Problem 1

Calculate and compare the expected value and standard deviation of price at time t (P_t), given each of the 3 types of price returns, assuming $r_t \sim N(0, \sigma^2)$. Simulate each return equation using $r_t \sim N(0, \sigma^2)$ and show the mean and standard deviation match your expectations.

The expected value and standard deviation of price at time t assuming a normality of returns:

Classic Brownian Motion

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

Expected Value:

$$E(P_t) = E(P_{t-1} + r_t)$$
$$E(P_t) = E(P_{t-1}) + E(r_t)$$

as defined in the question, $r_t \sim N(0, \sigma^2)$,

so we have:

$$E(P_t) = E(P_{t-1})$$

taking P_{t-1} as a known number

$$\Rightarrow E(P_t) = P_{t-1}$$

Standard Deviation:

$$std(P_t) = std(P_{t-1} + r_t)$$

assume P_{t-1} is uncorrelated with r_t, then.

$$std(P_t) = std(P_{t-1}) + std(r_t)$$

taking P_{t-1} as a known number, also

as defined in the question, $r_t \sim N(0, \sigma^2)$,

so we have:

$$std(P_t) = \sigma$$

Arithmetic Return System

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

Expected Value:

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

$$E[P_t] = E\left[P_{t-1} + P_{t-1}\cdot r_t\right]$$

$$E[P_t] = E[P_{t-1}] + E\left[P_{t-1}\cdot r_t\right]$$

assume P_{t-1} is uncorrelated with r_t, then:

$$E[P_t] = E[P_{t-1}] + E\left[P_{t-1}\right]\cdot E[r_t]$$

$$E[P_t] = E[P_{t-1}]\cdot(1+E[r_t])$$

assume that P_{t-1} is a known number, and $r_t \sim N(0, \sigma^2)$, we have:

$$E[P_t] = P_{t-1}\cdot(1+0)^{+}$$

$$\Rightarrow E[P_t] = P_{t-1}$$

Standard Deviation:

$$E[P_t] = P_{t-1}$$

$$E[P_t]^2 = P_{t-1}^2$$

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

$$P_t^2 = P_{t-1}^2\cdot(1+r_t)^2$$

$$E[P_t^2] = E\left[P_{t-1}^2 + 2P_{t-1}^2\cdot r_t + P_{t-1}^2\cdot r_t^2\right]$$

assume P_{t-1} and r_t are independent, $r_t \sim N(0, \sigma^2)$

$$E[P_t^2] = E[P_{t-1}^2] + E[P_{t-1}^2]\cdot E[r_t^2]$$

$$E[r_t^2] = Var(r_t) + E[r_t]^2 = \sigma^2$$

$$E[P_t^2] = E[P_{t-1}^2]\cdot(1+\sigma^2)$$

assume that P_{t-1} is a known number

$$E[P_t^2] = P_{t-1}^2\cdot(1+\sigma^2)$$

$$\Rightarrow var(P_t) = E[P_t^2] - E[P_t]^2 = P_{t-1}^2\left((1+\sigma^2)-1\right)$$

$$std(P_t) = P_{t-1}\cdot\sigma$$

Geometric Brownian Motion

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

Expected Value:

$E[P_t] = E[P_{t-1} \cdot e^{r_t}]$

$E[P_t] = E[P_{t-1}] \cdot E[e^{r_t}]$

look at $E[e^{r_t}]$, since $r_t \sim N(0, \sigma^2)$,
e^{r_t} becomes a log-normal distribution, and
using log-normal distribution properties —
$E[X] = \exp(\mu + \frac{1}{2}\sigma^2)$ — we have:

$E[P_t] = E[P_{t-1}] \cdot e^{\frac{1}{2}\sigma^2}$

assume that P_{t-1} is a known number:

$\Rightarrow E[P_t] = P_{t-1} \cdot e^{\frac{1}{2} \cdot \sigma^2}$

Standard Deviation:

$E[P_t] = P_{t-1} e^{\frac{1}{2} \cdot \sigma^2}$

$E[P_t]^2 = P_{t-1}^2 \cdot e^{\sigma^2}$

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

$P_t^2 = P_{t-1}^2 \cdot e^{2r_t}$

P_{t-1} and r_t are independent

$E[P_t^2] = E[P_{t-1}^2] \cdot E[e^{2r_t}]$

since $r_t \sim N(0, \sigma^2)$, $2 \cdot r_t \sim N(0, 4\sigma^2)$
e^{2r_t} is a log normal distribution
using log-normal distribution properties —
$E[X] = \exp(\mu + \frac{1}{2}\sigma^2)$ — we have:

$E[P_t^2] = E[P_{t-1}^2] \cdot e^{2\sigma^2}$

assume that P_{t-1} is a known number:

$E[P_t^2] = P_{t-1}^2 \cdot e^{2\sigma^2}$

$var(P_t) = P_{t-1}^2 \cdot e^{2\sigma^2} - P_{t-1}^2 \cdot e^{\sigma^2}$

$var(P_t) = P_{t-1}^2 (e^{2\sigma^2} - e^{\sigma^2})$

$\Rightarrow std(P_t) = P_{t-1} \cdot \sqrt{e^{2\sigma^2} - e^{\sigma^2}}$

Set P_{t-1} to be 100 and sigma to be 0.75 and run simulations:

Classic Brownian Motion

Expectations:

a. Expected Value

```
theory_mean = p_previous

theory_mean

100
```

b. Standard Deviation

```
theory_std = sigma

theory_std

0.75
```

Simulations:

a. Expected Value

```
sim_mean = sim_prices.mean()
```

```
sim_mean
```

```
99.99449634616789
```

b. Standard Deviation

```
sim_std = sim_prices.std()
```

```
sim_std
```

```
0.7558224706821802
```

Arithmetic Return System

Expectations:

a. Expected Value

```
theory_mean = p_previous
```

```
theory_mean
```

```
100
```

b. Standard Deviation

```
theory_std = p_previous * sigma
```

```
theory_std
```

```
75.0
```

Simulations:

a. Expected Value

```
sim_mean = sim_prices.mean()
```

```
sim_mean
```

```
99.95931785984811
```

b. Standard Deviation

```
sim_std = sim_prices.std()
```

```
sim_std
```

```
75.35227994963377
```

Geometric Brownian Motion

Expectations:

a. Expected Value

```
theory_mean = p_previous * math.exp(0.5 * pow(sigma,2))

theory_mean

132.47847587288655
```

b. Standard Deviation

```
theory_std = p_previous * math.sqrt(math.exp(2 * pow(sigma,2)) - math.exp(pow(sigma,2)))

theory_std

115.11568928507236
```

Simulations:

