VISUALIZATION

Data visualization lies at the heart of modern data science, analytics, and business intelligence. Whether you're creating interactive dashboards for your organization, performing exploratory data analysis on a new dataset, or presenting insights to stakeholders, great visualizations can make all the difference.

Data visualization is one of the most effective methods to communicate insights, discover patterns, and guide decision-making. Whether you're a data scientist exploring a new dataset or a business analyst building dashboards for executive reports, the ability to create clear, compelling visuals can make your data accessible and actionable.

- **Faster insights**: A well-crafted chart or graph can uncover relationships and patterns faster than combing through raw data.
- **Better decisions**: Visuals help decision-makers see trends and outliers at a glance, enabling them to act quickly.
- **Effective communication**: Visualizations make it easier to tell a story with data, ensuring your audience understands not just the "what," but also the "why" behind your findings.

In Python's data science ecosystem, two major visualization libraries—Matplotlib (the classic, foundational library) and Plotly (the interactive, modern library)—are indispensable tools.

What Is Matplotlib?

Released in 2003, Matplotlib is the bedrock of Python's data visualization world. Most other Python plotting libraries build on top of Matplotlib in some way (e.g., Seaborn). It excels at creating static, publication-quality plots, from basic line graphs to complex multi-paneled figures. While it's very flexible and easy to get started with, the learning curve can be steep, especially when you want advanced styling or complex subplots.

What Is Plotly?

Plotly is a relatively newer library that focuses on providing interactive visualizations in Python, JavaScript, R, and other languages. It has become increasingly popular for building dynamic and shareable dashboards. With Plotly's Python API, users can create beautiful visualizations with features such as hover tooltips, zooming, and clickable legend entries - functionality that usually requires more effort when using Matplotlib alone.

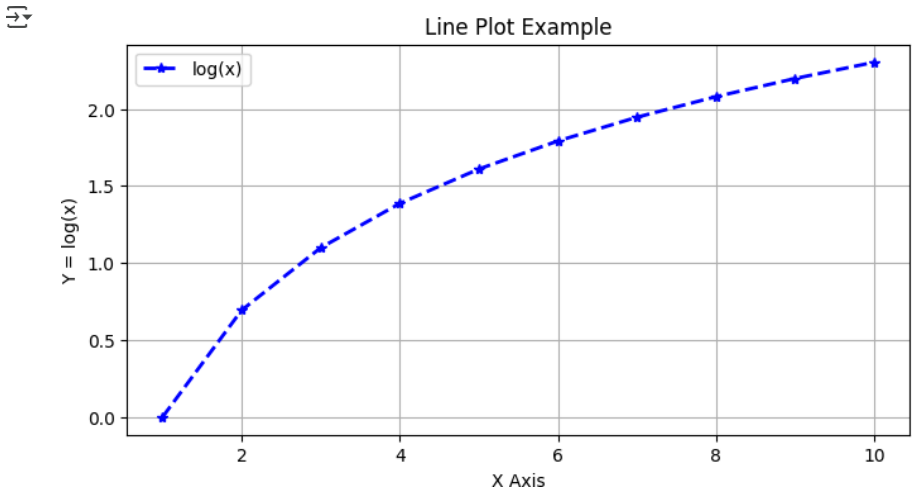
MATPLOTLIB library

```
import matplotlib.pyplot as plt
import numpy as np
```

1. Line Plot

```
x = np.arange(1, 11)
y = np.log(x)

plt.figure(figsize=(8, 4))
plt.plot(x, y, color='blue', linestyle='--', marker='*', linewidth=2, label='log(x)')
plt.title("Line Plot Example")
plt.xlabel("X Axis")
plt.ylabel("Y = log(x)")
plt.grid(True)
plt.legend()
plt.show()
```



A line plot is created using plt.plot(). It connects individual data points with straight lines. We can customize the line using parameters like linestyle (dotted or dashed), color, marker (to mark data points), and linewidth.

- x = np.arange(1, 11) creates a NumPy array x with values from 1 to 10.
- y = np.log(x) calculates the natural logarithm of each value in x and stores it in the y array.
- plt.figure(figsize=(8, 4)) creates a figure with a specific size (8 inches in width and 4 inches
- in height). plt.plot(x, y, ...) is the main function that creates the line plot.
- color='blue': Sets the line color to blue.
- linestyle='--': Makes the line dashed.
- marker='*': Places a star marker at each data point. • linewidth=2: Sets the thickness of the line to 2.
- label='log(x)': Assigns a label to the line, which will be shown in the
- legend. plt.title("Line Plot Example"): Sets the title of the plot.
- plt.xlabel("X Axis"): provides Label for x-axis.
- plt.ylabel("Y = log(x)"): provides Label for y-axis.
- plt.grid(True): gives a grid to the background of the plot.
- plt.legend(): Displays the legend, which in this case shows the label
- for the line. plt.show(): Renders and displays the plot.

Line plots are commonly used for tracking changes in stock prices, temperature over time, or website traffic

· 2. Bar Graph

```
categories = ['A', 'B', 'C', 'D']
values = [23, 45, 56, 78]
 plt.figure(figsize=(8, 4))
 plt.bar(categories, values, width=0.5, color='orange', edgecolor='black', label="Scores")
 plt.title("Bar Plot Example")
 plt.xlabel("Categories")
 plt.ylabel("Scores")
 plt.legend()
 plt.grid(True, axis='y', linestyle='--')
 plt.show()
```

The bar chart is created using the plt.bar() function. It is used to compare values across different categories by displaying vertical or horizontal rectangular bars. Parameters like color change the fill color of the bars, while width controls how thick the bars appear.

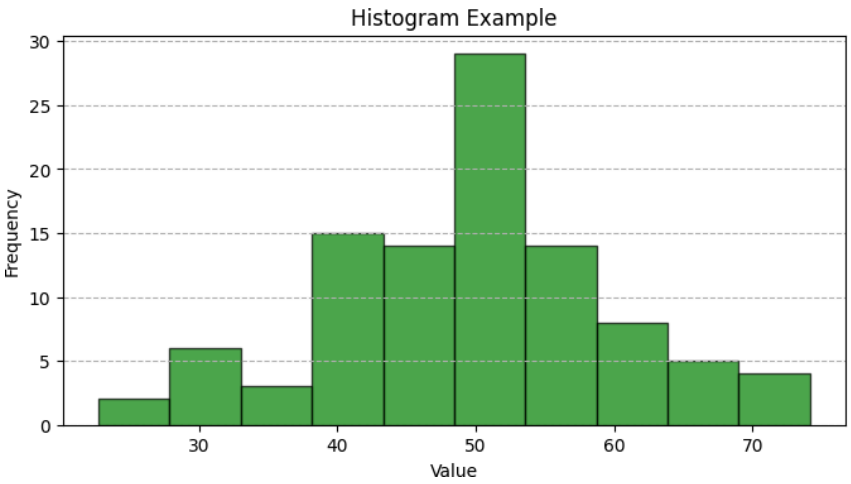
Here, the categories and values are the list of the x and y entries respectively.

plt.bar(x,y) is the function used to plot the bar graph where we provide our list instead of x and y along with different parameters like colour, thickness, edgecolor, label etc.

