Matplotlib

Intro to data visualization: -

Why visualize data?

- Data visualization allows us to **bring your data to life**: -
    - The human brain is built to interpret raw data as meaningless numbers and noise.
    - We need clear patterns and visual to help us quickly make sense of complex info.
- Humans are visual creatures by nature, we can identify and distinguish colours and patterns visually before our prefrontal cortex will catch up that is used for sensing raw data.
- Ideally in 10 sec your audience should be able to understand what your data visualization is trying to communicate.

The 3 key questions?

- The 3 key questions are a great way to help choose the right visual: -
    - **What type of data are you working with?** E.g. time series, categorical data, numeric data, hierarchical data.
    - **What do you want to communicate?** E.g. comparisons, composition, distribution, relationship.
    - **Who is the end user and what do they need?** E.g. analyst, manager, executive.

Essential visuals: -

- The essential visuals that all the analysts need to know how to produce are:-
    - **KPI card** → just simple text works best.
    - **Pie chart** → useful for determining how categories in our data contribute to the composition of another metric. These can be very controversial.

- **Table** → when we add colour-based formatting, often called a heat map, we can really start to extract some really interesting insights. Adding colour scale is essential for quickly understanding which are highest and lowest values in our table.
- **Line chart** → useful for determining trends over time.
- **Bar chart** → most basic chart used for comparisons.
- **Scatter plot** → it is helpful for determining the relationships between numeric variables like spend and store visits.
- **Area chart** → it combines the concept of composition and time series.
- **100% stacked** → if we want to take a look at how these categories contribute to the whole over time or across some other categories, we use this. 100% stacked bar charts show the composition between the categories in each bar, and allow for comparison with the rest of the bars.
- **Histogram** → very common and useful for looking at distributions.

Chart formatting and storytelling: -

- Chart formatting is critical to make sure that our audience or end user can **really understand the data** we are trying to present.
- Chart formatting should be **used to reduce noise and facilitate the understanding** on the part of our end user and audience.
- **Chart formatting** should be used to eliminate noise and facilitate understanding.
    - Remove the chart border and gridlines
    - Format the axis labels clearly
    - Add context with the chart title
- And depending on the purpose of the chart and the context in which we are going to be sharing our charts with others, we might want to take our charts a step further and really brings a storytelling aspect.
- **Descriptive titles** and **data labels** can be used to **tell a clear story** within your visuals.
    - Leverage the title to guide the audience towards specific insights.
    - Insert text and shapes directly inside the chart.
    - Use data labels and annotations to draw attention to the main data points.
    - Use colour strategically.

Common visualization mistakes: -

- Choosing the wrong visualization to represent the type of data.
    - E.g. using a line chart which is meant for time-series data, with categorical data gives the false sense of a trend.
    - **Treemap** is generally used for hierarchical data. While a Treemap can work, comparisons and compositions are harder to make than with a bar or pie chart. Its best to use them with hierarchical data.
    - Bar charts are great for showing comparisons and categorical data.
- Including too many series in a single visual.
- Providing **little to no context** with text and labels.
- Using **inconsistent colours** between related visual. Using different colours for the same series makes it difficult to associate them visually. Using the same colour makes them easier to understand.
- Consistency gains more importance as the number of visuals increases, making it critical for dashboards.

Key takeaways: -

- Always answer the 3 questions to choose the right visual. What type of data are you working with. What do you want to communicate. Who is the end user?
- Do not prioritize variety over effectiveness. Choose the chart types based on how clearly, they communicate the data underneath.
- Eliminate noise and distractions to facilitate understanding.
- Tell a story with the data to guide the user to the insights. Use titles, strategic labels and callouts to create a clear narrative.

