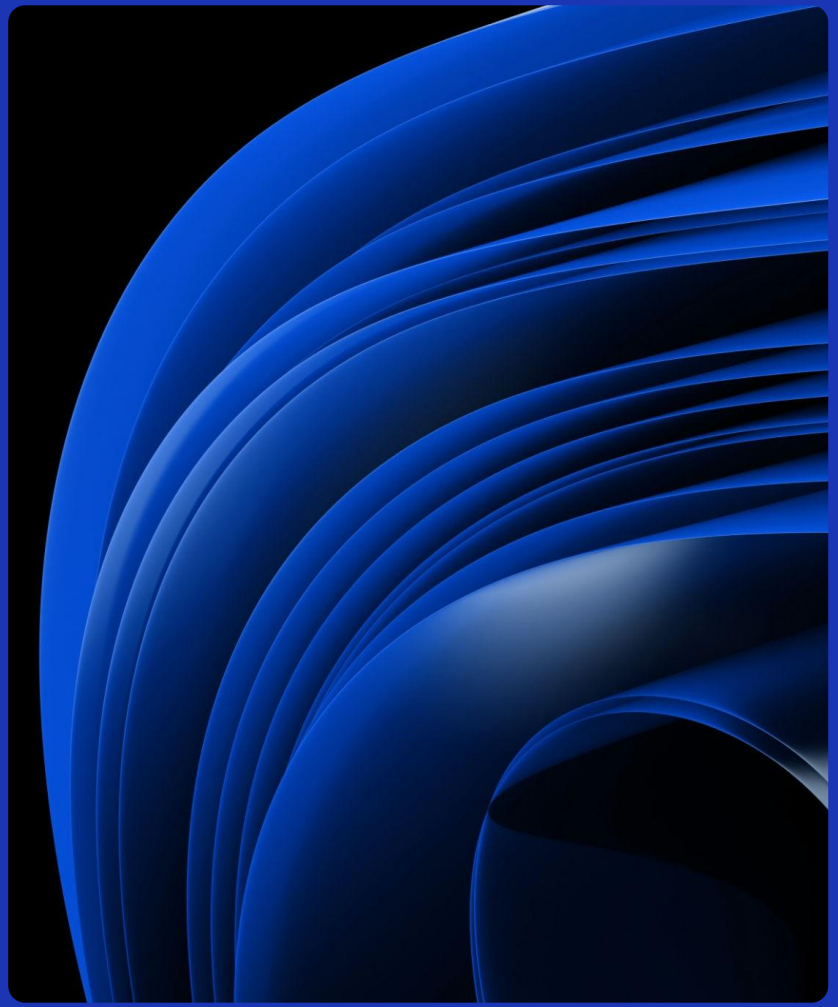
Visualization with Matplotlib

By Muhammad Umair Baig
Data & AI Trainer



Introduction to Matplotlib

- **Matplotlib is a multiplatform data visualization library built on NumPy arrays, part of the broader SciPy stack.**

- **Conceived by John Hunter in 2002 as a patch to IPython for MATLAB-style interactive plotting.**

- **Released version 0.1 in 2003 after John decided to develop it independently.**

Key Features of Matplotlib

Works across multiple operating systems and supports various graphics backends.

Supports dozens of backends and output types, ensuring compatibility and flexibility.

Large user base and active developer community contribute to its tools and ubiquity in scientific Python.

Challenges and Modern Alternatives

- Interface and style are perceived as outdated compared to newer tools like:
    - ggplot and ggvis (R language).
    - Web-based visualization libraries (e.g., D3.js, HTML5 canvas).
- Despite this, Matplotlib remains a strong, well-tested, cross-platform graphics engine.

Adaptation and Evolution

- Recent versions allow easier customization of global plotting styles (e.g., style sheets).
- Modern packages build on Matplotlib’s core to provide cleaner APIs, such as:
    - Seaborn.
    - ggpy, HoloViews.
    - Pandas as a wrapper around Matplotlib.
- Direct use of Matplotlib’s syntax is still valuable for precise adjustments.

Importance of Matplotlib

- **Despite competition, Matplotlib is likely to remain a crucial component of the Python data visualization ecosystem.**

- **Modern tools complement rather than completely replace Matplotlib, ensuring its continued relevance.**

Importing Matplotlib

- **Just as we use the np shorthand for NumPy and the pd shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:**

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Setting Styles

We will use the plt.style directive to choose appropriate aesthetic styles for our figures. Here we will set the classic style, which ensures that the plots we create use the classic Matplotlib style:

```
plt.style.use('classic')
```

show or No show? How to Display Your Plots

- **A visualization you can't see won't be of much use, but just how you view your Matplotlib plots depends on the context.**

- **The best use of Matplotlib differs depending on how you are using it; roughly, the three applicable contexts are using Matplotlib in a script, in an IPython terminal, or in a Jupyter notebook.**

Plotting from a Script

- **Purpose of `plt.show`:**
  - **Starts an event loop to display active figures in interactive windows.**

- **Example Script (myplot.py):**

```
import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))

plt.plot(x, np.cos(x))

plt.show()
```

Running the Script:

- Execute from the command line:

  ```
  $ python myplot.py
  ```

- Opens a window with the figure(s) displayed.

- Behavior of plt.show:
  - Interacts with the system's graphical backend, handling backend-specific details.
  - Should be used only once per Python session, typically at the script's end.

- Caution:
  - Multiple plt.show calls can cause unpredictable behavior depending on the backend.

Plotting from an IPython Shell

- **IPython Compatibility:**
  - **Matplotlib integrates seamlessly with the IPython shell.**

- **Activating Matplotlib Mode:**
  - **Use the %matplotlib magic command:**

    ```
    %matplotlib
    Using matplotlib backend: TkAgg
    ```

- **Workflow in IPython:**
  - **After enabling the mode, import Matplotlib:**

    ```
    import matplotlib.pyplot as plt
    ```
  - **Any plt plot command will open a figure window.**

- **Updating Plots:**
  - **Some changes (e.g., modifying drawn line properties) do not auto-update.**
  - **Use `plt.draw` to force a manual update.**
- **No Need for `plt.show`:**
  - **`plt.show` is unnecessary in IPython's Matplotlib mode.**

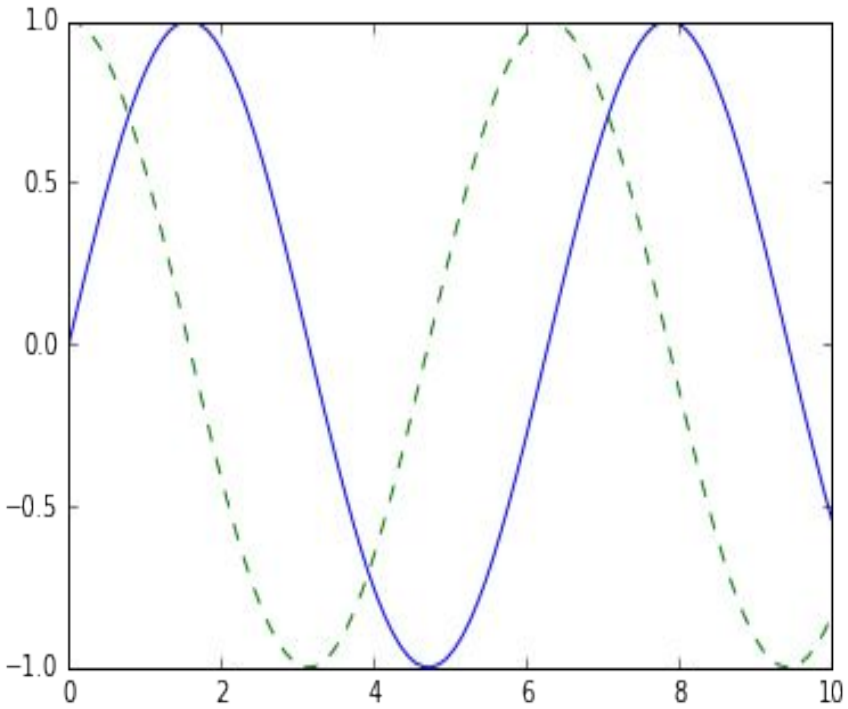
Plotting from a Jupyter Notebook

- **Jupyter Notebook Overview:**
  - **A browser-based tool combining narrative, code, graphics, HTML, and more into an executable document.**

- **Interactive Plotting in Notebooks:**
  - **Use `%matplotlib` magic command, similar to the IPython shell.**
  - **Options for embedding graphics:**
  - **`%matplotlib inline`: Static images of plots embedded in the notebook.**
  - **`%matplotlib notebook`: Interactive plots embedded in the notebook.**

Example with Static Embedding:

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

x = np.linspace(0, 10, 100)

fig = plt.figure() plt.plot(x, np.sin(x),
'-') plt.plot(x, np.cos(x), '--');
```

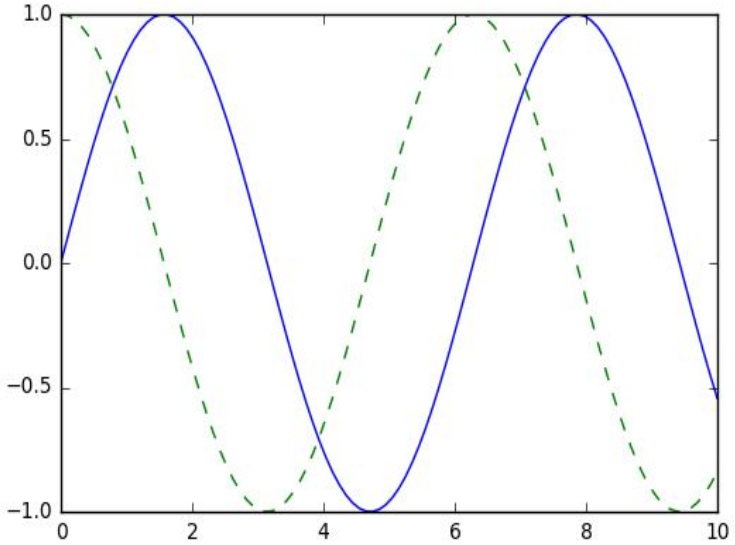


Saving Figures to File

- Use `savefig` to save figures in various formats.
- Example to save a plot as a PNG file:

```
fig.savefig('my_figure.png')
```

- Generates `my_figure.png` in the current working directory.



- Confirm Saved Content:

```
from IPython.display import Image
Image('my_figure.png')
```

Supported File Types:

- File format inferred from the file extension.
- Check supported formats using: `fig.canvas.get_supported_filetypes()`

  - Examples of supported formats:

    PNG, JPEG, PDF, SVG, EPS, TIFF, and more.

  - Key Note:
    - `plt.show` or similar commands are not necessary when saving figures.

Two Interfaces for the Price of One

MATLAB-Style Interface

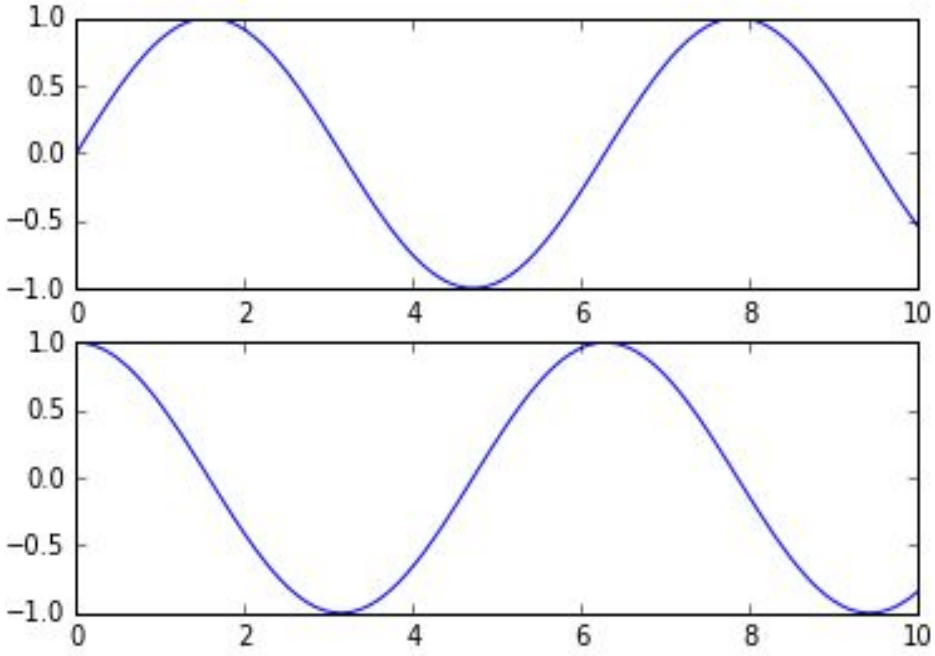
- Designed as a Python alternative for MATLAB users, using a stateful, convenient interface via `pyplot` (plt).
- Example code with two panels:

```
plt.figure()  # create a plot figure

# create the first of two panels and set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```

- Keeps track of the "current" figure and axes using `plt.gcf` (get current figure) and `plt.gca` (get current axes).

Limitations:

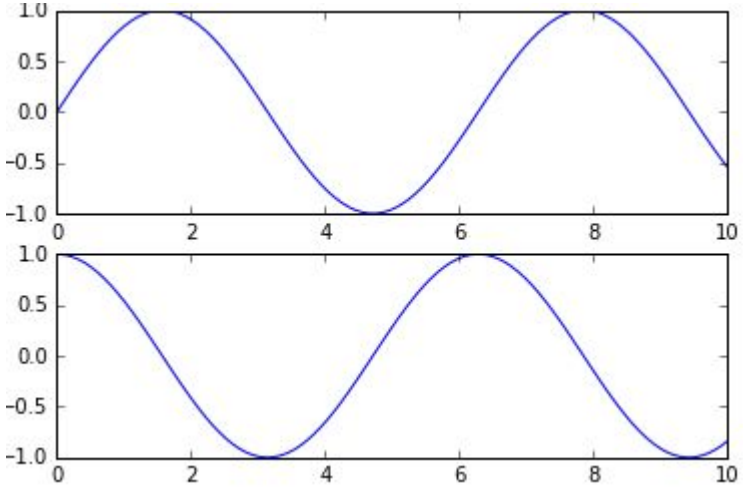
- **Issues arise when modifying previous panels or axes after creating new ones.**
- **While possible, it can be clunky for complex plots.**



Object-Oriented Interface

- **Overview:**
  - **Provides more control over figures and axes, especially for complex plots.**
  - **Plotting functions are methods of explicit `Figure` and `Axes` objects.**

```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```



Advantages:

- **Avoids reliance on an "active" figure or axes.**
- **Necessary for complicated plots requiring precise control.**

Comparison and Usage

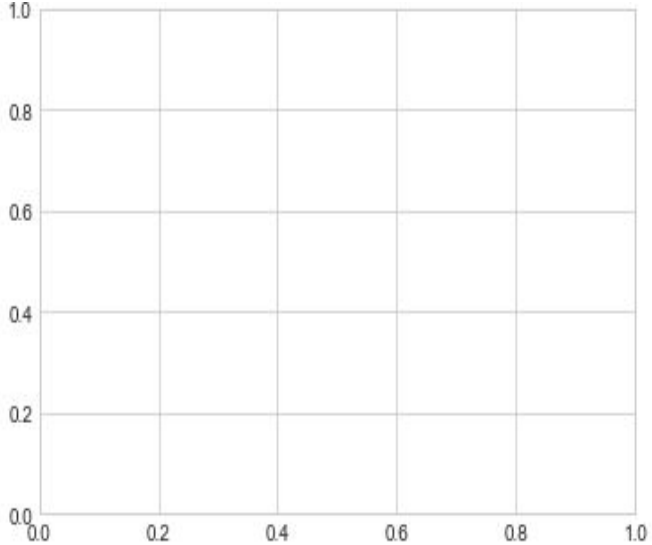
- **Simple Plots: Either style works, and preference determines choice.**
- **Complex Plots: Object-oriented approach becomes essential.**

Simple Line Plots

Creating a Simple Plot

- **Setup: Import necessary libraries and enable inline plotting.**

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-whitegrid')
import numpy as np
```



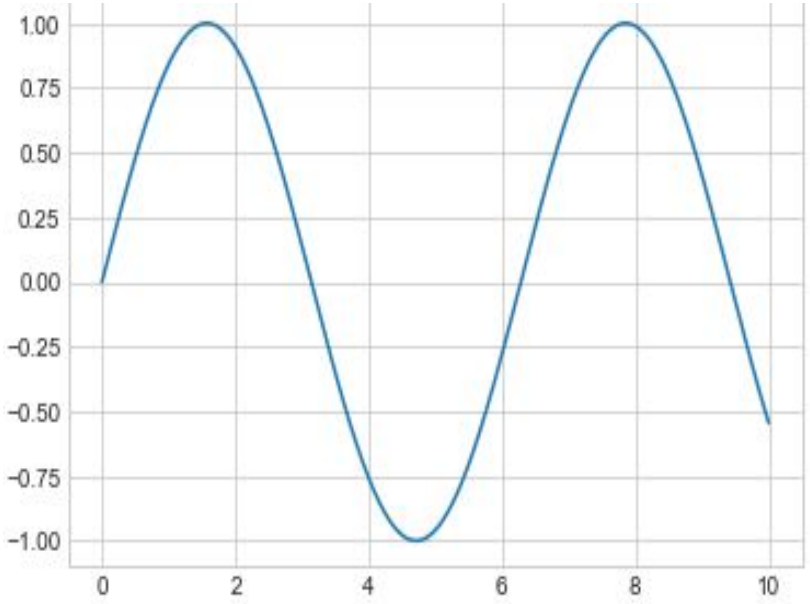
Creating a figure and axes:

- **The `figure` is a container for all plot elements, and `axes` is the area where the plot is drawn.**

Plotting a simple sinusoid:
Define an array of x-values and plot the corresponding sine values.

