Name: Niyomwungeri Parmenide ISHIMWE

Andrew-ID: parmenin

DATA, INFERENCE, AND APPLIED MACHINE LEARNING

18-785

ASSIGNMENT 1

5 SEPTEMBER 2022

I, the undersigned, have read the entire contents of the syllabus for course 18-785 (Data Inference and Applied Machine Learning) and agree with the terms and conditions of participating in this course, including adherence to CMU's AIV policy.

Signature: **Niyomwungeri Parmenide ISHIMWE**
Andrew ID: **parmenin**
Full Name: **Niyomwungeri Parmenide ISHIMWE**

The libraries used:

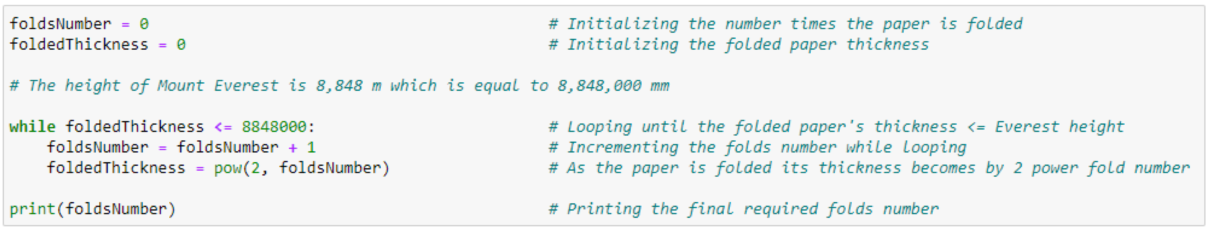
- import numpy as np
- import pandas as pd
- import matplotlib.pyplot as plt

QUESTION 1:

It was required to compute the number of times a piece of paper 1mm needs to be folded to exceed the height of Mount Everest which is 8,848 m equalling 8,848,000 mm.

While folding the paper, I found the relationship between the number of paper folds and the thickness of the folded paper which is:

Folded paper's thickness = 2 power folds number i.e., initially, the thickness is 1mm = 2^0 when you fold the paper once; the thickness becomes 2mm = 2^1, secondly, 4mm = 2mm = 2^2, thirdly, 8mm = 2mm = 2^3 and so forth. This should be implemented as a loop and stop the loop when the folded thickness equals or exceed 8,848,000 millimeters.



```
foldsNumber = 0                                      # Initializing the number times the paper is folded
foldedThickness = 0                                  # Initializing the folded paper thickness

# The height of Mount Everest is 8,848 m which is equal to 8,848,000 mm

while foldedThickness <= 8848000:                    # Looping until the folded paper's thickness <= Everest height
    foldsNumber = foldsNumber + 1                    # Incrementing the folds number while looping
    foldedThickness = pow(2, foldsNumber)            # As the paper is folded its thickness becomes by 2 power fold number

print(foldsNumber)                                   # Printing the final required folds number
```

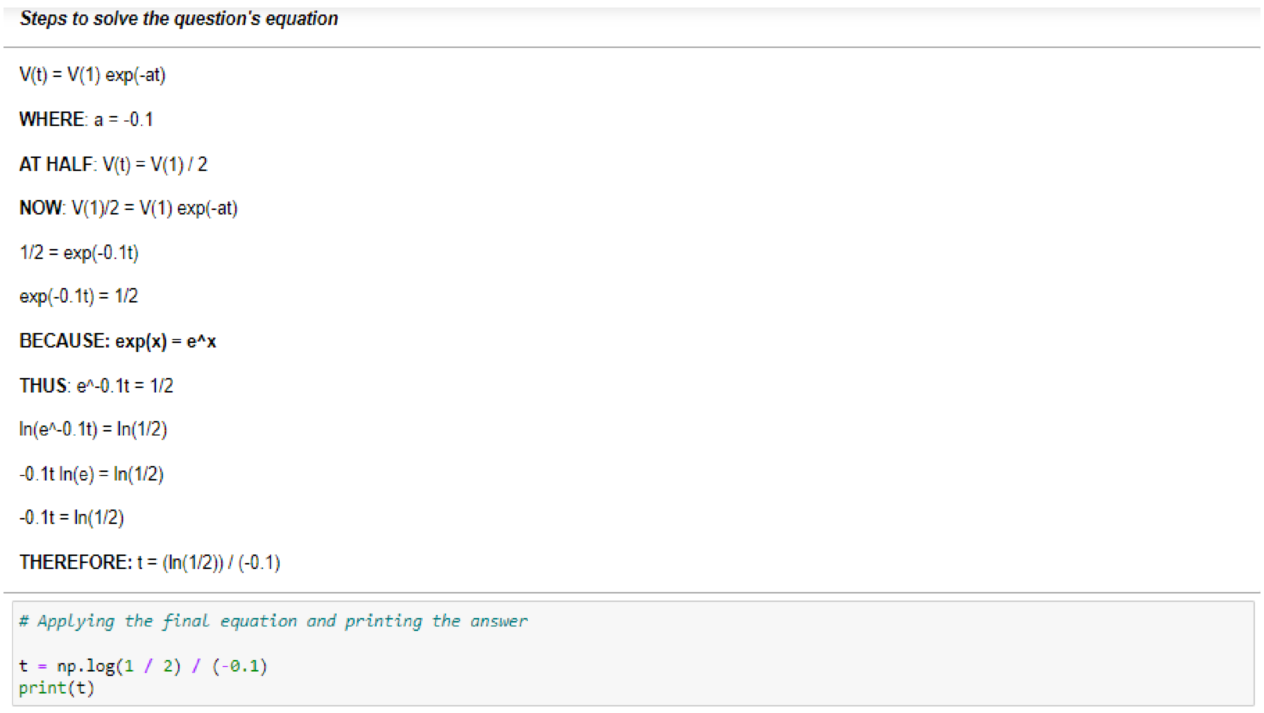
Picture 1: Implementation of paper thickness against mount Everest

