Cheatography

Matplotlib Cheat Sheet

by Arshdeep via cheatography.com/201979/cs/43037/

Line Plots

Line Plots	Line plots are fundamental in Matplotlib and are used to visualize data points connected by straight lines. They are particularly useful for displaying trends over time or any ordered data.

Basic Syntax	`import matplo tli b.p yplot as plt`
	`plt.plot(x, y)`
	`plt.show()`

Plotting Lines	The plot() function is used to create line plots. Pass arrays of data for the x-axis and y-axis.
Custom-izing Lines	You can customize the appearance of lines using parameters like color, linestyle, and marker.
Multiple Lines	Plot multiple lines on the same plot by calling plot() multiple times before show().
Adding Labels	Always add labels to the axes using xlabel() and ylabel() to provide context to the plotted data.
Example	`import matplo tli b.p yplot as plt`
Sample data	`x = [1, 2, 3, 4, 5]` `y = [2, 4, 6, 8, 10]`

Plotting the line	`plt.pl ot(x, y, color= 'blue',`
	`linestyle='-', marker ='o',`
	`label='Line 1')`

Adding labels and title	`plt.xl abe l(' X-a xis')`
	`plt.ylabel('Y-axis')`
	`plt.title('Example Line Plot')`

Adding legend	`plt.le gend()`
Display plot	`plt.show()`

Bar Plots

Bar Plots	Bar plots are used to represent categorical data with rectangular bars. They are commonly used to compare the quantities of different categories. Matplotlib provides a simple way to create bar plots using the bar() function.

Basic Bar Plot:	`import matplo tli b.p yplot as plt`
	`categories = ['A', 'B', 'C', 'D']`
	`values = [25, 30, 35, 40]`
	`plt.bar(categories, values)`
	`plt.xlabel('Categories')`
	`plt.ylabel('Values')`
	`plt.title('Basic Bar Plot')`
	`plt.show()`

Custom-izing Bar Plots	You can customize bar plots by changing colors, adding labels, adjusting bar width, and more using various parameters of the bar() function.
Grouped Bar Plots	To compare data across multiple categories, you can create grouped bar plots by plotting multiple sets of bars side by side.
Horizontal Bar Plots	Matplotlib also supports horizontal bar plots using the barh() function. These are useful when you have long category names or want to emphasize certain catego-ries.
Stacked Bar Plots	Stacked bar plots allow you to represent parts of the data as segments of each bar. This is useful for showing the composition of each category.

Handling Missing Data

Check for Missing Data	Before plotting, check your data for any missing values. This can be done using functions like isnull() or isna() from libraries like Pandas or NumPy.
Drop or Impute Missing Values	Depending on your analysis and the nature of missing data, you may choose to either drop the missing values using dropna() or impute them using techniques like mean, median, or interpolation.
Masking	Matplotlib supports masking, allowing you to ignore specific data points when plotting. You can create a mask array to filter out missing values from your dataset.
Handle Missing Data in Specific Plot Types	Different plot types may have different strategies for handling missing data. For example, in a line plot, you might interpolate missing values, while in a bar plot, you might choose to leave gaps or replace missing values with zeros.
Communicate Missing Data	Make sure your plots communicate clearly when data is missing. You can use annotations or legends to indicate where data has been removed or imputed.

Pie Charts

Pie Charts

Pie charts are circular statistical graphics that are divided into slices to i numerical proportions. Each slice represents a proportionate part of the data set.

Usage

Ideal for displaying the relative sizes of various categories within a data

Best suited for representing data with fewer categories (around 6 or few maintain clarity.

Creating a Pie Chart

```
import matplo tli b.p yplot as plt
labels = ['Category 1', 'Category 2', 'Category 3']
sizes = [30, 40, 30]
# Propor tions of each category
plt.pie(sizes, labels =la bels, autopc t=' %1.1 f%%'
plt.axis('equal')
# Equal aspect ratio ensures that pie is drawn as a
plt.show()
```

Customization

Colors: You can specify custom colors for each slice.

