Module 3: Introduction to Data Analysis

Quick Reference Guide

Learning Outcomes:

1. Discuss real-world contexts for the data science lifecycle
2. Manipulate data given a dataset
3. Generate visualizations using pandas, Seaborn, and Plotly
4. Customize plots using externally sourced documentation
5. Use pandas.groupby to manipulate data
6. Perform computations between dataframes using set index and reset index

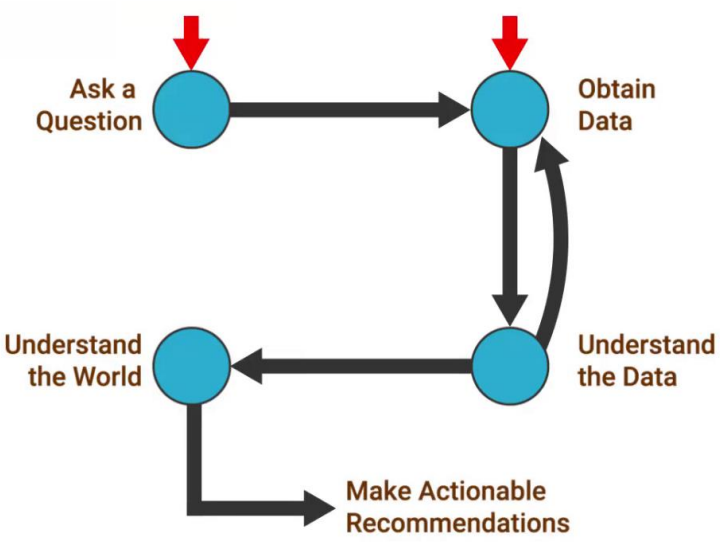
The Data Science Lifecycle

Building a model to solve real-world problems is a two-step process:

- Acquire and clean data to input into a model
- Use the data to train a model

For instance, to predict the sale price of a house, you can take the available data on the number of bedrooms, bathrooms, the location, the quality of the house's construction, and generate an estimate.

In the real world, the process is less linear. This is illustrated by the **data science lifecycle**.



The data science lifecycle usually **starts with a question**. For example, what will the sale price of a given house be? Next, we **find the data that we must understand**. What are the rows and columns of the dataset? How much of the data is useful? What data is missing? For example, after a first-pass analysis of the housing data, we might obtain crime statistics to improve the model. The goal is to **understand something about the world** and **make actionable recommendations or decisions**. There are two entry points into this lifecycle—asking a question and obtaining data. Sometimes we start with data, but no clear question. For example, we might have climate data and we want to make predictions about how the climate will change and recommendations to steer or prevent these changes.

pandas Basics

Some basic functions in pandas include:

- **pd.read_csv()**: A function that instructs pandas to read the imported CSV file.
- **df.head(n)**: A function that returns the first n rows of a table.

- **df.tail(n)**: A function that returns the last n rows of a table.
- **df.sample(n)**: A function that returns a random sample of n rows of a table.
- **df.["new column name"] = df["existing column name"] and an operator with its calculation**: A query that performs a specified calculation to return a new set of column values. The operator (+, -, *, and /) and calculation must be specified in each instance. For example, with df.["GDP"] = df["GDP (2010 US$)"] / 1_000_000, the new column references an existing column and divides those values by a million.
- **df[df["column name"] == "cell value"]**: A query that returns all rows with a particular cell value within the specified column.
- **df.["column name"] == "cell value"**: A query that returns a series of true/false values for each row, where the result is true if the cell value matches the cell value specified in the function and it is false when it does not.
- **(df.["column name"] == "cell value").iloc[row number 1:row number 2]**: A query that returns a series of true/false values for each row, where the result is true if the cell value matches the cell value specified in the function and it is false when it does not. The iloc parameter specifies the rows to be searched.
- **df[(df["column name 1"] == "cell value 1") & (df["column name 2"] == cell value 2)]**: A query that returns only the result(s) that conform to the search parameter specified by the Boolean arrays.
- **df.query()**: A query that compares the columns of a dataframe to a Boolean expression. A concise version of the query above.
- **name of list = ["cell value 1", "cell value 2", "cell value 3", "cell value 4"]:** A function to specify a list in which queries should be run.