From the picture above, I had to initialize the number of folds and folded thickness to zero. Then loop through by incrementing the number of the folds and increasing the thickness by two power the number of folds until the folded thickness is equal to or greater than 8,848,000 mm. When the loop reaches on **24th** time, it stops to mark that the height of Everest is exceeded on the **24th fold**.

QUESTION 2:

It was required to find how much time t will it take for the volume of water in the reservoir to decrease to less than one-half of the initial volume and that the water decreases at a rate of v(t) = v(0) exp(-at) with a=0.1.

While solving the question's equation, I reached the final equation to use in the codes for finding t which is **t = ln (1/2) / (-0.1)**.



Steps to solve the question's equation

V(t) = V(1) exp(-at)

WHERE: a = -0.1

AT HALF: V(t) = V(1) / 2

NOW: V(1)/2 = V(1) exp(-at)

1/2 = exp(-0.1t)

exp(-0.1t) = 1/2

BECAUSE: **exp(x)** = e^x

THUS: e^-0.1t = 1/2

ln(e^-0.1t) = ln(1/2)

-0.1t ln(e) = ln(1/2)

-0.1t = ln(1/2)

THEREFORE: t = (ln(1/2)) / (-0.1)

```
# Applying the final equation and printing the answer

t = np.log(1 / 2) / (-0.1)
print(t)
```

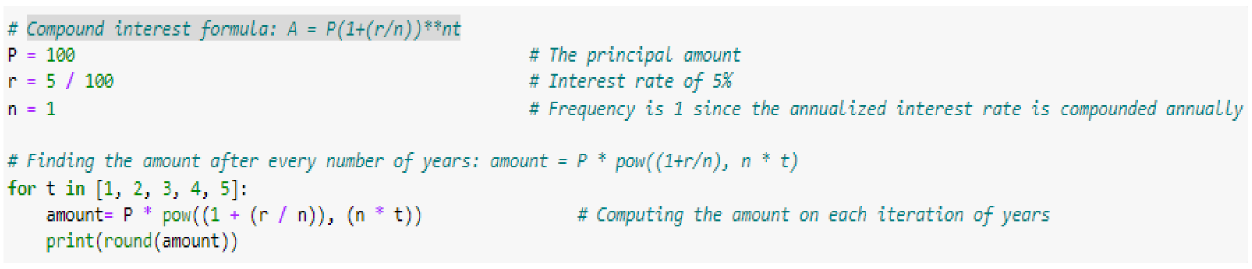
Picture 2: Solving the water reservoir equation

From the picture above, I had to implement the final formula as **t = np.log(1/2) / (-0.1)** and finally print the value of t which is approximately equal to **7**. Notice the use of **np.log** which is the Napierian or normal logarithm function available in the NumPy library. This means that it would take **7 amount of time** to reach the half of initial volume.

QUESTION 3:

It was asking to find how much interest will be made after depositing $100 in a bank account that offers an annualized interest rate of 5% compounded annually for 1, 2, 3, 4, and 5 years.