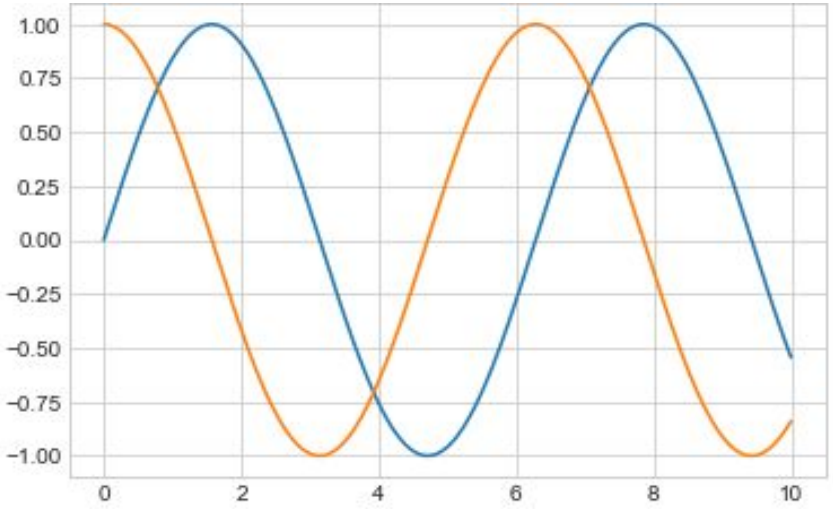
```
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
plt.plot(x, np.sin(x));
```

plt.plot(x, np.sin(x)) Automatically creates a figure and axes without explicitly defining them.



- **Plotting multiple lines:**
  Call `plt.plot` multiple times to add more lines to the same plot.

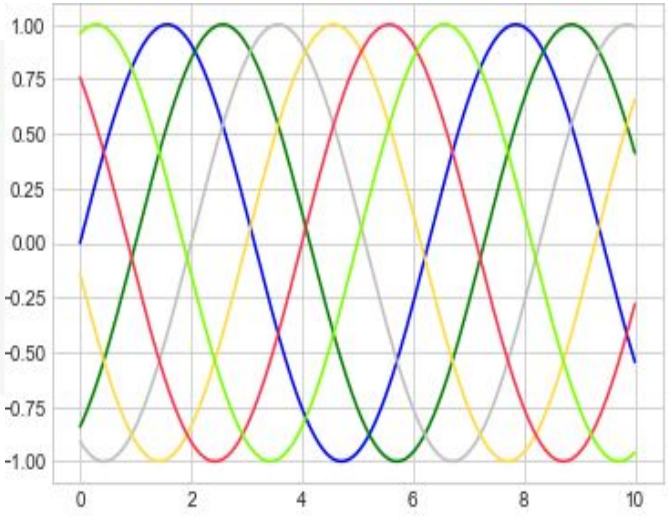
```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));  # Add a cosine curve
```



Adjusting Line Colors and Styles

- **Customizing colors:**
  Use the `color` argument to specify line colors in different formats:

```
plt.plot(x, np.sin(x - 0), color='blue')        # Name
plt.plot(x, np.sin(x - 1), color='g')           # Short color code (g for green)
plt.plot(x, np.sin(x - 2), color='0.75')        # Grayscale (0 to 1)
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # Hex code
plt.plot(x, np.sin(x - 4), color=(1.0, 0.2, 0.3)) # RGB tuple (0 to 1)
plt.plot(x, np.sin(x - 5), color='chartreuse'); # HTML color name
```

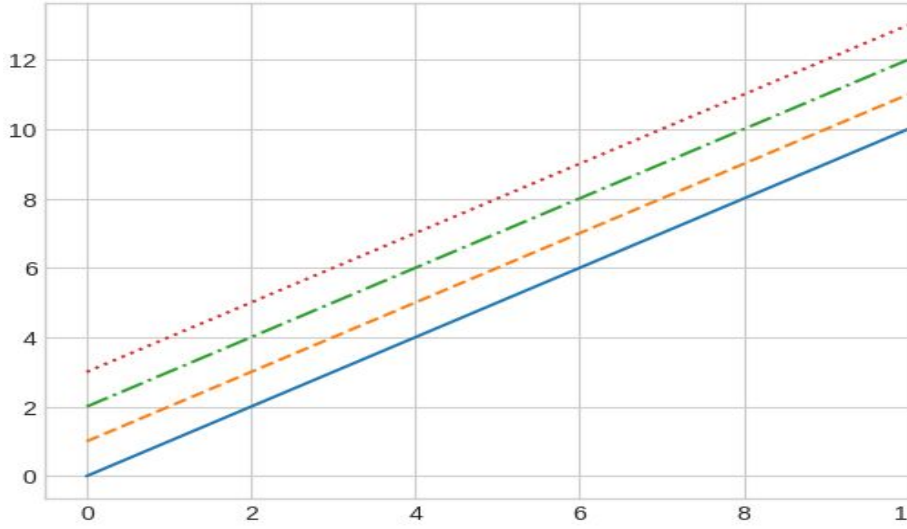


If no color is specified, Matplotlib cycles through default colors.

Customizing line styles:

- **Use `linestyle` to adjust the appearance of the line.**

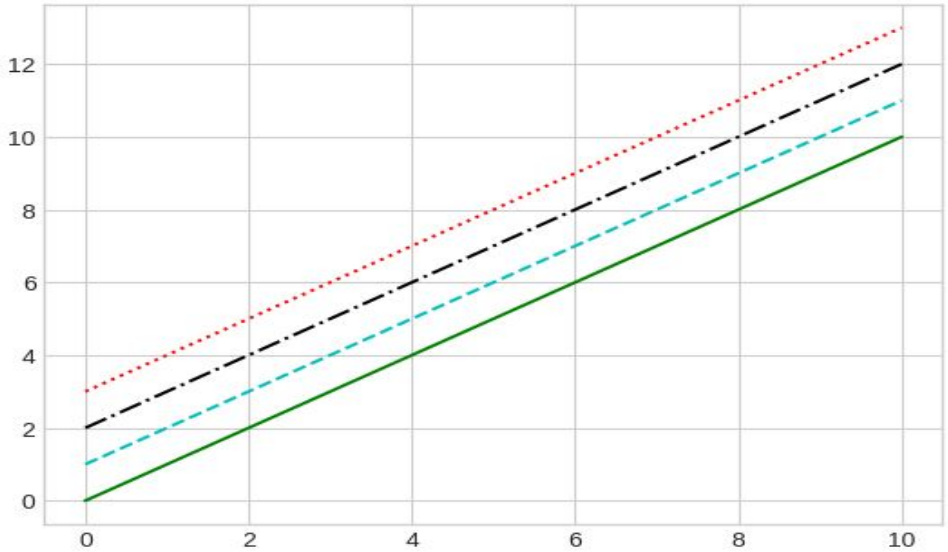
```
plt.plot(x, x + 0, linestyle='solid')  # Solid line
plt.plot(x, x + 1, linestyle='dashed')  # Dashed line
plt.plot(x, x + 2, linestyle='dashdot')  # Dash-dot line
plt.plot(x, x + 3, linestyle='dotted');  # Dotted line
```



Combining styles and colors:

- **Save effort by combining line styles and colors in shorthand.**

```
plt.plot(x, x + 0, '-g')   # Solid green
plt.plot(x, x + 1, '--c')  # Dashed cyan
plt.plot(x, x + 2, '-.k')  # Dash-dot black
plt.plot(x, x + 3, ':r');  # Dotted red
```

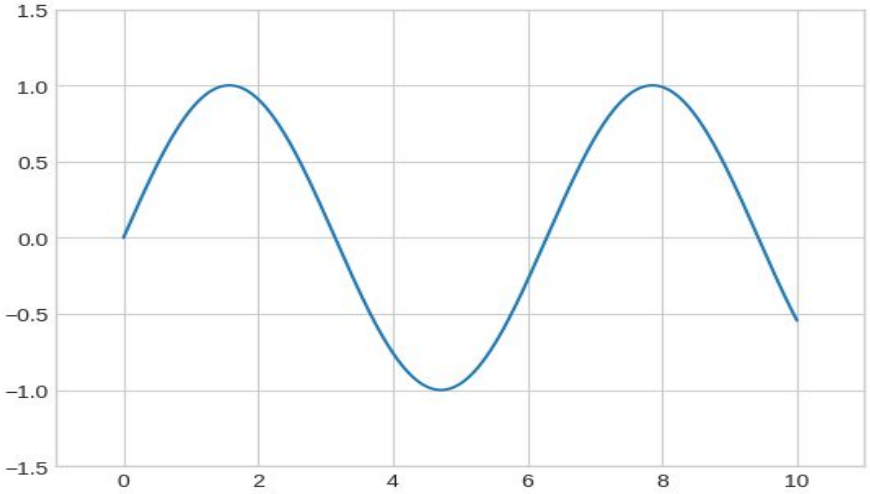


Adjusting Axes Limits

Matplotlib automatically adjusts axes limits, but you can override them:

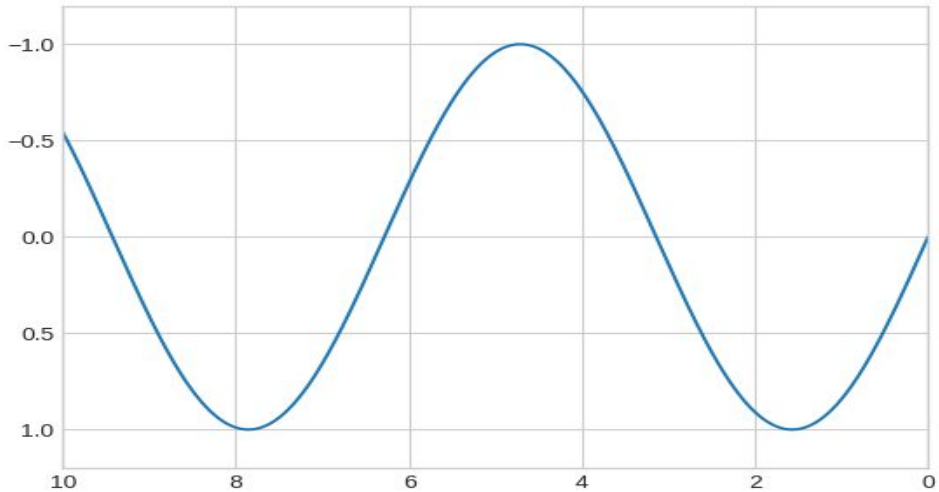
- **Manual limits:**

```
plt.plot(x, np.sin(x))
plt.xlim(-1, 11)  # x-axis range
plt.ylim(-1.5, 1.5);  # y-axis range
```



Reversing axes: Reverse the order of arguments to flip an axis direction.