Matplotlib fundamentals: -

Intro to Matplotlib: -

- **Matplotlib** is an open-source python library built for data visualization that lets you produce a wide variety of highly customizable charts and graphs.
- It is built as an open-source implementation of MATLAB plotting functions.
- For creating charts first, we import the matplotlib library as: -
    - **Import matplotlib.pyplot as plt**
- We will be using the **pyplot library** almost exclusively here.
- Then we will call the **plot() method** from the **piplot module** passing a list of integers and a line chart is plotted below: -
    - **Plt.plot([0,1,4,9,16,25])**
- The **.plot()** function **creates a line chart by default**, using the **index as the x-values** and the **list elements as y-values**.
- The data types that are compatible with matplotlib are: -
    - Matplotlib can plot many data types including **base python sequences (lists, tuples), numpy arrays and pandas' series and dataframes**.
- **plt.show()** → it is used to display the chart once we make it and it is used if we are not working in the jupyter notebook env.
- **plt.plot(labels, data);** → this is how we make line plots where labels are displayed on the x-axis and data on the y-axis.

Plotting methods: -

- There are 2 primary methods for building plots within matplotlib are: -
    - **PyPlot API** → charts are created with the **plot()** function and modified with additional features.
    - **Object-oriented** → charts are created by defining a **plot object**, and modified using the figure & axis methods.
- The **object-oriented approach** gives us more control over what's going on inside of our figures and helps us achieve extreme customization.

- To create a chart using the object-oriented method: -
    - **Import matplotlib.pyplot as plt**
      fig = plt.figure()
      ax = fig.add_subplot()
      ax.plot(y)
    - Another method
      fig, ax = plt.subplots() → this will create the figure and axis
      ax.plot(y) → plots "y"
      fig.suptitle('Overall title') → add a title to the figure
      ax.set_title('Chart title') → add a title to the axis
- So, for most part in the course, we are going to focus on the **object-oriented approach** as it provides more clear control over customization.
- **Object-oriented** plots are built by adding axes, or charts to a figure.
    - The **.subplots()** function lets us create the figure and axis in the single line of code.
    - We can then use the figure and axis methods to customize the different elements in the plot.

Plotting dataframes: -

- As analyst and data scientist, the most commonly used data structure we are going to use when analysing data is the **pandas dataframe**.
- When **plotting dataframes** using the object-oriented interface, Matplotlib will use the **index as the x-axis** and plot each **column as a separate series** by default.
- The general syntax of plot method is: -
    - **ax.plot(x-axis series, y-series values)**
- we can plot the dataframes like: -
    - **fig, ax = plt.subplots()**
      ax.plot(ca_housing.index, ca_housing['San Franscisco'])
      ax.plot(ca_housing.index, ca_housing['Los Angeles'])
- So, specifying your X and Y, even though we don't always need to specify that helps us in better customization and formatting of the plots.
- Its very helpful when we set our index to date when doing plotting, especially with line charts.
- We can use the **ls=" " parameter** inside the **.plot()** method to specify the line style of the chart.