Bar charts are useful in applications like comparing monthly sales, product ratings, or population statistics across cities.

• 3. Histogram

```
data = np.random.normal(50, 10, 100)
plt.figure(figsize=(8, 4))
plt.hist(data, bins=10, color='green', edgecolor='black', alpha=0.7)
plt.title("Histogram Example")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.grid(True, axis='y', linestyle='--')
plt.show()
```



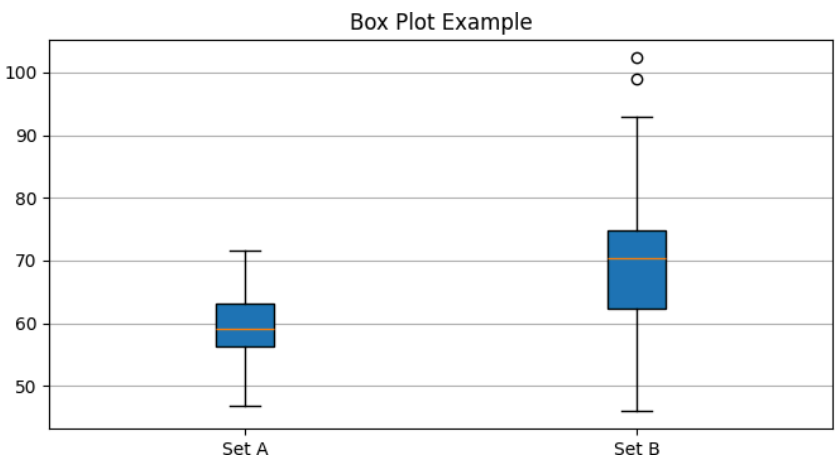
Histograms are created using plt.hist() and are used to display the distribution of a dataset.
plt.hist(data, bin=x) function accepts values for data to be plotted on y axis.

Number of bins are the number of bars in the histogram. Various parameters are applicable here as well. Here, alpha used is the opacity of the bins. here, data is generated using random function.

Histograms are ideal for visualizing test score distributions, customer ages, or product prices.

• 4. Box Plot

```
dataset = [np.random.normal(60, 5, 100), np.random.normal(70, 10, 100)]
plt.figure(figsize=(8, 4))
plt.boxplot(dataset, patch_artist=True, tick_labels=['Set A', 'Set B'])
plt.title("Box Plot Example")
plt.grid(True, axis='y')
plt.show()
```



here we use plt.boxplot() function to plot this graph. The box shows the middle 50% of data, the line in the middle is the median, and the whiskers show the range, excluding outliers.

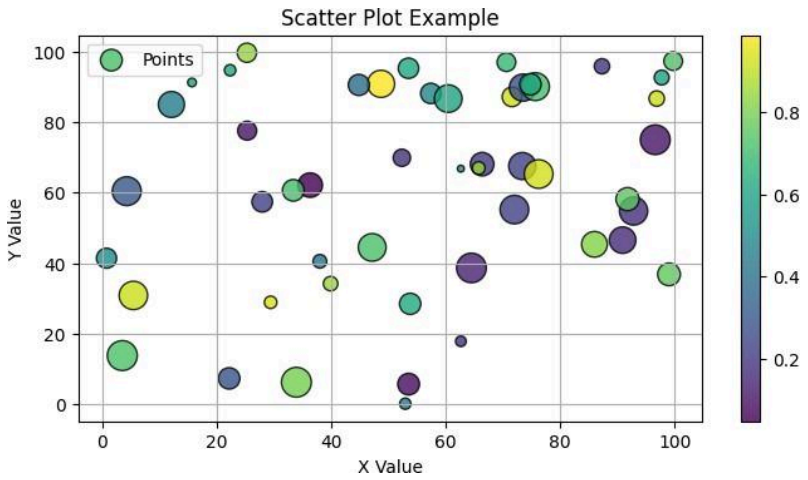
This creates a list of two datasets. Each dataset is generated using np.random.normal() which creates an array of 100 random numbers from a normal (Gaussian) distribution. The first dataset has a mean of 60 and a standard deviation of 5,

while the second has a mean of 70 and a standard deviation of 10. These are used in data analysis to identify spread, skewness, and outliers such as comparing test scores or income levels across groups.

• 5. Scatter Plot

```
x = np.random.rand(50) * 100
y = np.random.rand(50) * 100
sizes = np.random.rand(50) * 300
colors = np.random.rand(50)

plt.figure(figsize=(8, 4))
scatter = plt.scatter(x, y, s=sizes, c=colors, cmap='viridis', alpha=0.8, edgecolor='black', label='Points')
plt.colorbar(scatter)
plt.title("Scatter Plot Example")
plt.xlabel("X Value")
plt.ylabel("Y Value")
plt.grid(True)
plt.legend()
plt.show()
```



Scatter plots are created using plt.scatter() and are used to show the relationship between two variables. Each point on the graph represents a single observation. We can the point size using s, the color with c, and use alpha to control transparency, which helps when points overlap.

colors = np.random.rand(50): This creates a NumPy array colors with 50 random numbers between 0 and 1. This will be used to set the color of each point in the scatter plot.