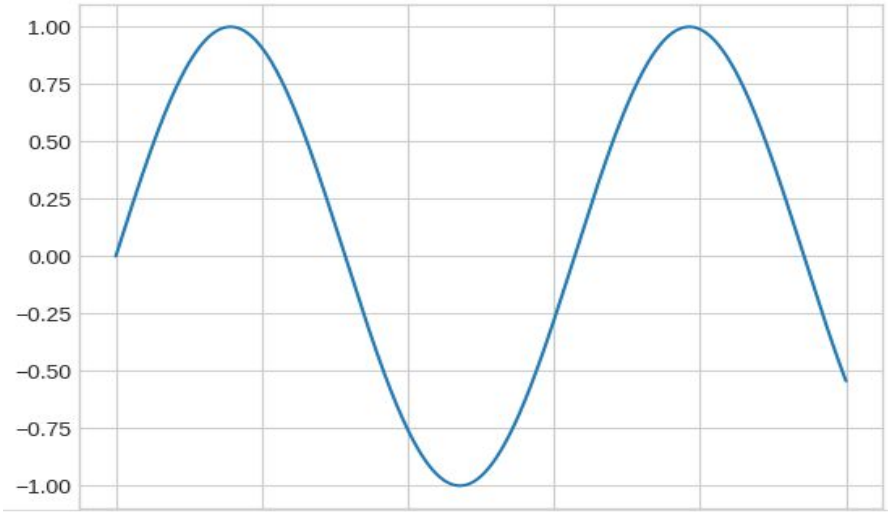
```
plt.plot(x, np.sin(x))
plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```



Automatic adjustments:

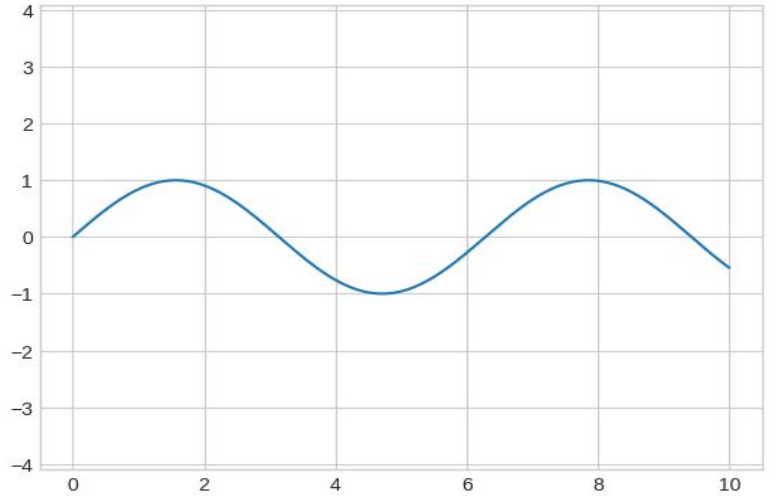
- `tight`: **Shrinks axes to fit the data snugly.**

```
plt.plot(x, np.sin(x))
plt.axis('tight');
```



- `equal`: **Ensures equal aspect ratio for x and y.**

```
plt.plot(x, np.sin(x))
plt.axis('equal');
```

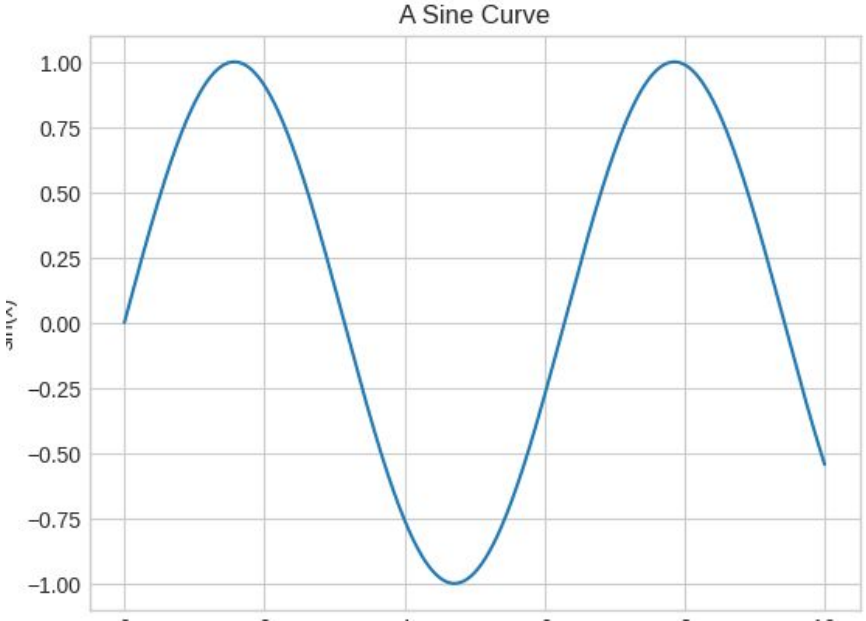


Labeling Plots

Add descriptive titles and axis labels to enhance clarity:

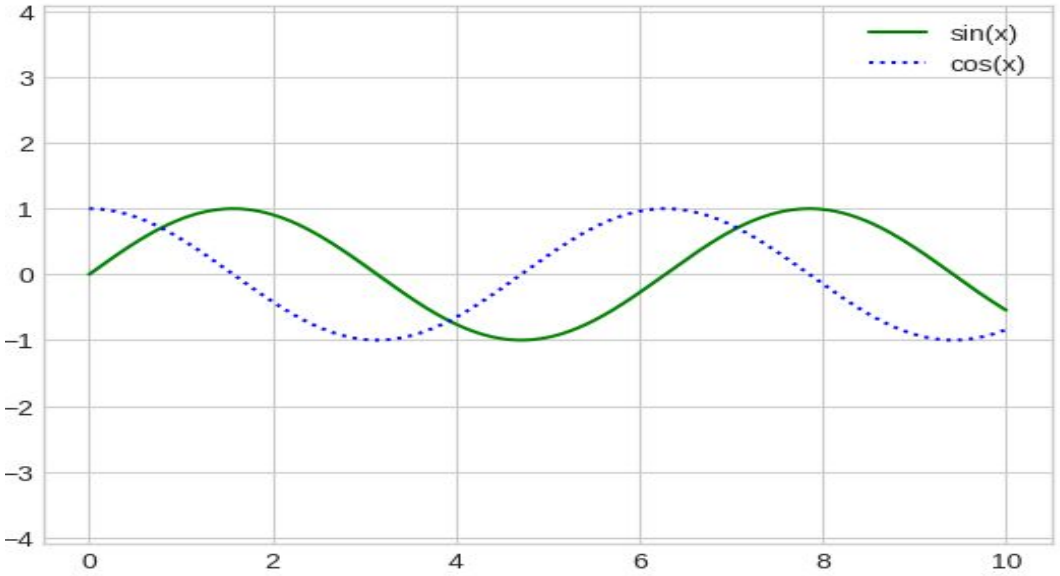
- **Title and axis labels:**

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")  # Plot title
plt.xlabel("x")            # x-axis label
plt.ylabel("sin(x)");      # y-axis label
```



- **Legends:**
- **Use `plt.legend` to identify different lines in a plot.**

```
plt.plot(x, np.sin(x), '-g', label='sin(x)')  # Label sine curve
plt.plot(x, np.cos(x), ':b', label='cos(x)')  # Label cosine curve
plt.axis('equal')
plt.legend();  # Display legend
```



Matplotlib Gotchas

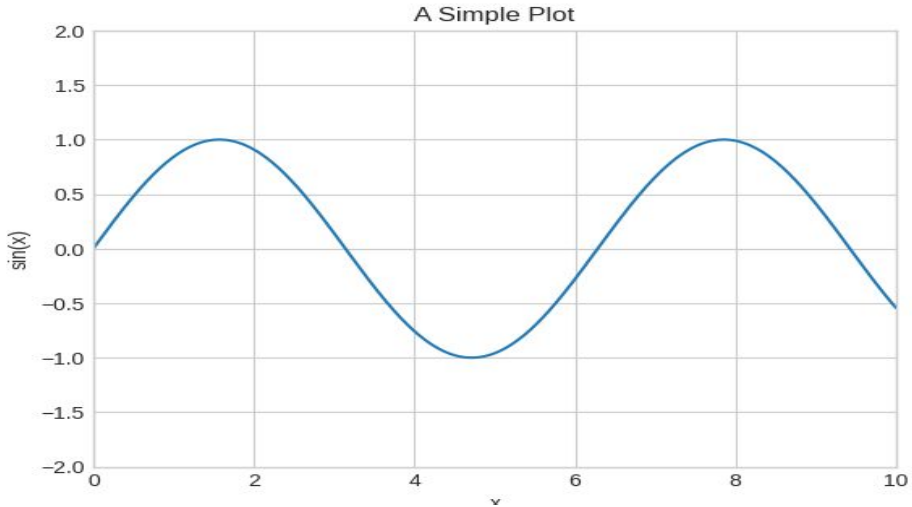
- **Transitioning from MATLAB-style to object-oriented commands:**