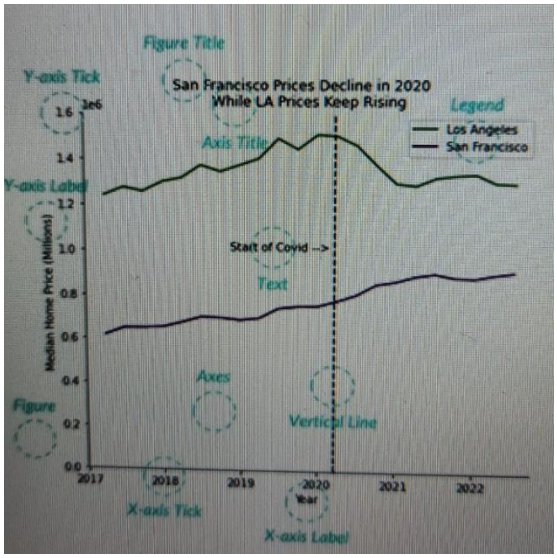
Assignment learnings: -

- **pandas to_datetime()** method helps to convert string Date time into Python Date time object.
- **pandas.to_timedelta()** function is used to convert argument to datetime. E.g. **pd.to_timedelta(hotels["DaysSinceCreation"], unit="D")** converts the DaysSinceCreation column into a Timedelta object, with each value representing a duration in days.
- **hotels["date"] = (pd.to_datetime("2018-12-31")- pd.to_timedelta(hotels["DaysSinceCreation"], unit="D")).astype("datetime64")** → in this code we are adding a column "date" to the dataframe.
- **daily_revenue.resample('M').sum()** → this will convert the daily revenue to the monthly revenue using the **resample() method** and the output is given on monthly basis.
- fig, ax = plt.subplots()
  ax.plot(daily_revenue) → this is how we plot the graph.
- **openpyxl is the python** library that is used to read the excel files in the **.xslx format**.

Anatomy of matplotlib figure: -

- Matplotlib has these formatting options for **pyplot** and **object-oriented** plots.
- Matplotlib defaults are pretty basic and oftentimes not the most visually pleasing.
- Some of the formatting options that can fill out our chart and make them effective visualization.
- The **fig** → is the blank canvas. This is the visualization object or image that we are going to layer everything else onto.
- The **axes** → it is a **layer or chart onto the figure**. And we can layer multiple charts onto the same figure. Once we built the chart, we're going to format the chart elements from there.
- The **figure title** → this is centered above the chart or at the top of the figure in the middle of the figure.
- The **axis title** → if we wanted to have the main headline, or main title with a subtitle then we can set a **figure title** and then add an **axis title** to that.

- The **y-axis label** → adding a y axis label is almost necessary for every single visualization.
- The **x-axis label** → it is often not necessary as a x-axis label, depending on what your x axis is.
- The **legend** → once we labelled all the axes and title the chart, a lot of charts also requires the legend. So which line represent what so we can add that by legend.
- We can also set the x and y axis limits using the methods **ax.set_xlim()** and **ax.set_ylim()**.



Option	Object-Oriented	PyPlot API
Figure Title	fig.suptitle()	plt.suptitle()
Chart Title	ax.set_title()	plt.subtitle()
X-Axis Label	ax.set_xlabel()	plt.xlabel()
Y-Axis Label	ax.set_ylabel()	plt.ylabel()
Legend	ax.legend()	plt.legend()
X-Axis Limit	ax.set_xlim()	plt.xlim()
Y-Axis Limit	ax.set_ylim()	plt.ylim()
X-Axis Ticks	ax.set_xticks()	plt.xticks()
Y-Axis Ticks	ax.set_yticks()	plt.yticks()
Vertical Line	ax.axvline()	plt.axvline()
Horizontal Line	ax.axhline()	plt.axhline()
Text	ax.text()	plt.text()
Spines (borders)	ax.spines['side']	plt.spines['side']

Chart titles and font sizes: -

- It is one of the most important pieces of chart formatting i.e. the **chart titles** and the **axes labels**.
- The **set_title()** and **set_label()** methods allow us to set the chart titles and axis labels.
- The **fig.suptitle()** method serves as an overall figure title. This can be helpful in creating a big headline with our chart title being a sub headline and when we move to multiple figures then it will be centered at the top of the figure.
- We can also modify the font sizes of our titles and labels with the help of **fontsize argument** in almost all of the methods. We can specify the size in points (10,12) or relative size (“smaller”, “x-large”).

- The **parse_dates parameter** in pandas is used when reading data from files (such as CSV, Excel, etc.) to automatically parse and convert specified columns into datetime objects.
- **str.contains(“ ”)** → this method is used to check if the string contains a particular string or not.
- **.assign()** → this method is used to add column to the dataframe.
- When **building line charts in particular** building a pivot table using the **pivot_table()** dataframe method is super helpful.