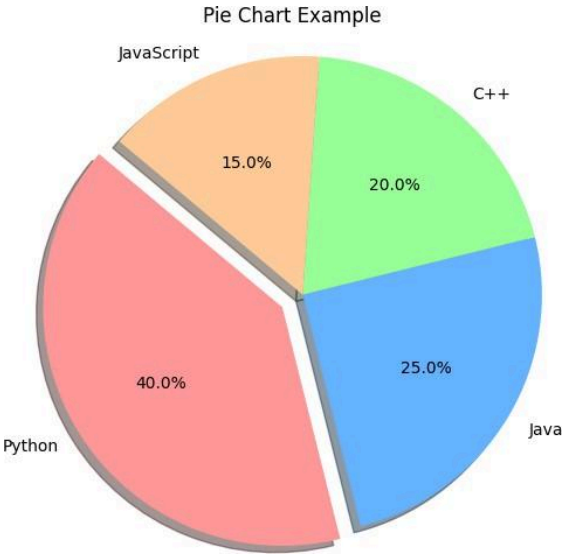
scatter = plt.scatter(x, y, s=sizes, c=colors, cmap='viridis', alpha=0.8, edgecolor='black', label='Points'): This is the main function that creates the scatter plot. cmap='viridis': This sets the colormap to be used for the points. alpha=0.8: This sets the transparency of the points.

These plots are ideal for applications like detecting patterns in exam scores vs. study hours, customer spending vs. income, or height vs. weight in health data.

· 6. Pie Chart

```
labels = ['Python', 'Java', 'C++', 'JavaScript']
sizes = [40, 25, 20, 15]
explode = (0.1, 0, 0, 0)

plt.figure(figsize=(6, 6))
plt.pie(sizes, labels=labels, explode=explode, autopct='%1.1f%%',
        shadow=True, startangle=140,
colors=['#ff9999','#66b3ff','#99ff99','#ffcc99']) plt.title("Pie Chart
Example")
plt.axis('equal')
plt.show()
```



The pie chart is built using the plt.pie() function. It shows how a whole is divided into different parts, using slices of a circle. The labels parameter names each slice, autopct shows the percentage of each part.

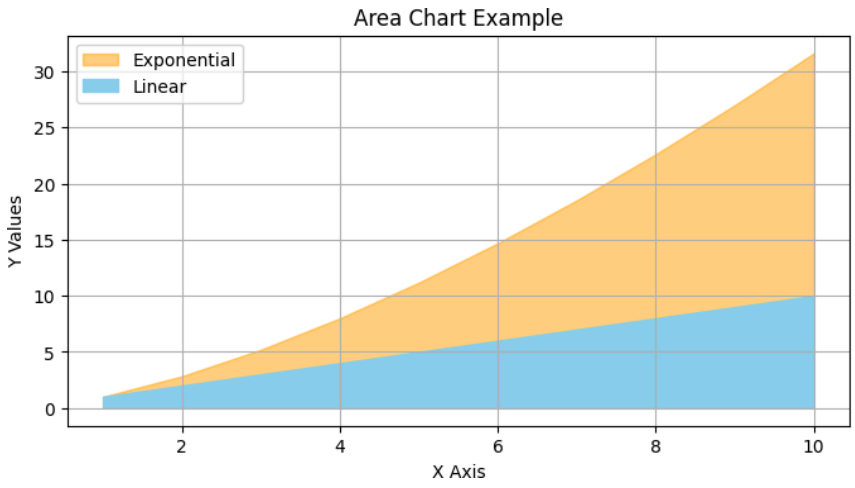
The explode parameter is very useful—it pulls one or more slices out to highlight them whereas autopct='%1.1f%%' provides us the percentage share of the section in total pie.

Pie charts are often used in business reports to show budget allocation, market share, or survey responses.

7. Area Chart

```
x = np.arange(1, 11)
y1 = np.arange(1, 11)
y2 = np.arange(1, 11) ** 1.5
plt.figure(figsize=(8, 4))
plt.fill_between(x, y2, color='orange', alpha=0.5, label='Exponential')
plt.fill_between(x, y1, color='skyblue', alpha=1, label='Linear')
plt.title("Area Chart Example")


plt.xlabel("X Axis")
plt.ylabel("Y Values")
plt.grid(True)
plt.legend()
plt.show()
```

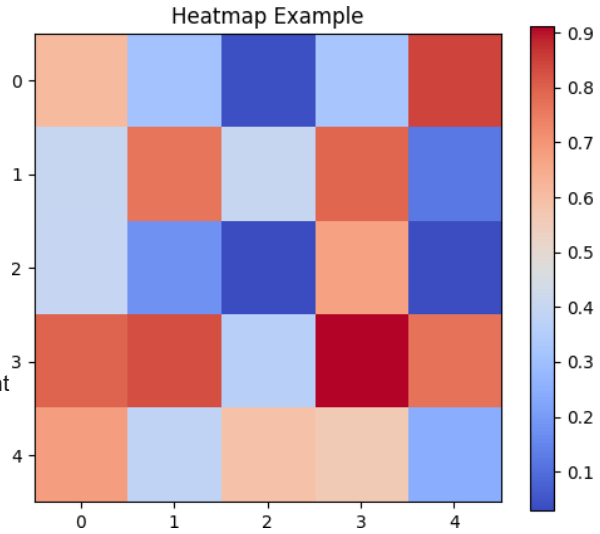


Area plots are similar to line plots but shade the area under the line. These can be created using plt.fill_between() or stackplot() if plotting multiple layers.

Area plots are useful when showing cumulative values, like population growth, sales over months, or resource usage over time.

8. Heatmap

```
matrix = np.random.rand(5, 5)
plt.figure(figsize=(6, 5))
plt.imshow(matrix, cmap='coolwarm', interpolation='nearest')
plt.title("Heatmap Example")
plt.colorbar()
plt.grid(False)
plt.show()
```



Though not a core Matplotlib function, heatmaps can be made using imshow().

A heatmap is a data visualization technique that uses color to represent the magnitude of a phenomenon across two dimensions. The "hotter" the color, the higher the value, and the "cooler" the color, the lower the value.

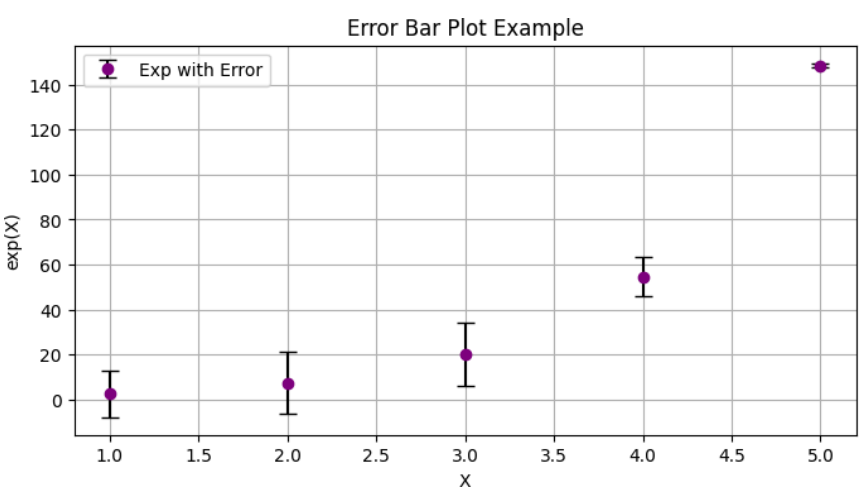
plt.imshow(matrix, cmap='coolwarm', interpolation='nearest'): This is the main function that creates the heatmap. matrix: This is the 2D array to be plotted. cmap='coolwarm': This sets the colormap to be used for the heatmap. interpolation='nearest': This displays the data as a grid of squares with no interpolation.

