Question 1

The most important part for this question is the formula of 3 classical way to calculate the price.

1. Classical Brownian Motion

$$P_t = P_{t-1} + r_t$$

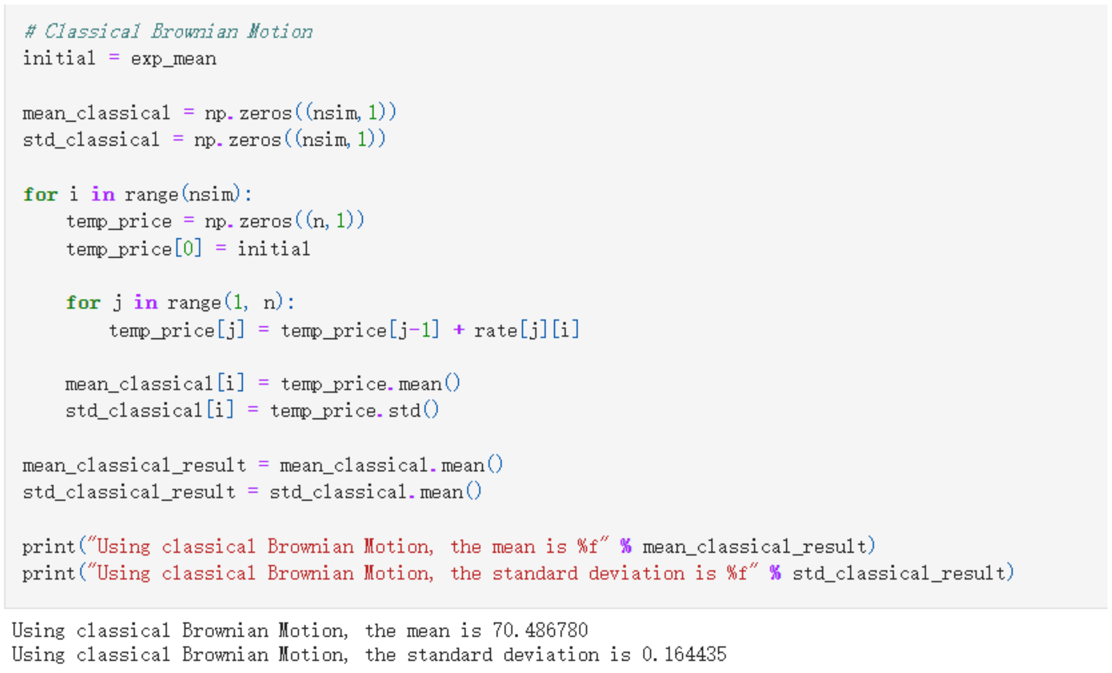
2. Arithmetic Return System

$$P_t = P_{t-1}\left(1 + r_t\right)$$

3. Log Return or Geometric Brownian Motion

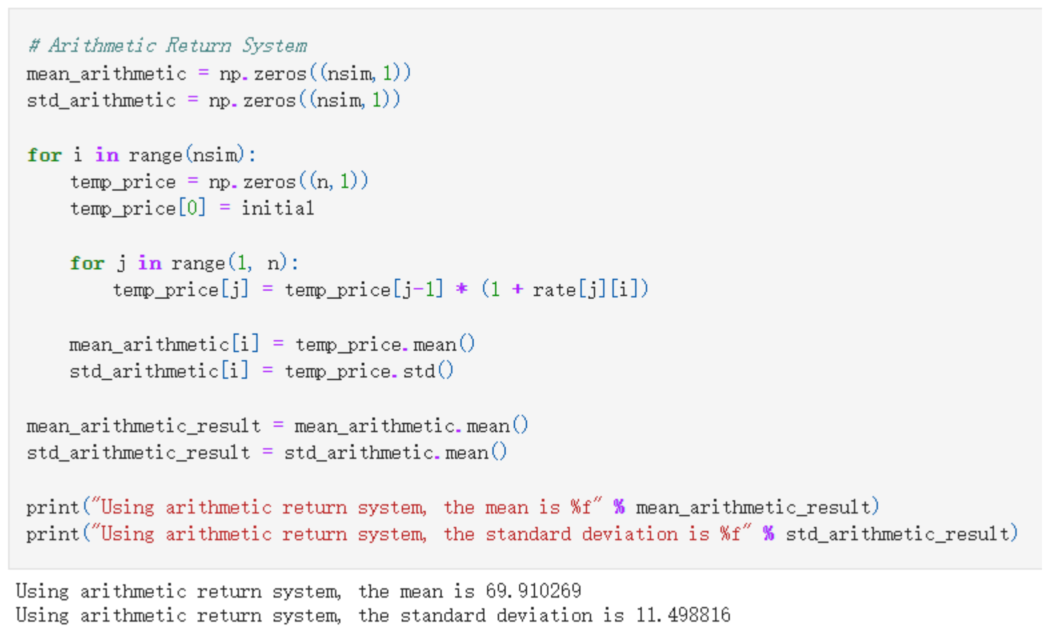
$$P_t = P_{t-1} e^{r_t}$$

For each of formula, I write a function to calculate. Here are my functions.