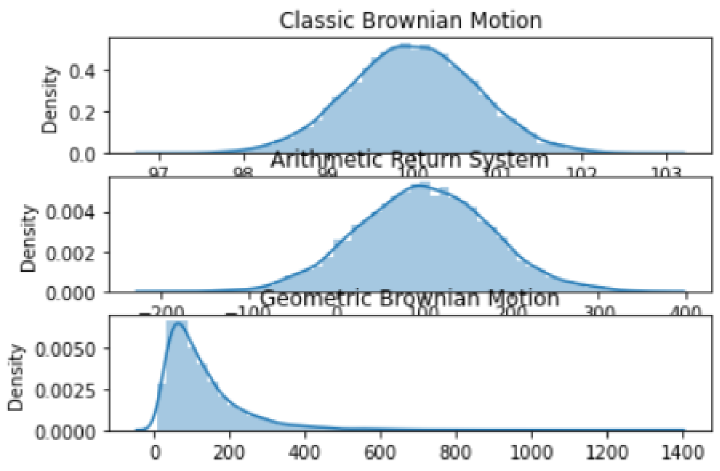
a. Expected Value

```
sim_mean

131.27768634748443
```

b. Standard Deviation

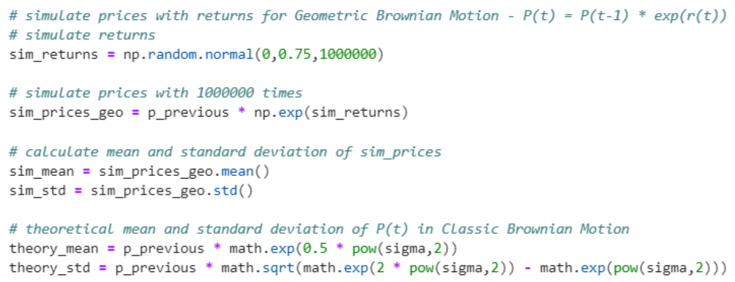
```
sim_std

113.59204878739429
```

Graphs of Distributions of Three Different Systems of Returns:

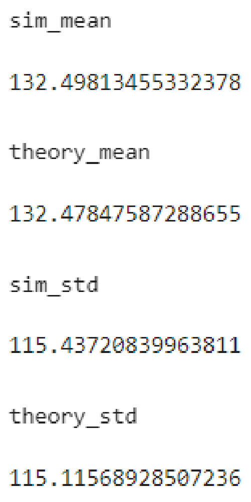


Conclusion:

From above, we could see that results from simulations are generally the same as results derived from analytical expressions. Note that there are nuances among results, especially for Geometric Brownian Motion – for one thing, there could be floating point errors; for another thing, only one time of 10000 random draws of returns was conducted – if conducting more draws, or more sets of 10000 draws, these nuances might be wiped out. For example, if 1000000 draws of returns are conducted, differences between expectations and results would be smaller:



```
# simulate prices with returns for Geometric Brownian Motion - P(t) = P(t-1) * exp(r(t))
# simulate returns
sim_returns = np.random.normal(0,0.75,1000000)

# simulate prices with 1000000 times
sim_prices_geo = p_previous * np.exp(sim_returns)

# calculate mean and standard deviation of sim_prices
sim_mean = sim_prices_geo.mean()
sim_std = sim_prices_geo.std()

# theoretical mean and standard deviation of P(t) in Classic Brownian Motion
theory_mean = p_previous * math.exp(0.5 * pow(sigma,2))
theory_std = p_previous * math.sqrt(math.exp(2 * pow(sigma,2)) - math.exp(pow(sigma,2)))
```



```
sim_mean

132.49813455332378

theory_mean

132.47847587288655

sim_std

115.43720839963811

theory_std

115.11568928507236
```

Overall, expectations and simulations are almost the same.

<u>Problem 2</u>

Implement a function similar to the “return_calculate()” in this week’s code. Allow the user to specify the method of return calculation.

Use DailyPrices.csv. Calculate the arithmetic returns for all prices.

Remove the mean from the series so that the mean(META)=0

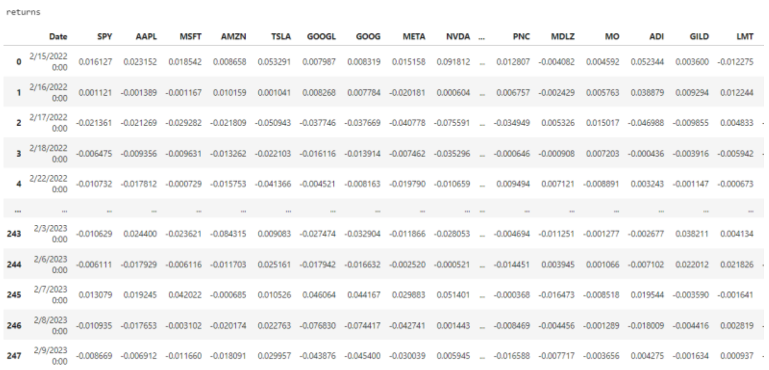
Calculate VaR

1. Using a normal distribution.
2. Using a normal distribution with an Exponentially Weighted variance ($\lambda = 0.94$)
3. Using a MLE fitted T distribution.
4. Using a fitted AR(1) model.
5. Using a Historic Simulation.

Compare the 5 values.

As defined in codes written for problem 2, function “return_cal” could be called to calculate returns of given prices with specified method of calculation – arithmetic or log.

For data in DailyPrices.csv, the arithmetic returns of all prices are (only part of the matrix is shown due to too many items):



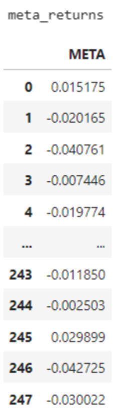
```
returns
```

	Date	SPY	AAPL	MSFT	AMZN	TSLA	GOOGL	GOOG	META	NVDA	...	PNC	MDLZ	MO	ADI	GILD	LMT
0	2/15/2022 0:00	0.016127	0.023152	0.018542	0.008658	0.053291	0.007987	0.008319	0.015158	0.091812	...	0.012807	-0.004082	0.004592	0.052344	0.003600	-0.012275
1	2/16/2022 0:00	0.001121	-0.001389	-0.001167	0.010159	0.001041	0.008268	0.007784	-0.020181	0.000604	...	0.006757	-0.002429	0.005763	0.038879	0.009294	0.012244
2	2/17/2022 0:00	-0.021361	-0.021269	-0.029282	-0.021809	-0.050943	-0.037746	-0.037669	-0.040778	-0.075591	...	-0.034949	0.005326	0.015017	-0.046988	-0.009855	0.004833
3	2/18/2022 0:00	-0.006475	-0.009356	-0.009631	-0.013262	-0.022103	-0.016116	-0.013914	-0.007462	-0.035296	...	-0.000646	-0.000908	0.007203	-0.000436	-0.003916	-0.005942
4	2/22/2022 0:00	-0.010732	-0.017812	-0.000729	-0.015753	-0.041366	-0.004521	-0.008163	-0.019790	-0.010659	...	0.009494	0.007121	-0.008891	0.003243	-0.001147	-0.000673
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
243	2/3/2023 0:00	-0.010629	0.024400	-0.023621	-0.084315	0.009083	-0.027474	-0.032904	-0.011866	-0.028053	...	-0.004694	-0.011251	-0.001277	-0.002677	0.038211	0.004134
244	2/6/2023 0:00	-0.006111	-0.017929	-0.006116	-0.011703	0.025161	-0.017942	-0.016632	-0.002520	-0.000521	...	-0.014451	0.003945	0.001066	-0.007102	0.022012	0.021826
245	2/7/2023 0:00	0.013079	0.019245	0.042022	-0.000685	0.010526	0.046064	0.044167	0.029883	0.051401	...	-0.000368	-0.016473	-0.008518	0.019544	-0.003590	-0.001641
246	2/8/2023 0:00	-0.010935	-0.017653	-0.003102	-0.020174	0.022763	-0.076830	-0.074417	-0.042741	0.001443	...	-0.008469	-0.004456	-0.001289	-0.018009	-0.004416	0.002819
247	2/9/2023 0:00	-0.008669	-0.006912	-0.011660	-0.018091	0.029957	-0.043876	-0.045400	-0.030039	0.005945	...	-0.016588	-0.007717	-0.003656	0.004275	-0.001634	0.000937