Chart legends: -

- The chart legend helps us to communicate what each series or line plotted in our chart represents.
- The **legend() method** allows use to add a **chart legend** to identify each series.
    - The series labels are used by default, but the custom values can also be passed through.
- **ax.legend(["SF", "LA"])** → this is how we can pass our custom legends in the **legend() method**.
- We can also change the location of legend with the **“loc”** or **“bbox_to_anchor” arguments**.
    - **“loc”** lets you set a predetermined location option. Location options are **best(default), upper left, upper right, upper center, lower right, lower left, lower center, center, center left, center right.**
    - **“bbox_to_anchor”** lets us set the specific (x,y) coordinates.
- **ax.legend(["SF", "LA"], loc="lower right")** → this is how we can change the location of the legend.
- **ax.legend(["SF", "LA"], bbox_to_anchor = (0,1), frameon=False)** → this is how we use the bbox_to_anchor parameter. The **frameon=False parameter** will remove the border around the legends.

Line style: -

- we can change the line style with the **“linestyle”, “linewidth” and “color” arguments.**
    - Common line styles are “solid”, “dashed”, or “dotted” (we can also use “-“, “- -“, or “:”
    - **ax.plot(**
      ca_housing_markets.index,
      ca_housing_markets['Los Angeles'],
      label = "Los Angeles",
      ls = ":",
      color = "purple",
      linewidth = 2) → this is how we can change the linestyle, linewidth and color in the plot.

Axis limits: -

- The **set_ylim()** and **set_xlim()** functions let you modify the **axis limits**.
    - **ax.set_xlim(lower limit, upper limit)**
- In general, we don’t tend to modify the X limit nearly as much as we’ll modify the Y limit.
    - **ax.set_xlim(17500, 19000)**
    - **ax.set_ylim(0, 1600000)**
- This is how we se the X limit as well as the Y limit.
- One thing that happens when we modify our X limit is that our X axis ticks may change interval size.
- Keeping the base of Y axis at 0 highlights the true magnitude of change across periods and the difference between series.

Figure size: -

- Another helpful formatting option is modifying the size of the figure.
- We can adjust the figure size with the **“figsize” argument**.
    - **figsize = (width, height)** – the default is 6.4 * 4.8 inches.
- Changing the figure size will increase the amount of space on our charts.
- **figsize = (7, 6)** → this is how we can set the figure size by mentioning it inside the **subplot() method**.

- Increasing the figure size is a great way to add white space without needing to change the format of things like x tick labels.

Custom X-Ticks: -

- There are some times when we want to customize our X axis ticks in particular. Usually, we don't change the Y axis ticks as much.
- We can apply custom X-ticks with the **set_xticks()** and **xticks() method**.
    - **ax.set_xticks(iterable)**
- This is how we set the X ticks and rotate them to see them clearly: -
    - **ax.set_xticks(ca_housing_markets.index[::12]);**
      plt.xticks(rotation = 45);

Adding vertical lines: -

- We can add the vertical lines to mark key points with the **axvline() function**.
- **ax.axvline(18341, ymin = .6, ymax = .4, color = "black", ls="--", label="Important date");** → this is how we plot the vertical line in the chart by mentioning the **days from 1 january 1970**.
- Using a vertical line can help show up before and after period and allow audience to really understand.
- If we want to get the number of days since 1970 then we can find it like: -
    - **min(ca_housing_markets.index) - datetime.datetime(1970,1,1)**
- We can also mention the length of the line using the parameters **ymin, ymax**.

Adding text: -

- A lot of the times we might want to add additional text beyond our chart titles or access labels or legends to give the audience more context.
- We can **add text** at specific coordinates with the **text() function**.
    - **ax.text(x-coordinate, y-coordinate, string, additional text formatting).**
- x-coordinate and y-coordinate is from where the text is going to start.
- **ax.text(17850, 1000000, "Start of covid -->")** → this is how we use the text() method to add text to our chart.