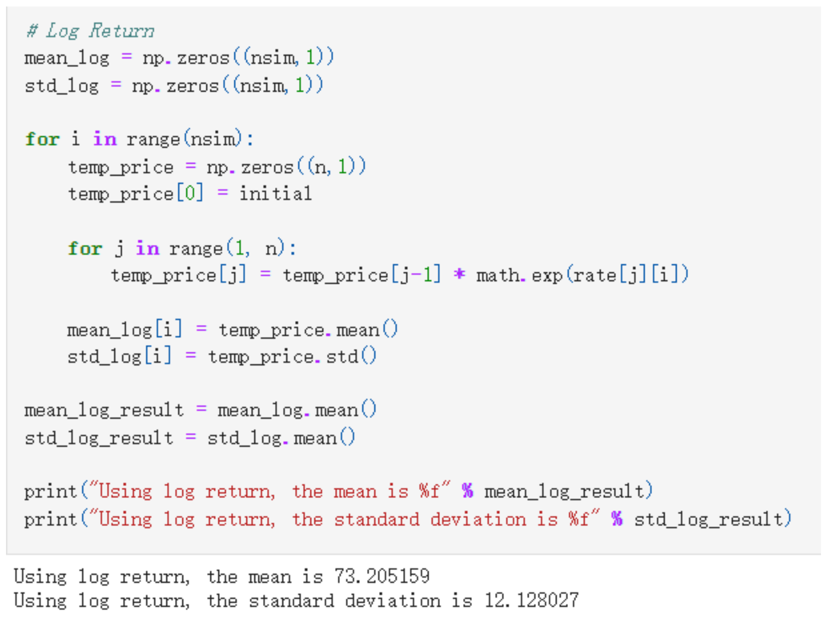
```
# Classical Brownian Motion
initial = exp_mean

mean_classical = np.zeros((nsim, 1))
std_classical = np.zeros((nsim, 1))

for i in range(nsim):
    temp_price = np.zeros((n, 1))
    temp_price[0] = initial

    for j in range(1, n):
        temp_price[j] = temp_price[j-1] + rate[j][i]

    mean_classical[i] = temp_price.mean()
    std_classical[i] = temp_price.std()

mean_classical_result = mean_classical.mean()
std_classical_result = std_classical.mean()

print("Using classical Brownian Motion, the mean is %f" % mean_classical_result)
print("Using classical Brownian Motion, the standard deviation is %f" % std_classical_result)
```

```
Using classical Brownian Motion, the mean is 70.486780
Using classical Brownian Motion, the standard deviation is 0.164435
```



```python
# Arithmetic Return System
mean_arithmetic = np.zeros((nsim, 1))
std_arithmetic = np.zeros((nsim, 1))

for i in range(nsim):
    temp_price = np.zeros((n, 1))
    temp_price[0] = initial

    for j in range(1, n):
        temp_price[j] = temp_price[j-1] * (1 + rate[j][i])

    mean_arithmetic[i] = temp_price.mean()
    std_arithmetic[i] = temp_price.std()

mean_arithmetic_result = mean_arithmetic.mean()
std_arithmetic_result = std_arithmetic.mean()

print("Using arithmetic return system, the mean is %f" % mean_arithmetic_result)
print("Using arithmetic return system, the standard deviation is %f" % std_arithmetic_result)
```

```
Using arithmetic return system, the mean is 69.910269
Using arithmetic return system, the standard deviation is 11.498816
```

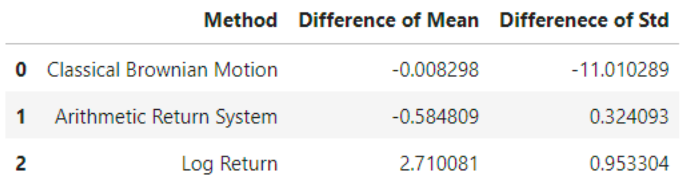


```python
# Log Return
mean_log = np.zeros((nsim, 1))
std_log = np.zeros((nsim, 1))

for i in range(nsim):
    temp_price = np.zeros((n, 1))
    temp_price[0] = initial

    for j in range(1, n):
        temp_price[j] = temp_price[j-1] * math.exp(rate[j][i])

    mean_log[i] = temp_price.mean()
    std_log[i] = temp_price.std()

mean_log_result = mean_log.mean()
std_log_result = std_log.mean()

print("Using log return, the mean is %f" % mean_log_result)
print("Using log return, the standard deviation is %f" % std_log_result)
```

```
Using log return, the mean is 73.205159
Using log return, the standard deviation is 12.128027
```

By calculating the mean and standard deviation of AdjClose price, I get the expectation of both value. The expected mean of AdjClose price is 70.495078. The expected standard deviation of AdjClose price is 11.174723.

Then, I compare mean and standard deviation from those three different method with expectation method to see the performance.