Heatmaps are often used for visualizing correlation matrices, confusion matrices, and geographic data density.

9. Error Bar Plot

```
x = np.arange(1, 6)
y = np.exp(x)
errors = np.random.rand(5) * 20

plt.figure(figsize=(8, 4))
plt.errorbar(x, y, errors=errors, fmt='o', color='purple', ecolor='black', capsize=5, label='Exp with Error')
plt.title("Error Bar Plot Example")
plt.xlabel("X")
plt.ylabel("exp(X)")
plt.grid(True)
plt.legend()
plt.show()
```



Error bar plots are made using plt.errorbar().

An error bar graph is a way to show the variability or uncertainty in your data. The length of this line represents the range of possible values or the standard deviation of the data.

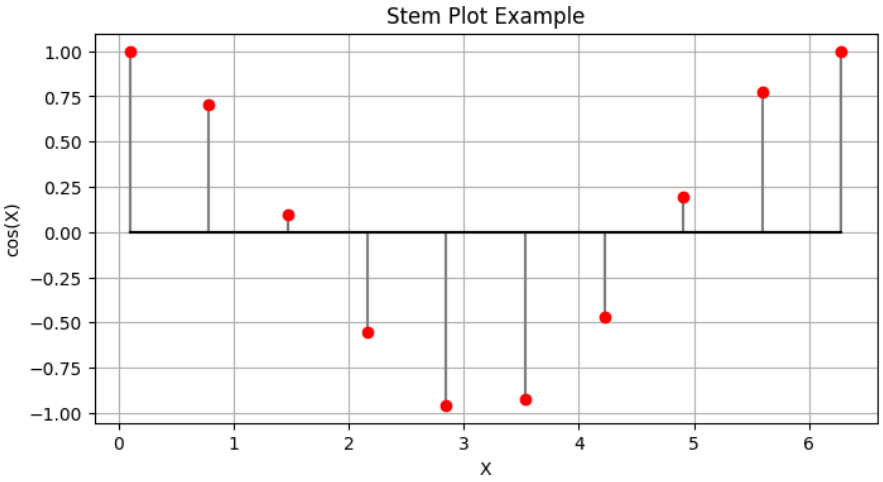
plt.errorbar(x, y, error=errors, fmt='o', color='purple', ecolor='black', capsize=5, label='Exp with Error'): This is the main function that creates the error bar plot. yerr=errors: This sets the size of the error bars for each data point.

These are important in scientific research to show the variability or precision of measurements.

10. Stem Plot

```
x = np.linspace(0.1, 2 * np.pi, 10)
y = np.cos(x)

plt.figure(figsize=(8, 4))
markerline, stemlines, baseline = plt.stem(x, y, linefmt='grey', markerfmt='ro', basefmt="k")
plt.title("Stem Plot Example")
plt.xlabel("X")
plt.ylabel("cos(X)")
plt.grid(True)
plt.show()
```



A stem plot is created using plt.stem(). A stem plot is a type of plot that displays data as vertical lines (the "stems") extending from a baseline to a marker at the data value.

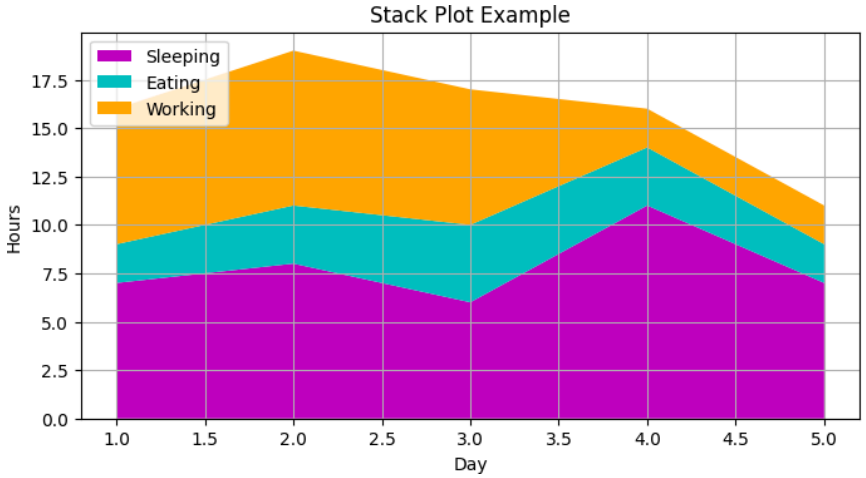
- Baseline: This is a horizontal line, usually at y=0, from which the stems originate.
- Stems: These are the vertical lines that extend from the baseline to the data points.
- Markers: These are the points at the end of each stem that indicate the value of the data point.

Stem plots are rarely used in business but common in digital signal processing or when plotting mathematical functions.

11. Stack Plot

```
days = [1, 2, 3, 4, 5]
sleeping = [7, 8, 6, 11, 7]
eating = [2, 3, 4, 3, 2]
working = [7, 8, 7, 2, 2]

plt.figure(figsize=(8, 4))
plt.stackplot(days, sleeping, eating, working, labels=['Sleeping', 'Eating', 'Working'], colors=['m', 'c', 'orange'])
plt.title("Stack Plot Example")
plt.xlabel("Day")
plt.ylabel("Hours")
plt.legend(loc='upper left')
plt.grid(True)
plt.show()
```



Stack plots are made using plt.stackplot().

A stack plot, also known as a stacked area chart, is a type of plot that shows how different components contribute to a whole over time, cumulative total of all the components, as well as the individual contribution of each component.

plt.stackplot(days, sleeping, eating, working, labels=['Sleeping', 'Eating', 'Working'], colors=['m', 'c', 'orange']).