Extract returns of “META”:

```
returns.loc[:,"META"].copy()

0      0.015158
1     -0.020181
2     -0.040778
3     -0.007462
4     -0.019790
         ...
243   -0.011866
244   -0.002520
245    0.029883
246   -0.042741
247   -0.030039
Name: META, Length: 248, dtype: float64
```

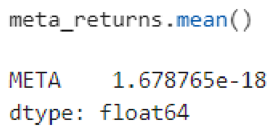
“De-mean” returns of “META”:



```
meta_returns
```

	META
0	0.015175
1	-0.020165
2	-0.040761
3	-0.007446
4	-0.019774
...	...
243	-0.011850
244	-0.002503
245	0.029899
246	-0.042725
247	-0.030022

Check if the mean of "META" returns is zero now:

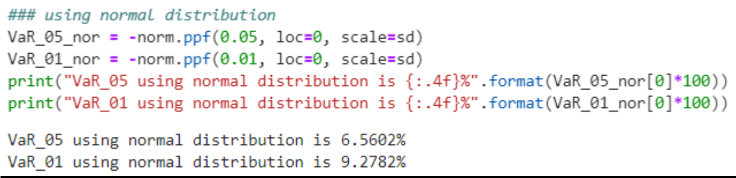


```
meta_returns.mean()

META    1.678765e-18
dtype: float64
```

As shown above, the mean is now effectively zero.

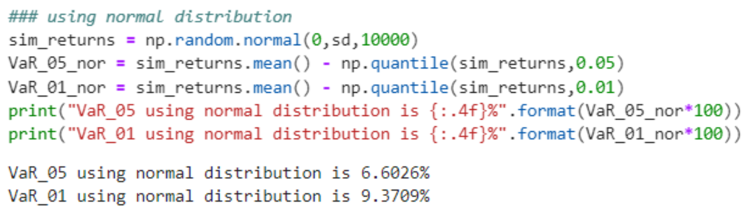
Start calculating VaR under circumstances indicated (note that the instrument in this problem is stock, so we may assume that returns are linear, and we could use quantile functions to calculate VaR rather than simulations):

a. <u>Using a normal distribution</u>



```
### using normal distribution
VaR_05_nor = -norm.ppf(0.05, loc=0, scale=sd)
VaR_01_nor = -norm.ppf(0.01, loc=0, scale=sd)
print("VaR_05 using normal distribution is {:.4f}%".format(VaR_05_nor[0]*100))
print("VaR_01 using normal distribution is {:.4f}%".format(VaR_01_nor[0]*100))

VaR_05 using normal distribution is 6.5602%
VaR_01 using normal distribution is 9.2782%
```

As shown in the screenshot, using a normal distribution, 5% VaR is 6.5602%, indicating a minimum loss of 6.5602% on a normal 5% bad day, and 1% VaR is 9.2782%, indicating a minimum loss of 9.2782% on a normal 1% bad day. Note that I did not simulate a number of returns under normal distribution with mean and standard deviation of returns of "META", and use np.quantile in python to derive VaR. In fact, results are quite similar derived in these two ways, as results of the other way shown below:

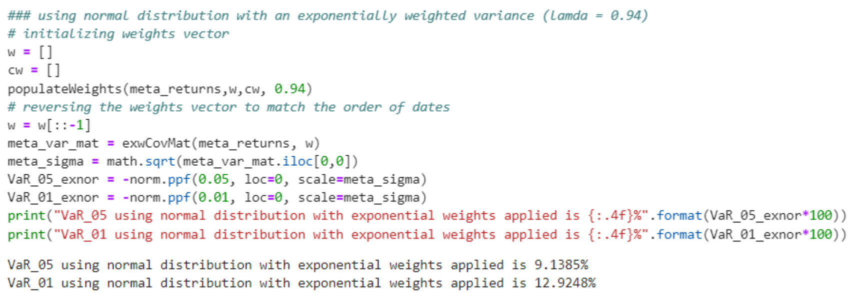


```
### using normal distribution
sim_returns = np.random.normal(0,sd,10000)
VaR_05_nor = sim_returns.mean() - np.quantile(sim_returns,0.05)
VaR_01_nor = sim_returns.mean() - np.quantile(sim_returns,0.01)
print("VaR_05 using normal distribution is {:.4f}%".format(VaR_05_nor*100))
print("VaR_01 using normal distribution is {:.4f}%".format(VaR_01_nor*100))

VaR_05 using normal distribution is 6.6026%
VaR_01 using normal distribution is 9.3709%
```

Therefore, it is acceptable to use the norm.ppf included in scipy package to derive VaR while using normal distribution (also applicable to t distribution later).

b. <u>Using a normal distribution with an Exponentially Weighted variance</u>

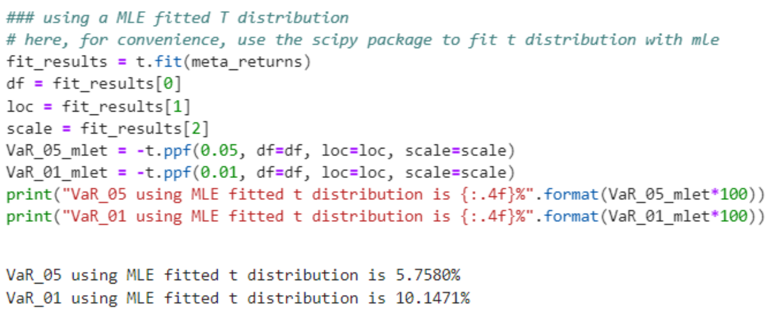
(lamda=0.94)



```
### using normal distribution with an exponentially weighted variance (lamda = 0.94)
# initializing weights vector
w = []
cw = []
populateWeights(meta_returns,w,cw, 0.94)
# reversing the weights vector to match the order of dates
w = w[::-1]
meta_var_mat = exwCovMat(meta_returns, w)
meta_sigma = math.sqrt(meta_var_mat.iloc[0,0])
VaR_05_exnor = -norm.ppf(0.05, loc=0, scale=meta_sigma)
VaR_01_exnor = -norm.ppf(0.01, loc=0, scale=meta_sigma)
print("VaR_05 using normal distribution with exponential weights applied is {:.4f}%".format(VaR_05_exnor*100))
print("VaR_01 using normal distribution with exponential weights applied is {:.4f}%".format(VaR_01_exnor*100))

VaR_05 using normal distribution with exponential weights applied is 9.1385%
VaR_01 using normal distribution with exponential weights applied is 12.9248%
```

As shown in the screenshot, using a normal distribution with exponentially weighted variance, 5% VaR is 9.1385%, indicating a minimum loss of 9.1385% on a normal 5% bad day, and 1% VaR is 12.9248%, indicating a minimum loss of 12.9248% on a normal 1% bad day. Note that the order of weights derived was reversed to match the order of dates of returns (nearest date is the last item).