Exploding Slices: Emphasize a particular slice by pulling it out of the pie

Labels: Adjust label font size, color, and position.

Shadow: Add a shadow effect for better visual appeal.

Legend: Include a legend to clarify the meaning of each slice.

Example

```
# Custom izing a Pie Chart
colors = ['gold', 'yellowgreen',
 'light coral']
explode = (0, 0.1, 0)
# Explode the 2nd slice (Category 2)
plt.pie(sizes, explod e=e xplode,
labels=labels, colors =co lors,
autopct='%1.1f%%', shadow =True)
plt.axis('equal')
plt.show()
```

Considerations

Avoid using pie charts for datasets with many categories, as slices bec and difficult to interpret.

Ensure that the proportions of the data are clear and easy to understan

Double-check labels and legend to avoid confusion.

Customizing Plots

Custom-izing Plots	You can customize the color, linestyle, and marker style of lines and markers using parameters like color, linestyle, and marker.
Line and Marker Properties	Adjust the width of lines with the linewidth parameter and the size of markers with markersize.
Axes Limits	Set the limits of the x and y axes using xlim and ylim to focus on specific regions of your data.
Axis Labels and Titles	Add descriptive labels to the x and y axes using xlabel and ylabel, and give your plot a title with title.
Grids	Display grid lines on your plot using grid.
Legends	Add a legend to your plot to label different elements using legend.
Ticks	Customize the appearance and positioning of ticks on the axes using functions like xticks and yticks.
Text Annotations	Annotate specific points on your plot with text using text or annotate.
Figure Size	Adjust the size of your figure using the figsize parameter when creating a figure.
Background Color	Change the background color of your plot using set_facecolor.

Introduction to Matplotlib

Matplotlib is a powerful Python library widely used for creating static, interactive, and publication-quality visualizations. It provides a flexible and comprehensive set of plotting tools for generating a wide range of plots, from simple line charts to complex 3D plots.

Key Features:

Simple Interface: Matplotlib offers a straightforward interface for creating plots with just a few lines of code.

Flexibility: Users have fine-grained control over the appearance and layout of their plots, allowing for customization according to specific needs.

Wide Range of Plot Types: Matplotlib supports various plot types, including line plots, scatter plots, bar plots, histograms, pie charts, and more.

Integration with NumPy: Matplotlib seamlessly integrates with NumPy, making it easy to visualize data stored in NumPy arrays.

Publication-Quality Output: Matplotlib produces high-quality, publication-ready plots suitable for both digital and print media.

Extensibility: Users can extend Matplotlib's functionality through its object-oriented interface, enabling the creation of custom plot types and enhancements.

```
import matplo tli b.p yplot as plt
# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
# Create a line plot
plt.plot(x, y)
plt.xlabel('X-axis label')
plt.ylabel('Y-axis label')
plt.title('Simple Line Plot')
plt.show()
```

Annotations and Text

Adding Text	`plt.te xt(x, y, 'Your Text Here', fontsi ze e')`
Annota-tions	`plt.an not ate ('I mpo rtant Point', xy=(x, y), xytext =(x _text, y_text), arrowprops=dict(facecolor='black', arrowstyle='->'), fontsi ze=10)`
Text Properties	Matplotlib allows you to customize text properties such as style. Use keyword arguments like fontsize, color, fontweig appearance.

Annotations and Text (cont)

Latex Support	`plt.te xt(x, y, r'$\alpha > \beta$', fontsi ze=12)`
Multiline Text	`plt.te xt(x, y, 'Line 1\nLine 2', fontsize=12)`
Rotation	`plt.te xt(x, y, 'Rotated Text', fontsize=12, rotati on=45)`