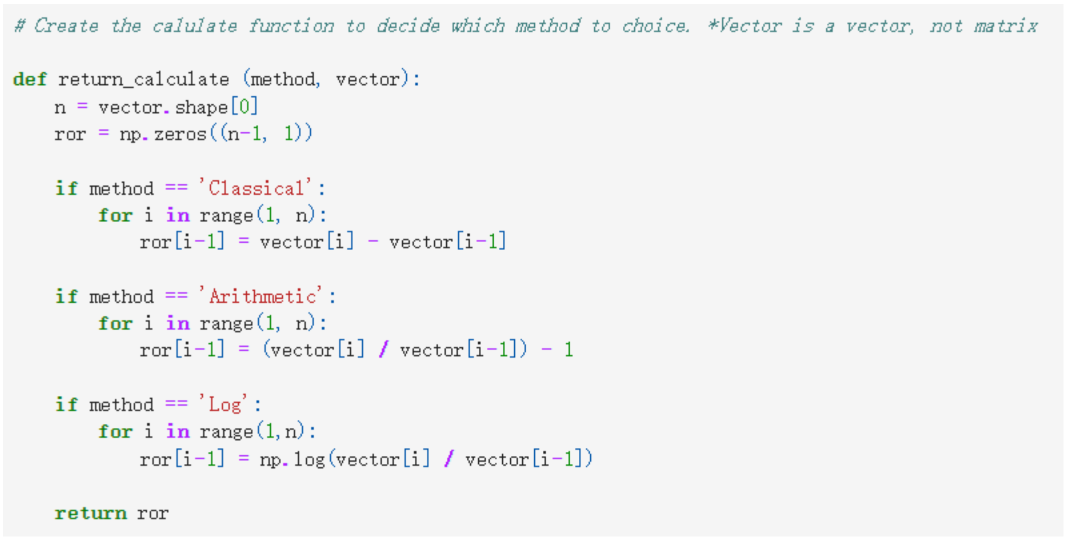


	Method	Difference of Mean	Differenece of Std
0	Classical Brownian Motion	-0.008298	-11.010289
1	Arithmetic Return System	-0.584809	0.324093
2	Log Return	2.710081	0.953304

From this table, I can see that using classical Brownian motion method, we get the most similar mean with expectation. Arithmetic return system gives the best performance on standard deviation. Also, classical Brownian motion gives a very bad result on standard deviation.

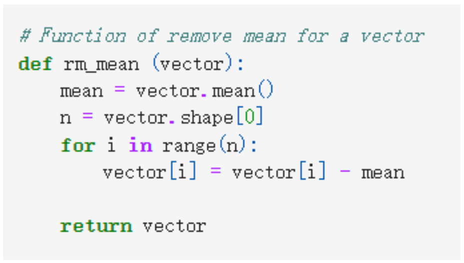
Question 2

Following the instruction, I create a return_calculate() function to give the option for user to choose which method they want to use to calculate the rate of return. We can get the rate of return formula by transferring the above price formulas.



```python
# Create the calulate function to decide which method to choice. *Vector is a vector, not matrix

def return_calculate (method, vector):
    n = vector.shape[0]
    ror = np.zeros((n-1, 1))

    if method == 'Classical':
        for i in range(1, n):
            ror[i-1] = vector[i] - vector[i-1]

    if method == 'Arithmetic':
        for i in range(1, n):
            ror[i-1] = (vector[i] / vector[i-1]) - 1

    if method == 'Log':
        for i in range(1,n):
            ror[i-1] = np.log(vector[i] / vector[i-1])

    return ror
```

Then, I remove the mean for stock INTC to see the rate of return of INTC by using arithmetic method.

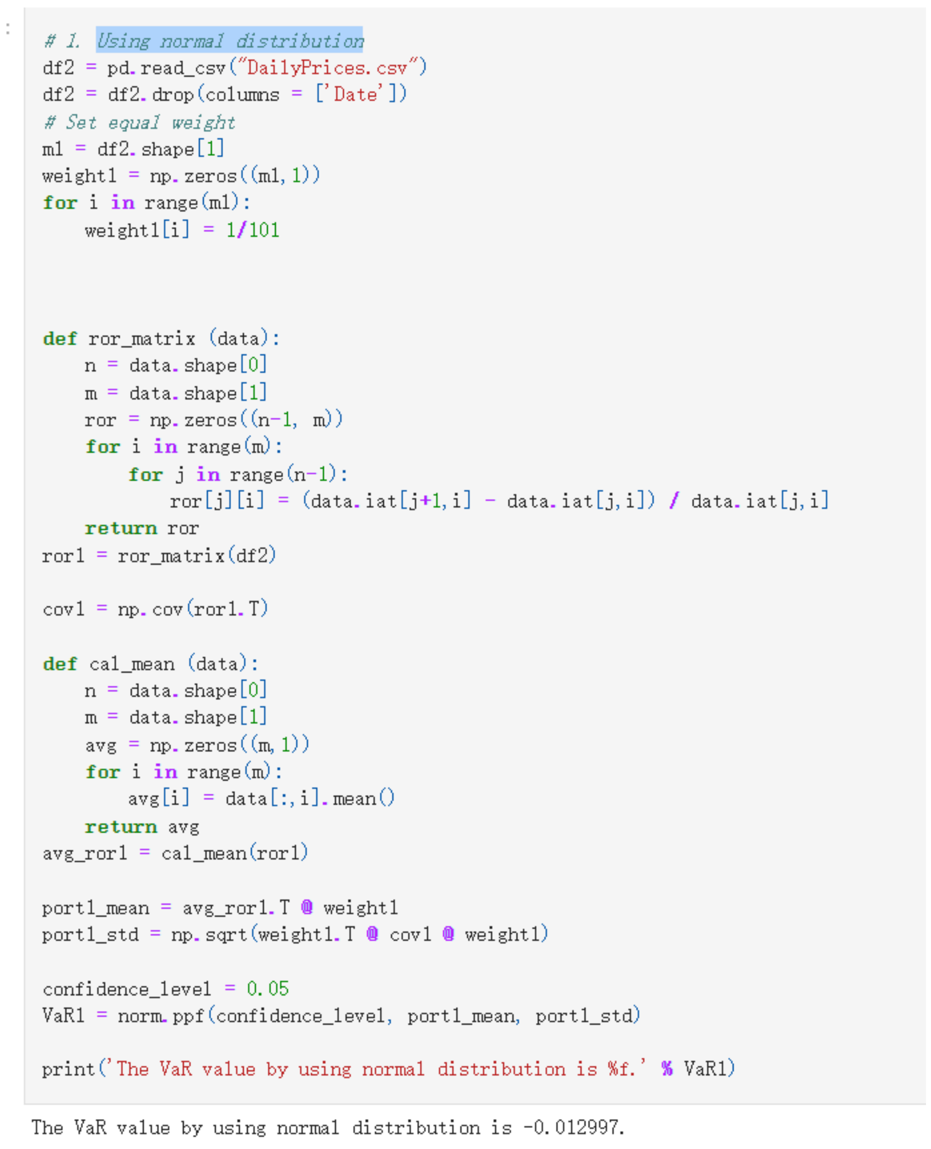


```python
# Function of remove mean for a vector
def rm_mean (vector):
    mean = vector.mean()
    n = vector.shape[0]
    for i in range(n):
        vector[i] = vector[i] - mean

    return vector
```