- **fig.text(.95, .8, "This chart shows \n the severe drop in \n prices in san francisco", fontsize = 16, color = "red")** → this is how we can also mention text about what our figure is about to give more descriptive understanding to the user.

Text annotations: -

- Previously we have put text into our chart using the **.text()** method.
- Another way to do this that will give us a little prettier arrow is using **annotations**.
- **Annotations** are a great way to call-out and label important datapoints.
    - **ax.annotate(string, datapoint coordinate, text coordinate, arrow style dictionary, text formatting).**
- This is how we use **ax.annotate()**: -
    - **ax.annotate("Start of covid",**
      xy=(18300, 1300000),
      xytext=(17600, 1300000),
      arrowprops=dict(facecolor="black",
      width=1,
      headwidth=8,
      connectionstyle = "angle3, angleA=270, angleB=0"),
      verticalalignment="center")

Removing Borders: -

- It is helpful in cleanup our chart appearance and add a little bit of visual polish.
- We can remove specific chart borders with **ax.spines[ ].set_visible(False).**
- This is how we can remove borders from the chart we plot: -
    - **ax.spines['right'].set_visible(False)**
      ax.spines['top'].set_visible(False)

Line charts: -

- Line charts are **used for showing trends over time**.
- To create a line chart, we use the method: -
    - **ax.plot(x-axis series, series values, formatting options)**
- Pro tip for setting up the dataframe to create a line chart properly is pivot your tabular data to turn each unique series into a dataframe column, and **set the datetime as the index**.
- Divide your series by the appropriate units while plotting to simplify the y-axis scale.

Stacked Line charts: -

- It is a variation of the line chart.
- We use the **.stackplot()** method to create a **stacked line chart**, which lets you visualize overall trend over time as well as its composition by series.
- Use the bottom series in the stacked line chart to draw focus to its individual trend, it's the most visible.
- This is how we plot the **.stackplot()**: -
    - **ax.stackplot(**
      ca_housing_pivot.index,
      ca_housing_pivot["Los Angeles"],
      ca_housing_pivot["San Diego"],
      ca_housing_pivot["San Francisco"],
      labels = labels,
      colors=colors
)

Dual axis charts: -

- We use the **.twinx()** method to create dual axis chart, which lets us plot series with values on significantly different scales inside a single visual.
- We will add the legend to the figure level which will allow use to pick up both the series even if they are plotted on separate axes like **fig.legend().**

- This is how we use the .twinx() method: -
    - **fig, ax = plt.subplots()**
      ax.plot(sd_dual["inventory"],
      label="Inventory",
      c = "orange")
      ax.set_ylim(0,10000)

      ax2 = ax.twinx()

      ax2.plot(sd_dual["median_active_list_price"],
      label="Price")
      ax2.set_ylim(0,1000000)
      fig.legend(bbox_to_anchor=(.35,.25))

Bar charts: -

- **Bar charts** are use the compare values **across different categories**.
- To use the bar chart, we use the method: -
    - **ax.bar(category label, bar heights, formatting options)**.
- Use the **groupby()** and **add()** to aggregate our data by category and push the labels into the index.
- One way to structure our dataframe is to groupby() the category you want to perform some aggregation().
- Use the **.axhline() method** to add a horizontal line at a specified y-value on a bar chart.
    - This will typically be something to benchmark against, like mean or target.
- Use the **.barh() method** to create a horizontal bar chart.
- Use the **'color' argument** to highlight the series you'd like to focus on.
- **ax.bar(total_sold.index, total_sold['total_homes_sold'], color = colors)** → this is how we plot the bar charts.