Here, days are the x axis values, whereas sleeping, eating and working are shown as shaded region along y axis.

They are ideal for showing how different components (e.g., energy sources, expenses, or traffic sources) contribute to a whole over time.

PLOTLY library

```
import numpy as np
import plotly.graph_objects as go
import plotly.express as px
import numpy as np
import pandas as pd
```

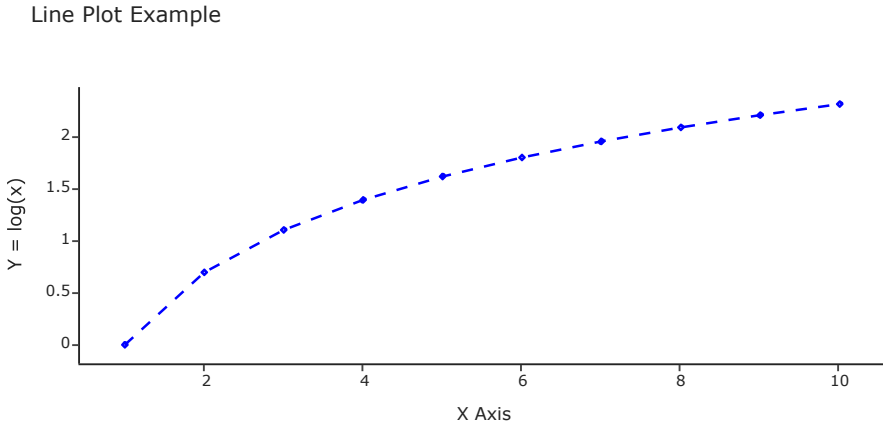
1. Line Plot

```
import plotly.graph_objects as go
import numpy as np

x = np.arange(1, 11)
y = np.log(x)

fig = go.Figure()
fig.add_trace(go.Scatter(x=x, y=y, mode='lines+markers',
                         line=dict(color='blue', dash='dash', width=2),
                         marker=dict(symbol='circle-dot'),
                         name='log(x)'))

fig.update_layout(title="Line Plot Example",
                  xaxis_title="X Axis",
                  yaxis_title="Y = log(x)",
                  template='simple_white',
                  width=800, height=400)
fig.show()
```



This graph illustrates a line plot using Plotly's go.Scatter().

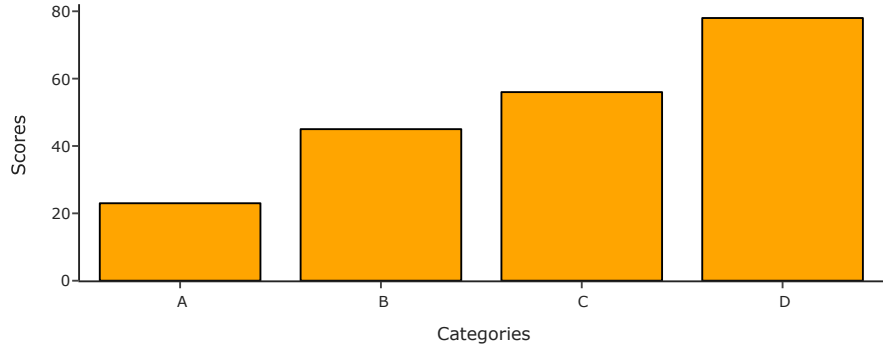
- fig.add_trace(go.Scatter(...)): This adds a scatter trace to the figure. In Plotly, go.Scatter is used for both line plots and scatter plots. mode='lines+markers': This tells Plotly to draw both lines and markers at the data points.
- line=dict(color='blue', dash='dash', width=2): This sets the properties of the line, including its color, style (dashed), and width. marker=dict(symbol='circle-dot'): This sets the style of the markers to be filled circles with a dot in the center.

The x-values (x) range from 1 to 10, and the y-values (y) are calculated as the log of x.

2. Bar PLot

```
categories = ['A', 'B', 'C', 'D']
values = [23, 45, 56, 78]

fig = go.Figure([go.Bar(x=categories, y=values, name="Scores",
                        marker=dict(color='orange', line=dict(color='black', width=1)))])
fig.update_layout(title="Bar Plot Example",
                  xaxis_title="Categories",
                  yaxis_title="Scores",
                  template='simple_white',
                  width=800, height=400)
fig.show()
```



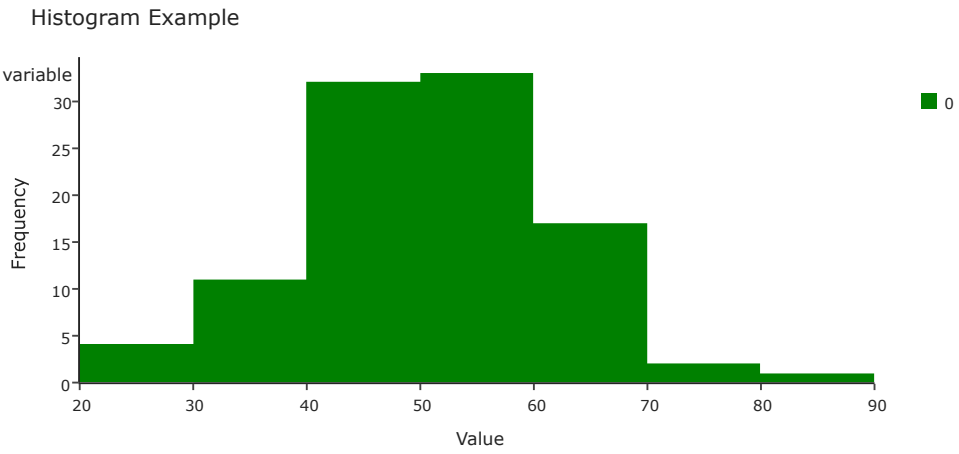
A bar plot displays categorical data with rectangular bars representing each category's value. Fucntion used here go.Bar().

values = [23, 45, 56, 78]: This creates a list of numbers that will be used as the values for the y-axis. fig = go.Figure([go.Bar(...)]): This creates a figure object and adds a bar trace to it.

Other characteristics: Horizontal and vertical axes are labeled, and a legend distinguishes the series.