Advanced Plotting Techniques

Multiple Axes and Figures	Use plt.subplots() to create multiple plots in a single figure. Control layout with plt.subplot() or plt.GridSpec().
Customizing Line Styles	Change line styles with linestyle parameter (e.g., '-', '--', '-.', ':'). Adjust line width using linewidth.
Color Mapping and Colormaps	Utilize colormaps for visualizing data with color gradients. Apply colormaps using the cmap parameter in functions like plt.scatter() or plt.imshow().
Error Bars and Confidence Intervals	Represent uncertainties in data with error bars. Add error bars using plt.errorbar() or ax.errorbar().
Layering Plots	Overlay plots to visualize multiple datasets in one figure. Use plt.plot() or ax.plot() multiple times with different data.
Plotting with Logarithmic Scale	Use logarithmic scales for axes with plt.xscale() and plt.yscale(). Helpful for visualizing data that spans several orders of magnitude.

Advanced Plotting Techniques (cont)

Polar Plots	Create polar plots with plt.subplot() and projection='polar'. Useful for visualizing cyclic data, such as compass directions or periodic phenomena.
3D Plotting	Visualize 3D data with mpl_toolkits.mplot3d. Create 3D scatter plots, surface plots, and more.
Animations	Animate plots using FuncAnimation. Ideal for displaying dynamic data or simulations.
Streamplots	Visualize vector fields with streamlines using plt.streamplot(). Useful for displaying fluid flow or electromagnetic fields.

Interactive Plotting

Zooming and Panning	Users can zoom in on specific regions of the plot to examine details more closely or pan across the plot to explore different sections.
Data Selection	Interactive plots allow users to select and highlight specific data points or regions of interest within the plot.
Interactive Widgets	Matplotlib provides widgets such as sliders, buttons, and text input boxes that enable users to dynamically adjust plot parameters, such as plot range, line styles, or data filters.

Cheatography

Matplotlib Cheat Sheet

by Arshdeep via cheatography.com/201979/cs/43037/

Interactive Plotting (cont)

Dynamic Updates	Plots can update dynamically in response to user interactions or changes in the underlying data, providing real-time feedback and visualization.
Custom Interactivity	Users can define custom interactive behavior using Matplotlib's event handling system, allowing for complex interactions tailored to specific use cases.

Adding Labels and Titles

Adding Axis Labels	`plt.xl abe l("X -axis Label")` `plt.ylabel("Y-axis Label")`
Adding Titles	`plt.ti tle ("Plot Title")`
Customizing Labels and Titles	`plt.xl abe l("X -axis Label", fontsize=12, fontwe igh t=' bold ', color='blue')` `plt.title("Plot Title", fontsize=14, fontwe igh t=' bold ', color='green')`
Mathematical Expressions in Labels	`plt.xl abe l(r " $\a lph a$")` `plt.title(r"$\beta$")`

3D Plotting

To create a 3D plot in Matplotlib, you typically start by importing the necessary modules:	`import matplo tli b.p yplot as plt` `from mpl_to olk its.mp lot3d import Axe s3D`
Then, you can create a 3D axes object using plt.figure() and passing the projection='3d' argument:	`fig = plt.fi gure()` `ax = fig.ad d_s ubp lot (111, projection='3d')`

3D Plotting (cont)

Once you have your 3D axes object, you can plot various types of 3D data using methods such as plot(), scatter(), bar3d(), etc. For example, to create a simple 3D scatter plot:	`ax.sca tte r(x _data, y_data, z_da ta)`

You can also customize the appearance of your 3D plot by setting properties such as labels, titles, axis limits, colors, and more:	`ax.set _xl abel('X Label')`
	`ax.set_ylabel('Y Label')`
	`ax.set_zlabel('Z Label')`
	`ax.set_title('3D Scatter Plot')`
	`# Set axis limits`
	`ax.set_xlim(x_min, x_max)`
	`ax.set_ylim(y_min, y_max)`
	`ax.set_zlim(z_min, z_max)`
	`# Customize colors`
	`ax.scatter(x_data, y_data, z_data, c=colo r_data, cmap='viridis')`

Plotting Images

Example	`import matplo tli b.p yplot as plt` `import numpy as np`
Create a random image	`image_data = np.random.random(( 100, 100))`

Plot the image	`plt.im sho w(i mag e_data, cmap=' gr ay')`
	`plt.axis('off')`
	`# Turn off axis`
	`plt.show()`

Working with Different Plot Styles

Default Style	Matplotlib's default style is functional and simple. Suitable for most basic plots without any specific styling requirements.