This requires using the compound interest formula: **A** = $\boldsymbol{P(1+\left(\frac{r}{n}\right))^{nt}}$ where **A** is the interest amount, **P** is the principal amount deposited, **r** is the interest rate, **n** is the frequency the interest rate was compounded annually, and **t** is the time or number of years.



```
# Compound interest formula: A = P(1+(r/n))**nt
P = 100                                     # The principal amount
r = 5 / 100                                 # Interest rate of 5%
n = 1                                       # Frequency is 1 since the annualized interest rate is compounded annually

# Finding the amount after every number of years: amount = P * pow((1+r/n), n * t)
for t in [1, 2, 3, 4, 5]:
    amount= P * pow((1 + (r / n)), (n * t))            # Computing the amount on each iteration of years
    print(round(amount))
```

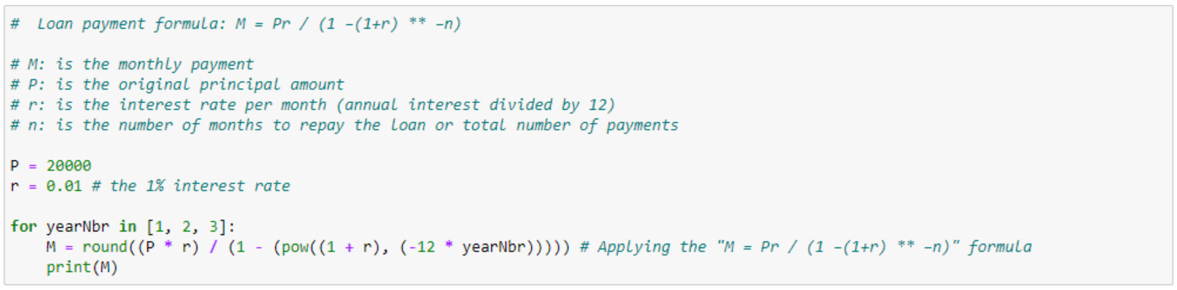
Picture 3: Finding the compound interest

As depicted above, I used the formula to compute the interest amount for each number of years by multiplying it into the formula. This is done inside a loop that also implies that the amount to have is **105** after the first year, **110** after the second, **116** after the third, **122** after the fourth and **128** for the fifth year.

QUESTION 4:

It was required to find the monthly payment to pay off the debt in one, two, and three years when a loan of $20,000 is provided from a financial institution at an interest rate of 1%.

For this question, the loan payment formula needs to be applied for finding the monthly payment of each of the number of years. The loan payment formula is **M** = $\left(\frac{\boldsymbol{Pr}}{(\mathbf{1}-(\mathbf{1}+\boldsymbol{r}))-\boldsymbol{n}}\right)$ helps to find the monthly payments where **M** is the monthly payment, **P** is the original principal amount, **r** is the interest rate per month i.e., the annual interest divided by 12, and **n** is the total number of months to repay the loan or the total number of payments.



```
#  Loan payment formula: M = Pr / (1 -(1+r) ** -n)

# M: is the monthly payment
# P: is the original principal amount
# r: is the interest rate per month (annual interest divided by 12)
# n: is the number of months to repay the loan or total number of payments

P = 20000
r = 0.01 # the 1% interest rate

for yearNbr in [1, 2, 3]:
    M = round((P * r) / (1 - (pow((1 + r), (-12 * yearNbr))))) # Applying the "M = Pr / (1 -(1+r) ** -n)" formula
    print(M)
```

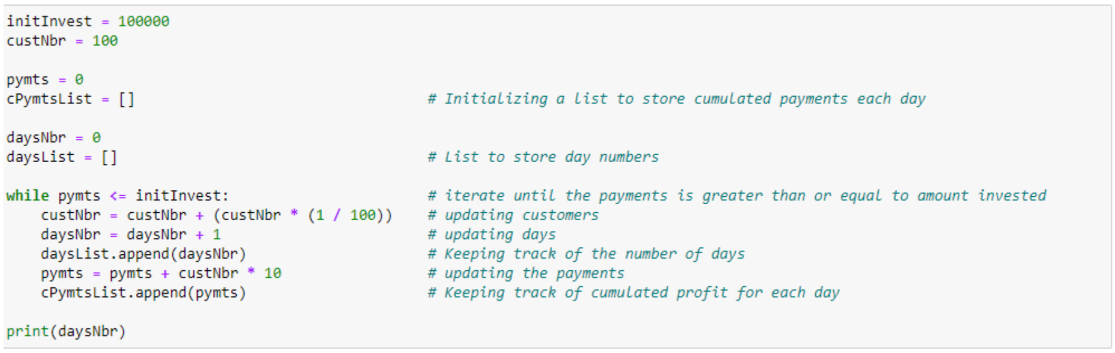
Picture 4: Loan payment calculation

As shown in the picture, I used the formula to find the monthly payment to pay off the debt which is done inside a loop where the formula is applied, and results reflect the number of years. This also made with a loop that computes the monthly payments and prints it on every iteration. The results show that in one year the payment would be **1777**, **941** for two years and **664** for three years.