3. Histogram

```
data = np.random.normal(50, 10, 100)
fig = px.histogram(data, nbins=10, color_discrete_sequence=['green'])
fig.update_layout(title="Histogram Example",
                  xaxis_title="Value",
                  yaxis_title="Frequency",
                  template='simple_white',
                  width=800, height=400)
fig.show()
```



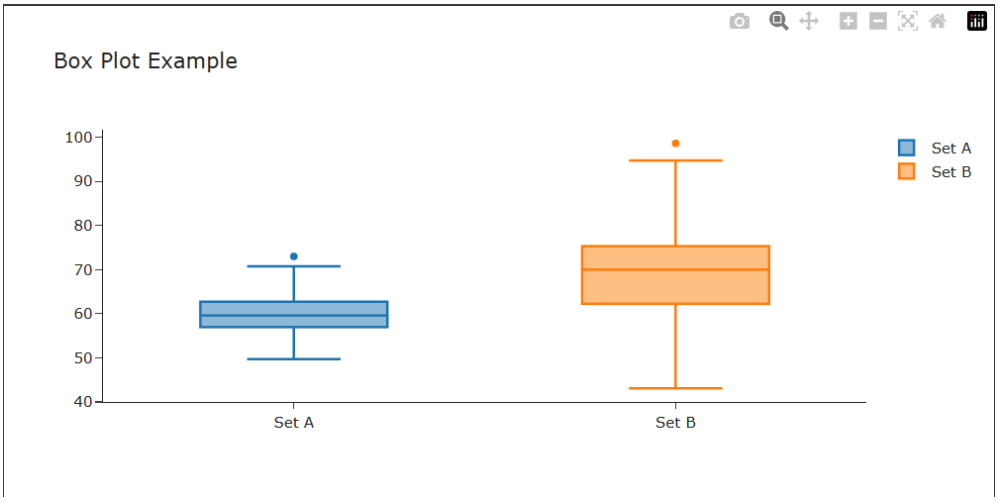
Plotly histograms use go.Histogram(). They group continuous data into intervals (bins) and show the frequency.

Histograms are widely used in statistics for analyzing exam scores, customer ages, or distribution of any numerical variable.

• 4. Box Plot

```
set_a = np.random.normal(60, 5, 100)
set_b = np.random.normal(70, 10, 100)

fig = go.Figure()
fig.add_trace(go.Box(y=set_a, name='Set A', boxpoints='outliers'))
fig.add_trace(go.Box(y=set_b, name='Set B', boxpoints='outliers'))
fig.update_layout(title="Box Plot Example",
                  template='simple_white',
                  width=800, height=400)
fig.show()
```



Created with go.Box(), box plots visualize the spread and skewness of data using median, quartiles, and outliers.

- fig.add_trace(go.Box(y=set_a, name='Set A', boxpoints='outliers')): This adds a box trace for set_a to the
- figure. boxpoints='outliers': This tells Plotly to show the outliers as individual points.

We use these plots in data analysis to compare income levels, exam scores, or performance metrics.

5. Scatter Plot

```
x = np.random.rand(50) * 100
y = np.random.rand(50) * 100
sizes = np.random.rand(50) * 30
colors = np.random.rand(50)

fig = px.scatter(x=x, y=y, size=sizes, color=colors,
                 color_continuous_scale='viridis',
                 labels={'x':'X Value', 'y':'Y Value'},
                 title="Scatter Plot
                 Example",
                  width=800, height=400)
fig.show()
```

The go.Scatter function in Plotly is used to create line charts, scatter plots, or a combination of both.

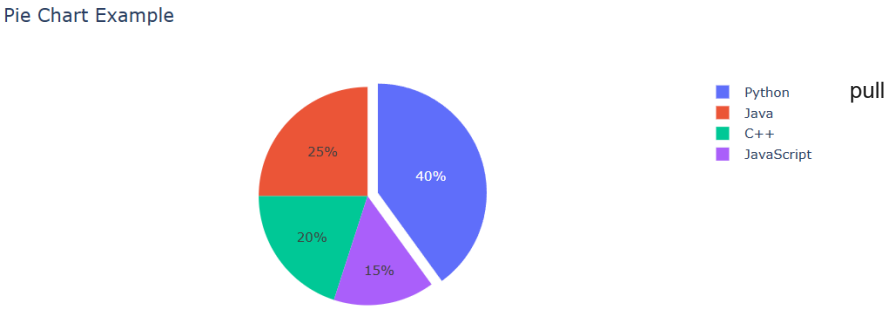
- colors = np.random.rand(50): This creates a NumPy array colors with 50 random numbers between 0 and 1. This will be used to set the color of each point in the scatter plot.
- fig = px.scatter(...): This is the main function that creates the scatter plot.
  color_continuous_scale='viridis': This sets the colormap to be used for the points.

6. Pie Plot

```
labels = ['Python', 'Java', 'C++', 'JavaScript']
sizes = [40, 25, 20, 15]

fig = go.Figure(data=[go.Pie(labels=labels, values=sizes,
                             pull=[0.1, 0, 0, 0],
                             marker=dict(colors=['r','g','b','o']))])


fig.update_layout(title="Pie Chart Example", width=800, height=400)
fig.show()
```



Pie chart shows the proportion of different programming languages, here were using go.Pie() function. The parameter used here helps in establishing emphasize on that section.

General used to represent simpler data like part in company shares etc with smaller number of characteristics.

pull

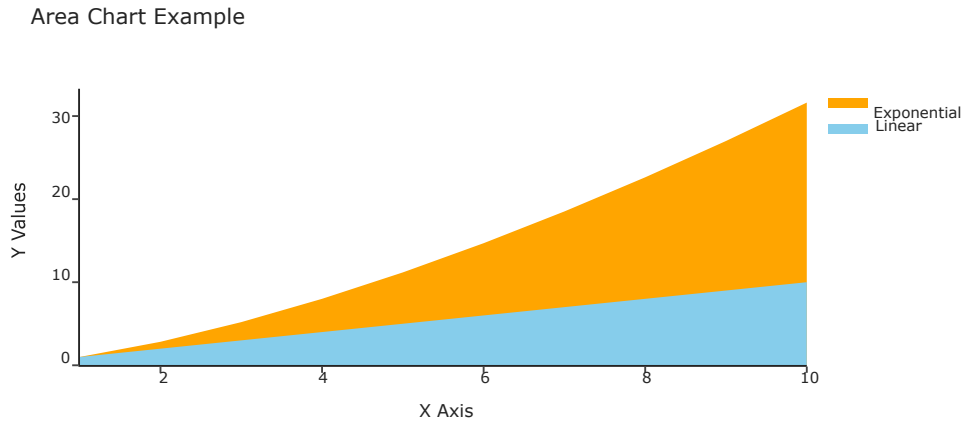
- fig = go.Figure(data=[go.Pie(...)]): This creates a figure object and adds a pie trace to it. labels=labels, values=sizes: These are the lists for the labels and values of the slices
- pull=[0.1, 0, 0, 0]: This "pulls" the first slice out from the center of the pie chart to emphasize it.

