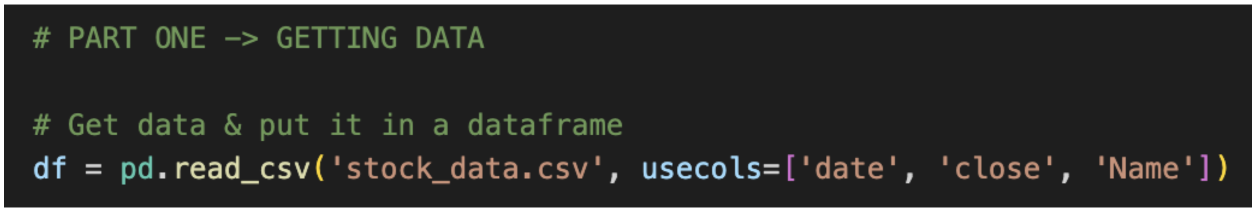
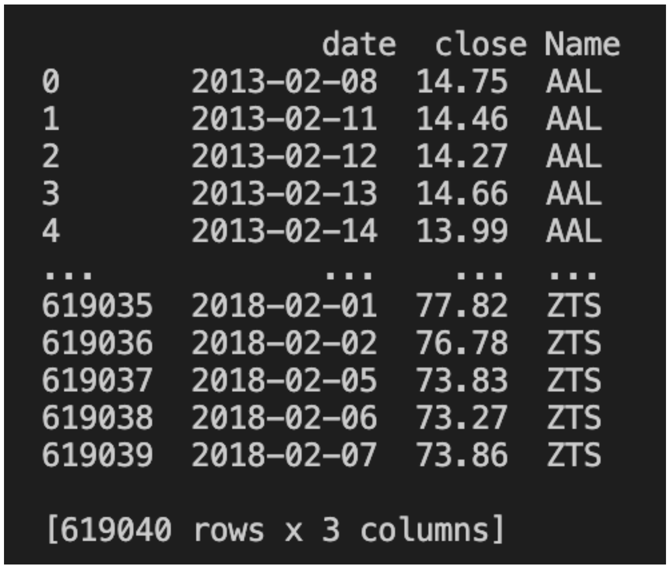
PART ONE

Code:



```
# PART ONE -> GETTING DATA

# Get data & put it in a dataframe
df = pd.read_csv('stock_data.csv', usecols=['date', 'close', 'Name'])
```

Output:



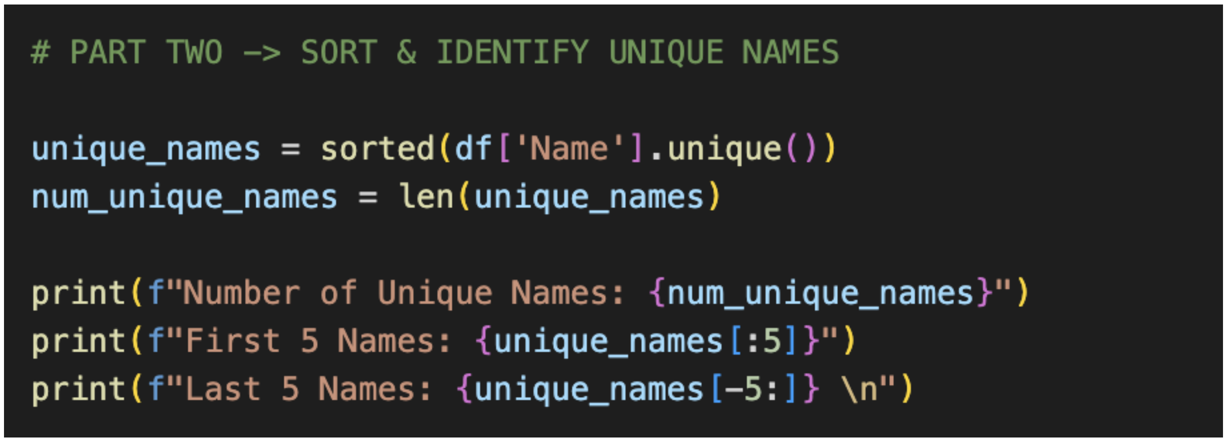
```
               date  close Name
0       2013-02-08  14.75  AAL
1       2013-02-11  14.46  AAL
2       2013-02-12  14.27  AAL
3       2013-02-13  14.66  AAL
4       2013-02-14  13.99  AAL
...            ...    ...  ...
619035  2018-02-01  77.82  ZTS
619036  2018-02-02  76.78  ZTS
619037  2018-02-05  73.83  ZTS
619038  2018-02-06  73.27  ZTS
619039  2018-02-07  73.86  ZTS

[619040 rows x 3 columns]
```

Description

In this piece of code, I use the read_csv method from the pandas library to load into a pandas data frame the “stock_data.csv” dataset and only use the ‘date’, ‘close’, and ‘Name’ columns. Disregarding the other columns in the CSV file.

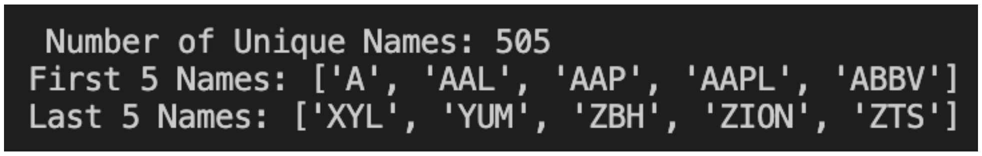
PART TWO

Code:



```python
# PART TWO -> SORT & IDENTIFY UNIQUE NAMES

unique_names = sorted(df['Name'].unique())
num_unique_names = len(unique_names)

print(f"Number of Unique Names: {num_unique_names}")
print(f"First 5 Names: {unique_names[:5]}")
print(f"Last 5 Names: {unique_names[-5:]} \n")
```

Output:



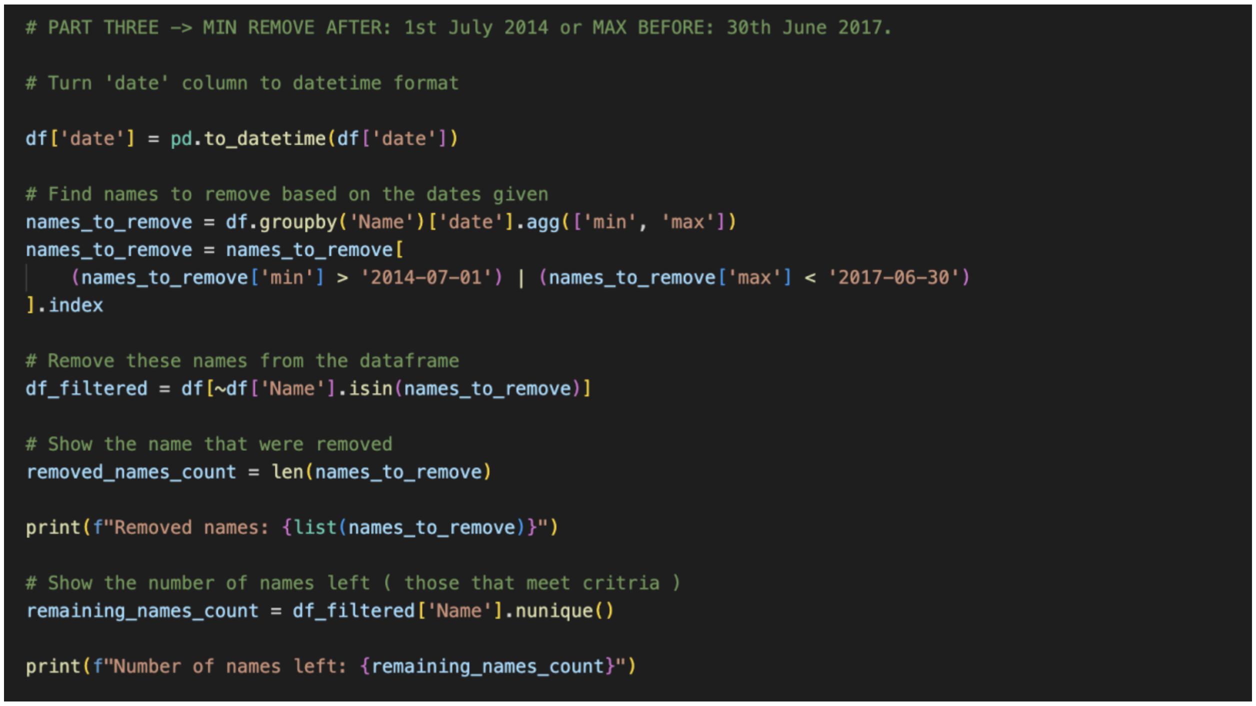
```
 Number of Unique Names: 505
First 5 Names: ['A', 'AAL', 'AAP', 'AAPL', 'ABBV']
Last 5 Names: ['XYL', 'YUM', 'ZBH', 'ZION', 'ZTS']
```

Description:

Using the unique function from the panda’s library I’m able to get all instance of the “Name” column which are unique and then sort them using the sorted function to easily find the first and last 5 names.

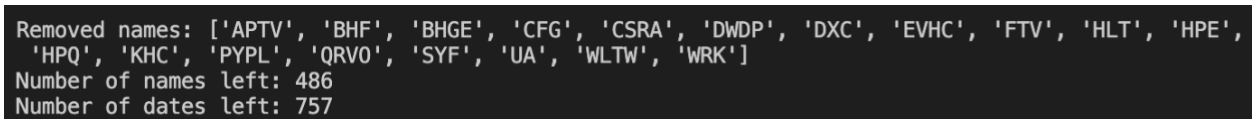
PART THREE

Code:



```
# PART THREE -> MIN REMOVE AFTER: 1st July 2014 or MAX BEFORE: 30th June 2017.

# Turn 'date' column to datetime format

df['date'] = pd.to_datetime(df['date'])

# Find names to remove based on the dates given
names_to_remove = df.groupby('Name')['date'].agg(['min', 'max'])
names_to_remove = names_to_remove[
    (names_to_remove['min'] > '2014-07-01') | (names_to_remove['max'] < '2017-06-30')
].index

# Remove these names from the dataframe
df_filtered = df[~df['Name'].isin(names_to_remove)]

# Show the name that were removed
removed_names_count = len(names_to_remove)

print(f"Removed names: {list(names_to_remove)}")

# Show the number of names left ( those that meet critria )
remaining_names_count = df_filtered['Name'].nunique()

print(f"Number of names left: {remaining_names_count}")
```

Output:



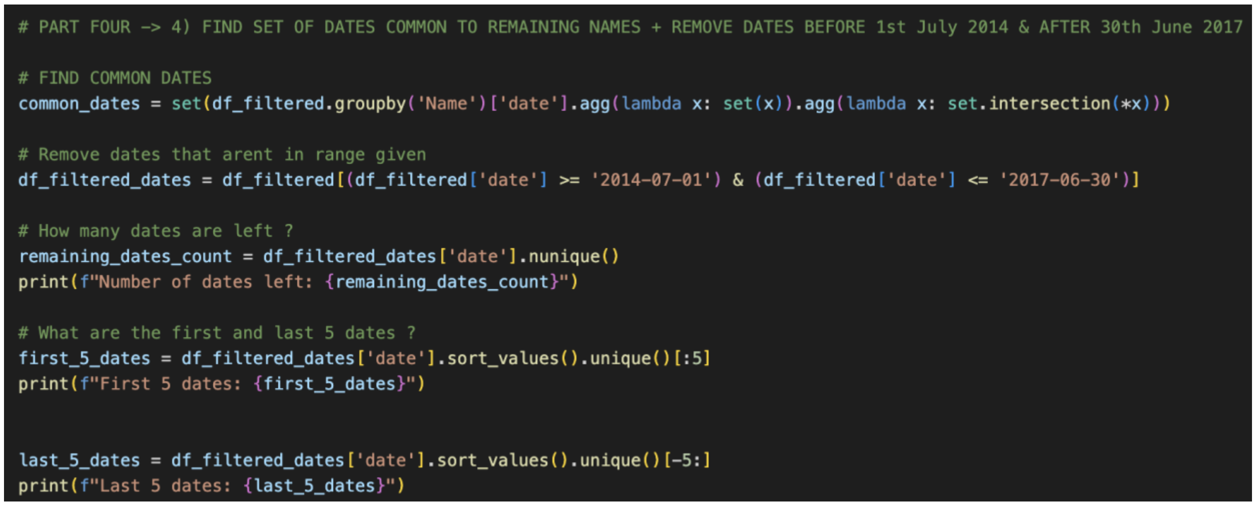
```
Removed names: ['APTV', 'BHF', 'BHGE', 'CFG', 'CSRA', 'DWDP', 'DXC', 'EVHC', 'FTV', 'HLT', 'HPE',
 'HPQ', 'KHC', 'PYPL', 'QRVO', 'SYF', 'UA', 'WLTW', 'WRK']
Number of names left: 486
Number of dates left: 757
```

Description:

In this code I filter the data according to the predefined parameters. This is done using the ‘groupby’ function to: for each name find the associated dates and if the first date is after 1st July 2014 or the last date is before 30th June 2017 the stock is removed from the dataset as there is not enough information on that stock.