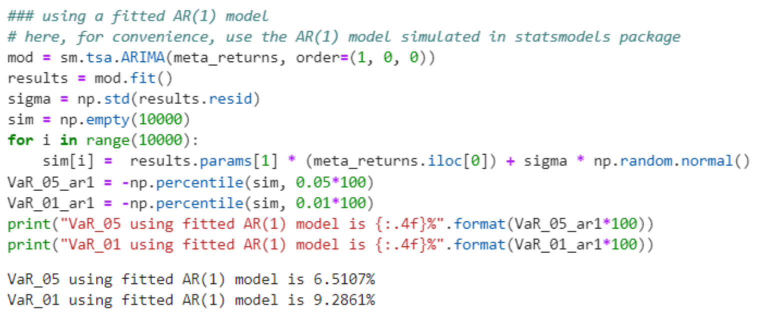
c. Using a MLE fitted T distribution



```
### using a MLE fitted T distribution
# here, for convenience, use the scipy package to fit t distribution with mle
fit_results = t.fit(meta_returns)
df = fit_results[0]
loc = fit_results[1]
scale = fit_results[2]
VaR_05_mlet = -t.ppf(0.05, df=df, loc=loc, scale=scale)
VaR_01_mlet = -t.ppf(0.01, df=df, loc=loc, scale=scale)
print("VaR_05 using MLE fitted t distribution is {:.4f}%".format(VaR_05_mlet*100))
print("VaR_01 using MLE fitted t distribution is {:.4f}%".format(VaR_01_mlet*100))


VaR_05 using MLE fitted t distribution is 5.7580%
VaR_01 using MLE fitted t distribution is 10.1471%
```

As shown in the screenshot, using a MLE fitted t distribution, 5% VaR is 5.7580%, indicating a minimum loss of 5.7580% on a normal 5% bad day, and 1% VaR is 10.1471%, indicating a minimum loss of 10.1471% on a normal 1% bad day.

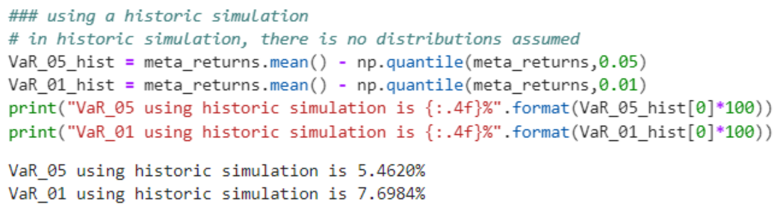
d. Using a AR(1) fitted model



```
### using a fitted AR(1) model
# here, for convenience, use the AR(1) model simulated in statsmodels package
mod = sm.tsa.ARIMA(meta_returns, order=(1, 0, 0))
results = mod.fit()
sigma = np.std(results.resid)
sim = np.empty(10000)
for i in range(10000):
    sim[i] =  results.params[1] * (meta_returns.iloc[0]) + sigma * np.random.normal()
VaR_05_ar1 = -np.percentile(sim, 0.05*100)
VaR_01_ar1 = -np.percentile(sim, 0.01*100)
print("VaR_05 using fitted AR(1) model is {:.4f}%".format(VaR_05_ar1*100))
print("VaR_01 using fitted AR(1) model is {:.4f}%".format(VaR_01_ar1*100))

VaR_05 using fitted AR(1) model is 6.5107%
VaR_01 using fitted AR(1) model is 9.2861%
```

As shown in the screenshot, using a AR(1) fitted model, 5% VaR is 6.5107%, indicating a minimum loss of 6.5107% on a normal 5% bad day, and 1% VaR is 9.2861%, indicating a minimum loss of 9.2861% on a normal 1% bad day. Note that in this part, after fitting data into the AR(1) model, I take one step further – making a simulation of the return starting from its initial value to 10000th return, and use the output return vector for VaR calculation.

e. Using a Historical Simulation


```
### using a historic simulation
# in historic simulation, there is no distributions assumed
VaR_05_hist = meta_returns.mean() - np.quantile(meta_returns,0.05)
VaR_01_hist = meta_returns.mean() - np.quantile(meta_returns,0.01)
print("VaR_05 using historic simulation is {:.4f}%".format(VaR_05_hist[0]*100))
print("VaR_01 using historic simulation is {:.4f}%".format(VaR_01_hist[0]*100))

VaR_05 using historic simulation is 5.4620%
VaR_01 using historic simulation is 7.6984%
```

As shown in the screenshot, using a historical simulation, 5% VaR is 5.4620%, indicating a minimum loss of 5.4620% on a normal 5% bad day, and 1% VaR is 7.6984%, indicating a minimum loss of 7.6984% on a normal 1% bad day.

Conclusion:
From results above, regardless of 5% or 1% (there is one exception: though using normal distribution results in a higher 5% VaR than using the MLE fitted t distribution, MLE fitted t distribution leads to a higher 1% VaR; this might be the result of flatter tails of t distribution, comparing to a normal distribution), VaR is highest when using a normal distribution with exponentially weighted variance. The second highest VaR comes from using a normal distribution. Then, using AR(1) fitted model has the next highest VaR. The last two methods generate lowest VaR – using MLE fitted t distribution generates higher VaR than using historical simulation.

Problem 3

Using Portfolio.csv and DailyPrices.csv. Assume the expected return on all stocks is 0.

This file contains the stock holdings of 3 portfolios. You own each of these portfolios. Using an exponentially weighted covariance with lambda = 0.94, calculate the VaR of each portfolio as well as your total VaR (VaR of the total holdings). Express VaR as a $.

Discuss your methods and your results.

Choose a different model for returns and calculate VaR again. Why did you choose that model? How did the model change affect the results?

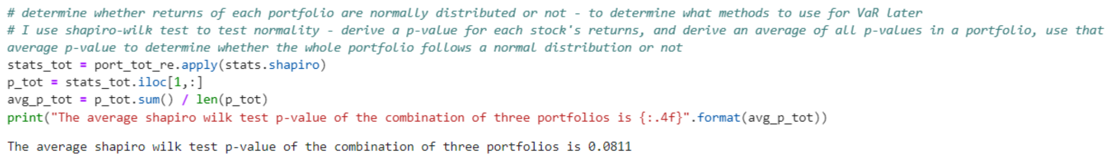
In this problem, I first copied codes used to calculate returns, exponential weights, and variance-covariance matrix from problem 2 for later use in this problem.

After reading data from provided csv files, I first extracted holdings of stocks of each portfolio and stored them separately.