7. Area Chart

```
x = np.arange(1, 11)
y1 = np.arange(1, 11)
y2 = np.arange(1, 11) ** 1.5

fig = go.Figure()

fig.add_trace(go.Scatter(x=x, y=y2, fill='tozeroy', name='Exponential', mode='none',
                         fillcolor='orange', opacity=1))
fig.add_trace(go.Scatter(x=x, y=y1, fill='tozeroy', name='Linear', mode='none',
                         fillcolor='skyblue',opacity=0.25))
fig.update_layout(title="Area Chart Example",
                  xaxis_title="X Axis",
                  yaxis_title="Y Values",
                  template='simple_white', width=800, height=400)
fig.show()
```



This scatter plot charts individual points in two-dimensional space, with marker size and color encoding extra variables.

- fig.add_trace(go.Scatter(...)): This adds two scatter traces to the figure to create the area chart.
- The first trace plots y2 and fills the area between the line and the x-axis (fill='tozeroy') with an orange color.

The second trace plots y1 and fills the area between the line and the x-axis with a sky blue color.

8. Heatmap

```
matrix = np.random.rand(5, 5)

fig = go.Figure(data=go.Heatmap(
    z=matrix,
    colorscale='balance',
    x=['A', 'B', 'C', 'D', 'E'],
    y=['1', '2', '3', '4', '5'],
    text=np.round(matrix, 2),
    texttemplate="%{text}",
    hoverinfo="x+y+z" ))
fig.update_layout(
    title="Heatmap Example",
    xaxis_title="Categories",
    yaxis_title="Groups",
    width=800, height=400)
fig.show()
```

A heatmap displays a 5x5
array (matrix) of random values, using color to represent magnitude. here, we use go.Heatmap() function.

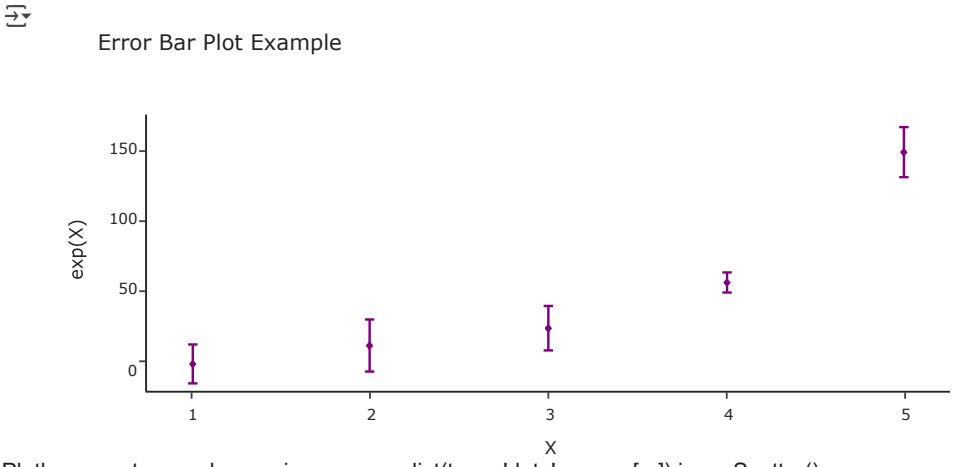
- fig = go.Figure(): This creates an empty figure object.
- fig.add_trace(go.Scatter(...)): This adds two scatter traces to the figure to create the area chart.
- The first trace plots y2 and fills the area between the line and the x-axis (fill='tozeroy') with an orange color.

Variable z stores the dimension of the map. The second trace plots y1 and fills the area between the line and the x-axis with a sky blue color

9. Error Bar plot

```
x = np.arange(1, 6)
y = np.exp(x)
errors = np.random.rand(5) * 20

fig = go.Figure(data=go.Scatter(
    x=x, y=y,
    error_y=dict(type='data', array=errors, visible=True),
    mode='markers',
    marker=dict(color='purple'),
    name='Exp with Error'
))
fig.update_layout(title="Error Bar Plot Example",
                  xaxis_title="X", yaxis_title="exp(X)",
                  template='simple_white', width=800, height=400)
fig.show()
```



Plotly supports error bars using error_y=dict(type='data', array=[...]) in go.Scatter().

fig = go.Figure(data=go.Scatter(...)): This creates a figure object and adds a scatter trace to it. error_y=dict(type='data', array=errors, visible=True): This sets the properties of the error bars.

These plots are crucial in scientific reports where uncertainty or standard deviation must be visualized (e.g., measurement errors or test variations).

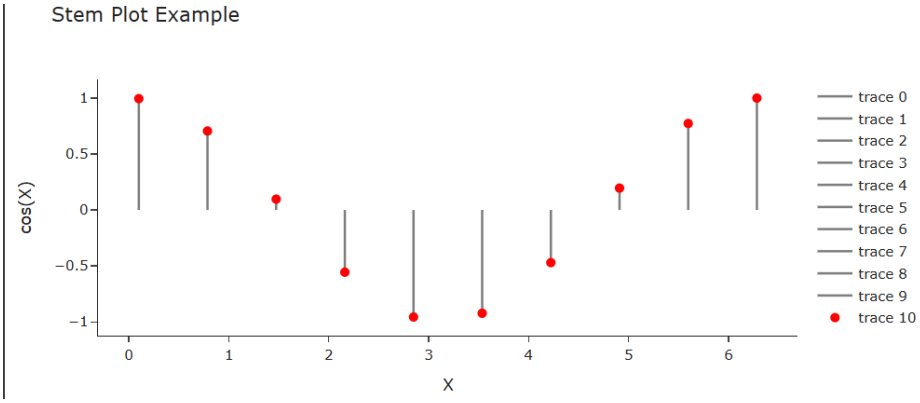
• 10. Stem Plot

```
x = np.linspace(0.1, 2 * np.pi, 10)
y = np.cos(x)

fig = go.Figure()
for xi, yi in zip(x, y):
    fig.add_trace(go.Scatter(x=[xi, xi], y=[0, yi],
                             mode="lines", line=dict(color='grey')))
fig.add_trace(go.Scatter(x=x, y=y, mode='markers', marker=dict(color='red', size=8)))

fig.update_layout(title="Stem Plot Example", xaxis_title="X", yaxis_title="cos(X)",
                  template='simple_white', width=800, height=400)
fig.show()
```

Plotly doesn't have a direct stem plot like Matplotlib, we can simulate it using go.Scatter() with vertical lines and markers.