QUESTION 5:

Here the task was to provide the number of days it will take to repay the initial investment of $100,000 invested in a new business, that expects 100 customers that grows at a daily rate of 1% where each customer brings $10 profits.

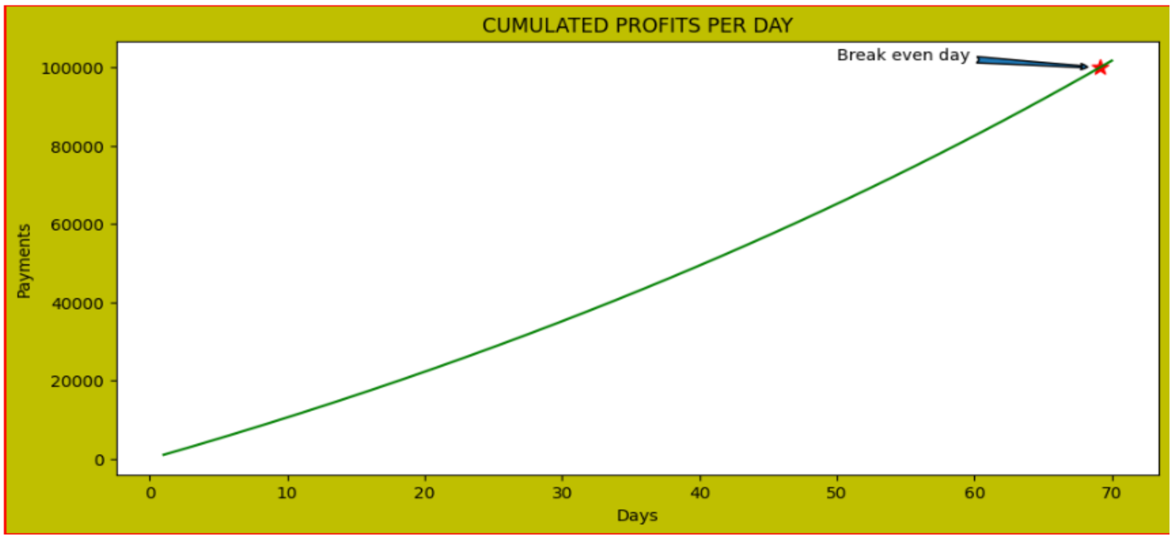
This was approached by increasing the number of customers at the 1% rate, the days' number, and the amount of the payments by adding the customer number times 10. This is done inside a loop that keeps iterating until the payments are not less or equal to the initial investment. Therefore, it will take **70 days** to repay the initial investment.



```
initInvest = 100000
custNbr = 100

pymts = 0
cPymtsList = []                                       # Initializing a list to store cumulated payments each day

daysNbr = 0
daysList = []                                         # List to store day numbers

while pymts <= initInvest:                            # iterate until the payments is greater than or equal to amount invested
    custNbr = custNbr + (custNbr * (1 / 100))         # updating customers
    daysNbr = daysNbr + 1                             # updating days
    daysList.append(daysNbr)                          # Keeping track of the number of days
    pymts = pymts + custNbr * 10                      # updating the payments
    cPymtsList.append(pymts)                          # Keeping track of cumulated profit for each day

print(daysNbr)
```

Picture 5: Repayment of initial investment

As required to also plot the profits per day, showing the initial investment and marking the breakeven day, the days' list and cumulated payment list were declared and used inside the loop to append on them the day numbers, and payments respectively. These lists along with the initial investment helps in plotting the graph and also finding and showing the break even day by the use of NumPy's linear interpolation **np.interp(initInvest, cPymtsList , daysList)**. As depicted bellow, the break-even point is approximately **69.1612861613676.**