Working with Different Plot Styles (cont)

FiveThirtyEight Style	Mimics the style used by the FiveThirtyEight website. Bold colors and thicker lines for emphasis.
ggplot Style	Emulates the style of plots generated by the ggplot library in R. Clean and modern appearance with gray backgrounds.
Seaborn Style	Similar to the default Seaborn plotting style. Features muted colors and grid backgrounds.
Dark Background Styles	Various styles with dark backgrounds, suitable for presentations or dashboards. Examples include 'dark_background' and 'Solarize_Light2'.
XKCD Style	Creates plots with a hand-drawn, cartoonish appearance. Adds a playful touch to visualizations.
Example	`import matplo tli b.p yplot as plt` `plt.style.use('ggplot')`

Saving Plots

Using savefig() Function	`import matplo tli b.p yplot as pl t`
	`# Plotting code here`
	`plt.savefig('plot.png')`
	`# Save the plot as 'plot.png'`

Customizing Output	`plt.sa vef ig( 'pl ot.p ng', dpi= 300,`
	`bbox_inches='tight',`
	`transparent=True)`

Supported Formats	`plt.sa vef ig( 'pl ot.p df')`
	`# Save as PDF`
	`plt.savefig('plot.svg')`
	`# Save as SVG`

Interactive Saving	`plt.sa vef ig( 'pl ot.p ng',`
	`bbox_inches='tight',`
	`pad_inches=0)`

Legends

For example:	`import matplo tli b.p yplot as pl t`
Plotting data	`plt.pl ot(x1, y1, label= 'Line 1' )` `plt.plot(x2, y2, label= 'Line 2')`
Adding legend	`plt.le gend()` `plt.show()`

You can customize the appearance of the legend by specifying its location, adjusting the font size, changing the background color, and more.	`plt.le gen d(l oc= 'upper right',`
	`fontsize='large', shadow =True,`
	`facecolor='lightgray')`

Subplots

Subplots	Subplots allow you to display multiple plots within the sam useful for comparing different datasets or visualizing relate

Creating Subplots	`import matplo tli b.p yplot as plt`
	`# Create a figure with 2 rows and 2 columns`
	`fig, axs = plt.su bpl ots(2, 2)`

Accessing Subplot Axes	`ax1 = axs[0, 0]`
	`# Top-left subplot`
	`ax2 = axs[0, 1]`
	`# Top-right subplot`
	`ax3 = axs[1, 0]`
	`# Bottom -left subplot`
	`ax4 = axs[1, 1]`
	`# Bottom -right subplot`

Plotting on Subplots	`ax1.pl ot(x1, y1)`
	`ax2.scatter(x2, y2)`
	`ax3.bar(x3, y3)`
	`ax4.hist(data)`

Subplots (cont)

Customizing Subplots	`ax1.se t_t itl e('Plot 1')`
	`ax2.set_xlabel('X Label')`
	`ax3.set_ylabel('Y Label')`

Adjusting Layout	`plt.su bpl ots _ad jus t(h spa ce=0.5, wspac =0.5)` `# Adjust horizontal and vertical spacing`
Conclusion	Subplots are a powerful feature in Matplotlib for creating multi-panel figures, allowing you to efficiently visualize and compare multiple datasets within the same plot.

Histograms

Histograms

Histograms are graphical representations of the distribution of data. They display the frequency or probability of occurrence of different values in a dataset, typically depicted as bars. Histograms are commonly used to visualize the distribution of continuous data.