- fig.add_trace(go.Scatter(x=[xi, xi], y=[0, yi], mode="lines", line=dict(color='grey'))): This adds a vertical line (stem) for each data point, extending from the baseline (y=0) to the data value.
- fig.add_trace(go.Scatter(x=x, y=y, mode='markers', marker=dict(color='red', size=8))): This adds markers at the end of each stem to indicate the data value.

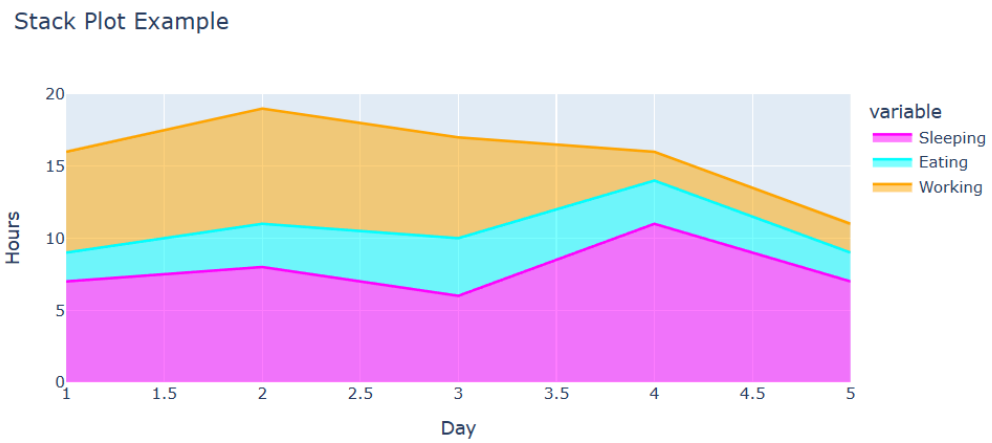
It's used in signal processing or discrete mathematical data to visualize individual values.

• 11. Stack Plot

```
days = [1, 2, 3, 4, 5]
sleeping = [7, 8, 6, 11, 7]
eating = [2, 3, 4, 3, 2]
working = [7, 8, 7, 2, 2]

df = pd.DataFrame({
    'Day': days,
    'Sleeping': sleeping,
    'Eating': eating,
    'Working': working
})

fig = px.area(df, x="Day", y=["Sleeping", "Eating", "Working"],
              labels={'value':'Hours'}, title="Stack Plot Example",
              colour_discrete_sequence = [‘magenta’, ‘cyan’, ‘orange’])
              fig.show()
```



We use multiple go.Scatter() traces with stackgroup='one' to create stack plots in Plotly.

df = pd.DataFrame(...): This creates a pandas DataFrame from the lists.

fig = px.area(...): This is the main function that creates the stack plot.

df: This is the DataFrame containing the data.

x="Day": This sets the "Day" column as the x-axis.
y=["Sleeping", "Eating", "Working"]: This sets the "Sleeping", "Eating", and "Working" columns as the y-axis values.

COMPARISON

MatplotLib Vs Plotly

- ***MatplotLib :***
  1. Matplotlib has extensive customization capabilities. We can control every aspect of your plot—axes, ticks, fonts, colors, etc
  2. **Performance**: Matplotlib is generally performant for generating static plots, though rendering can slow down if you're handling extremely large datasets (e.g., millions of points).
  3. **Ecosystem**: Being one of the oldest libraries, Matplotlib's ecosystem is vast—Seaborn, pandas plotting, scikit-learn's built-in plotting functions, etc., all rely on Matplotlib under the hood
  4. **Primarily static**: Most plots in Matplotlib are static images by default (PNG, PDF, SVG). We can incorporate some interactive elements in Jupyter notebooks (e.g., %matplotlib notebook), but the level of interactivity is generally limited compared to Plotly.
  5. **Third-party tools**: Tools like mpld3 and bokeh can add interactivity on top of Matplotlib, but these require additional effort and are not as seamless as Plotly's built-in approach.

- ***Plotly :***
  1. **Built-in themes**: Plotly offers sleek default themes, and we can switch among them easily.
  2. **Interactive elements**: Hover labels, legends, tooltips, and pop-ups are part of the core Plotly design, which can be further styled to match your needs. If our data contains thousands of data points and it's important for a user to be able to zoom in to a specific part of the plot, this interactivity will be very important.
  3. **Customization**: Although interactive styling can feel more complex to navigate, Plotly's figure objects can be tweaked to an impressive level of detail
  4. **Performance**: Plotly's performance is also robust. Plotly does provide ways to optimize or reduce data resolution for large-scale visualizations but you may find that working with hundreds of thousands or millions of points on a chart can start to cause issue.

Aspect / Feature	***Matplotlib***	***Plotly***
Type of Plots	Mainly static plots (line, bar, scatter, pie, etc.)	Highly interactive plots (line, bar, scatter, 3D, maps, etc.)
Interactivity	Plots are static by default. Limited interactivity, some via third-party tools or mplcursors.	Interactive by default: hover, zoom, pan, tooltips, legend toggles
Customization	Very high: every element (fonts, scales, ticks, lines, etc.) can be customized.	High, but some advanced customizations may take more work. Defaults are visually appealing.
Functions Set	100+ plotting functions (e.g., plot, scatter, hist, bar, pie, boxplot, imshow, etc.).	Rich set of expressive functions (plotly.express: scatter, line, bar, pie, map, 3D, etc.) plus lower-level plotly.graph_objects.
Parameters	Each function accepts a large set of keyword arguments (color, linestyle, marker, label, alpha, linewidth, etc.) for full control.	Parameters are optimized for both rapid prototyping and advanced configs (color, symbol, size, labels, hover info, templates, etc.).
Options & Appearance	Suited for publication-quality images—custom color maps, fine-tuned axes, subplots, LaTeX text, export in many formats (PNG, SVG, PDF).	Polished web-ready visuals, rich color schemes, layout options, smooth transitions. Better for dashboards and quick prototyping.