MATLAB-Style Command	Object-Oriented Command
`plt.xlabel`	`ax.set_xlabel`
`plt.ylabel`	`ax.set_ylabel`
`plt.xlim`	`ax.set_xlim`
`plt.ylim`	`ax.set_ylim`
`plt.title`	`ax.set_title`

- **Setting multiple properties at once:**
  The object-oriented interface simplifies setting properties.

```
ax = plt.axes()
ax.plot(x, np.sin(x))
ax.set(xlim=(0, 10), ylim=(-2, 2),
       xlabel='x', ylabel='sin(x)',
       title='A Simple Plot');
```

ylim: Any



Simple Scatter Plots:

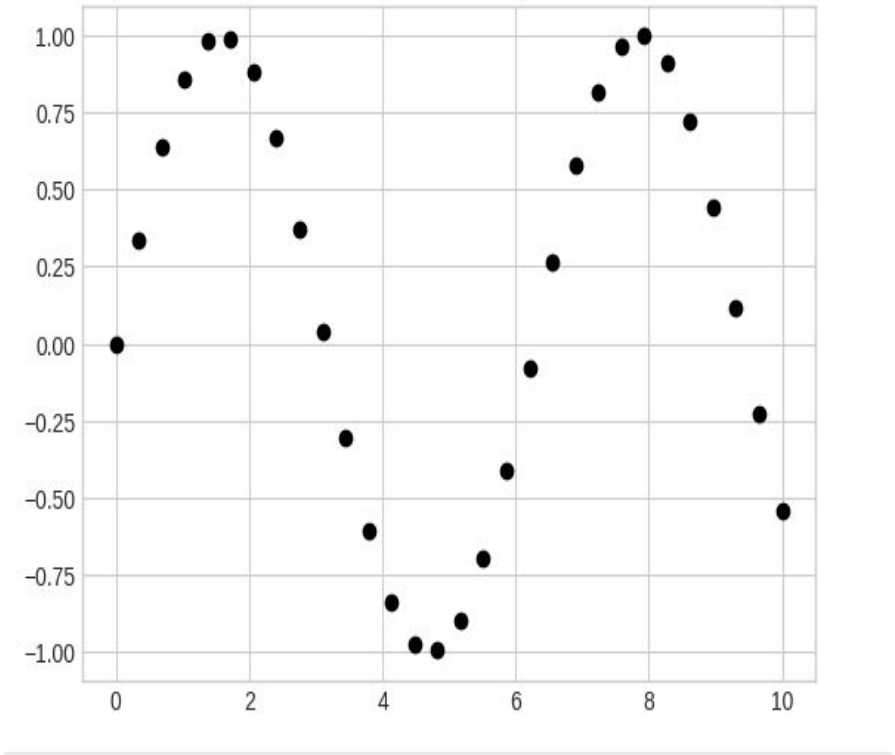
- **Setup**
- **To create scatter plots, begin by importing necessary libraries and setting the style:**

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-whitegrid')
import numpy as np
```

- **Scatter Plots with `plt.plot`**
- **The `plt.plot` function is versatile, allowing scatter plots by specifying marker styles.**

```
x = np.linspace(0, 10, 30)  # Generate 30 points between 0 and 10
y = np.sin(x)               # Apply sine function

plt.plot(x, y, 'o', color='black');  # 'o' specifies circle markers
```

- **Explanation: The `'o'` argument defines the marker style as circles. The `color='black'` argument sets the marker color.**
- **Graph: Points are displayed as black circles without connecting lines.**

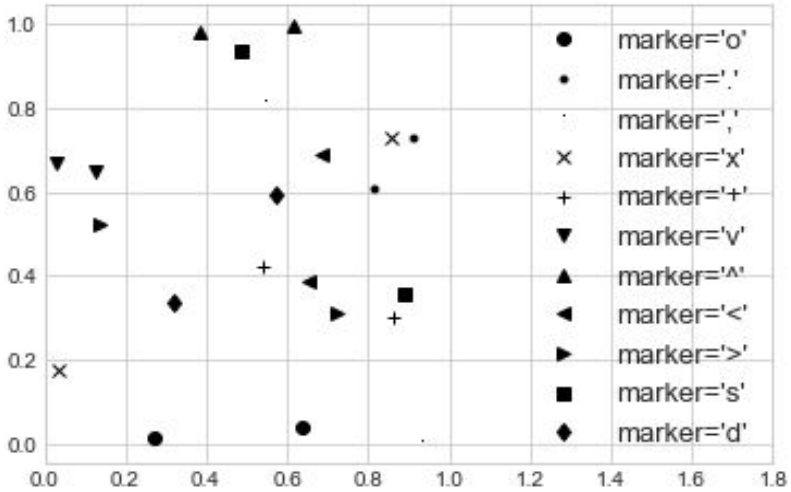


Marker Styles

- You can experiment with various marker styles like `'x'`, `'^'`, `s`, etc.

```
rng = np.random.default_rng(0)  # Random number generator
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.random(2), rng.random(2), marker, color='black',
             label=f"marker='{marker}'")
plt.legend(numpoints=1, fontsize=13)  # Add legend
plt.xlim(0, 1.8);
```

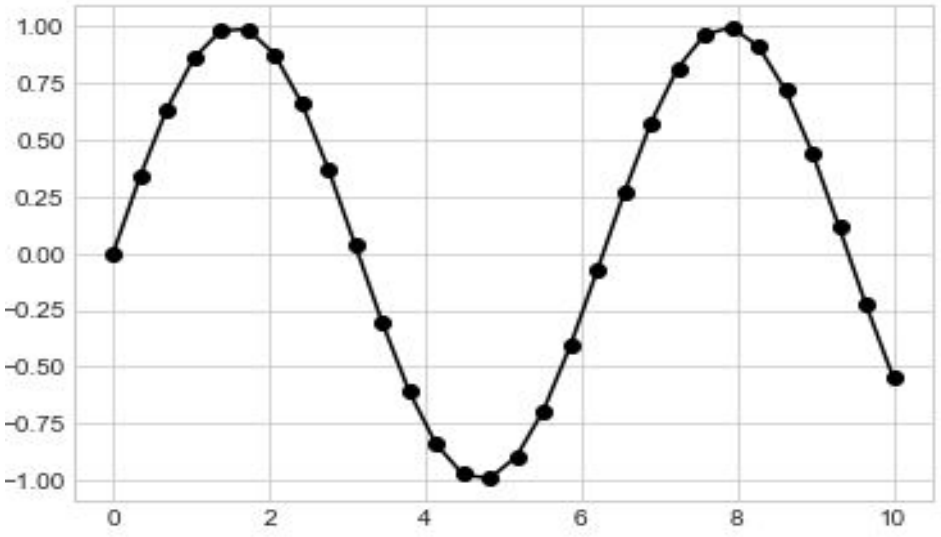
- The loop iterates over marker styles.
- The `label` argument identifies each marker in the legend.
- The `plt.legend()` function displays a legend for the markers.



Combining Markers with Lines

- **Markers can be used alongside lines for enhanced visuals.**

```
plt.plot(x, y, '-ok')  # Black line ('-') with circle markers ('o')
```

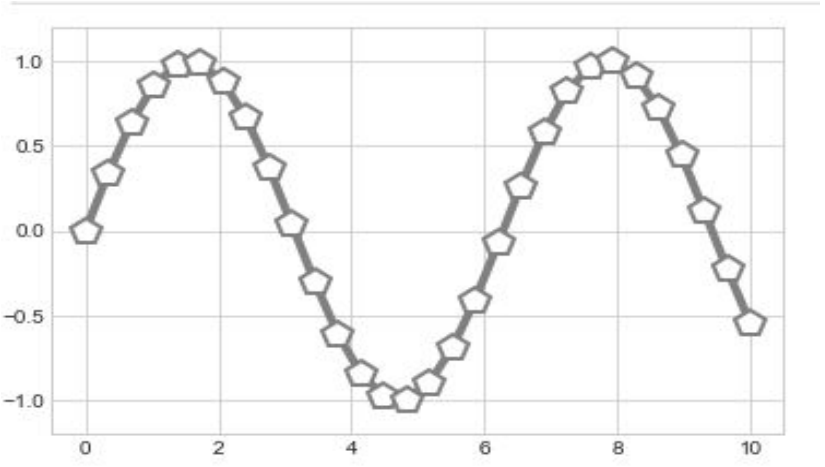


- **`'-ok'`: Combines a solid line (`-`), black color (`k`), and circle markers (`o`).**

- **Customizing Marker Appearance**
- **Markers can be adjusted further using properties like size, edge color, and face color.**

```
plt.plot(x, y, '-p', color='gray', markersize=15, linewidth=4,
         markerfacecolor='white', markeredgecolor='gray',
         markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