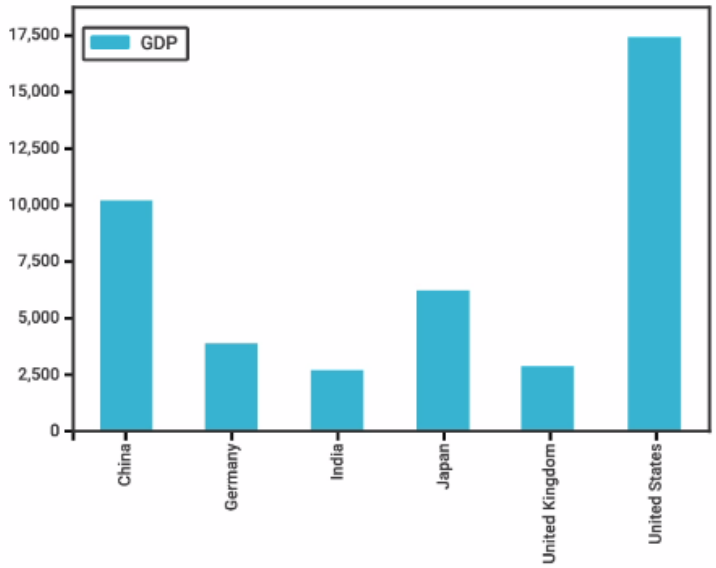
- **df[df["column name"].isin(name of list)]:** A query that returns rows of interest related to a specified list. The list name is first specified and then the query is run.
- **df.query('column name in @name of list')**: A query that returns rows of interest related to a specified list. The list name is first specified and then the query is run.

Introduction to Visualization Libraries

Visualizations are used to understand and better interpret data.

The function **df.plot()** from the built-in **pandas** plotting library is used to visualize data by representing it as diagrams or charts. When calling the plotting function on the dataframe, you specify the values to plot along the X- and Y axes, and instruct the function on the type of plot to create, for instance, a bar or line plot.

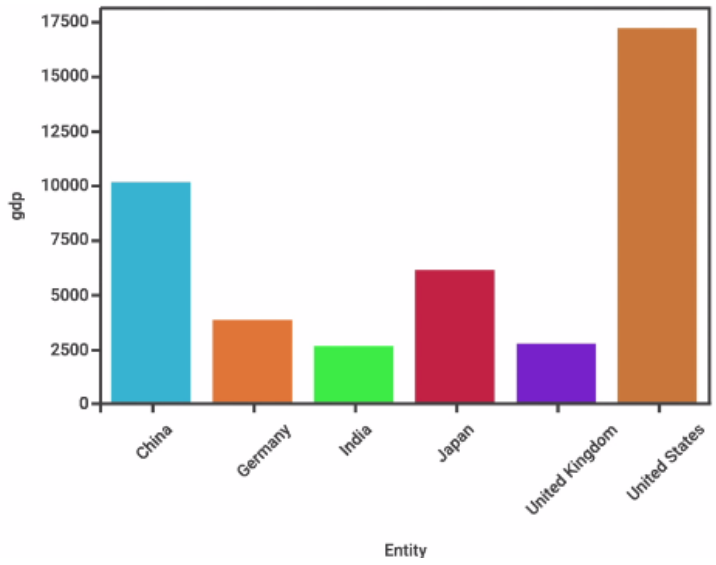
This is an example of a visualization created using the built-in pandas plotting functionality.