Picture 6: Plotting the break-even day

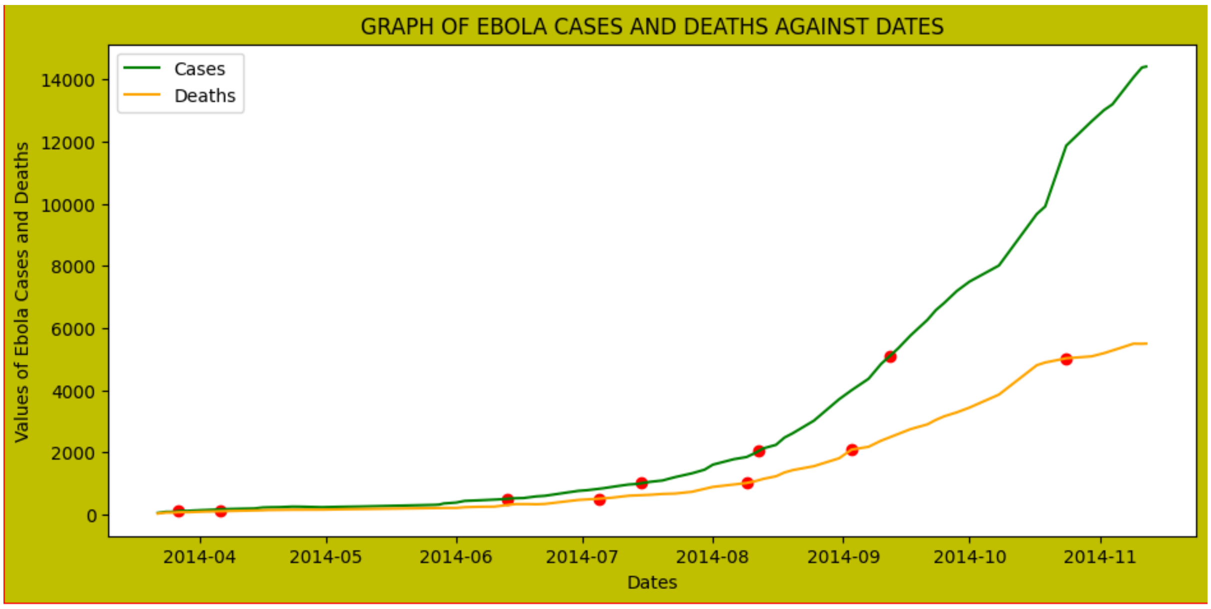
<u>QUESTION 6:</u>

I was tasked to estimate the dates when the number of cases and deaths due to Ebola exceeded the 100, 500, 1000, 2000, and 5000 thresholds using the Ebola excel data given and interpolation.

This is done by the use first reading data from an excel file, extracting only the data needed to use i.e., dates, cases, and deaths, sorting them by date, and then interpolating them to fill in the missing values. Interpolation is done by first creating a date range including also skipped dates using the **date_range()** function, setting the date column of our extracted data set as the index, and using the **reindex()** function to integrate the date range into our dataset and fill the fields corresponding to skipped dates with NaN, interpolating linearly to get values for **NaN** (missing) values from the dataset for cases and deaths using the **interpolate()** function, and then use **rename_axis()** and **reset_index()** functions to put the date titles again and remove the index. Notice that all those functions are from the **Pandas** library.

After having data to use from interpolation, next I find the first case numbers or death numbers that exceed the threshold numbers by using nested looping through thresholds and the cases or deaths column of our data. I also need to find dates those thresholds exceeded, and it is done by looping into our data set cases or death to find the dates corresponding to the values that exceed the thresholds. The dates for cases and death are maintained separately in their list as well as it is done for cases and death numbers.

After getting the exceeded lists of cases and deaths and their dates, I then plot the graph of Ebola interpolated data and their dates, and I mark using scatter plotting the cases that are exceeding the thresholds.

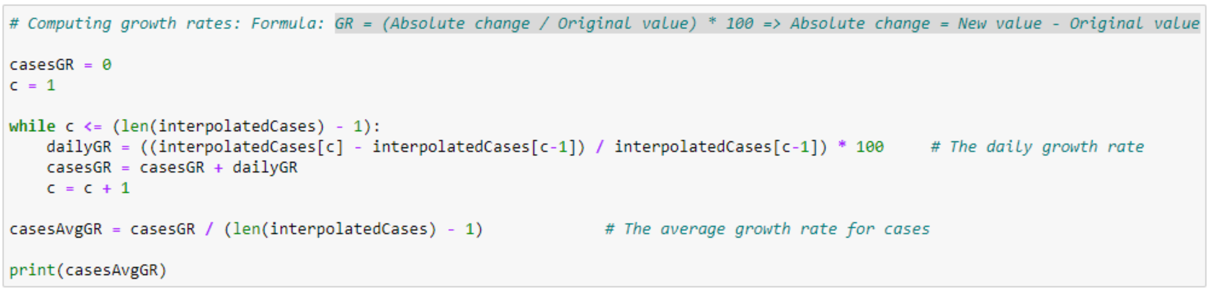


Picture 7: Plot of Ebola cases and deaths and thresholds

As the plot shows, the date when the values exceeded the threshold are shown as **2014-03-27; 2014-06-13; 2014-07-15; 2014-08-12; 2014-09-12** for **111.0, 502.8, 1004.0, 2051.0, 5092.5** cases respectively. In addition, **2014-04-06; 2014-07-05; 2014-08-09; 2014-09-03; 2014-10-24** dates are for **102.16666666666667, 508.75, 1013.0, 2089.0, 5026.0** deaths respectively. This implies that the deaths due to Ebola and Ebola cases were increasing as time moved.