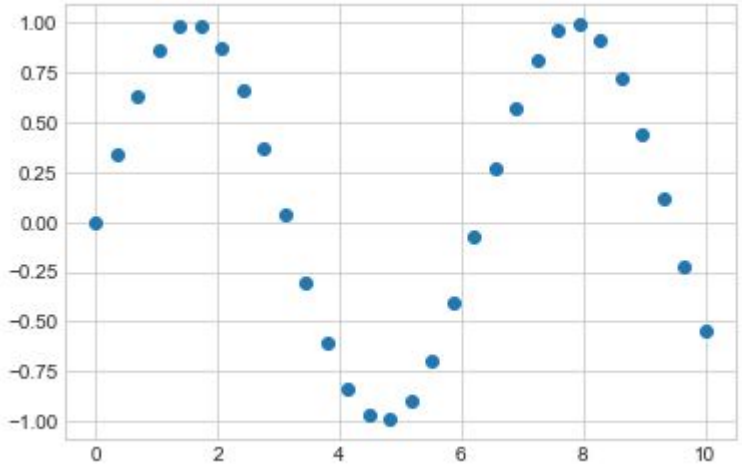
- `markersize=15`: **Enlarges markers.**
- `linewidth=4`: **Thickens the connecting line.**
- `markerfacecolor='white'`: **Sets the marker fill color to white.**
- `markeredgecolor='gray'`: **Sets the marker edge color to gray.**
- `markeredgewidth=2`: **Sets the thickness of the marker edge.**



Scatter Plots with `plt.scatter`

- The `plt.scatter` function offers more flexibility for scatter plots, allowing per-point customization.

```
plt.scatter(x, y, marker='o');
```



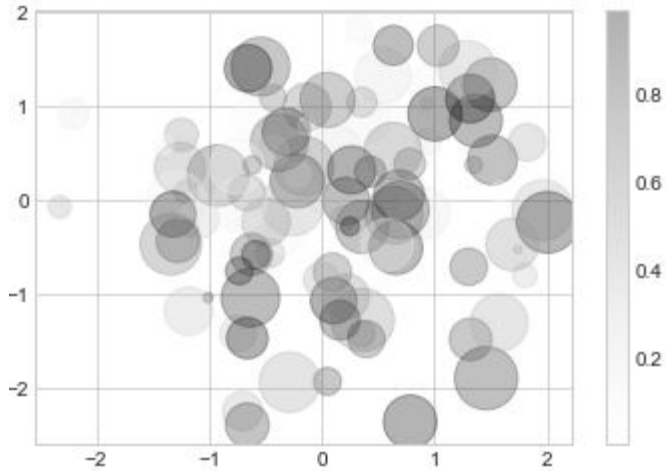
- Explanation: The `plt.scatter` function plots the same points as `plt.plot`, but with more customization options.

Customizing Points

- **Scatter plots can convey additional information using point size, color, and transparency.**

```
rng = np.random.default_rng(0)
x = rng.normal(size=100)  # Generate 100 random x-coordinates
y = rng.normal(size=100)  # Generate 100 random y-coordinates
colors = rng.random(100)  # Generate random colors
sizes = 1000 * rng.random(100)  # Generate random sizes for markers

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3)  # Add transparency
plt.colorbar();  # Display a color scale
```

- `c=colors`: **Sets the color of each point based on an array.**
- `s=sizes`: **Sets the size of each point.**
- `alpha=0.3`: **Makes points partially transparent.**
- `plt.colorbar()`: **Adds a color bar to explain the color mapping.**

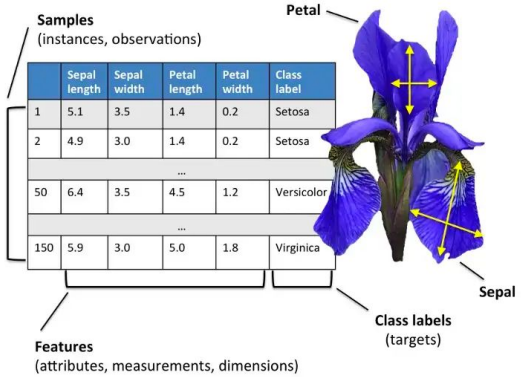


Multidimensional Data with `plt.scatter`

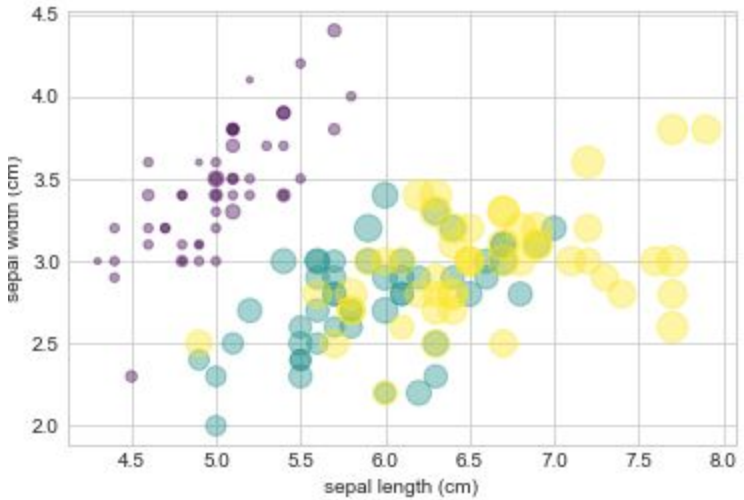
Scatter plots are excellent for visualizing multidimensional datasets.

Example: Iris Dataset

```
from sklearn.datasets import load_iris
iris = load_iris()  # Load the dataset
features = iris.data.T  # Transpose to separate features

plt.scatter(features[0], features[1], alpha=0.4,
            s=100 * features[3], c=iris.target, cmap='viridis')
plt.xlabel(iris.feature_names[0])  # Label x-axis
plt.ylabel(iris.feature_names[1]); # Label y-axis
```



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica



- **X-axis → Sepal length.**
- **Y-axis → Sepal width.**
- **Point size → Petal width.**
- **Point color → Flower species.**
- **`cmap='viridis'`: Sets the colormap for color mapping.**

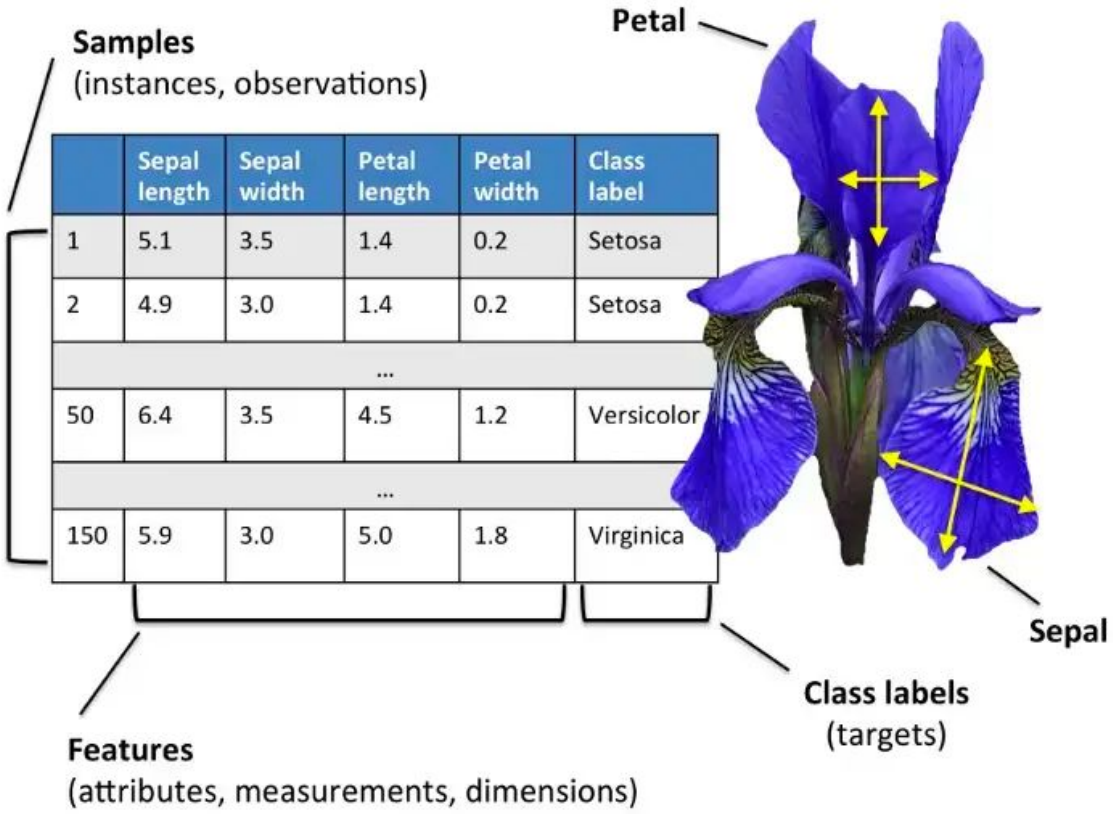
Multidimensional

Scatter plots are ex
multidimensional da

Example: Iris Datas

```
from sklearn.datasets
iris = load_iris()  #
features = iris.data.T

plt.scatter(features[0
            s=100 * fe
plt.xlabel(iris.featur
plt.ylabel(iris.featur
```

- **X-axis → S**
- **Y-axis → S**
- **Point size**
- **Point colo**
- **`cmap='vi`**
  mapping.