Then, I calculate VaR by four different methods. For all those four methods, I calculate the portfolio VaR rather than one specific stock's VaR. In my portfolio, I involve all 101 stocks. The reason why I use the whole portfolio rather than one stock is because that I think that it is more reasonable. In the real world, I believe that most people will buy many different stocks in the portfolio rather than one stock. Thus, using the VaR of the whole portfolio will give me strong information of the market.

1. Using normal distribution

   In question 3, I used this method to illustrate. I will explain how I create this function in question 3.

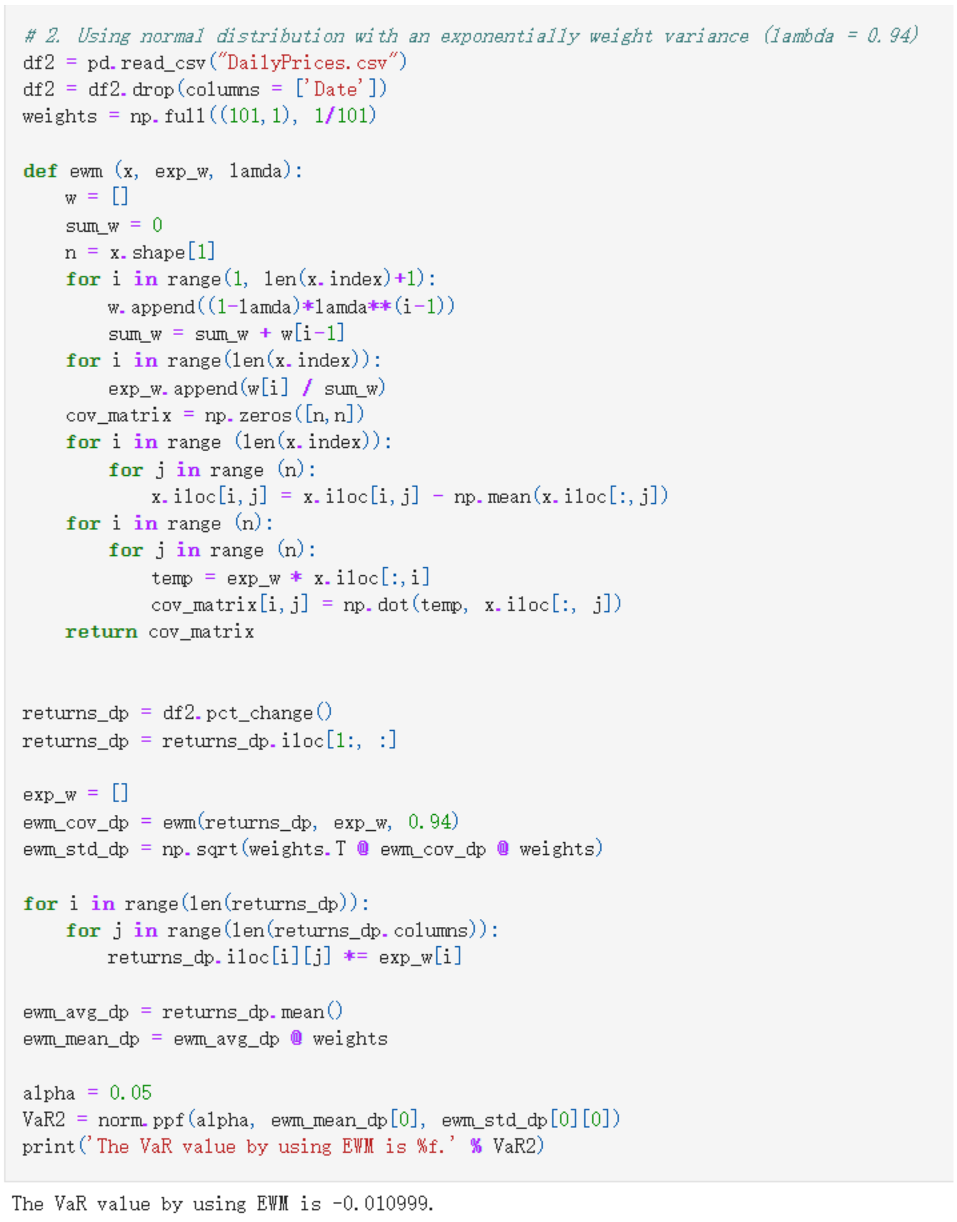


```python
# 1. Using normal distribution
df2 = pd.read_csv("DailyPrices.csv")
df2 = df2.drop(columns = ['Date'])
# Set equal weight
m1 = df2.shape[1]
weight1 = np.zeros((m1,1))
for i in range(m1):
    weight1[i] = 1/101



def ror_matrix (data):
    n = data.shape[0]
    m = data.shape[1]
    ror = np.zeros((n-1, m))
    for i in range(m):
        for j in range(n-1):
            ror[j][i] = (data.iat[j+1,i] - data.iat[j,i]) / data.iat[j,i]
    return ror
ror1 = ror_matrix(df2)

cov1 = np.cov(ror1.T)

def cal_mean (data):
    n = data.shape[0]
    m = data.shape[1]
    avg = np.zeros((m,1))
    for i in range(m):
        avg[i] = data[:,i].mean()
    return avg
avg_ror1 = cal_mean(ror1)

port1_mean = avg_ror1.T @ weight1
port1_std = np.sqrt(weight1.T @ cov1 @ weight1)

confidence_level = 0.05
VaR1 = norm.ppf(confidence_level, port1_mean, port1_std)

print('The VaR value by using normal distribution is %f.' % VaR1)
```

```
The VaR value by using normal distribution is -0.012997.
```