Creating a Histogram

```
import matplo tli b.p yplot as plt
data = [1, 2, 3, 3, 4, 4, 4, 5, 5, 6, 7, 7,
8, 8, 8]
plt.hist(data, bins=5, color= 'sk yblue',
edgecolor='black')
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram of Data')
plt.show()
```

Parameters

data: The input data to be plotted.

bins: Number of bins or intervals for the histogram.

color: Color of the bars.

edgecolor: Color of the edges of the bars.

Customizations

Adjust the number of bins to control the granularity of the histogram.

Change colors, edge colors, and bar width for aesthetic appeal.

Add labels and titles for clarity.

Interpretation

Histograms help in understanding the distribution of data, including its central tendency, spread, and shape.

They are useful for identifying patterns, outliers, and data skewness.

Histograms (cont)

Histograms are often used in exploratory data analysis and statistical analysis.

Scatter Plots

Scatter Plot	A Scatter Plot is a type of plot that displays values for two variables as points on a Cartesian plane. Each point represents an observation in the dataset, with the x-coordinate corresponding to one variable and the y-coordinate corresponding to the other variable.
Visualizing Relationships	Scatter plots are particularly useful for visualizing relationships or patterns between two variables. They can reveal trends, clusters, correlations, or outliers in the data.
Marker Style and Color	Points in a scatter plot can be customized with different marker styles, sizes, and colors to enhance visualization and highlight specific data points or groups.
Adding Third Dimension	Sometimes, scatter plots can incorporate additional dimensions by mapping variables to marker size, color intensity, or shape.
Regression Lines	In some cases, regression lines or curves can be added to scatter plots to indicate the overall trend or relationship between the variables.
Example	`import matplo tli b.p yplot as plt`
Sample data	`x = [1, 2, 3, 4, 5]` `y = [2, 3, 5, 7, 11]`

Cheatography

Matplotlib Cheat Sheet

by Arshdeep via cheatography.com/201979/cs/43037/

Scatter Plots (cont)

Create a scatter plot	`plt.sc att er(x, y, color= 'blue', marker='o', s=100)`

Adding labels and title	`plt.xl abe l(' X-axis Label')`
	`plt.ylabel('Y-axis Label')`
	`plt.title('Scatter Plot Example')`

Show plot	`plt.show()`
Data Scaling	Ensure that both variables are on a similar scale to avoid distortion in the visualization.
Data Explor-ation	Use scatter plots as an initial step in data exploration to identify potential patterns or relationships before further analysis.
Interp-retation	Interpretation of scatter plots should consider the overall distribution of points, any evident trends or clusters, and the context of the data.

Basic Plotting

Importing Matplotlib	`import matplo tli b.p yplot as plt`
Creating a Plot	`plt.pl ot( x_v alues, y_values)`
Displaying the Plot	`plt.show()`

Adding Labels and Title	`plt.xl abe l(' X-axis Label')`
	`plt.ylabel('Y-axis Label')`
	`plt.title('Plot Title')`

Customizing Plot Appearance	`plt.pl ot( x_v alues, y_values, color='red', linest yle ='--', marker='o')`
Adding Gridlines	`plt.gr id( True)`
Saving the Plot	`plt.sa vef ig( 'pl ot.p ng')`

Plotting with Dates

Datetime Objects	Matplotlib accepts datetime objects for plotting dates. You can create datetime objects using Python's datetime module.
Date Formatting	You can customize the appearance of dates on the plot using formatting strings. Matplotlib's DateFo-rmatter class enables you to specify the format of date labels.
Plotting Time Series	Matplotlib provides various plotting functions like plot(), scatter(), and bar() that accept datetime objects as input for the x-axis.
Custom-izing Date Axes	You can customize the appearance of the date axis, including the range, tick frequency, and formatting. Matplotlib's DateLocator class helps in configuring date ticks on the axis.
Handling Time Zones	Matplotlib supports handling time zones in date plotting. You can convert datetime objects to different time zones using Python libraries like pytz and then plot them accordingly.
Plotting Date Ranges	Matplotlib allows you to plot specific date ranges by filtering your dataset based on date values before passing them to the plotting functions.