distribution, and hence, could have better predicted stock market movements.

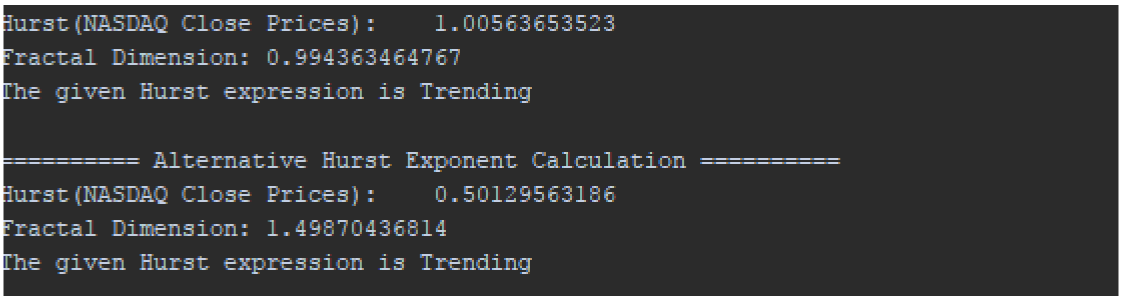
The hurst component was calculated using the 'hurst' function, which is as follows:

```
def hurst(size, nasdaq_close_price):
    nasdaq_close_price = nasdaq_close_price[0:size]  # NASDAQ prices for n period
    yn = nasdaq_close_price - np.mean(nasdaq_close_price)  # Calculation of mean
adjusted series for this period
    zn = np.cumsum(yn)  # Calculation of the cumulative sum
    Rn = np.max(zn) - np.min(zn)  # Calculation of range
    Sn = np.std(nasdaq_close_price)  # Calculation of std deviation
    En = Rn / Sn  # Calculation of the rescaled range
    return np.log(En)
```

With the help of Hurst component, we plot a rescaled range analysis for our “Prices Movements” (where we use close prices, because the above arguments and assumptions made in 1-4 still hold true).



```
xx = sm.add_constant(x)
model = sm.OLS(y, xx)
results = model.fit()
print(results.summary())
yn = generateYn(xx, results)
plt.plot(xx, np.reshape(yn, (6, 2)), '-')
plt.scatter(x, y)
plt.grid(True)
plt.title("Rescaled Range Analysis for NASDAQ Prices for 10 Years")
plt.show()

# Get the Hurst Exponent of the Series:
nasdaq_hurst = results.params[1]
print("Hurst(NASDAQ Close Prices):    %s" % str(nasdaq_hurst))
fractal_dim = 2 - nasdaq_hurst
print("Fractal Dimension: %s" % str(fractal_dim))
if nasdaq_hurst > 0.5 and fractal_dim < 1.5:
    print('The given Hurst expression is Trending')
elif nasdaq_hurst < 0.5 and fractal_dim > 1.5:
    print('The given Hurst expression is anti-persisting, that is, it is mean 
reverting')
```



```
Hurst(NASDAQ Close Prices):    1.00563653523
Fractal Dimension: 0.994363464767
The given Hurst expression is Trending

========== Alternative Hurst Exponent Calculation ==========
Hurst(NASDAQ Close Prices):    0.50129563186
Fractal Dimension: 1.49870436814
The given Hurst expression is Trending
```

Therefore, from the output, we can say that our time series is predictable, because it has a trending behaviour. Since we have established the fact that the prices are trending, we can use this for predicting stock market returns rather than log-normal distribution, because as we have observed, prices do not follow log-normal distribution, and hence, would be bad for prediction.

There are several key takeaways:

The Hurst Exponent is used for checking if the time series are predictable.

Fractal Dimension = 2 – Hurst exponent

Hurst Exponent takes values between 0 and 1.

There are three possible situations:

Fractal Dimension of 1.50 (Hurst = 0.50) – indicates a random walk process, there is no long memory and the time series is hard to predict its future evolution

Fractal Dimension > 1.50 (Hurst<0.50) – anti-persistent behavior.

Fractal Dimension <1.50 hurst="">0.50) – Trend behavior.

Therefore, we don't need to explicitly plot a fractal image and perform analysis on it. A Hurst exponent and fractal dimensionality can help use determine whether the series is trending or not.

The goal of the Hurst Exponent is to provide us with a scalar value that will help us to identify (within the limits of statistical estimation) whether a series is mean reverting, random walking or trending.

The idea behind the Hurst Exponent calculation is that we can use the variance of a log price series to assess the rate of diffusive behaviour

I have also presented an alternative Hurst Exponent calculation based on the formulas present at (https://www.quantstart.com/articles/Basics-of-Statistical-Mean-Reversion-Testing)

```python
print('\n========== Alternative Hurst Exponent Calculation ==========')
nasdaq_hurst = hurst_ts(np.log(nasdaq_close_price))
print("Hurst(NASDAQ Close Prices):    %s" % str(nasdaq_hurst))
fractal_dim = 2 - nasdaq_hurst
print("Fractal Dimension: %s" % str(fractal_dim))

if nasdaq_hurst > 0.5 and fractal_dim < 1.5:
    print('The given Hurst expression is Trending')
elif nasdaq_hurst < 0.5 and fractal_dim > 1.5:
    print('The given Hurst expression is anti-persisting, that is, it is mean 
reverting')
```

A key note to remember about fractals is:

1. **Fractals are lagging indicators. Lagging indicators usually change after the market registers a particular trend. Therefore they are used for confirming trends.**