Here is a link to the pandas dataframe plotting libraries documentation: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html

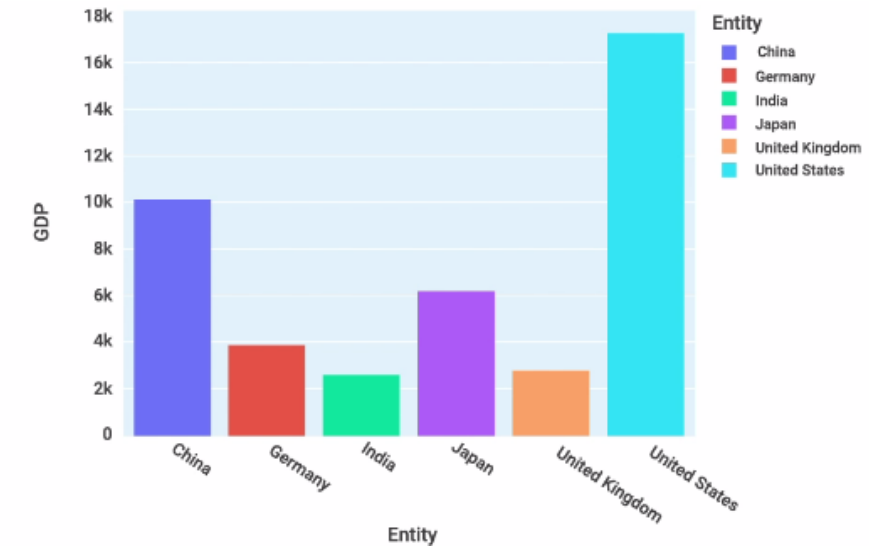
Seaborn is another popular plotting library that can be used to create statistical or standard plot types, such as bar and line plots. The bar plotting function in Seaborn is **sns.barplot()**. Seaborn is built on top of Matplotlib, a plotting library that uses Python to create static, animated, and interactive visualizations — so it is worth noting that advanced use of Seaborn requires Matplotlib knowledge to improve the appearance of plots.

This is an example of a visualization created using the Seaborn plotting functionality.



Plotly is powerful visualization library with attractive graphs, good documentation, and helpful interactivity features. But on the downside, its statistics visualization is suboptimal and it has a smaller support community. The function to plot bar charts in Plotly is **px.bar()**.

This is an example of a visualization created using Plotly.



Aggregation Operations

When working with real-world datasets, often the data may require some preprocessing to clean, understand, or reorganize it. You can use the **groupby()** operation in pandas to subdivide the data and then combine it anew or 'aggregate' it based on the questions you are asking as part of the data science lifecycle.

The **groupby()** operation does not involve any loops or iterations and is based on aggregate operations. With this style of programming, the declarative paradigm, the programming language does all the iteration for you.

Aggregation operations in pandas include:

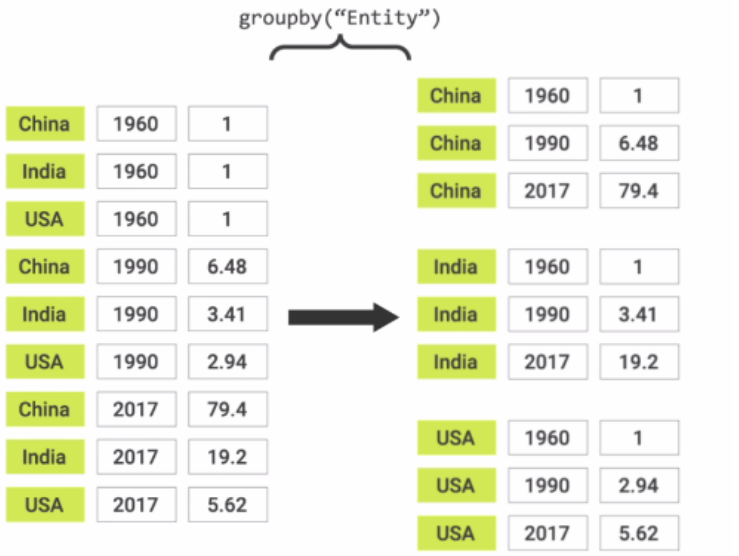
- **groupby()**: Creates a dataframe to present information more concisely.
- **df.groupby().agg(sum)**: Returns an aggregated dataframe with summed data for a given series.

- **df.groupby().agg(sum)["column name"]**: Returns only the aggregated data for the column specified. Series can be created with [] notation. Dataframes can be created with [[]] notation.
- **df.groupby().agg(max)**: Returns an aggregated dataframe with the maximum data in a given series.
- **df.groupby().agg(min)**: Returns an aggregated dataframe with the minimum data in a given series.