Arithmetic Returns with Delta Normal VaR:

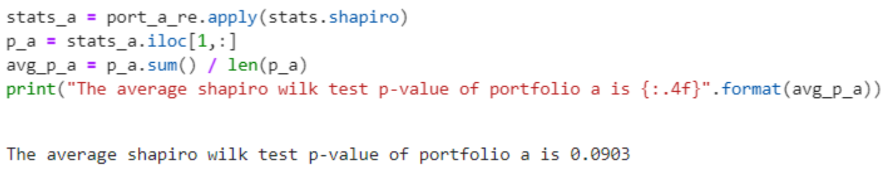
I filtered prices and returns (calculated with arithmetic system) accordingly. Then, I conducted Shapiro-Wilk test upon each stock's returns in each portfolio, and generated an average p-value of Shapiro-Wilk test for each portfolio to determine whether returns in each portfolio generally follow a normal distribution or not:

Combined Portfolio:



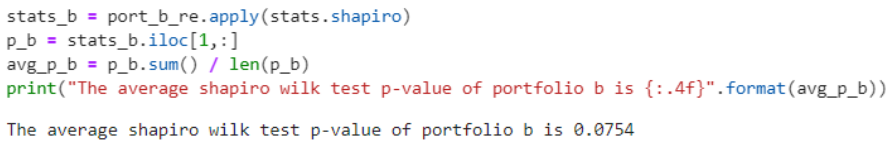
```
# determine whether returns of each portfolio are normally distributed or not - to determine what methods to use for VaR later
# I use shapiro-wilk test to test normality - derive a p-value for each stock's returns, and derive an average of all p-values in a portfolio, use that
average p-value to determine whether the whole portfolio follows a normal distribution or not
stats_tot = port_tot_re.apply(stats.shapiro)
p_tot = stats_tot.iloc[1,:]
avg_p_tot = p_tot.sum() / len(p_tot)
print("The average shapiro wilk test p-value of the combination of three portfolios is {:.4f}".format(avg_p_tot))

The average shapiro wilk test p-value of the combination of three portfolios is 0.0811
```

Portfolio A:



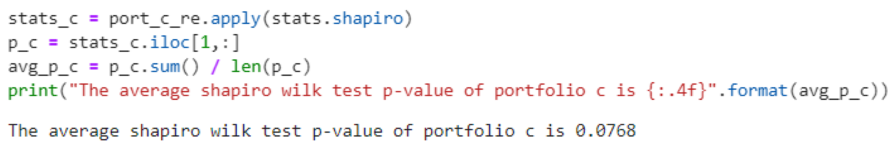
```
stats_a = port_a_re.apply(stats.shapiro)
p_a = stats_a.iloc[1,:]
avg_p_a = p_a.sum() / len(p_a)
print("The average shapiro wilk test p-value of portfolio a is {:.4f}".format(avg_p_a))


The average shapiro wilk test p-value of portfolio a is 0.0903
```

Portfolio B:



```
stats_b = port_b_re.apply(stats.shapiro)
p_b = stats_b.iloc[1,:]
avg_p_b = p_b.sum() / len(p_b)
print("The average shapiro wilk test p-value of portfolio b is {:.4f}".format(avg_p_b))

The average shapiro wilk test p-value of portfolio b is 0.0754
```

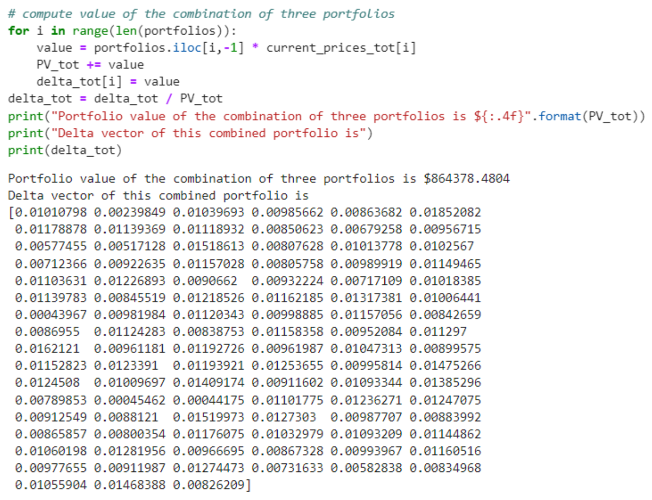
Portfolio C:



```
stats_c = port_c_re.apply(stats.shapiro)
p_c = stats_c.iloc[1,:]
avg_p_c = p_c.sum() / len(p_c)
print("The average shapiro wilk test p-value of portfolio c is {:.4f}".format(avg_p_c))

The average shapiro wilk test p-value of portfolio c is 0.0768
```

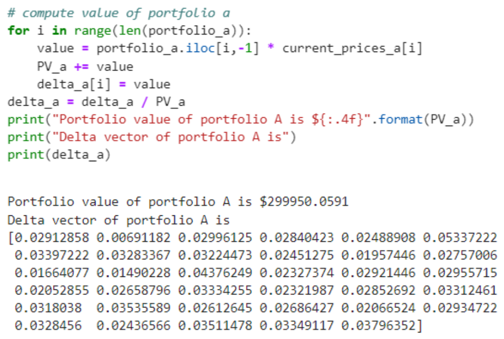
As shown in screenshots above, average p-values of Shapiro-Wilk test for four portfolios (three individual portfolios and one combination) are all greater than 0.05, and we do not enough evidence to reject that these returns in portfolios are normally distributed. Since they are normally distributed, and returns of stocks are linear, I chose Delta Normal VaR as the way I calculated VaR in later parts. Followingly, I derived total value for each portfolio, and correspondingly computed delta vector that would be used later via dividing each stock's value by total value of the portfolio within each portfolio – total value and delta vector of each portfolio are:

Combined Portfolio:



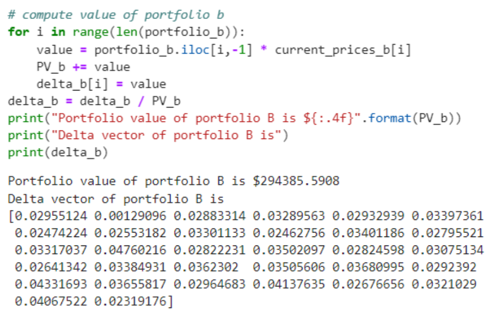
```
# compute value of the combination of three portfolios
for i in range(len(portfolios)):
    value = portfolios.iloc[i,-1] * current_prices_tot[i]
    PV_tot += value
    delta_tot[i] = value
delta_tot = delta_tot / PV_tot
print("Portfolio value of the combination of three portfolios is ${:.4f}".format(PV_tot))
print("Delta vector of this combined portfolio is")
print(delta_tot)

Portfolio value of the combination of three portfolios is $864378.4804
Delta vector of this combined portfolio is
[0.01010798 0.00239849 0.01039693 0.00985662 0.00863682 0.01852082
 0.01178878 0.01139369 0.01118932 0.00850623 0.00679258 0.00956715
 0.00577455 0.00517128 0.01518613 0.00807628 0.01013778 0.0102567
 0.00712366 0.00922635 0.01157028 0.00805758 0.00989919 0.01149465
 0.01103631 0.01226893 0.0090662  0.00932224 0.00717109 0.01018385
 0.01139783 0.00845519 0.01218526 0.01162185 0.01317381 0.01006441
 0.00043967 0.00981984 0.01120343 0.00998885 0.01157056 0.00842659
 0.0086955  0.01124283 0.00838753 0.01158358 0.00952084 0.011297
 0.0162121  0.00961181 0.01192726 0.00961987 0.01047313 0.00899575
 0.01152823 0.0123391  0.01193921 0.01253655 0.00995814 0.01475266
 0.0124508  0.01009697 0.01409174 0.00911602 0.01093344 0.01385296
 0.00789853 0.00045462 0.00044175 0.01101775 0.01236271 0.01247075
 0.00912549 0.0088121  0.01519973 0.0127303  0.00987707 0.00883992
 0.00865857 0.00800354 0.01176075 0.01032979 0.01093209 0.01144862
 0.01060198 0.01281956 0.00966695 0.00867328 0.00993967 0.01160516
 0.00977655 0.00911987 0.01274473 0.00731633 0.00582838 0.00834968
 0.01055904 0.01468388 0.00826209]
```