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

Efficiency: `plt.plot` vs `plt.scatter`

- `plt.plot`: More efficient for large datasets; markers are uniform, reducing rendering overhead.
- `plt.scatter`: Slower but allows customization of individual points (size, color, etc.).

Performance Tip

Use `plt.plot` for datasets with thousands of points, and `plt.scatter` when detailed customization is required.

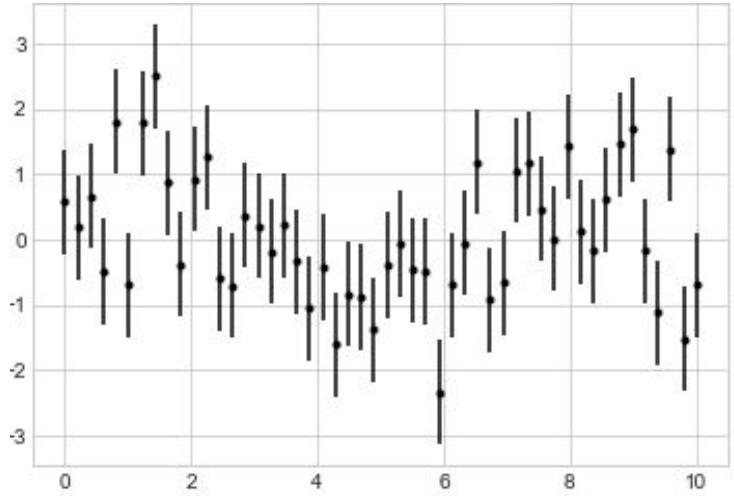
Importance of Visualizing Uncertainties

- **Accurate reporting of uncertainties is crucial for interpreting scientific measurements.**
- **Example:**
- **Without uncertainties: A measured value of 74 (km/s)/Mpc compared to a literature value of 70 (km/s)/Mpc cannot be assessed for consistency.**
- **With uncertainties: Adding uncertainties like 70 ± 2.5 and 74 ± 5 allows quantitative analysis of consistency.**

Basic Errorbars

Errorbars are a standard way to represent uncertainties visually.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-whitegrid')
import numpy as np

x = np.linspace(0, 10, 50)
dy = 0.8  # Define uncertainty
y = np.sin(x) + dy * np.random.randn(50)  # Data with noise

plt.errorbar(x, y, yerr=dy, fmt='.k');  # Plot with errorbars
```

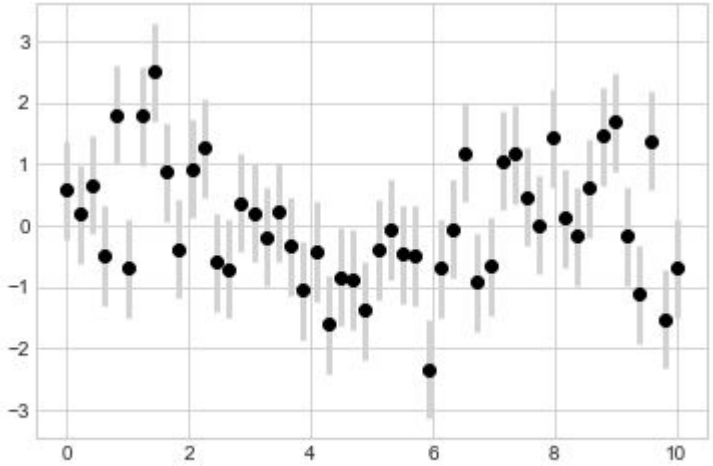


- `yerr=dy`: Specifies vertical error magnitude.
- `fmt='.k'`: Defines marker style (`.`) and color (`k` for black).

Graph: Points with vertical errorbars.

Customized Errorbars

```
plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
             ecolor='lightgray', elinewidth=3, capsize=0);
```



- `color='black'`: Black marker color.
- `ecolor='lightgray'`: Light gray errorbar color.
- `elinewidth=3`: Thicker errorbar lines.
- `capsize=0`: Removes caps at the ends of errorbars.

This customization enhances clarity, especially in crowded plots.

Continuous Errors

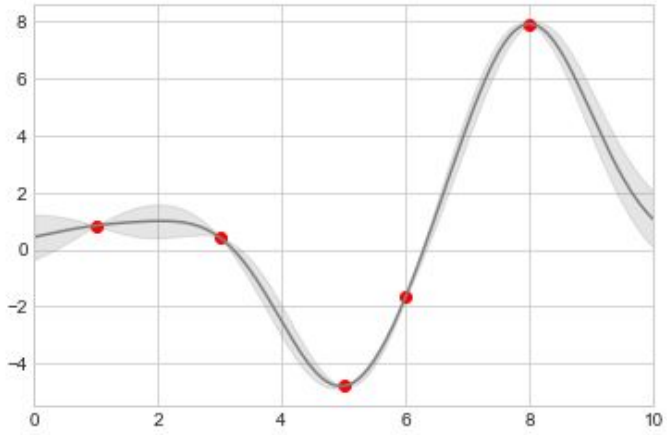
For continuous quantities, uncertainty can be visualized using a shaded region.

```
from sklearn.gaussian_process import GaussianProcessRegressor

# Define the model and generate data
model = lambda x: x * np.sin(x)
xdata = np.array([1, 3, 5, 6, 8])
ydata = model(xdata)

# Compute the Gaussian process fit
gp = GaussianProcessRegressor()
gp.fit(xdata[:, np.newaxis], ydata)

xfit = np.linspace(0, 10, 1000)  # Fine grid for fit
yfit, dyfit = gp.predict(xfit[:, np.newaxis], return_std=True)  # Predictions
```

- **Gaussian Process Regression: Fits a flexible model to the data, providing predictions (`yfit`) and associated uncertainties (`dyfit`).**

Visualizing Continuous Errors

```
# Visualize results
plt.plot(xdata, ydata, 'or')  # Data points in red
plt.plot(xfit, yfit, '-', color='gray')  # Fit line in gray
plt.fill_between(xfit, yfit - dyfit, yfit + dyfit,
                 color='gray', alpha=0.2)  # Shaded region for uncertainty
plt.xlim(0, 10);
```



- `plt.fill_between`: Fills the area between the bounds of uncertainty.
- `yfit - dyfit` and `yfit + dyfit`: Define the lower and upper bounds.
- `alpha=0.2`: Sets transparency for the shaded region.

Graph:

- **Data points in red.**
- **A gray line represents the model fit.**
- **Shaded regions show uncertainty, with wider areas indicating greater uncertainty far from data points.**

Use Cases

- **Errorbars: Ideal for discrete data points with uncertainties.**
- **Continuous Errors: Suitable for functions or predictions with continuous uncertainties.**

Tools for Advanced Visualization

- `plt.fill_between`**: Customizable and precise for low-level control of shaded error regions.**
- **Seaborn: Offers a more streamlined API for creating similar plots effortlessly.**

Density and Contour Plots

Visualizing Three-Dimensional Data in Two Dimensions

Three Matplotlib functions are commonly used for visualizing three-dimensional data in two dimensions:

1. `plt.contour`: **Line-based contour plots.**
2. `plt.contourf`: **Filled contour plots.**
3. `plt.imshow`: **Image plots with color-coding.**

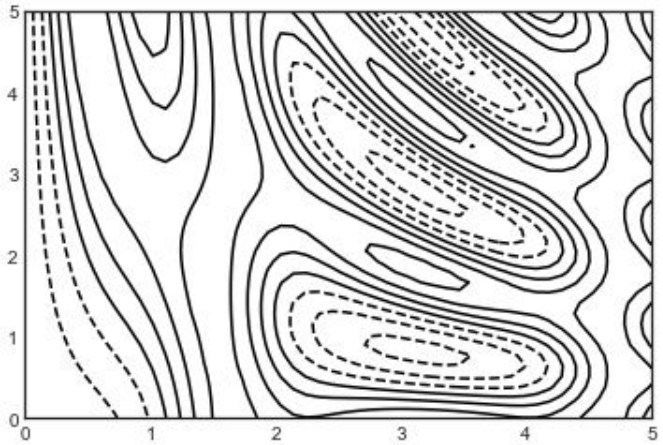
```
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-white')
import numpy as np

def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

Explanation: The function f(x,y)f(x, y)f(x,y) is defined as a combination of sine and cosine terms for a complex surface.

Generating Data for the Plot

```
x = np.linspace(0, 5, 50)  # X-axis values
y = np.linspace(0, 5, 40)  # Y-axis values

X, Y = np.meshgrid(x, y)  # Create 2D grids
Z = f(X, Y)  # Evaluate the function
plt.contour(X, Y, Z, colors='black');
```

- `np.meshgrid`: Constructs 2D coordinate grids from 1D arrays.
- ZZZ: The function values at all points in the grid.



Contour Plots

Basic Line Contours

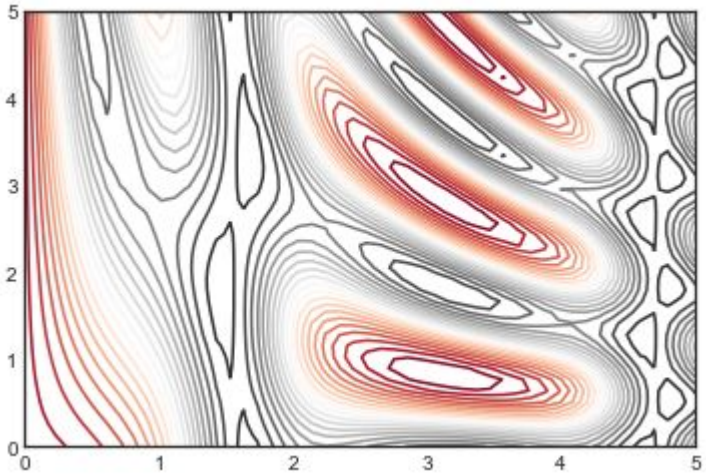
```
plt.contour(X, Y, Z, colors='black');
```

Explanation:

- `colors='black'`: Draws all contours in black.
- Graph: Solid lines represent positive values, dashed lines for negative values.