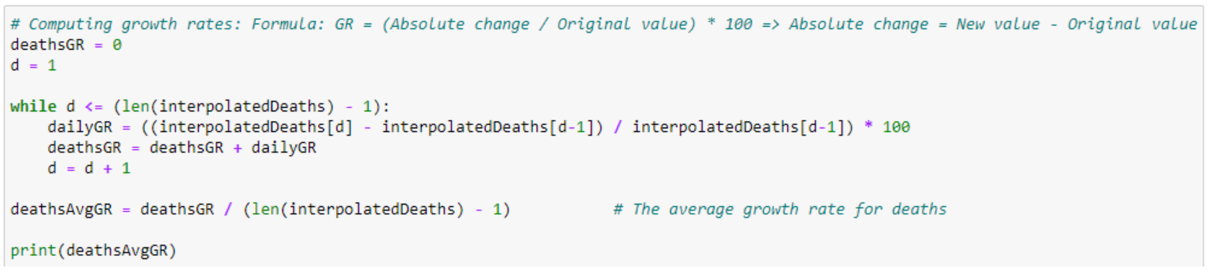
QUESTION 7:

The question was asked to find the average growth rate per day of Ebola cases and deaths using the data from the previous question. This required to use of the growth rate formula where Growth Rate = (Absolute change / Original value) * 100 where the Absolute change = New value - Original value which is done for both cases and death.



```
# Computing growth rates: Formula: GR = (Absolute change / Original value) * 100 => Absolute change = New value - Original value

casesGR = 0
c = 1

while c <= (len(interpolatedCases) - 1):
    dailyGR = ((interpolatedCases[c] - interpolatedCases[c-1]) / interpolatedCases[c-1]) * 100     # The daily growth rate
    casesGR = casesGR + dailyGR
    c = c + 1

casesAvgGR = casesGR / (len(interpolatedCases) - 1)            # The average growth rate for cases

print(casesAvgGR)
```

Picture 8: Implementation of the growth rates for cases



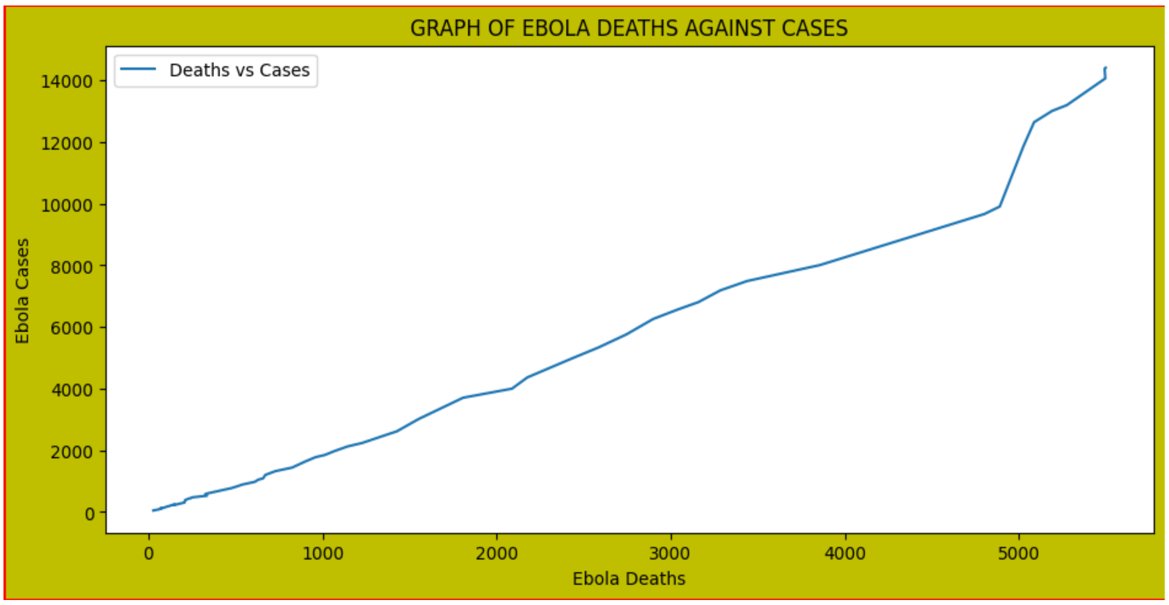
```
# Computing growth rates: Formula: GR = (Absolute change / Original value) * 100 => Absolute change = New value - Original value
deathsGR = 0
d = 1

while d <= (len(interpolatedDeaths) - 1):
    dailyGR = ((interpolatedDeaths[d] - interpolatedDeaths[d-1]) / interpolatedDeaths[d-1]) * 100
    deathsGR = deathsGR + dailyGR
    d = d + 1

deathsAvgGR = deathsGR / (len(interpolatedDeaths) - 1)           # The average growth rate for deaths

print(deathsAvgGR)
```

Picture 9: Implementation of the growth rates for death

As depicted above, first I found the daily growth rates, where I took the interpolated data from the previous question and apply them to the formula for both cases and death and inside the loop then sum the daily results to use for finding the averages. After I looped, divided the totals by the length of the data to obtain the final average growth rates. This concludes that Ebola cases grown by around **2.5%** and death due to Ebola grown at about **2.3%**.