2. Using normal distribution with an exponentially weight variance (lambda = 0.94)

   This function is edited from last homework function. I get VaR is equal to -0.010999

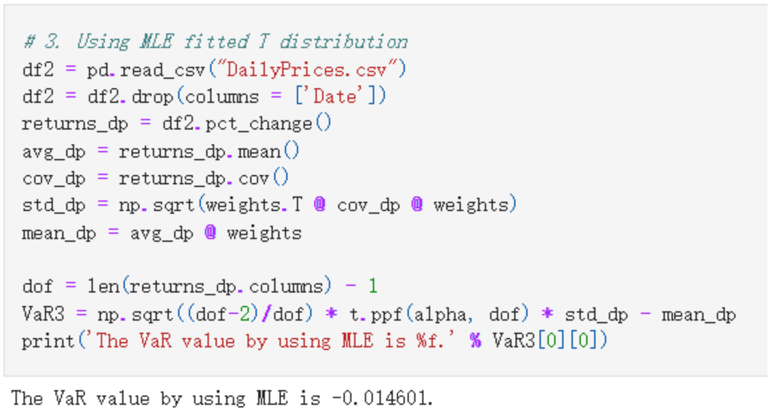


```
# 2. Using normal distribution with an exponentially weight variance (lambda = 0.94)
df2 = pd.read_csv("DailyPrices.csv")
df2 = df2.drop(columns = ['Date'])
weights = np.full((101,1), 1/101)

def ewm (x, exp_w, lamda):
    w = []
    sum_w = 0
    n = x.shape[1]
    for i in range(1, len(x.index)+1):
        w.append((1-lamda)*lamda**(i-1))
        sum_w = sum_w + w[i-1]
    for i in range(len(x.index)):
        exp_w.append(w[i] / sum_w)
    cov_matrix = np.zeros([n,n])
    for i in range (len(x.index)):
        for j in range (n):
            x.iloc[i,j] = x.iloc[i,j] - np.mean(x.iloc[:,j])
    for i in range (n):
        for j in range (n):
            temp = exp_w * x.iloc[:,i]
            cov_matrix[i,j] = np.dot(temp, x.iloc[:, j])
    return cov_matrix


returns_dp = df2.pct_change()
returns_dp = returns_dp.iloc[1:, :]

exp_w = []
ewm_cov_dp = ewm(returns_dp, exp_w, 0.94)
ewm_std_dp = np.sqrt(weights.T @ ewm_cov_dp @ weights)

for i in range(len(returns_dp)):
    for j in range(len(returns_dp.columns)):
        returns_dp.iloc[i][j] *= exp_w[i]

ewm_avg_dp = returns_dp.mean()
ewm_mean_dp = ewm_avg_dp @ weights

alpha = 0.05
VaR2 = norm.ppf(alpha, ewm_mean_dp[0], ewm_std_dp[0][0])
print('The VaR value by using EWM is %f.' % VaR2)
```

```
The VaR value by using EWM is -0.010999.
```

3. Using MLE fitted T distribution

   This function is inspired from the week 2 assignment. I get VaR value equal to -0.014601 from this question.