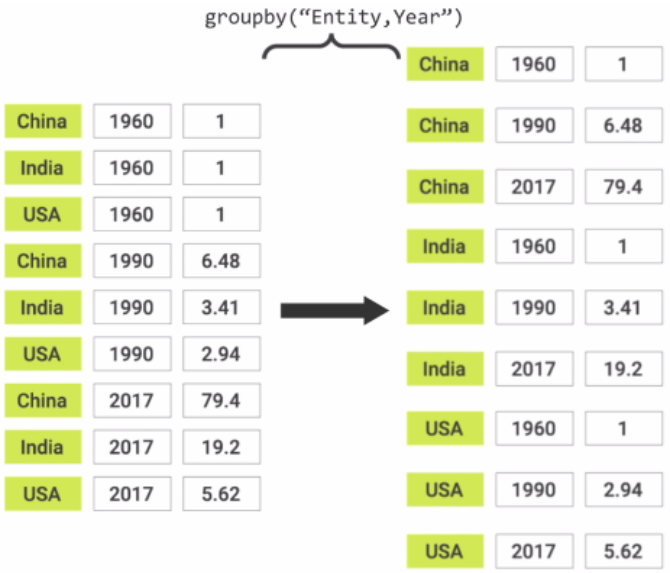
The **agg()** function can take any arbitrary function to aggregate the data. For instance, you can create a function that computes the ratio from largest to smallest in a given series and then combine this with the **agg()** function. Additionally, you can use the **line()** function to create a line plot in pandas to visualize the data.

A Visual Depiction of Aggregation Operations

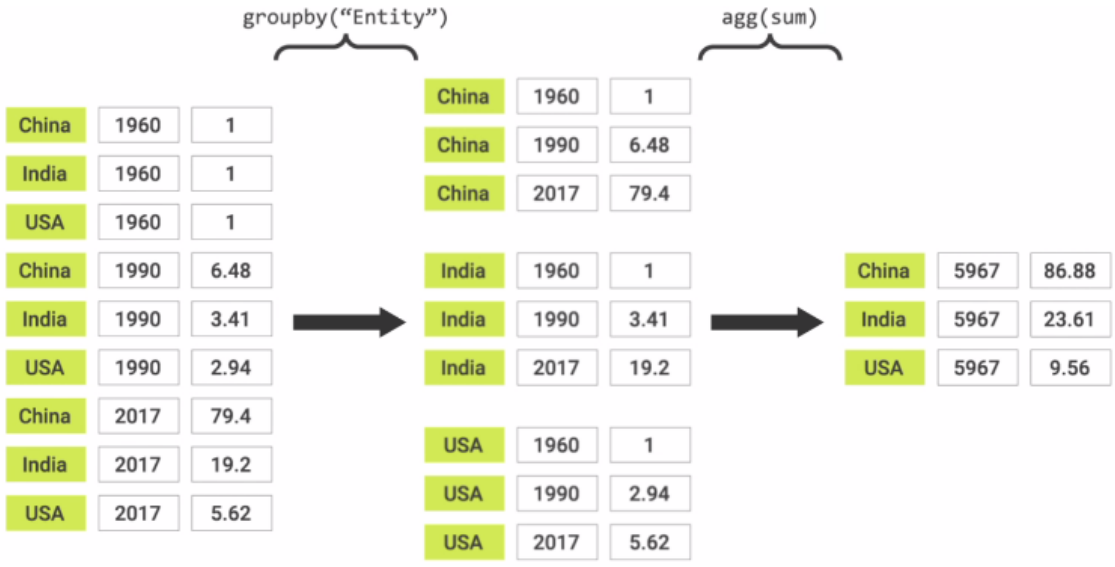
The following illustrations showcase how **groupby()** and **agg()** work.