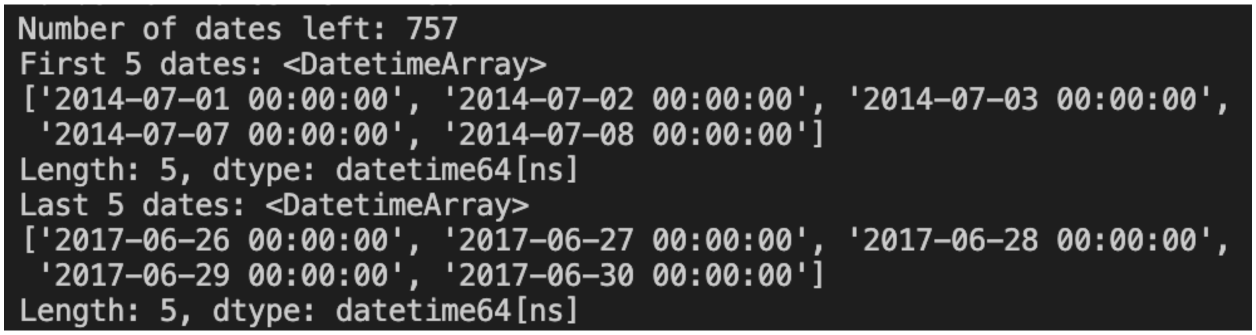
PART FOUR

Code:



```
# PART FOUR -> 4) FIND SET OF DATES COMMON TO REMAINING NAMES + REMOVE DATES BEFORE 1st July 2014 & AFTER 30th June 2017

# FIND COMMON DATES
common_dates = set(df_filtered.groupby('Name')['date'].agg(lambda x: set(x)).agg(lambda x: set.intersection(*x)))

# Remove dates that arent in range given
df_filtered_dates = df_filtered[(df_filtered['date'] >= '2014-07-01') & (df_filtered['date'] <= '2017-06-30')]

# How many dates are left ?
remaining_dates_count = df_filtered_dates['date'].nunique()
print(f"Number of dates left: {remaining_dates_count}")

# What are the first and last 5 dates ?
first_5_dates = df_filtered_dates['date'].sort_values().unique()[:5]
print(f"First 5 dates: {first_5_dates}")


last_5_dates = df_filtered_dates['date'].sort_values().unique()[-5:]
print(f"Last 5 dates: {last_5_dates}")
```

Output:



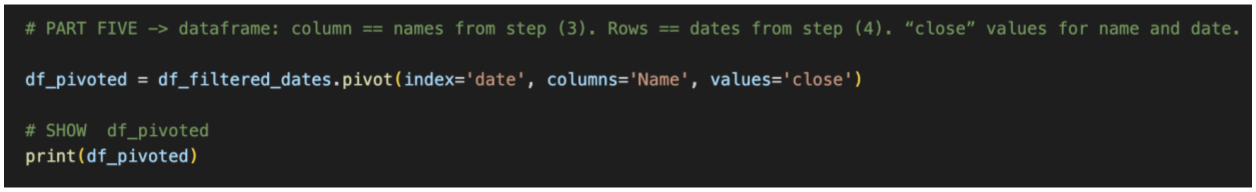
```
Number of dates left: 757
First 5 dates: <DatetimeArray>
['2014-07-01 00:00:00', '2014-07-02 00:00:00', '2014-07-03 00:00:00',
 '2014-07-07 00:00:00', '2014-07-08 00:00:00']
Length: 5, dtype: datetime64[ns]
Last 5 dates: <DatetimeArray>
['2017-06-26 00:00:00', '2017-06-27 00:00:00', '2017-06-28 00:00:00',
 '2017-06-29 00:00:00', '2017-06-30 00:00:00']
Length: 5, dtype: datetime64[ns]
```

Description:

During the fourth part of this assignment, I am trying to remove date outside the given range and the dates common to the remaining names. How do I do this? Again, I begin with grouping the data (names and the associated dates) and we are trying to find what is common amongst all of the dates. So, using a simple lambda agg function we can then find the intersection (commonality between all the dates). Then for the second part I just simple filter the data according to the criteria.

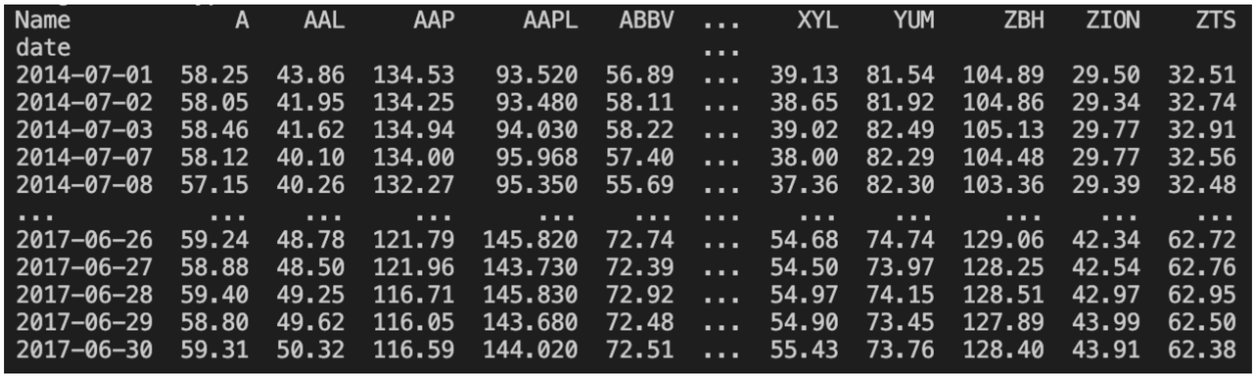
PART FIVE

Code:



```
# PART FIVE -> dataframe: column == names from step (3). Rows == dates from step (4). “close” values for name and date.

df_pivoted = df_filtered_dates.pivot(index='date', columns='Name', values='close')

# SHOW  df_pivoted
print(df_pivoted)
```

Output:



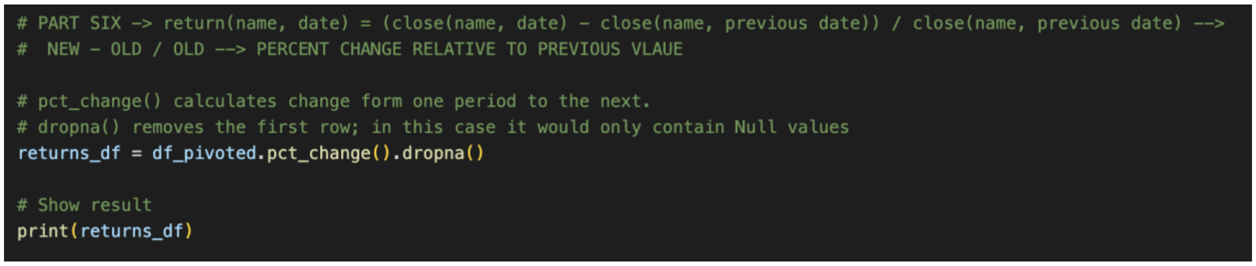
```
Name            A    AAL     AAP     AAPL   ABBV  ...    XYL    YUM     ZBH   ZION    ZTS
date                                              ...
2014-07-01  58.25  43.86  134.53   93.520  56.89  ...  39.13  81.54  104.89  29.50  32.51
2014-07-02  58.05  41.95  134.25   93.480  58.11  ...  38.65  81.92  104.86  29.34  32.74
2014-07-03  58.46  41.62  134.94   94.030  58.22  ...  39.02  82.49  105.13  29.77  32.91
2014-07-07  58.12  40.10  134.00   95.968  57.40  ...  38.00  82.29  104.48  29.77  32.56
2014-07-08  57.15  40.26  132.27   95.350  55.69  ...  37.36  82.30  103.36  29.39  32.48
...           ...    ...     ...      ...    ...  ...    ...    ...     ...    ...    ...
2017-06-26  59.24  48.78  121.79  145.820  72.74  ...  54.68  74.74  129.06  42.34  62.72
2017-06-27  58.88  48.50  121.96  143.730  72.39  ...  54.50  73.97  128.25  42.54  62.76
2017-06-28  59.40  49.25  116.71  145.830  72.92  ...  54.97  74.15  128.51  42.97  62.95
2017-06-29  58.80  49.62  116.05  143.680  72.48  ...  54.90  73.45  127.89  43.99  62.50
2017-06-30  59.31  50.32  116.59  144.020  72.51  ...  55.43  73.76  128.40  43.91  62.38
```

Description:

Pivot table with the rows being the dates and the columns being the names. And the data being the close days for each date.

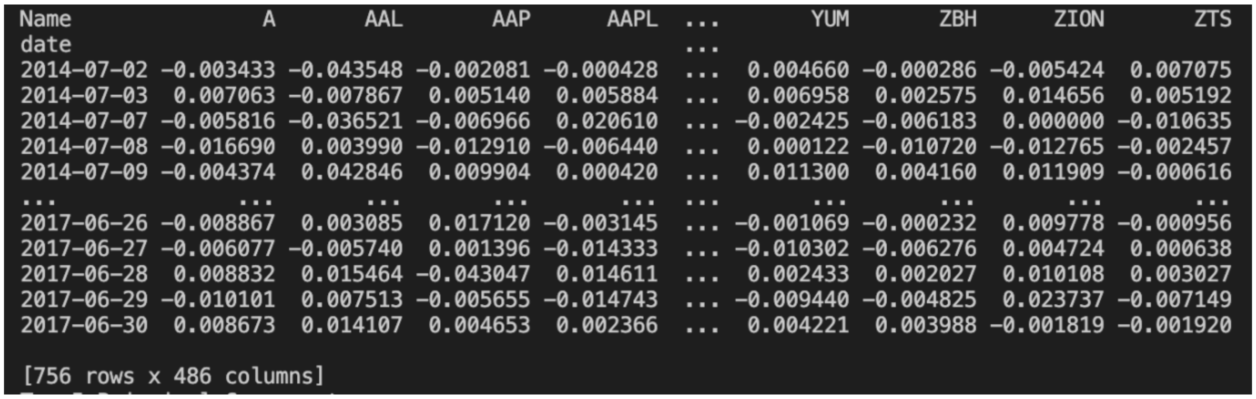
PART SIX

Code:



```
# PART SIX -> return(name, date) = (close(name, date) - close(name, previous date)) / close(name, previous date) -->
#  NEW - OLD / OLD --> PERCENT CHANGE RELATIVE TO PREVIOUS VLAUE

# pct_change() calculates change form one period to the next.
# dropna() removes the first row; in this case it would only contain Null values
returns_df = df_pivoted.pct_change().dropna()

# Show result
print(returns_df)
```

Output:



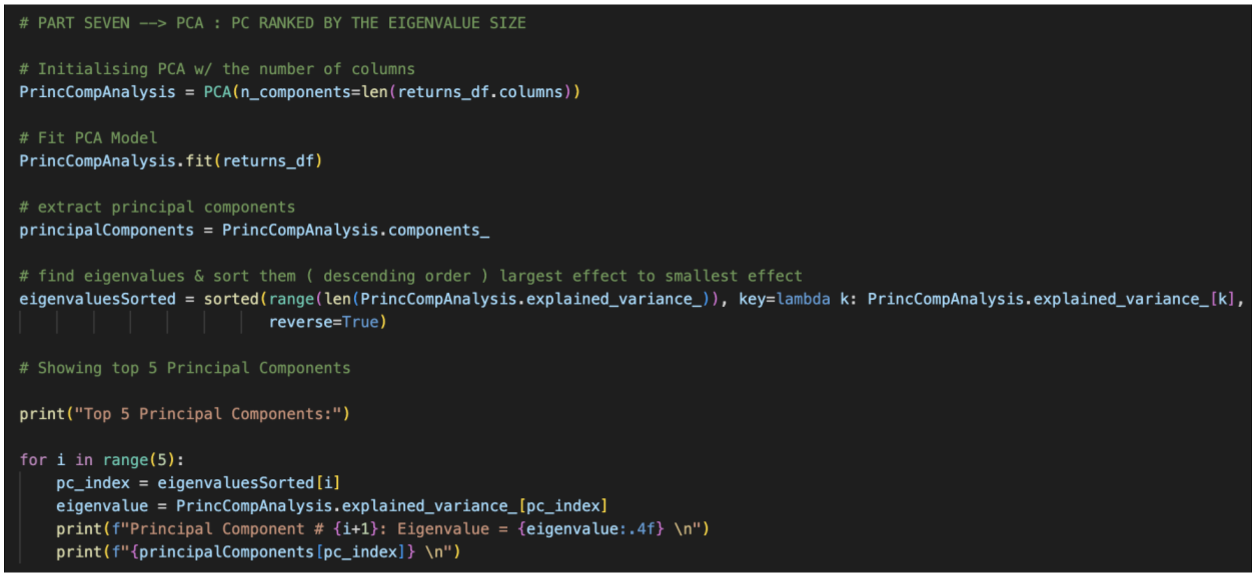
```
Name               A       AAL       AAP      AAPL  ...       YUM       ZBH      ZION       ZTS
date                                                ...
2014-07-02 -0.003433 -0.043548 -0.002081 -0.000428  ...  0.004660 -0.000286 -0.005424  0.007075
2014-07-03  0.007063 -0.007867  0.005140  0.005884  ...  0.006958  0.002575  0.014656  0.005192
2014-07-07 -0.005816 -0.036521 -0.006966  0.020610  ... -0.002425 -0.006183  0.000000 -0.010635
2014-07-08 -0.016690  0.003990 -0.012910 -0.006440  ...  0.000122 -0.010720 -0.012765 -0.002457
2014-07-09 -0.004374  0.042846  0.009904  0.000420  ...  0.011300  0.004160  0.011909 -0.000616
...              ...       ...       ...       ...  ...       ...       ...       ...       ...
2017-06-26 -0.008867  0.003085  0.017120 -0.003145  ... -0.001069 -0.000232  0.009778 -0.000956
2017-06-27 -0.006077 -0.005740  0.001396 -0.014333  ... -0.010302 -0.006276  0.004724  0.000638
2017-06-28  0.008832  0.015464 -0.043047  0.014611  ...  0.002433  0.002027  0.010108  0.003027
2017-06-29 -0.010101  0.007513 -0.005655 -0.014743  ... -0.009440 -0.004825  0.023737 -0.007149
2017-06-30  0.008673  0.014107  0.004653  0.002366  ...  0.004221  0.003988 -0.001819 -0.001920

[756 rows x 486 columns]
```