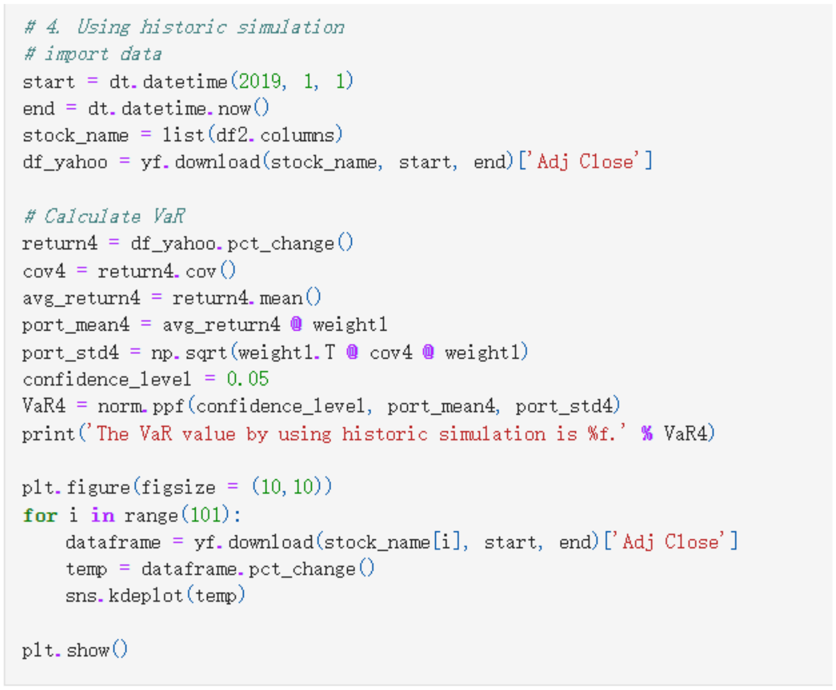


```
# 3. Using MLE fitted T distribution
df2 = pd.read_csv("DailyPrices.csv")
df2 = df2.drop(columns = ['Date'])
returns_dp = df2.pct_change()
avg_dp = returns_dp.mean()
cov_dp = returns_dp.cov()
std_dp = np.sqrt(weights.T @ cov_dp @ weights)
mean_dp = avg_dp @ weights

dof = len(returns_dp.columns) - 1
VaR3 = np.sqrt((dof-2)/dof) * t.ppf(alpha, dof) * std_dp - mean_dp
print('The VaR value by using MLE is %f.' % VaR3[0][0])
```

```
The VaR value by using MLE is -0.014601.
```