QUESTION 8:

Continuing from the previous two questions, this one asks to plot the number of deaths versus the number of cases, where I used the interpolated data from the previous question and then estimated the average ratio of Ebola deaths to cases.



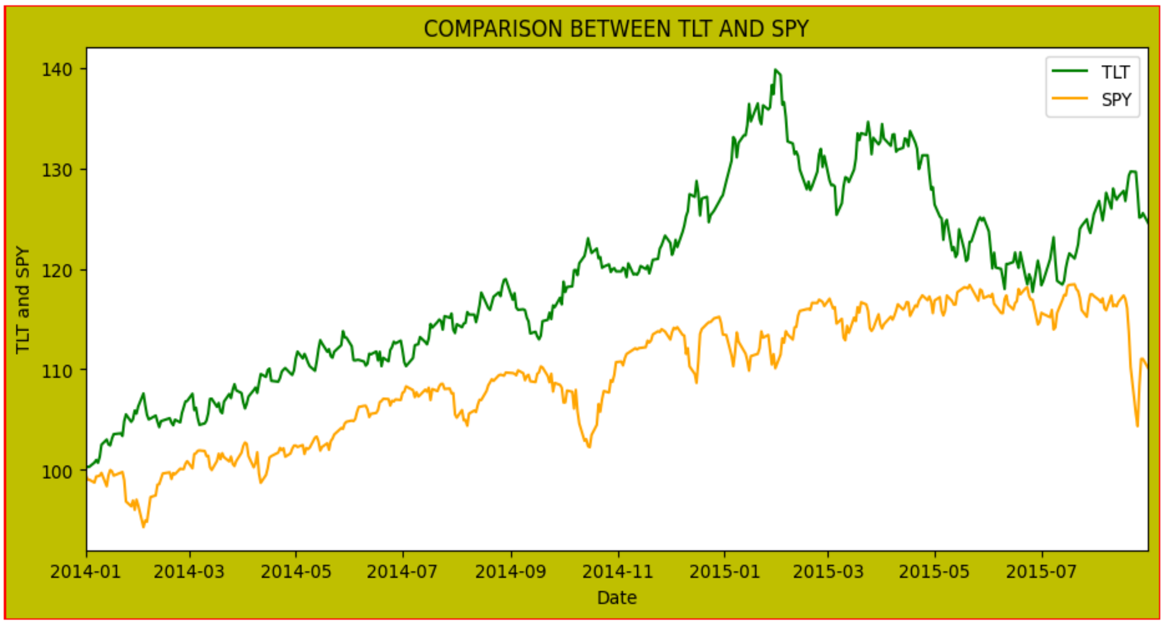
Picture 10: Implementation of the growth rates for death

As depicted above, the graph generally shows that the number of deaths due to Ebola increases as the number of Ebola cases increases. This is also shown by the average ratio of Ebola deaths to cases which is about **0.5577992908998354.** This can imply that half of Ebola cases lead to death.

QUESTION 9:

At this time, I was asked to plot the two-time series during 01/01/2014 – 08/31/2015 and make them comparable by starting from prices of $100 on the first day on 01/01/2014 –08/31/2015. So, the graph below shows the prices at TLT and SPY ETFs.

This is done by first, reading data first from excel file, extracting date and adjusted date, and then normalizing each data object by hundred for both TLT and SPY and saving them into lists of normalized data. After normalizing the data, I then go ahead and plot them on the graph for comparison.