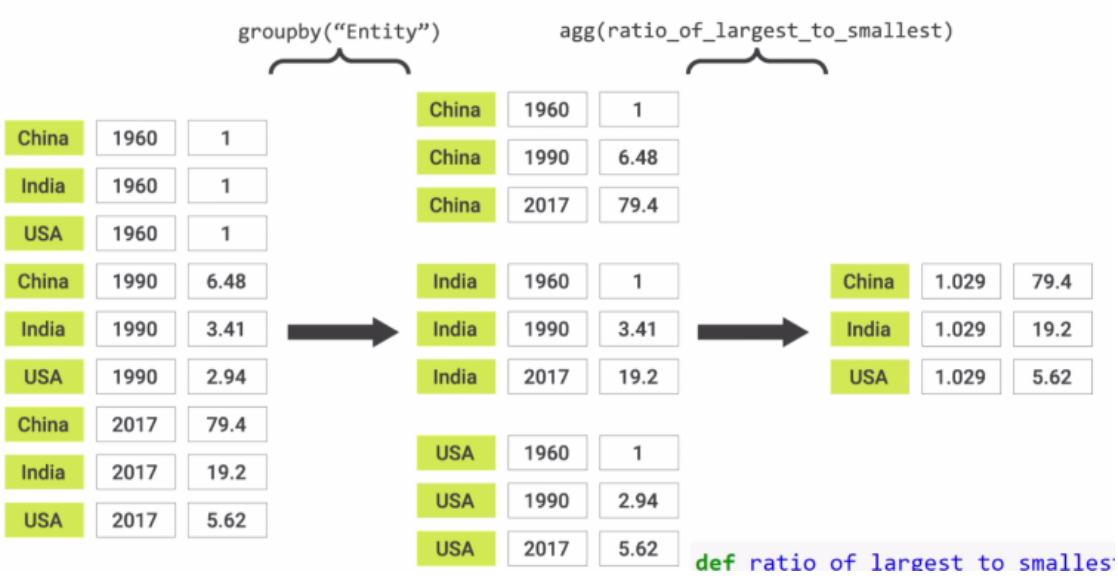
If you call **groupby()** to retrieve an entity, imagine that pandas reorganizes the data to cluster rows with the same entity. The data is not actually moved around, but it is a convenient fiction to illustrate the groupby operation. Effectively, you can assume the original dataframe is broken into sub dataframes, each of which contains all rows corresponding to a single entity.



If grouped by multiple columns, then each sub dataframe would be a group of rows, where all the selected columns are equal.



If you call **agg(sum)** on the results of the groupby operation, pandas looks at each sub dataframe and computes a single value for each column that represents the entire dataframe. For example, for the China sub dataframe, the year value of 5,967 is just the sum of all years in the sub dataframe and the GDP ratio is the sum of all GDP ratio values in the sub dataframe. Note that this example exists purely to showcase how the **groupby()** and **agg()** functions work and the resulting table has no practical value.



You can use a different function to aggregate the data; for instance, the ratio of largest to smallest. Now representatives for the year and GDP ratio are 2017 divided by 1960, or 1.029 and 79.4, divided by 1, or 79.4. As before, this output table is not useful for any practical purpose and is provided simply so you can see how **groupby()** and **agg()** work in an example

Sorting and Aggregating

Data can be sorted and aggregated to answer questions. For instance, why did the world's smallest economy suddenly get smaller in 1990? The naive

approach is to take the entire dataframe, group it by year, and then call **agg(min)**. But in this example, the results indicate that in 1960 Algeria had a GDP of 0.098736. The bad news is this does not reflect the GDP of Algeria in the year 1960.

If you run **df.query()**, you find that Algeria had a GDP of 27 billion in 1960. The issue is what **groupby("Year").agg(min)** is doing is computing the minimum entity in the year 1960. And since Algeria came alphabetically earliest, it is the one getting picked.

So how do you figure out the smallest economy in the year 1960? First, you sort the dataframe by the GDP using **sort_values()**. The results indicate that the smallest economy was Tuvalu in 1990. Next, you can consider each year and get the first value in that year. Since the dataframe is ordered by GDP, the first value for a given year is the smallest. To get the first item, you can define a function that retrieves the first item from a series and use **return s.iloc[0]**.