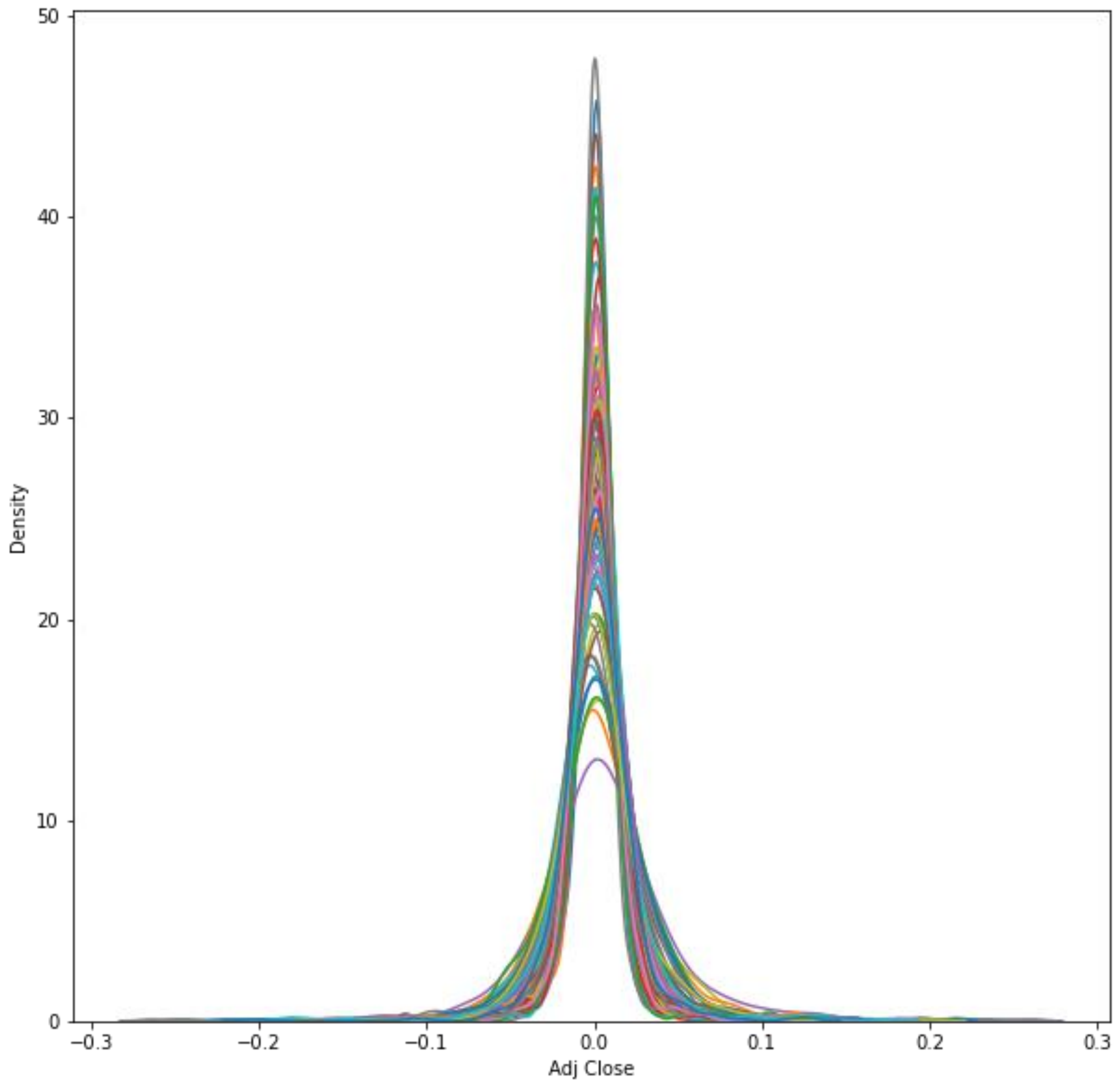
4. Using historic simulation

In this part, I read Yahoo Finance's data starting from 2019/1/1 to now. I still use the 101 stocks which in the the daily price file. From Yahoo Finance, I only read the adjust close price, since I only need to use this to calculate the VaR.

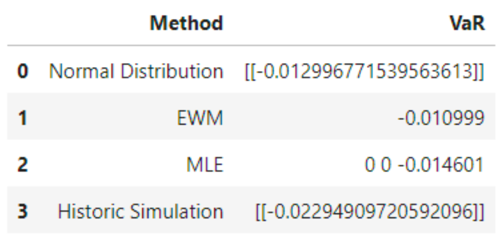


```
# 4. Using historic simulation
# import data
start = dt.datetime(2019, 1, 1)
end = dt.datetime.now()
stock_name = list(df2.columns)
df_yahoo = yf.download(stock_name, start, end)['Adj Close']

# Calculate VaR
return4 = df_yahoo.pct_change()
cov4 = return4.cov()
avg_return4 = return4.mean()
port_mean4 = avg_return4 @ weight1
port_std4 = np.sqrt(weight1.T @ cov4 @ weight1)
confidence_level = 0.05
VaR4 = norm.ppf(confidence_level, port_mean4, port_std4)
print('The VaR value by using historic simulation is %f.' % VaR4)

plt.figure(figsize = (10,10))
for i in range(101):
    dataframe = yf.download(stock_name[i], start, end)['Adj Close']
    temp = dataframe.pct_change()
    sns.kdeplot(temp)

plt.show()
```

Here is the KDE plot for each individual stock.



After using those four method to calculate the VaR, I create a table to clearly see the difference between each method.



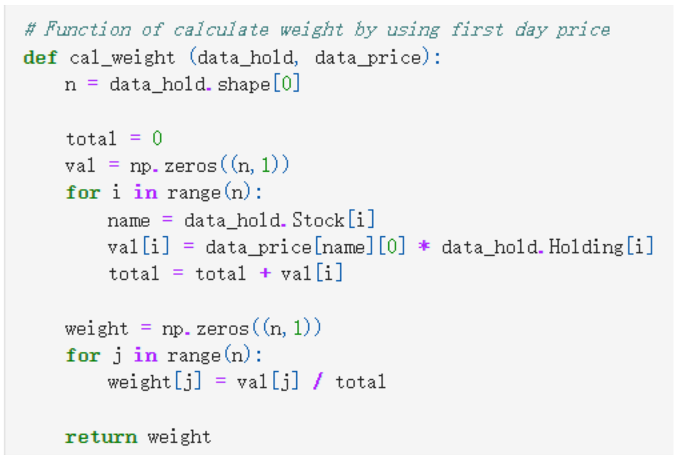
	Method	VaR
0	Normal Distribution	[[-0.012996771539563613]]
1	EWM	-0.010999
2	MLE	0 0 -0.014601
3	Historic Simulation	[[-0.02294909720592096]]

We can see that EWM method give the smallest VaR and historic simulation shows the largest VaR.

Question 3

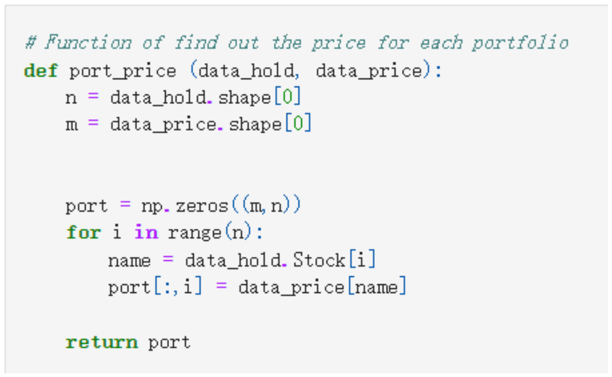
*In this question, I separate the portfolio file into three different files (portfolio_A, portfolio_B, portfolio_C) and load into Python. Please download those files before running the program.

I calculate the weight by using the first price times each stocks' holding number, then divides the total values of the whole portfolio.



