

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 matplotlib.pyplot 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.

Customizing 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 matplotlib.pyplot as plt
```

Sample data

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
```

Plotting the line

```
plt.plot(x, y, color='blue',
         linestyle='-', marker='o',
         label='Line 1')
```

Adding labels and title

```
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Example Line Plot')
```

Adding legend

```
plt.legend()
```

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 matplotlib.pyplot 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()
```

Customizing 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 categories.

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.



By **Arshdeep**
cheatography.com/arshdeep/

Not published yet.
 Last updated 14th April, 2024.
 Page 1 of 8.

Sponsored by **ApolloPad.com**
 Everyone has a novel in them. Finish Yours!
<https://apollopod.com>

Handling Missing Data

Check for Missing Data	Before plotting, check your data for any missing values. This can be done using functions like <code>isnull()</code> or <code>isna()</code> 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 <code>dropna()</code> 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 represent 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 set.

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

Creating a Pie Chart

```
import matplotlib.pyplot as plt
labels = ['Category 1', 'Category 2', 'Category 3']
sizes = [30, 40, 30]
# Proportions of each category
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.axis('equal')
# Equal aspect ratio ensures that pie is drawn as a circle.
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 chart.

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

```
# Customizing a Pie Chart
colors = ['gold', 'yellowgreen', 'lightcoral']
explode = (0, 0.1, 0)
# Explode the 2nd slice (Category 2)
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%1.1f%%', shadow=True)
plt.axis('equal')
plt.show()
```

Considerations

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

Ensure that the proportions of the data are clear and easy to understand.

Double-check labels and legend to avoid confusion.



Customizing Plots

Customizing 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 matplotlib.pyplot 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.text(x, y, 'Your Text Here', fontsize)`

Annotations `plt.annotate('Important Point', xy=(x, y), xytext=(x_text, y_text), arrowprops=dict(facecolor='black', arrowstyle='->'), fontsize=10)`

Text Properties Matplotlib allows you to customize text properties such as style. Use keyword arguments like `fontsize`, `color`, `fontweight`, and `fontstyle` to control the appearance.



Annotations and Text (cont)

Latex Support	<code>plt.text(x, y, r'\$\alpha > \beta\$', fontsize=12)</code>
Multiline Text	<code>plt.text(x, y, 'Line 1\nLine 2', fontsize=12)</code>
Rotation	<code>plt.text(x, y, 'Rotated Text', fontsize=12, rotation=45)</code>

Advanced Plotting Techniques

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

Advanced Plotting Techniques (cont)

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



By Arshdeep

cheatography.com/arshdeep/

Not published yet.

Last updated 14th April, 2024.

Page 4 of 8.

Sponsored by [ApolloPad.com](https://apollopad.com)

Everyone has a novel in them. Finish Yours!

<https://apollopad.com>

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.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
```

Adding Titles

```
plt.title("Plot Title")
```

Customizing Labels and Titles

```
plt.xlabel("X-axis Label",
fontsize=12, fontweight='bold',
color='blue')
plt.title("Plot Title",
fontsize=14, fontweight='bold',
color='green')
```

Mathematical Expressions in Labels

```
plt.xlabel(r"$\alpha$")
plt.title(r"$\beta$")
```

3D Plotting

To create a 3D plot in Matplotlib, you typically start by importing the necessary modules:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Then, you can create a 3D axes object using `plt.figure()` and passing the `projection='3d'` argument:

```
fig = plt.figure()
ax = fig.add_subplot(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:

You can also customize the appearance of your 3D plot by setting properties such as labels, titles, axis limits, colors, and more:

```
ax.set_xlabel('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=color_data,
cmap='viridis')
```

Plotting Images

Example

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

Create a random image

```
image_data = np.random.random((
100, 100))
```

Plot the image

```
plt.imshow(image_data, cmap='gray')
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	<pre>import matplotlib.pyplot as plt plt.style.use('ggplot')</pre>

Saving Plots

Using savefig() Function	<pre>import matplotlib.pyplot as plt # Plotting code here plt.savefig('plot.png') # Save the plot as 'plot.png'</pre>
Customizing Output	<pre>plt.savefig('plot.png', dpi=300, bbox_inches='tight', transparent=True)</pre>
Supported Formats	<pre>plt.savefig('plot.pdf') # Save as PDF plt.savefig('plot.svg') # Save as SVG</pre>
Interactive Saving	<pre>plt.savefig('plot.png', bbox_inches='tight', pad_inches=0)</pre>

Legends

For example:	<pre>import matplotlib.pyplot as plt</pre>
Plotting data	<pre>plt.plot(x1, y1, label='Line 1') plt.plot(x2, y2, label='Line 2')</pre>
Adding legend	<pre>plt.legend() plt.show()</pre>
You can customize the appearance of the legend by specifying its location, adjusting the font size, changing the background color, and more.	<pre>plt.legend(loc='upper right', fontsize='large', shadow=True, facecolor='lightgray')</pre>

Subplots

Subplots	Subplots allow you to display multiple plots within the same figure, which is useful for comparing different datasets or visualizing relationships.
Creating Subplots	<pre>import matplotlib.pyplot as plt # Create a figure with 2 rows and 2 columns fig, axes = plt.subplots(2, 2)</pre>
Accessing Subplot Axes	<pre>ax1 = axes[0, 0] # Top-left subplot ax2 = axes[0, 1] # Top-right subplot ax3 = axes[1, 0] # Bottom-left subplot ax4 = axes[1, 1] # Bottom-right subplot</pre>
Plotting on Subplots	<pre>ax1.plot(x1, y1) ax2.scatter(x2, y2) ax3.bar(x3, y3) ax4.hist(data)</pre>



Subplots (cont)

```
Custom-izing Subplots
ax1.set_title('Plot 1')
ax2.set_xlabel('X Label')
ax3.set_ylabel('Y Label')

Adjusting Layout
plt.subplots_adjust(hspace=0.5, wspace=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 matplotlib.pyplot as plt
data = [1, 2, 3, 3, 4, 4, 4, 5, 5, 6, 7, 7, 8, 8, 8]
plt.hist(data, bins=5, color='skyblue', 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 matplotlib.pyplot as plt`

Sample data `x = [1, 2, 3, 4, 5]`
`y = [2, 3, 5, 7, 11]`



Scatter Plots (cont)

Create a scatter plot

```
plt.scatter(x, y, color='blue',
            marker='o', s=100)
```

Adding labels and title

```
plt.xlabel('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 Exploration Use scatter plots as an initial step in data exploration to identify potential patterns or relationships before further analysis.

Interpretation 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 matplotlib.pyplot as plt
```

Creating a Plot

```
plt.plot(x_values, y_values)
```

Displaying the Plot

```
plt.show()
```

Adding Labels and Title

```
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Plot Title')
```

Customizing Plot Appearance

```
plt.plot(x_values, y_values,
         color='red', linestyle='--',
         marker='o')
```

Adding Gridlines

```
plt.grid(True)
```

Saving the Plot

```
plt.savefig('plot.png')
```

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 DateFormatter 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.

Customizing 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.