Next, if you use **df.sort_values().groupby().agg(first item)**, you'll get a table with the first item in each sub dataframe. The results, in this example, indicate that Belize was the smallest economy in 1960 and you can conjecture that at some point Tuvalu entered the dataset and was not there before. To confirm this conjecture, you can take a look at Tuvalu by running **df.query()**. The result indicates that data for Tuvalu was only added to the dataset in 1990.

You do not need to create your own function to get the first item of a series. You can use the built-in **groupby.first()** instead of the **agg()** function. For instance, **df.sort_values().groupby().first()**. There are also **max(), min()**, and **sum()** functions that you can incorporate in a similar manner.

Indexing

As another demonstration of the **groupby()** feature in pandas, consider a problem: What fraction of the world's GDP did each country generate in a given year? For example, in 1995, what fraction of the world's economy did the United States occupy?

You may think you can compute the total GDP for each year and divide each country's GDP by the total GDP. But if you do, the result is not a number (NaN). The reason is because the total GDP that was calculated does not have the same structure as the original data. The total GDP is indexed by year whereas the original dataframe is indexed by an arbitrary number. As a result, when pandas tries to divide the two tables, it gets confused. The solution is to set the index of each country using **set_index()**. For instance, you can set the index of the original data to year, similar to the total GDP.

Just note, whenever you set the index of a dataframe it does not change the original dataframe. Rather, it creates a copy that is indexed by what you specify. You need to create a new dataframe equal to **set_index()** to make the change permanent.

But if you want to set the index for all countries simultaneously, you need to do **multi-indexing**. For instance, if you wanted to set the index of an example for both year and country, you specify it in the brackets — **set_index("Year", "Country")**. With multi-index, pandas considers all the values you specified as important for organizing the data. Now, when you divide the dataframe with each country's GDP by the dataframe with the total GDP, you get the exact result you want.

Next, you can take the resulting dataframe and reset the index using **reset_index()**. This resets the index to the original format from the imported CSV, but the column with the GDP ratio you calculated remains in the table.

You can visualize the dataframe using **px.line().** If it is too messy, remember you can choose to visualize certain data within the dataframe. For instance, only the top six economies.

It is important to understand data through visualization and querying because you want to look out for anomalies. One of the most difficult tasks when doing machine learning or data science is ensuring you have nice, clean data. Exploratory visualizations can really help you understand what is going on with your data.

Another Groupby and Aggregation Example

Here is another example of how to use **groupby()** to solve interesting problems. Let us answer the question: By what fraction has each economy grown in a given year? This is done to determine which country has grown the most.

First, the naive approach, along with its resultant complications. If you run **query(),** you get six countries of interest from the dataframe.

Next, you imagine that you should multi-index – for instance, setting the index for country and year using **set_index().** Then you can use **groupby().first()** to find the first values for each country. Since the CSV file was ordered by year, you end up getting all the earliest years as a table. The naive solution divides the reset table by the groupby table to deliver the result you want. While that may work in principle, there is a flaw.

For instance, Germany only has data from 1970 onward. The German data will be skewed because it is not comparing Germany's current economy to its 1960s value, but to its 1970s value. Using the naive approach, you might end up with an analysis that is deceptive.

To avoid this problem, you can create a new temporary table that only has the 1960s values. You can get the GDP value in 1960 for each country, indexed by country, using **df.query().set_index()**. Then, if you take the table indexed by country and year and divide it by the new table with only the 1960s values, you get a GDP ratio. Now, you can verify how each country's economy has grown or shrunk in comparison to its 1960s value.

If you try to plot this data using **px.line()**, you get an error. The error is because you have not reset the index. For instance, the x you provided, year, is not a column in the dataframe. You can only use columns in your dataframe. So, you should reset the index for the table with the top six countries using **reset_index().** But remember, you need to assign a copy to make a change. Thereafter, you can plot the data.