Portfolio A:



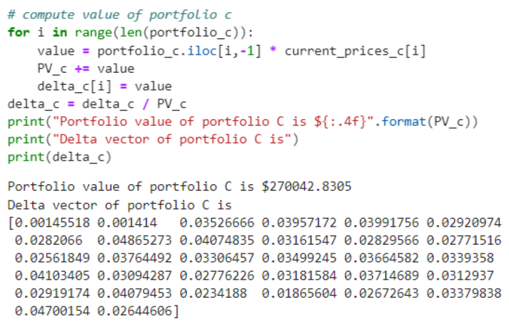
```
# compute value of portfolio a
for i in range(len(portfolio_a)):
    value = portfolio_a.iloc[i,-1] * current_prices_a[i]
    PV_a += value
    delta_a[i] = value
delta_a = delta_a / PV_a
print("Portfolio value of portfolio A is ${:.4f}".format(PV_a))
print("Delta vector of portfolio A is")
print(delta_a)


Portfolio value of portfolio A is $299950.0591
Delta vector of portfolio A is
[0.02912858 0.00691182 0.02996125 0.02840423 0.02488908 0.05337222
 0.03397222 0.03283367 0.03224473 0.02451275 0.01957446 0.02757006
 0.01664077 0.01490228 0.04376249 0.02327374 0.02921446 0.02955715
 0.02052855 0.02658796 0.03334255 0.02321987 0.02852692 0.03312461
 0.0318038  0.03535589 0.02612645 0.02686427 0.02066524 0.02934722
 0.0328456  0.02436566 0.03511478 0.03349117 0.03796352]
```

Portfolio B:



```
# compute value of portfolio b
for i in range(len(portfolio_b)):
    value = portfolio_b.iloc[i,-1] * current_prices_b[i]
    PV_b += value
    delta_b[i] = value
delta_b = delta_b / PV_b
print("Portfolio value of portfolio B is ${:.4f}".format(PV_b))
print("Delta vector of portfolio B is")
print(delta_b)

Portfolio value of portfolio B is $294385.5908
Delta vector of portfolio B is
[0.02955124 0.00129096 0.02883314 0.03289563 0.02932939 0.03397361
 0.02474224 0.02553182 0.03301133 0.02462756 0.03401186 0.02795521
 0.03317037 0.04760216 0.02822231 0.03502097 0.02824598 0.03075134
 0.02641342 0.03384931 0.0362302  0.03505606 0.03680995 0.0292392
 0.04331693 0.03655817 0.02964683 0.04137635 0.02676656 0.0321029
 0.04067522 0.02319176]
```

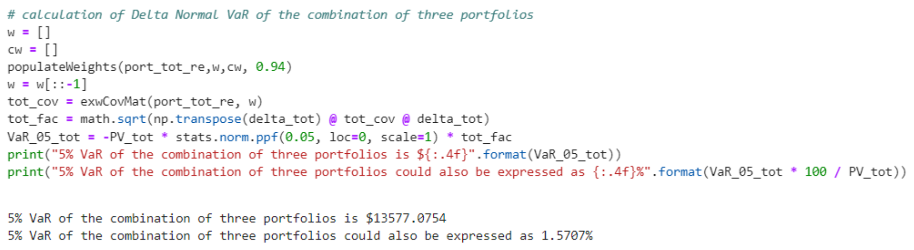
Portfolio C:



```
# compute value of portfolio c
for i in range(len(portfolio_c)):
    value = portfolio_c.iloc[i,-1] * current_prices_c[i]
    PV_c += value
    delta_c[i] = value
delta_c = delta_c / PV_c
print("Portfolio value of portfolio C is ${:.4f}".format(PV_c))
print("Delta vector of portfolio C is")
print(delta_c)

Portfolio value of portfolio C is $270042.8305
Delta vector of portfolio C is
[0.00145518 0.001414   0.03526666 0.03957172 0.03991756 0.02920974
 0.0282066  0.04865273 0.04074835 0.03161547 0.02829566 0.02771516
 0.02561849 0.03764492 0.03306457 0.03499245 0.03664582 0.0339358
 0.04103405 0.03094287 0.02776226 0.03181584 0.03714689 0.0312937
 0.02919174 0.04079453 0.0234188  0.01865604 0.02672643 0.03379838
 0.04700154 0.02644606]
```

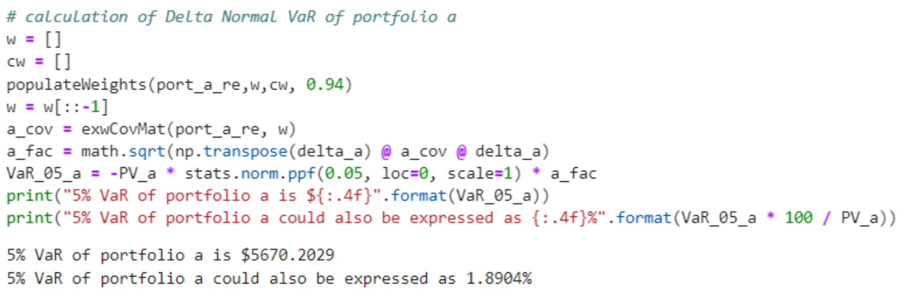
Eventually, with results derived all above, Delta Normal VaRs could be computed (note that when we are using exponentially weighted variance-covariance matrix to calculate Delta Normal VaR, we need to match the order of exponential weights with the order of dates. In this case, dates list from farthest to nearest, so we need to reverse the order of exponential weights through calling "w = w[::-1]" as shown below in screenshots):

Combined Portfolio:



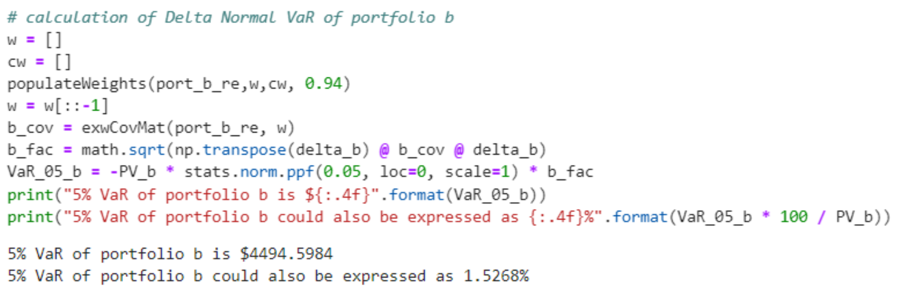
```
# calculation of Delta Normal VaR of the combination of three portfolios
w = []
cw = []
populateWeights(port_tot_re,w,cw, 0.94)
w = w[::-1]
tot_cov = exwCovMat(port_tot_re, w)
tot_fac = math.sqrt(np.transpose(delta_tot) @ tot_cov @ delta_tot)
VaR_05_tot = -PV_tot * stats.norm.ppf(0.05, loc=0, scale=1) * tot_fac
print("5% VaR of the combination of three portfolios is ${:.4f}".format(VaR_05_tot))
print("5% VaR of the combination of three portfolios could also be expressed as {:.4f}%".format(VaR_05_tot * 100 / PV_tot))

5% VaR of the combination of three portfolios is $13577.0754
5% VaR of the combination of three portfolios could also be expressed as 1.5707%
```