Stacked Bar charts: -

- In matplotlib, we literally going to build stacked bar charts by stacking our series from different categories together to produce a single bar.
- We can create a **stacked bar chart** by setting the **“bottom” argument** for the **second “stacked” series** as the values from the bars below it.
    - This will use those values as the baseline for the stacked bar series instead of the x-axis.
- This is how we plot the stacked bar chart: -
    - **fig, ax = plt.subplots()**
      ax.bar(ca_or.index, ca_or["CA"], label="California")
      ax.bar(ca_or.index, ca_or["OR"], label="Oregon", bottom = ca_or["CA"])
      ax.bar(ca_or.index, ca_or["WA"], label="Washington", bottom=ca_or["CA"]+ca_or["OR"])
      ax.set_title("Unit sales of Homes in CA and OR")
      ax.set_ylabel("Homes sold")
      ax.legend()
- To create a **100% stacked bar chart**, convert the dataframe to row-level percentages before plotting.
- Pandas **.cut() function** is used to separate the array elements **into different bins**. The syntax of the .cut() function is: -
    - ***cut(x, bins, right=True, labels=None, retbins=False, precision=3, include_lowest=False, duplicates=”raise”,)***
- The **pd.query()** method Query the columns of a DataFrame with a boolean expression. The **query() method** takes a query expression as a string parameter, which has to evaluate to either True of False. It returns the DataFrame where the result is True according to the query expression.

Grouped Bar charts: -

- The difference in grouped bar chart and the stacked bar chart is that with **stacked bar chart**, we are stacking the values that each subcategory contributes to **create a single bar**.
- With grouped bar chart we are allowing there to be one bar for each of our underlying categories without stacking them.

- We can create the **grouped bar chart** by **reducing the width of each series** and shifting them evenly around their corresponding label.
- These are much easier to create by using **seaborn** or **pandas matplotlib API**.
- **ca_or.plot.bar()** → this is how we create grouped bar chart using the pandas matplotlib API.
- This is how we plot grouped bar chat using matplotlib: -
    - **fig, ax = plt.subplots()**
      width = .25
      x1 = np.arange(len(ca_or))
      x2 = [x + width for x in x1]
      ax.bar(
        x1,
        ca_or["CA"],
        label = "California",
        width = width
)
      ax.bar(
        x2,
        ca_or["OR"],
        label = "Oregon",
        width = width
)

Combo charts: -

- Combo charts **mean it has multiple charts in it**.
- We can create a **combo chart** by specifying **different chart types in a dual axis plot**.
- All we need to do to create a combination chart with one line chart and one bar chart plotted on a left and right axis is the same thing. But instead of calling the line chart function twice, we are going to call our line chart function once and bar chart function once.
- Its very clear to your audience that a bar and a line are extremely visually different, and that's going to trigger and immediate response that, these are different pieces of information being plotted.

- If we want to produce a dual axis chart with very different pieces of information, then use the combination chart where one is a line and one is a bar.
- The **alpha parameter** represents the **data transparency,** and by default alpha equals to 1, which is solid completely visible. Alpha = 0 will set the data to be completely invisible.
- **countries_subset_pcts = countries_subset.apply(lambda x: x*100 / sum(x), axis = 1)** → this is how we convert a chart into 100% stacked chart.
- This is how we plot combo charts: -
    - **fig, ax = plt.subplots()**
      width = 200
      ax.plot(
      sd_dual2["median_active_list_price"],
      color="green",
      alpha = .4,
      label = "Price"
)
      ax.legend()
      ax2 = ax.twinx()
      ax2.bar(
      sd_dual2.index,
      sd_dual2["inventory"],
      width = width,
      alpha = .3,
      label = "Inventory"
)
      ax2.legend()

Pie and Donut charts: -

- **Pie charts** are used to compare proportions totalling 100%. What % of the pie each category takes up. To create pie, we use the method: -
    - **ax.pie(series value, labels = , statrangle=, autopct = , pctdistance=, explode=)**
- keep the number of slices in the pie chart low to enhance readability. We can group “others” into a single slice.