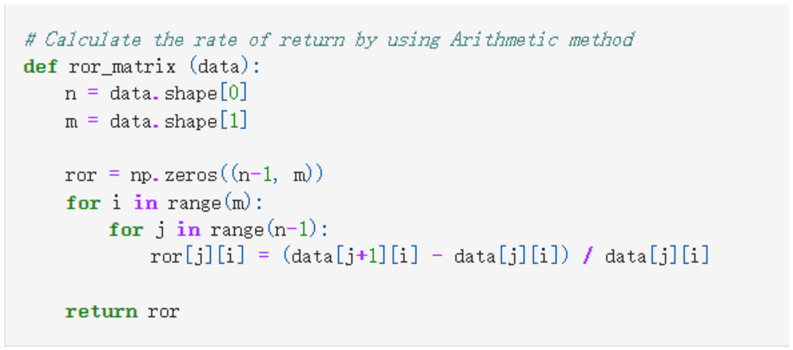
```python
# Function of calculate weight by using first day price
def cal_weight (data_hold, data_price):
    n = data_hold.shape[0]

    total = 0
    val = np.zeros((n,1))
    for i in range(n):
        name = data_hold.Stock[i]
        val[i] = data_price[name][0] * data_hold.Holding[i]
        total = total + val[i]

    weight = np.zeros((n,1))
    for j in range(n):
        weight[j] = val[j] / total

    return weight
```

Then, I need to find out the price for each stocks in different portfolio.



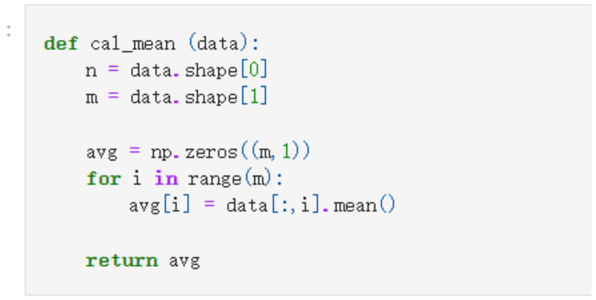
```python
# Function of find out the price for each portfolio
def port_price (data_hold, data_price):
    n = data_hold.shape[0]
    m = data_price.shape[0]


    port = np.zeros((m,n))
    for i in range(n):
        name = data_hold.Stock[i]
        port[:,i] = data_price[name]

    return port
```

Then, I calculate the rate of return for each portfolio.

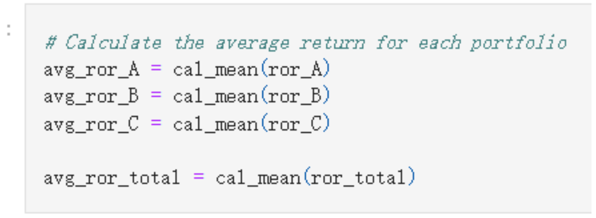


```python
# Calculate the rate of return by using Arithmetic method
def ror_matrix (data):
    n = data.shape[0]
    m = data.shape[1]

    ror = np.zeros((n-1, m))
    for i in range(m):
        for j in range(n-1):
            ror[j][i] = (data[j+1][i] - data[j][i]) / data[j][i]

    return ror
```

Then, I use covariance function to calculate the covariance matrix and calculate the mean of rate of return matrix.

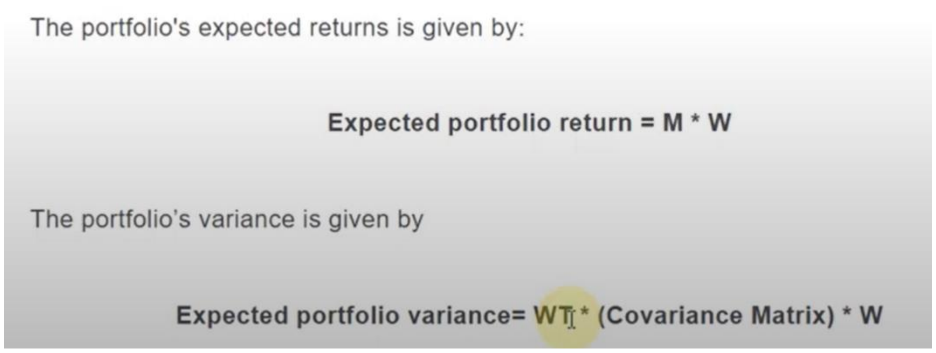


```
def cal_mean (data):
    n = data.shape[0]
    m = data.shape[1]

    avg = np.zeros((m, 1))
    for i in range(m):
        avg[i] = data[:, i].mean()

    return avg
```



```
# Calculate the average return for each portfolio
avg_ror_A = cal_mean(ror_A)
avg_ror_B = cal_mean(ror_B)
avg_ror_C = cal_mean(ror_C)

avg_ror_total = cal_mean(ror_total)
```

Then, I calculate the portfolio mean and standard deviation.



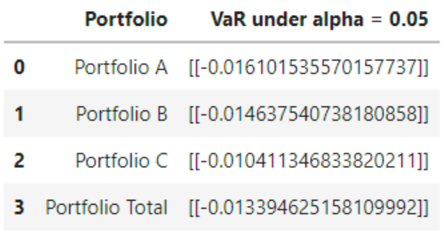
The portfolio's expected returns is given by:

Expected portfolio return = M * W

The portfolio's variance is given by

Expected portfolio variance= WT * (Covariance Matrix) * W

Finally, I create a table to see the VaR under different portfolio with alpha = 0.05.



	Portfolio	VaR under alpha = 0.05
0	Portfolio A	[[-0.016101535570157737]]
1	Portfolio B	[[-0.014637540738180858]]
2	Portfolio C	[[-0.010411346833820211]]
3	Portfolio Total	[[-0.013394625158109992]]

From the table, we can see that portfolio C's VaR is the smallest and portfolio A's VaR is the largest.