Portfolio A:



```
# calculation of Delta Normal VaR of portfolio a
w = []
cw = []
populateWeights(port_a_re,w,cw, 0.94)
w = w[::-1]
a_cov = exwCovMat(port_a_re, w)
a_fac = math.sqrt(np.transpose(delta_a) @ a_cov @ delta_a)
VaR_05_a = -PV_a * stats.norm.ppf(0.05, loc=0, scale=1) * a_fac
print("5% VaR of portfolio a is ${:.4f}".format(VaR_05_a))
print("5% VaR of portfolio a could also be expressed as {:.4f}%".format(VaR_05_a * 100 / PV_a))

5% VaR of portfolio a is $5670.2029
5% VaR of portfolio a could also be expressed as 1.8904%
```

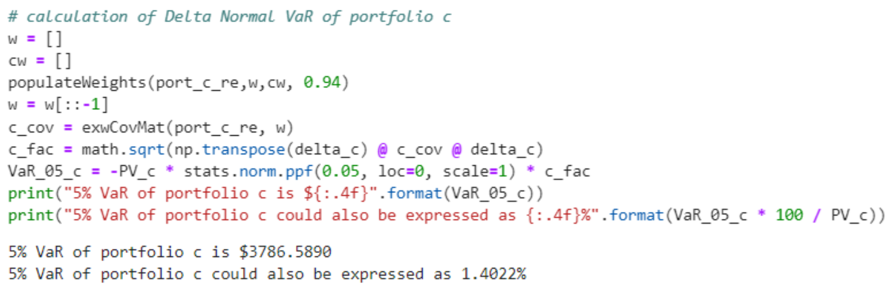
Portfolio B:



```
# calculation of Delta Normal VaR of portfolio b
w = []
cw = []
populateWeights(port_b_re,w,cw, 0.94)
w = w[::-1]
b_cov = exwCovMat(port_b_re, w)
b_fac = math.sqrt(np.transpose(delta_b) @ b_cov @ delta_b)
VaR_05_b = -PV_b * stats.norm.ppf(0.05, loc=0, scale=1) * b_fac
print("5% VaR of portfolio b is ${:.4f}".format(VaR_05_b))
print("5% VaR of portfolio b could also be expressed as {:.4f}%".format(VaR_05_b * 100 / PV_b))
```

```
5% VaR of portfolio b is $4494.5984
5% VaR of portfolio b could also be expressed as 1.5268%
```

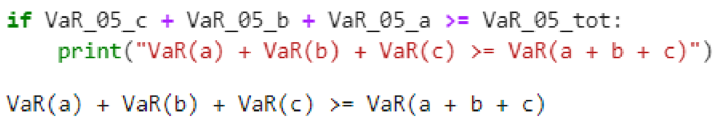
Portfolio C:



```
# calculation of Delta Normal VaR of portfolio c
w = []
cw = []
populateWeights(port_c_re,w,cw, 0.94)
w = w[::-1]
c_cov = exwCovMat(port_c_re, w)
c_fac = math.sqrt(np.transpose(delta_c) @ c_cov @ delta_c)
VaR_05_c = -PV_c * stats.norm.ppf(0.05, loc=0, scale=1) * c_fac
print("5% VaR of portfolio c is ${:.4f}".format(VaR_05_c))
print("5% VaR of portfolio c could also be expressed as {:.4f}%".format(VaR_05_c * 100 / PV_c))
```

```
5% VaR of portfolio c is $3786.5890
5% VaR of portfolio c could also be expressed as 1.4022%
```

From screenshots above, we could see that among individual portfolios, portfolio A has the highest 5% VaR, and portfolio B has the second highest, followed by portfolio C in the last place. Notice that, in this case, VaR is a subadditive measure:



```
if VaR_05_c + VaR_05_b + VaR_05_a >= VaR_05_tot:
    print("VaR(a) + VaR(b) + VaR(c) >= VaR(a + b + c)")
```

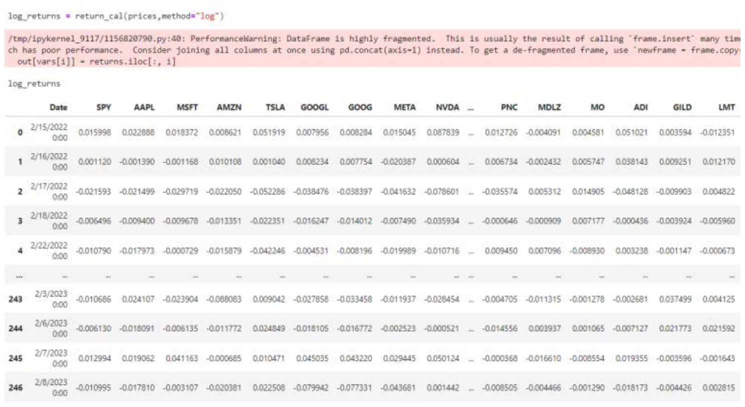
```
VaR(a) + VaR(b) + VaR(c) >= VaR(a + b + c)
```

The reason behind that could be returns, in this case, are normally distributed, and normal distribution is elliptical. Therefore, subadditive nature holds here.

Results above are derived from returns calculated with arithmetic system which has an advantage that arithmetic returns are summable across portfolios, but if returns are derived with log system, or under assuming Geometric Brownian Motion, things could be different. Log returns are sometimes preferred to arithmetic returns, since they avoid being affected by extreme values, and log returns are like arithmetic returns with continuously compounding which is exactly what sometimes happened in real markets. Also, I would switch from Delta Normal VaR to Historic VaR below, since Historic VaR addresses problems Delta Normal VaR has – linear assumption of portfolio value with returns and assumption of normally distributed returns – and it would sometimes be closer to real markets with breaking these assumptions.