- If we are going to compare categories, we should use bar charts. Pie charts are really only useful for showing how they make up the whole.
- We can create a **donut chart** by **adding a "hole" to a pie chart** and shifting the labels.
- The **pandas.concat()** function does all the heavy lifting of performing concatenation operations along with an axis of Pandas objects while performing optional set logic (union or intersection). This is how we use **.concat() display(pd.concat([series1, series2])).**
- This is how we create a pie chart: -
    - **fig, ax = plt.subplots()**
      ax.pie(
        x = sales_total["total_homes_sold"][::-1],
        startangle=90,
        labels=["San Francisco", "San Diego", "Los Angeles"],
        autopct="%.0f%%",
        explode = (0, .12, 0)
)
- This is how we create a donut chart: -
    - **fig, ax = plt.subplots()**
      ax.pie(
        x = sales_total["total_homes_sold"][::-1],
        startangle=90,
        labels=["San Francisco", "San Diego", "Los Angeles"],
        autopct="%.0f%%",
        explode = (0, 0, 0),
        pctdistance=0.85
)
      hole = plt.Circle((0,0), 0.70, fc = 'white')
      fig = plt.gcf()
      fig.gca().add_artist(hole)

Scatter plot and bubble charts: -

- **Scatter plots** are used to visualize the relationships between numerical values. To build the scatter plots we use the method: -
    - **ax.scatter(x-axis series, y-axis series, size=, alpha=)**
- Scatter plots tend to be one of the chart types where the aggregation isn't always needed.
- Modify the alpha level to make overlapping points more visible. Bubble charts can be useful in some cases, but they often add confusion rather than clarity.
- To create a **bubble chart**, specify a third series in the **"size" argument** of .scatter().
- This is how we plot the scatter plots: -
    - **fig, ax = plt.subplots()**
      ax.scatter(
      la_housing["months_of_supply"],
      la_housing["median_active_list_price"]
)
      ax.set_title("Price vs supply in LA")
      ax.set_xlabel("Months Of Supply")
      ax.set_ylabel("Median List Price")

Histogram: -

- Histograms are a great way to learn a **lot about a numeric variable or column of a data** in a short amount of time.
- Histograms are use to visualize the distribution of a numeric variable and the create a histogram we use: -
    - **ax.hist(series, density=, alpha=, bins=)**
- we are going to plot more than one distribution in the same chart. Set the alpha parameter to be more transparent that the solid histogram.
- Set the density = True to use the relative frequencies in the y axis.
- Histograms are powerful for understanding the spread or distribution of our data.

Key takeaways: -

- Both the .join() and .merge() methods are use to combine the dataframes.
- **.join()** method combine the dataframes on the basis of their indexes.
- **.merge()** it allow us to specify columns besides the index to join on for both the dataframes. It is ideal for more complex joins, including those on multiple columns.

Advance Customization: -

Subplots: -

- **Subplots** allow us to **create a grid of equally sized charts** in a single figure.
    - **fig, ax = subplots(rows, columns)** → this will create a grid with the specified rows and columns. By default, this is 1, 1.
    - In order to populate our charts, we want to make sure we specify the correct coordinate before plotting.
    - The coordinates are like (0,0), (1,0), (0,1), (1,1) ….. and so on.
- One other helpful argument for subplots is the **'sharex'** and **'sharey' arguments** to set the same axis limits on all the plots.
    - This can be set as **'none'** by default, but can be set to **'all', 'row',** or '**col'**.
- Subplots **can be any chart types**, and do not have to be the same type.
- This is how we create multiple figures using **subplots()**: -
    - **fig, ax = plt.subplots(2,1, sharex="all", sharey="all")**
      ax[0].scatter(
      housing_raw.loc[:,"total_homes_sold"], housing_raw.loc[:, "inventory"]
)
      ax[1].scatter(
      housing_raw.loc[:,"months_of_supply"], housing_raw.loc[:, "inventory"]
)
- **plt.tight_layout()** → this function will make sure that our axis tick labels don't overlap with the other charts.

- We could also create some kind of loop to generate our subplots programmatically.
- **ax[1,2].set_axis_off()** → this is used to remove the extra chart that is not in use.