Filtering

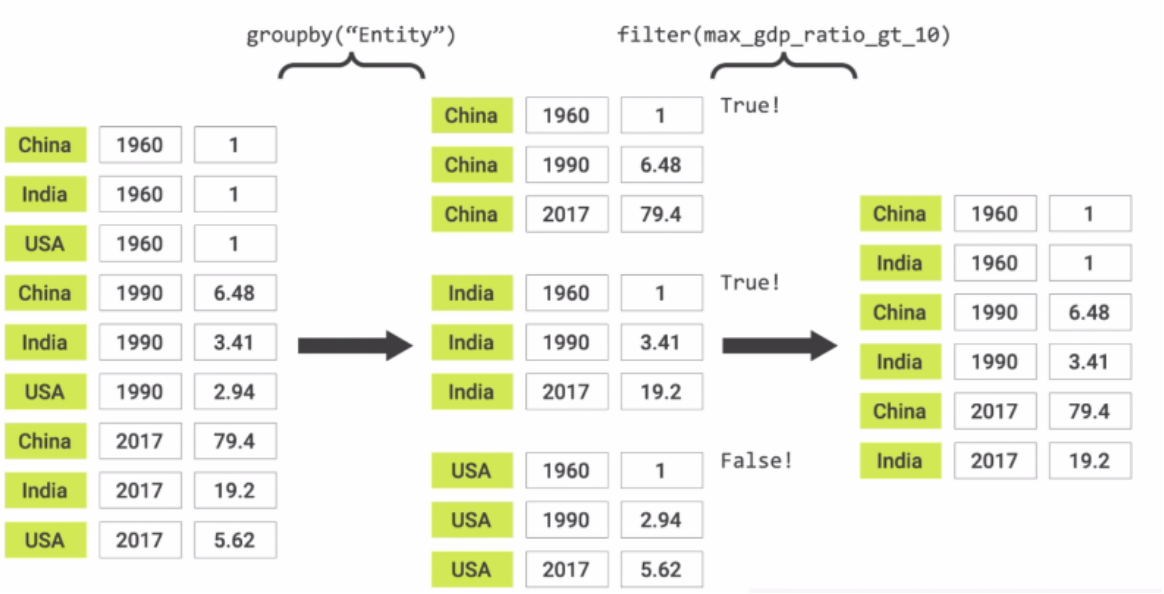
Filtering is another operation to process your data. Suppose you want to see which country has grown the most.

First, you set the index for the entire dataframe using **set_index()**. Some of the results may be not a number (**NaN**) because there is no 1960s value. If this occurs, you can drop the rows with NaNs using **dropna**() and get a table with no NaNs. Using various query operations, you can determine which countries are missing. That may be important if you are compiling a report.

You can then create a plot using **px.line()** to visualize the ratio of the current size of each country's economy to its 1960 value.

Now suppose you want to generate a plot with only the countries that are at least ten times as large as their 1960 value. To do this, you create a function that returns true if the maximum GDP ratio is greater than ten, otherwise, it returns false. Once you have written this function, take the dataframe, group it by country, and filter it for countries where the maximum GDP ratio is greater than ten using **df.groupby().filter()**. The resulting table will only have countries with a GDP that is ten times as large as their 1960 value. Then if you generate the plot using **px.line()**, it only keeps the countries which have grown by at least ten times since 1960.

Here is an illustration that showcases how **groupby()** and **filter()** work together.



The **groupby()** operation effectively breaks the original dataframe into sub dataframes. When you call **filter(),** pandas creates a new dataframe according to the specified rule. For each sub dataframe, pandas computes

the results of the function passed to the filter operation. If the result is true, the filter command keeps that sub dataframe. But if the result is false, the filter command will reject that sub dataframe. Ultimately, the output of the filter command will be the union of all of the sub dataframes that were kept. Note that this process has no effect on the original dataframe. In other words, the rejected rows are not removed from the original dataframe.

Conclusion

In this module you have learned how to:

- Read, query, visualize, aggregate, and filter tabular data using pandas. In demonstrations, only CSV files were used, but other formats work in pandas using the same basic principles.
- Create visualizations using the pandas, Seaborn, and Plotly visualization libraries.
- Perform operations with pandas.groupby to aggregate and filter data. Groupby operations are more efficient than iterative loops.
- Set and reset the indices of dataframes, allowing for convenient operations between dataframes.