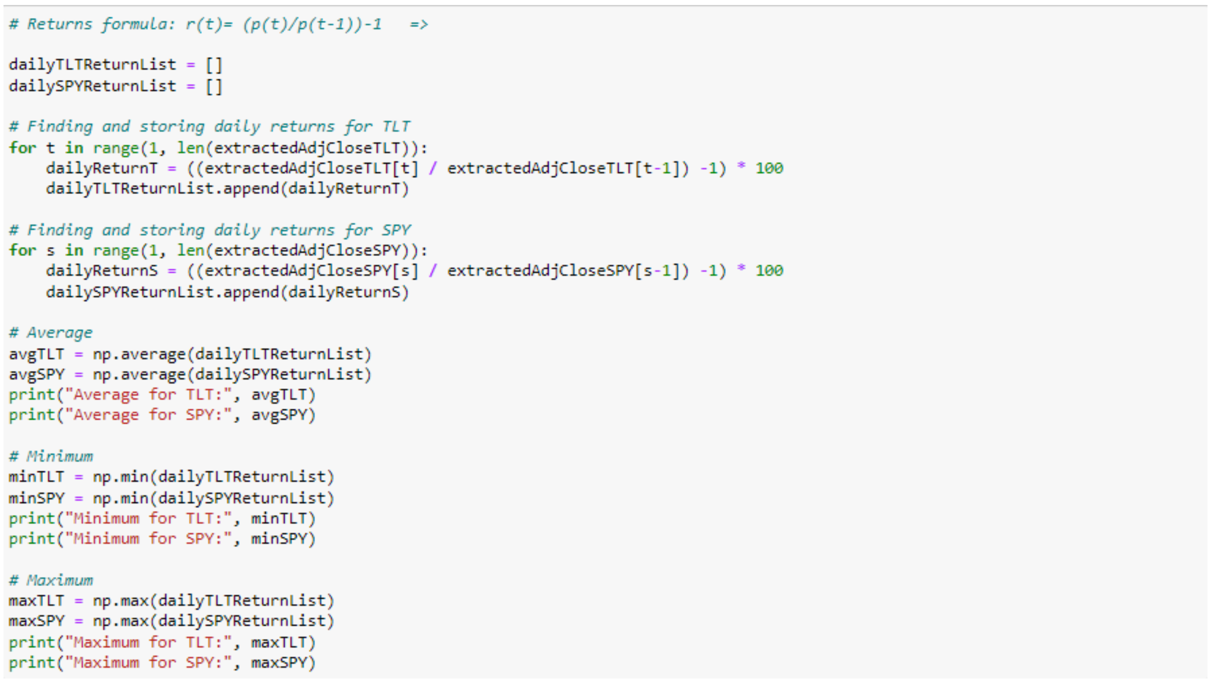
Picture 11: Comparison between prices of TLT and SPY

In general, the graph indicates that the prices for TLT ETF were running high compared to that of SPY. Even though they all started at **100** on **01/01/2014** date**,** as time goes on, they both increased prices until they become close to being equal near the start of **July 2015.** However, TLT shows a sharp increase in prices again over SPY which was also decreasing in prices until **08/31/2015**.

QUESTION 10:

Here, the task is to calculate daily returns using the formula **r(t)= p(t)/p(t-1)-1** for each trading day in the same period as above and calculate the average, min, and max daily return for each of the two ETFs during the period and express them as percentages.

This is done using two lists that maintain daily returns for both ETFs, then using loops to append each calculated daily return in percentage to corresponding list. This helps to later calculate the averages, min, and max for both ETFs.



```python
# Returns formula: r(t)= (p(t)/p(t-1))-1   =>

dailyTLTReturnList = []
dailySPYReturnList = []

# Finding and storing daily returns for TLT
for t in range(1, len(extractedAdjCloseTLT)):
    dailyReturnT = ((extractedAdjCloseTLT[t] / extractedAdjCloseTLT[t-1]) -1) * 100
    dailyTLTReturnList.append(dailyReturnT)

# Finding and storing daily returns for SPY
for s in range(1, len(extractedAdjCloseSPY)):
    dailyReturnS = ((extractedAdjCloseSPY[s] / extractedAdjCloseSPY[s-1]) -1) * 100
    dailySPYReturnList.append(dailyReturnS)

# Average
avgTLT = np.average(dailyTLTReturnList)
avgSPY = np.average(dailySPYReturnList)
print("Average for TLT:", avgTLT)
print("Average for SPY:", avgSPY)

# Minimum
minTLT = np.min(dailyTLTReturnList)
minSPY = np.min(dailySPYReturnList)
print("Minimum for TLT:", minTLT)
print("Minimum for SPY:", minSPY)

# Maximum
maxTLT = np.max(dailyTLTReturnList)
maxSPY = np.max(dailySPYReturnList)
print("Maximum for TLT:", maxTLT)
print("Maximum for SPY:", maxSPY)
```

Picture 12: Average, min, and max daily return for TLT and SPY

In general, the results shows that TLT experienced a big average of returns of around **0.056%** compared to that of SPY which is around **0.026%.** In addition, TLT also shows the highest minimum daily return of around **-2.432%** compared to around **-4.210%** of SPY. However, SPY experienced greatest maximum daily return of around **3.839%** compared to that of TLT.