Description:

First row removed using dropna () function and the relative change gotten using pct_change () function.

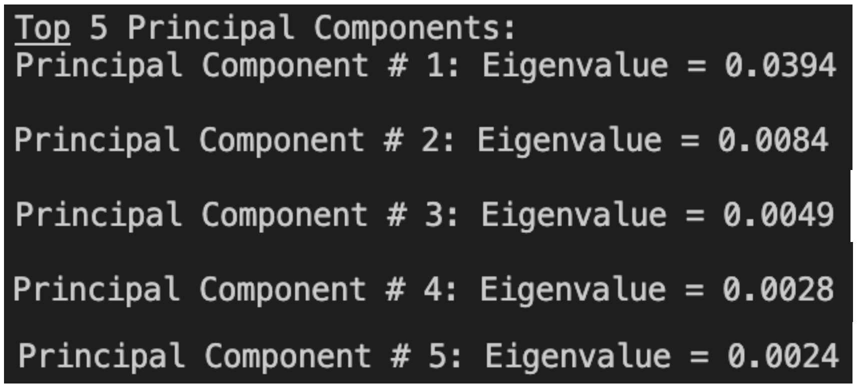
PART SEVEN

Code:



```
# PART SEVEN --> PCA : PC RANKED BY THE EIGENVALUE SIZE

# Initialising PCA w/ the number of columns
PrincCompAnalysis = PCA(n_components=len(returns_df.columns))

# Fit PCA Model
PrincCompAnalysis.fit(returns_df)

# extract principal components
principalComponents = PrincCompAnalysis.components_

# find eigenvalues & sort them ( descending order ) largest effect to smallest effect
eigenvaluesSorted = sorted(range(len(PrincCompAnalysis.explained_variance_)), key=lambda k: PrincCompAnalysis.explained_variance_[k],
                           reverse=True)

# Showing top 5 Principal Components

print("Top 5 Principal Components:")

for i in range(5):
    pc_index = eigenvaluesSorted[i]
    eigenvalue = PrincCompAnalysis.explained_variance_[pc_index]
    print(f"Principal Component # {i+1}: Eigenvalue = {eigenvalue:.4f} \n")
    print(f"{principalComponents[pc_index]} \n")
```

Output:



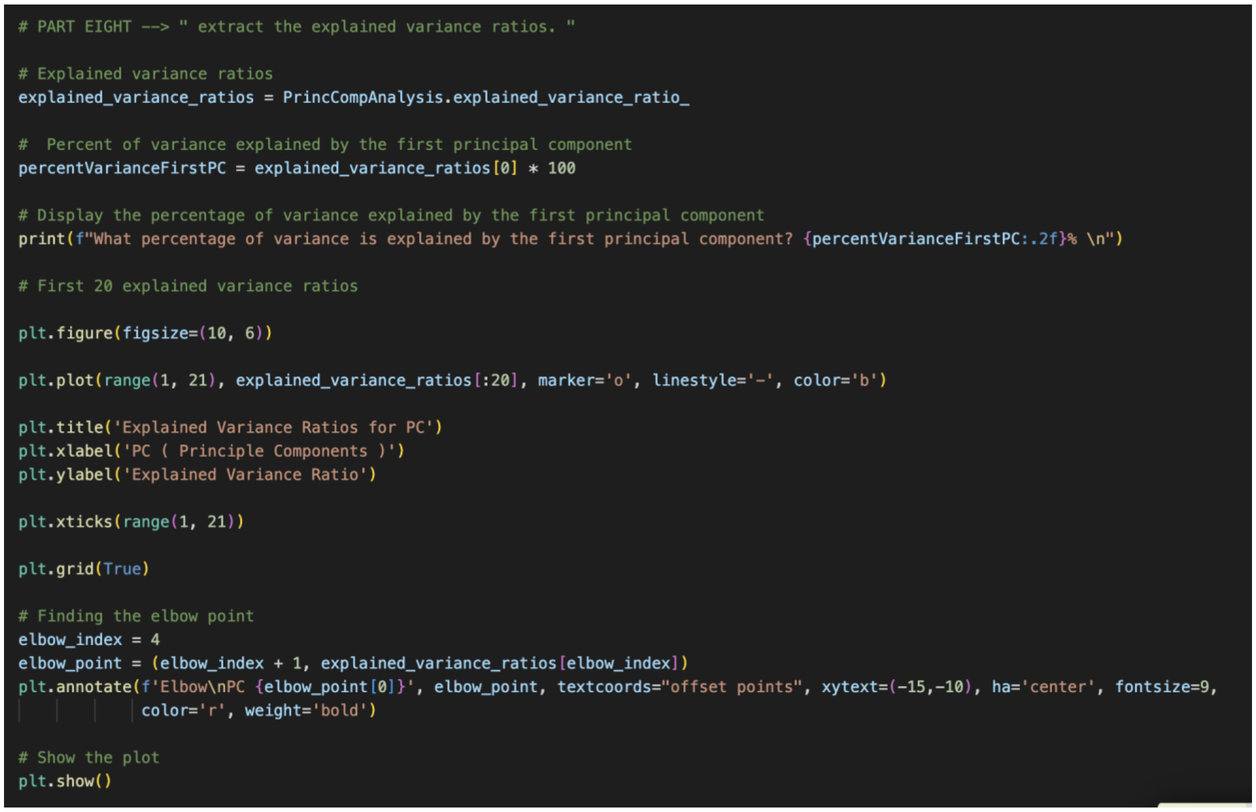
```
Top 5 Principal Components:
Principal Component # 1: Eigenvalue = 0.0394

Principal Component # 2: Eigenvalue = 0.0084

Principal Component # 3: Eigenvalue = 0.0049

Principal Component # 4: Eigenvalue = 0.0028

Principal Component # 5: Eigenvalue = 0.0024
```

Description:

PCA (Principal Component Analysis) is a statistical procedure that allows you to summarize the information content in large data tables by means of a smaller set of “summary indices” that can be more easily visualized and analyzed.