<u>Log Returns with Historic VaR:</u>

I first compute returns by calling defined function "return_cal" with specifying method to be "log" (part of the matrix due to too many items):



```
log_returns = return_cal(prices,method="log")
```

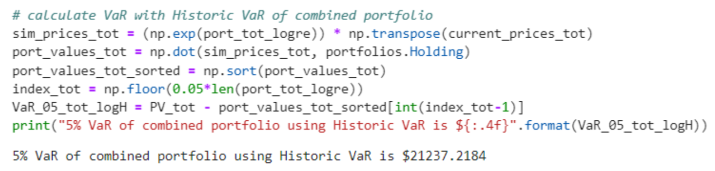
```
/tmp/ipykernel_9117/1156820790.py:40: PerformanceWarning: DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many tim
ch has poor performance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `newframe = frame.copy
  out[vars[i]] = returns.iloc[:, i]
```

```
log_returns
```

	Date	SPY	AAPL	MSFT	AMZN	TSLA	GOOGL	GOOG	META	NVDA	...	PNC	MDLZ	MO	ADI	GILD	LMT
0	2/15/2022 0:00	0.015998	0.022888	0.018372	0.008621	0.051919	0.007956	0.008284	0.015045	0.087839	...	0.012726	-0.004091	0.004581	0.051021	0.003594	-0.012351
1	2/16/2022 0:00	0.001120	-0.001390	-0.001168	0.010108	0.001040	0.008234	0.007754	-0.020387	0.000604	...	0.006734	-0.002432	0.005747	0.038143	0.009251	0.012170
2	2/17/2022 0:00	-0.021593	-0.021499	-0.029719	-0.022050	-0.052286	-0.038476	-0.038397	-0.041632	-0.078601	...	-0.035574	0.005312	0.014905	-0.048128	-0.009903	0.004822
3	2/18/2022 0:00	-0.006496	-0.009400	-0.009678	-0.013351	-0.022351	-0.016247	-0.014012	-0.007490	-0.035934	...	-0.000646	-0.000909	0.007177	-0.000436	-0.003924	-0.005960
4	2/22/2022 0:00	-0.010790	-0.017973	-0.000729	-0.015879	-0.042246	-0.004531	-0.008196	-0.019989	-0.010716	...	0.009450	0.007096	-0.008930	0.003238	-0.001147	-0.000673
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
243	2/3/2023 0:00	-0.010686	0.024107	-0.023904	-0.088083	0.009042	-0.027858	-0.033458	-0.011937	-0.028454	...	-0.004705	-0.011315	-0.001278	-0.002681	0.037499	0.004125
244	2/6/2023 0:00	-0.006130	-0.018091	-0.006135	-0.011772	0.024849	-0.018105	-0.016772	-0.002523	-0.000521	...	-0.014556	0.003937	0.001065	-0.007127	0.021773	0.021592
245	2/7/2023 0:00	0.012994	0.019062	0.041163	-0.000685	0.010471	0.045035	0.043220	0.029445	0.050124	...	-0.000368	-0.016610	-0.008554	0.019355	-0.003596	-0.001643
246	2/8/2023 0:00	-0.010995	-0.017810	-0.003107	-0.020381	0.022508	-0.079942	-0.077331	-0.043681	0.001442	...	-0.008505	-0.004466	-0.001290	-0.018173	-0.004426	0.002815

With log returns derived, I filtered returns according to different portfolios, and started calculating Historic VaR for each portfolio:

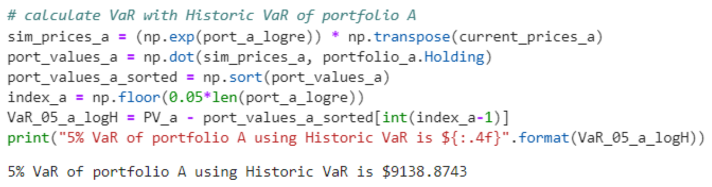
Combined Portfolio:



```
# calculate VaR with Historic VaR of combined portfolio
sim_prices_tot = (np.exp(port_tot_logre)) * np.transpose(current_prices_tot)
port_values_tot = np.dot(sim_prices_tot, portfolios.Holding)
port_values_tot_sorted = np.sort(port_values_tot)
index_tot = np.floor(0.05*len(port_tot_logre))
VaR_05_tot_logH = PV_tot - port_values_tot_sorted[int(index_tot-1)]
print("5% VaR of combined portfolio using Historic VaR is ${:.4f}".format(VaR_05_tot_logH))
```

```
5% VaR of combined portfolio using Historic VaR is $21237.2184
```

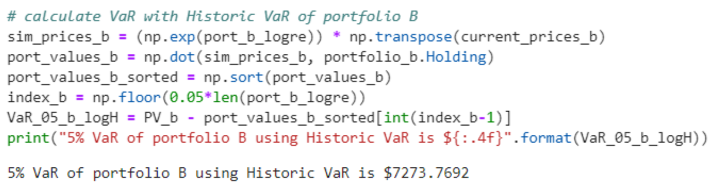
Portfolio A:



```
# calculate VaR with Historic VaR of portfolio A
sim_prices_a = (np.exp(port_a_logre)) * np.transpose(current_prices_a)
port_values_a = np.dot(sim_prices_a, portfolio_a.Holding)
port_values_a_sorted = np.sort(port_values_a)
index_a = np.floor(0.05*len(port_a_logre))
VaR_05_a_logH = PV_a - port_values_a_sorted[int(index_a-1)]
print("5% VaR of portfolio A using Historic VaR is ${:.4f}".format(VaR_05_a_logH))
```

```
5% VaR of portfolio A using Historic VaR is $9138.8743
```

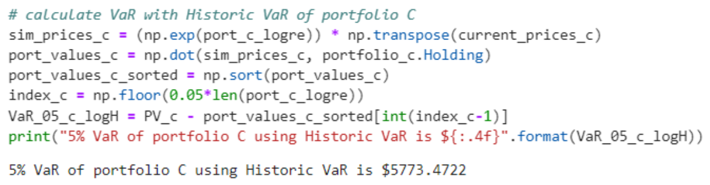
Portfolio B:



```
# calculate VaR with Historic VaR of portfolio B
sim_prices_b = (np.exp(port_b_logre)) * np.transpose(current_prices_b)
port_values_b = np.dot(sim_prices_b, portfolio_b.Holding)
port_values_b_sorted = np.sort(port_values_b)
index_b = np.floor(0.05*len(port_b_logre))
VaR_05_b_logH = PV_b - port_values_b_sorted[int(index_b-1)]
print("5% VaR of portfolio B using Historic VaR is ${:.4f}".format(VaR_05_b_logH))
```

```
5% VaR of portfolio B using Historic VaR is $7273.7692
```

Portfolio C:



```
# calculate VaR with Historic VaR of portfolio C
sim_prices_c = (np.exp(port_c_logre)) * np.transpose(current_prices_c)
port_values_c = np.dot(sim_prices_c, portfolio_c.Holding)
port_values_c_sorted = np.sort(port_values_c)
index_c = np.floor(0.05*len(port_c_logre))
VaR_05_c_logH = PV_c - port_values_c_sorted[int(index_c-1)]
print("5% VaR of portfolio C using Historic VaR is ${:.4f}".format(VaR_05_c_logH))
```

```
5% VaR of portfolio C using Historic VaR is $5773.4722
```

From results above, we may notice that subadditive nature still holds, and among individual portfolios, portfolio A has the highest 5% VaR, and portfolio B comes next, followed by portfolio C in the last place. This huge raise in VaRs in all portfolios may be resulted in use of log returns and change from Delta Normal VaR to Historic VaR.

On the one hand, returns of these portfolios might not be well-fitted into normal distribution, and Delta Normal VaR could be not precise. On the other hand, historical data may not accurately reflect current market conditions, and using historic VaR in this sense could lead to inaccurate estimation of risk.