Question Answer:

2/ b)

What is the meaning of variables in the context of the data set I've used for my work?

- A variable in general is the column in a data set, of which describes the different data points provided in the set. In the case for my data set it's all the different Commodity Price Indexes (an index that measures the general price level of the commodities) and the dates of their recording. In a general summary a variable is also referred to as a characteristic or attribute for the observations in the dataset. An example from my dataset would be the Food Price Index(variable) for the date 1980-01-01(observation).

What is the meaning of observations in the context of the data set I've used for my work?

- Starting generally, observations are the rows provided in a dataset each having its own meaning. In the case of the dataset I chose to explore, it is a set of data consisting of the index prices for different commodities such as Industrial Inputs, Food Price ,etc… (each commodity (i.e. each column) would be referred to as a variable) for a specific date (example: January 1, 1980).

4-/

Brief description if df.shape (com.shape in my case) and df.describe() (com.describe() in my case)

- df.shape reports back the number of rows and columns regardless if they are full or empty (null or non-null). In other words it's not the size of the data in the dataset we are getting back, but rather just the size of the data frame all together.

- df.describe() on the other hand only reports back the summary statistics (like count, mean, standard deviation, etc.). Though we have to put in mind that this is only for columns that consist of numerical data. non-numeric columns with string/object data types (i.e. dates in my case) can not be processed only if explicitly specified.

The reason behind the discrepancy between df.shape (com.shape in my case) and df.describe() (com.describe() in my case):

- If you ever see discrepancies in the number of columns between df.shape and df.describe() is more than likely due to the column/s being non numerical. We need to remember that non numerical columns will be ignored by describe(). Only if it is explicitly stated that non-numerical data should be included, that describe() won't report it.
- Furthermore, if the count value for a column is less in description than the number of rows given by df.shape, it's due the count value in df.describe()

displaying the number of non-null entries for the column, not the total number of rows. In other words if the column has missing values in a column df.describe() won't include it in the count number.

6-\ the meaning of 'count', 'mean', 'std', 'min', '25%', '50%', '75%', and 'max' when the method describe() is used.

- **Count:** it's the number of non-null values that have been observed by the method. In other words the number of data points used, while not including null (missing) values in the count.

- **mean:** the average of the data points included a specific column. This is done through the sum of all the values divided by the count (number of values). The overall purpose is to have a central value of the data set which helps in understanding the overall value of the data.

- **std:** This refers to the standard deviation in a set, which is the measure of dispersion of values in the column. Std is also the square root of the average squared deviation of each value from the mean. This value helps in understanding the spread of the values in respect to the mean. A higher value given by std will indicate greater variability in the set (the opposite is also true).

- **min:** is the smallest value in the column. This is meant as a mean to help identify the lower ranges for your set of data.

- **25%:** the value of which below it 25% of the observations fall.Gives us an insight in the distribution of the lower quartile of the dataset.

- **50%:** the middle value or commonly referred to the median. The value of which the data set will be divided, half the values will be above and half will be below this given value. This value will represent the central value of the data and is less susceptible to outliers contrary to the mean.

- **75%:** the value of which 75% of all the data will fall below it. Gives us an insight in the distribution of the upper quartile of the dataset.

- **max:** this would display the largest value in the dataset.

7.

1. **Provide an example of a "use case" in which using df.dropna() might be preferred over using del df['col']**

- Let's say the data you have at hand is the number of delayed flights for airlines. The airline name would be the variables and the dates would be the observations. In this case if we use del df['col'] we lose a whole airline carrier from the dataset even though they are important for the overall analysis. So for this case it might be better to use the df.dropna() to remove any years where data might be missing. This will help us understand the overall performance of the airlines given that they have the same number of observations. Overall the point is that even though a column might be missing data, the column itself might contain useful data for other rows. With the removal of rows instead of columns we ensure the dropping of incomplete data in a set observation (row), without sacrificing entire information from a set variable (column).

2. **Provide an example of "the opposite use case" in which using del df['col'] might be preferred over using df.dropna()**

    - A counter example could be of the same airline dataset as I referred to above. Though this time you want to observe the overall delayed flights for a certain time period. In this case using del df['col'] to remove columns that don't meet that requirement might be adequate. If we were to use df.dropna() it might lead to the removal of observations that might have been vital to the overall picture of the analysis.

3. **Discuss why applying del df['col'] before df.dropna() when both are used together could be important**

    - del df['col'] before df.dropna() when both are used together is extremely important if you want to avoid unintentionally removal for entire rows that have missing values in that column. This could lead to the loss of important data from other columns that don't contain missing values in those rows. Though this all is dependent on what the intention is for the analyzation of the data.

4. **Remove all missing data from one of the datasets you're considering using some combination of del df['col'] and/or df.dropna() and give a justification for your approach, including a "before and after" report of the results of your approach for your dataset.**

- Before: my data set was missing a lot of observation points for certain variables, for example All Commodity Price Index Non-Fuel Price. So when I came to analyze the com.describe() I really couldn't really compare all the price indexes since the number of counts differed for each one. So I chose to use df.dropna() to help even the number of counts.

- After: the number of observations in my set of data decreased by about 144. Though when inputting com.describe(0 the only value that seemed to change was the mean. The rest of the values such as the std, 25% 50%, etc.. did not change. Though overall we even count the number between the different variables so if we ever want to compare the set values it will be possible, since all refer back to the same observation periods.

8-

1. **The function df.groupby("col1")["col2"].describe() helps us associate two sets of data to each other. In my case I will be explaining**

- df.groupby("sibsp")["fage"].describe(). The column sibsp ("sibsp") dataframe will be split up and turned into multiple groups (grouping the different data into observation points) . While column fare ["age"] will be split up and the data will be used to perform subsequent describe() operations such as the mean, min, max, std, etc… . So in summary df.groupby("sibsp")["age"].describe() will give you a detailed summary for ages values in retrospect of the different sibsp categories.

2. **Why do these capture something fundamentally different from the values in the count that result from doing something like df.groupby("col1")["col2"].describe()?**

- The difference between the two counts is that the count from df.describe() is the total sum of all the numeral data points for a particular variable/column. In df.groupby("col1")["col2"].describe(), the count is different in the sense that the column itself is split up into different categories (observation points), so the count is for the set of data variables in the column that fit a certain category. So if we add all the counts in df.groupby("col1")["col2"].describe() we will get the count given by col2 in df.describe().

LINK TO CHATBOT:

CB1:

https://chatgpt.com/share/2a1376b1-bc21-4261-98d7-20564d02d33f

CB2:

https://chatgpt.com/share/51cf4e34-8e95-4412-bdbf-eadc3fd30aaa

CB3:

https://chatgpt.com/share/4b5a9122-4426-404a-a9b7-f80e476256b1