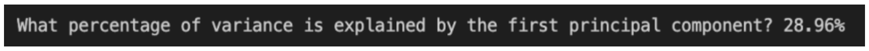
PART EIGHT

Code:

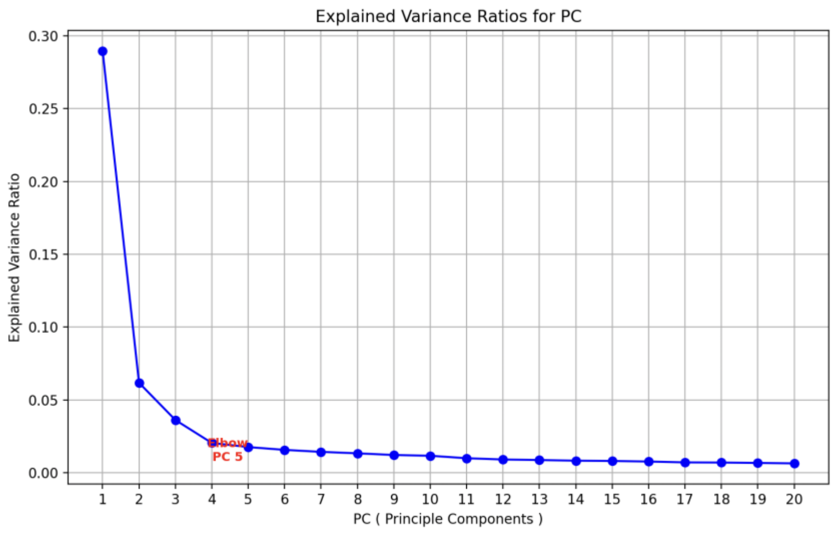


```python
# PART EIGHT --> " extract the explained variance ratios. "

# Explained variance ratios
explained_variance_ratios = PrincCompAnalysis.explained_variance_ratio_

#  Percent of variance explained by the first principal component
percentVarianceFirstPC = explained_variance_ratios[0] * 100

# Display the percentage of variance explained by the first principal component
print(f"What percentage of variance is explained by the first principal component? {percentVarianceFirstPC:.2f}% \n")

# First 20 explained variance ratios

plt.figure(figsize=(10, 6))

plt.plot(range(1, 21), explained_variance_ratios[:20], marker='o', linestyle='-', color='b')

plt.title('Explained Variance Ratios for PC')
plt.xlabel('PC ( Principle Components )')
plt.ylabel('Explained Variance Ratio')

plt.xticks(range(1, 21))

plt.grid(True)

# Finding the elbow point
elbow_index = 4
elbow_point = (elbow_index + 1, explained_variance_ratios[elbow_index])
plt.annotate(f'Elbow\nPC {elbow_point[0]}', elbow_point, textcoords="offset points", xytext=(-15,-10), ha='center', fontsize=9,
             color='r', weight='bold')

# Show the plot
plt.show()
```

Output:



```
What percentage of variance is explained by the first principal component? 28.96%
```



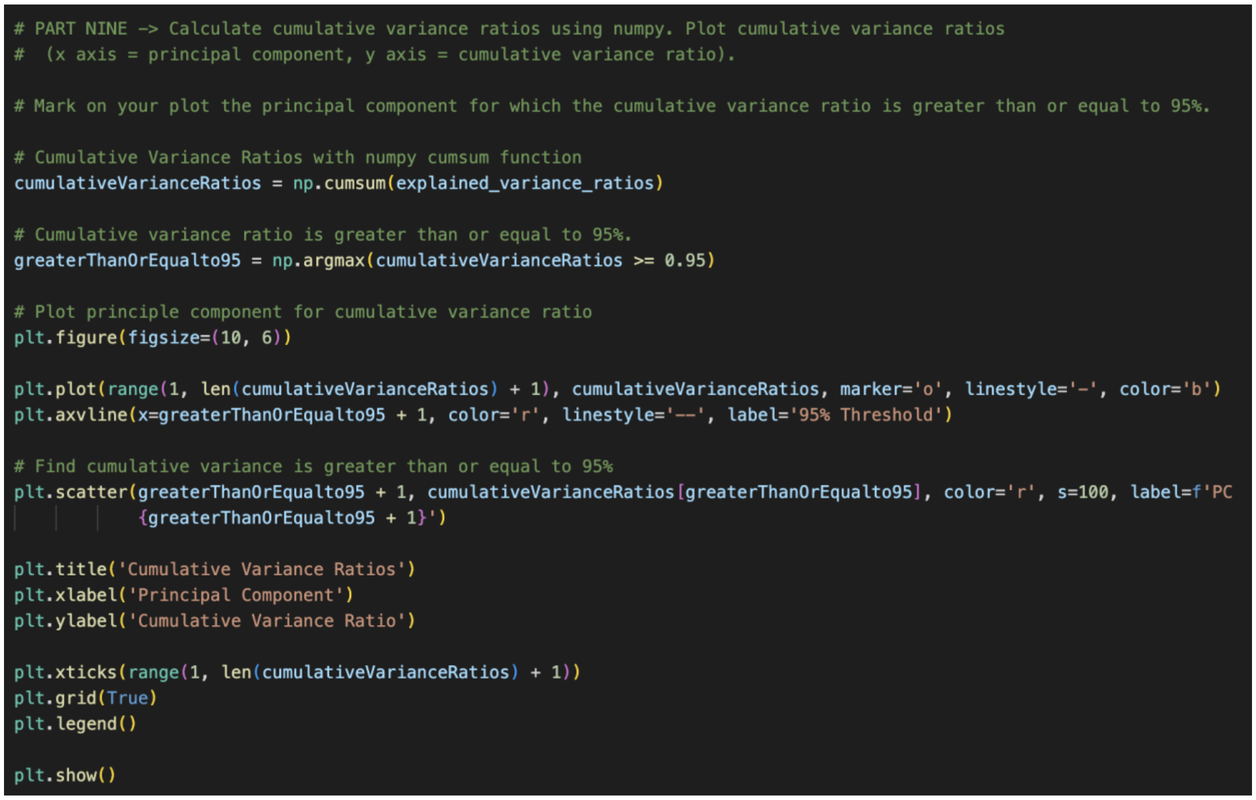
Description.

The code extracts the explained variance ratios from the PCA, which showcase the proportion of variance explained by each principal component. Then it simply calculates the percentage of variance explained by the first principal component and prints the result.