GridSpec: -

- We can build **layouts with charts of varying sizes** by setting a **gridspec object**.
    - This **creates a grid of specified number of rows and columns**.
    - Each axis, chart can then occupy a group of squares in the grid.
- To use the **gridspec object** we have to import the **gridspec module** from the matplotlib library like **from matplotlib.gridspec import GridSpec**.
- This is how we use the **GridSpec object**: -
    - ```
      fig = plt.figure(constrained_layout = True)
      grid = GridSpec(8, 8, figure=fig)
      ax1 = fig.add_subplot(grid[2:7, 0:6])
      ax1.scatter(
        diamonds["carat"],
        diamonds["price"],
        alpha=.3
      )
      ax2 = fig.add_subplot(grid[0:2, 0:6])
      ax2.hist(diamonds["carat"])
      ax3 = fig.add_subplot(grid[2:7, 6:])
      ax3.hist(
        diamonds["price"],
        orientation = "horizontal"
      )
      ```
- **to_datetime:** Accepts **strings or numeric values** that **represent dates or times.** Useful for working with **actual date and time values**.
- **to_timedelta:** Accepts strings or numeric values that **represent durations**. Useful for working with **time differences or durations**.
      ```

Colour Options: -

- The simplest way to **modify the colors in the chart** is pass a colors to a plot **by assigning them to a list**.
- We can also loop through a list of colors to pass them to separate series in a plot.
- Hex codes can be used to supply the specific color pantones.
- We can also **modify the entire color palette** for the series in a plot.
- Default color map is like: -
    - **Blue → Orange → Green → Red → Purple → Brown → Pink → Grey → Yellow → Turquoise**
- **plt.rcParams["axes.prop_cycle"] = plt.cycler(color=["red","orange","grey","green"])** → this is how we can se the color palette for our series.
- **plt.rcParams["axes.prop_cycle"] = plt.cycler("color", plt.cm.Dark2.colors)** → this is how we specify colors from the color palette mentioned on official matplotlib documentation.

Style sheets: -

- We've seen a lot of different ways to customize and format charts. One great shortcut to **formatting our charts is using style sheets**.
- **Style sheet** modify the default settings of the matplotlib and apply to all of our charts.
- **Matplotlib and Seaborn have style sheets** that can be used instead of default.
- There are lot of style sheets available in both the libraries.
- **plt.style.use('style_name')** → this is how we can apply style sheets to our charts.

rcParameters: -

- These are also known as **runtime configuration parameters**.
- So, by **setting a style in the charts**, we are **modifying the** bunch of the underlying **default parameters** of matplotlib. We can also modify the parameters just like the style sheet did by **modifying the rcParameters**.

- **Plt.style.library['style name']** → this is how we can see all the parameters of a stylesheet.
- There are 300+ parameters which can be modified. Just every attribute that we put onto a plot can be modified.
- There are **2 ways to modify** the parameters: -
    - We can **change the individual parameters** via assignment.
    - We can **change multiple parameters from the same group** with the **rc() function**.
- This is how we can change the default parameters using the **rc() function**: -
  plt.rc("axes.spines", top=False, right=False) → rc() function
  plt.rcParams["lines.linestyle"] = "--" → assignment
  fig, ax = plt.subplots()
  ax.plot(data);
- **plt.rcParams.keys()** → it is used to see all the default parameters in the rc dictionary.

Saving figures: -

- There are some frameworks for embedding matplotlib figures into web apps and even making them interactive. For that we can use the **python plotly library**.
- One way to save the figures is to save them as image files. This allows us to email them, paste them into presentation etc.
- The **savefig() function** will save figures as an image file.
    - Simply specify the desired filename and fileformat. **Png is the default** file format.
- **plt.savefile('filename', dpi=1000)**. The default resolution is 100 and if we want to modify that use the **dpi parameter inside savefile()**.