The code then creates a plot for the first 20 explained variance ratios, with markers, and lines connecting them and an elbow point in the plot is added using an index (elbow_index). showing where the explained variance ratio plateaus.
The resulting plot helps visualize the diminishing returns of adding more principal components, and the annotation all us to draw insight on the potential number of principal components to keep.

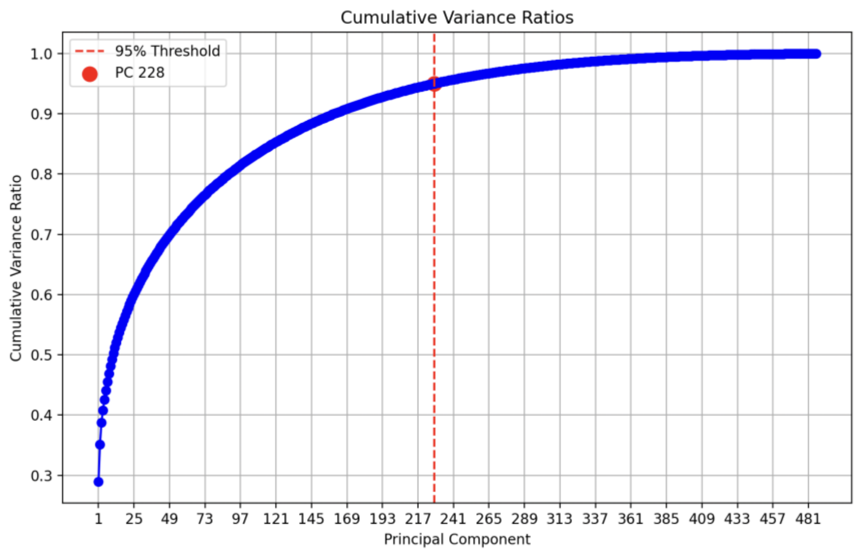
PART NINE

Code:



```python
# PART NINE -> Calculate cumulative variance ratios using numpy. Plot cumulative variance ratios
#  (x axis = principal component, y axis = cumulative variance ratio).

# Mark on your plot the principal component for which the cumulative variance ratio is greater than or equal to 95%.

# Cumulative Variance Ratios with numpy cumsum function
cumulativeVarianceRatios = np.cumsum(explained_variance_ratios)

# Cumulative variance ratio is greater than or equal to 95%.
greaterThanOrEqualto95 = np.argmax(cumulativeVarianceRatios >= 0.95)

# Plot principle component for cumulative variance ratio
plt.figure(figsize=(10, 6))

plt.plot(range(1, len(cumulativeVarianceRatios) + 1), cumulativeVarianceRatios, marker='o', linestyle='-', color='b')
plt.axvline(x=greaterThanOrEqualto95 + 1, color='r', linestyle='--', label='95% Threshold')

# Find cumulative variance is greater than or equal to 95%
plt.scatter(greaterThanOrEqualto95 + 1, cumulativeVarianceRatios[greaterThanOrEqualto95], color='r', s=100, label=f'PC
            {greaterThanOrEqualto95 + 1}')

plt.title('Cumulative Variance Ratios')
plt.xlabel('Principal Component')
plt.ylabel('Cumulative Variance Ratio')

plt.xticks(range(1, len(cumulativeVarianceRatios) + 1))
plt.grid(True)
plt.legend()

plt.show()
```

Output:

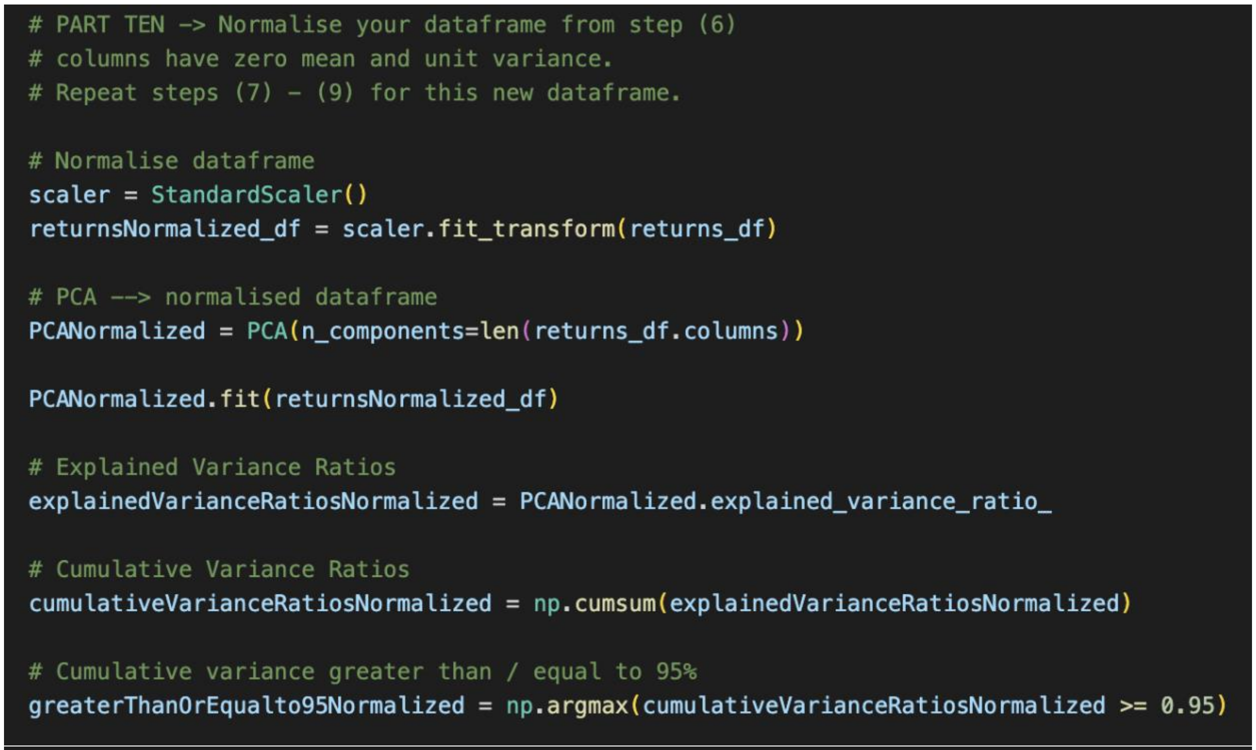


Description:

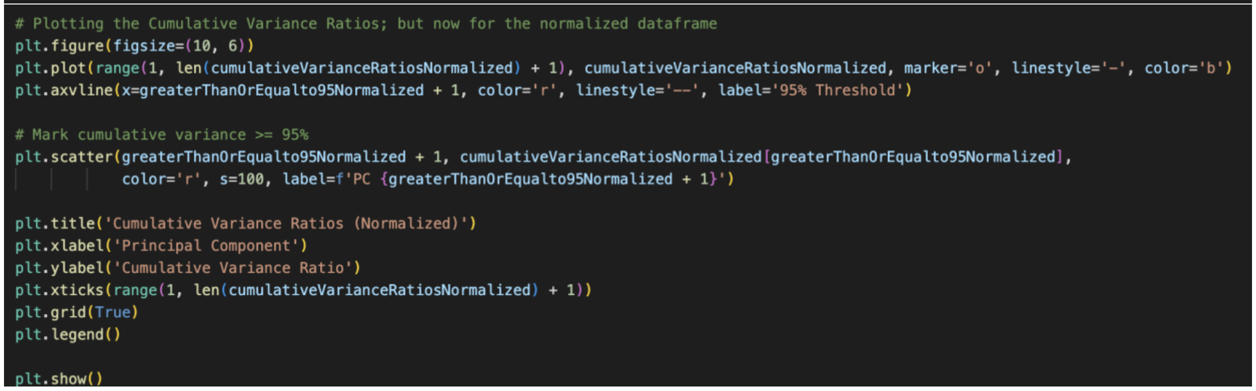
I use **numpy.cumsum** to calculate the cumulative sum of the explained variance ratios. I also used the **np.argmax** function to identify the principal component index where the cumulative variance ratio is greater than or equal to 95%. Finally, I create a plot of the cumulative variance ratios, with a red dashed line showing the 95% threshold and a red point marking the principal component when the threshold is seen.

<u>PART TEN</u>

Code:

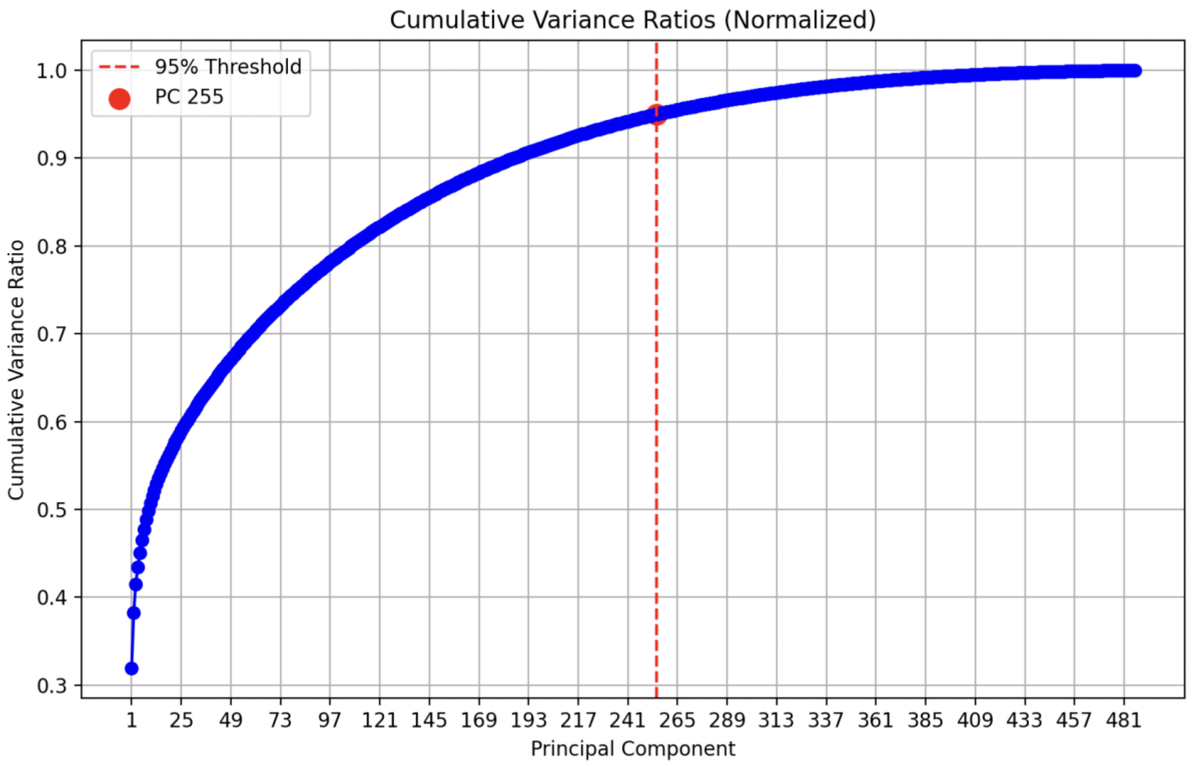


```python
# PART TEN -> Normalise your dataframe from step (6)
# columns have zero mean and unit variance.
# Repeat steps (7) - (9) for this new dataframe.

# Normalise dataframe
scaler = StandardScaler()
returnsNormalized_df = scaler.fit_transform(returns_df)

# PCA --> normalised dataframe
PCANormalized = PCA(n_components=len(returns_df.columns))

PCANormalized.fit(returnsNormalized_df)

# Explained Variance Ratios
explainedVarianceRatiosNormalized = PCANormalized.explained_variance_ratio_

# Cumulative Variance Ratios
cumulativeVarianceRatiosNormalized = np.cumsum(explainedVarianceRatiosNormalized)

# Cumulative variance greater than / equal to 95%
greaterThanOrEqualto95Normalized = np.argmax(cumulativeVarianceRatiosNormalized >= 0.95)
```



```python
# Plotting the Cumulative Variance Ratios; but now for the normalized dataframe
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(cumulativeVarianceRatiosNormalized) + 1), cumulativeVarianceRatiosNormalized, marker='o', linestyle='-', color='b')
plt.axvline(x=greaterThanOrEqualto95Normalized + 1, color='r', linestyle='--', label='95% Threshold')

# Mark cumulative variance >= 95%
plt.scatter(greaterThanOrEqualto95Normalized + 1, cumulativeVarianceRatiosNormalized[greaterThanOrEqualto95Normalized],
            color='r', s=100, label=f'PC {greaterThanOrEqualto95Normalized + 1}')

plt.title('Cumulative Variance Ratios (Normalized)')
plt.xlabel('Principal Component')
plt.ylabel('Cumulative Variance Ratio')
plt.xticks(range(1, len(cumulativeVarianceRatiosNormalized) + 1))
plt.grid(True)
plt.legend()

plt.show()
```

Output:



Description:

First, I normalize the returns of the data frame using StandardScaler function that then returns the normalized data frame. Then I perform PCA on the data frame; extract the explained variance ratios; calculate cumulative variance ratios; find 95% threshold of cumulative variance ratio; and finally plot the cumulative variance ratio. All of these are taken for their early instances, but now applied to the normalized data frame.