Colormap and Custom Intervals

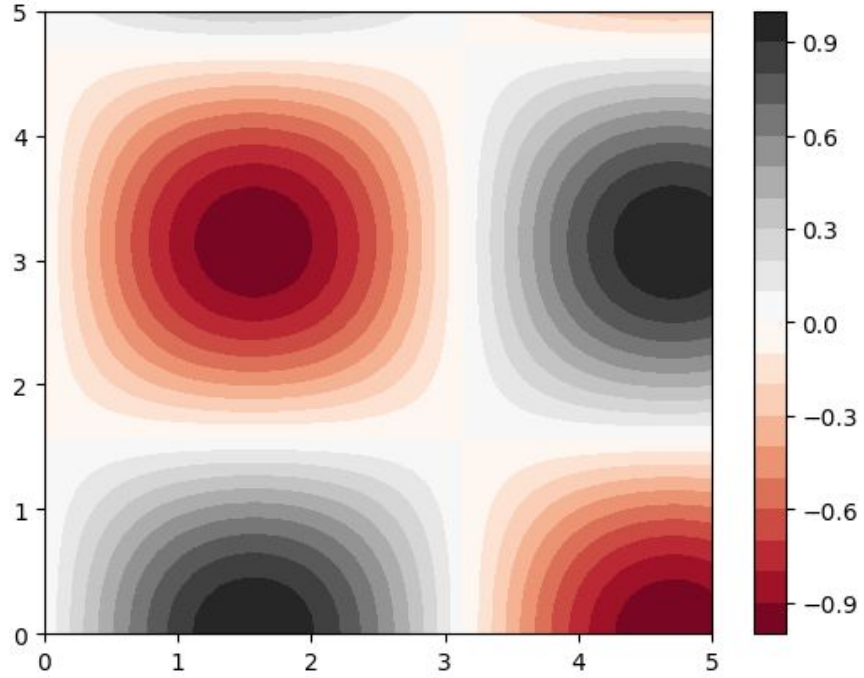
```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```



- `20`: Number of contour levels.
- `cmap='RdGy'`: Red-Gray colormap, ideal for data varying around zero.
- Graph: Lines are color-coded, providing more intuitive visual differentiation.

Filled Contour Plots

```python
import numpy as np
import matplotlib.pyplot as plt  # Ensure pyplot is imported as plt
# Define the grid
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)
X, Y = np.meshgrid(x, y)

# Define the function
def f(x, y):
    return np.sin(x) * np.cos(y)

# Evaluate the function
Z = f(X, Y)
plt.contourf(X, Y, Z, 20, cmap='RdGy')  # Filled contours
plt.colorbar();  # Add colorbar
```

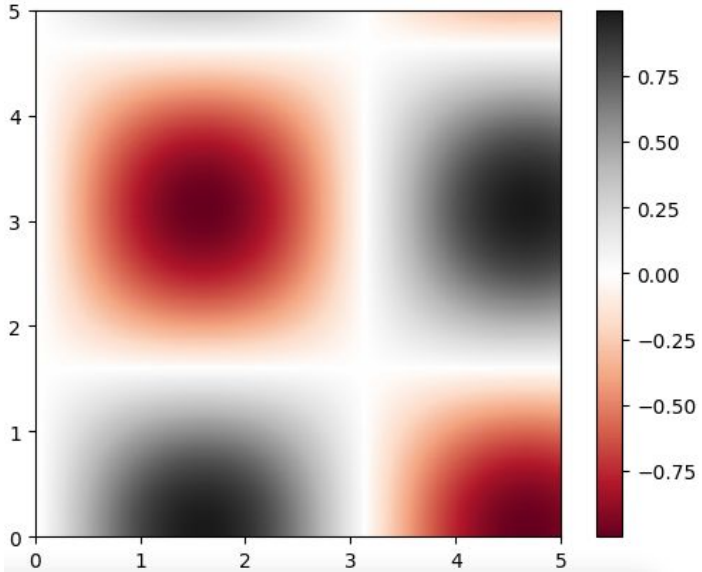


- `plt.contourf`: Creates filled regions between contour levels.
- `plt.colorbar`: Displays a color scale indicating the data range.
- Graph: Clearer depiction of "peaks" (black) and "valleys" (red) using filled colors.

Smoothing Discrete Steps

```
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower', cmap='RdGy',
           interpolation='gaussian', aspect='equal')
plt.colorbar();
```

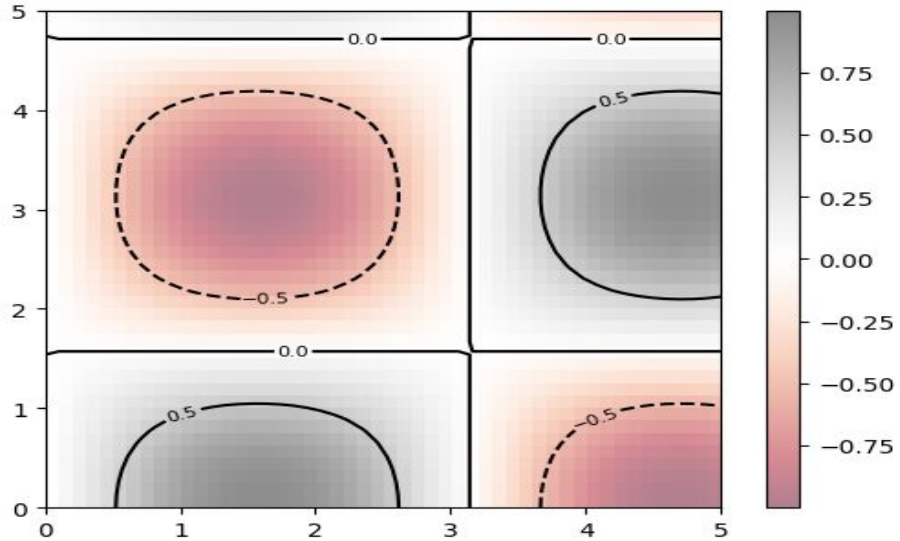
- `plt.imshow`: Provides a smooth 2D representation of ZZZ.
- `interpolation='gaussian'`: Smooths the data for continuous shading.
- `extent=[xmin, xmax, ymin, ymax]`: Maps the grid to specific axis ranges.
- Graph: A visually smooth surface without discrete contour steps.



Combining Contour and Image Plots

```
contours = plt.contour(X, Y, Z, 3, colors='black')  # Add contours
plt.clabel(contours, inline=True, fontsize=8)  # Label contours

plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
           cmap='RdGy', alpha=0.5)  # Add transparent background
plt.colorbar();
```

- `plt.contour` and `plt.imshow` are combined for enhanced visualization.
- `alpha=0.5`: Adds transparency to the background image.
- `plt.clabel`: Labels contours directly on the plot.
- Graph: Highlights contours over a smooth color-coded background.



Considerations with `plt.imshow`

1. **Gridless Axes:**
   - `plt.imshow` doesn't use xxx and yyy grids; manual extents must be specified.
2. **Origin:**
   - Defaults to the top-left (`origin='upper'`). Use `origin='lower'` for gridded data.
3. **Aspect Ratio:**
   - Automatically adjusts to data; override with `aspect='equal'`.

Use Cases

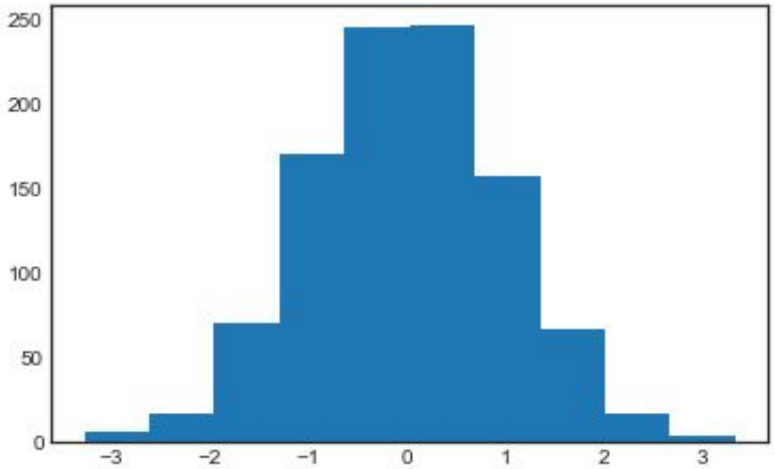
- `plt.contour`: Highlight specific data levels.
- `plt.contourf`: Ideal for filled visualizations.
- `plt.imshow`: Suitable for smooth color transitions.
- Combining Plots: Achieves detailed and visually appealing representations

Histograms, Binnings, and Density

One-Dimensional Histograms

Histograms provide a visual representation of data distribution. Using Matplotlib's `plt.hist` function, you can quickly create a histogram.

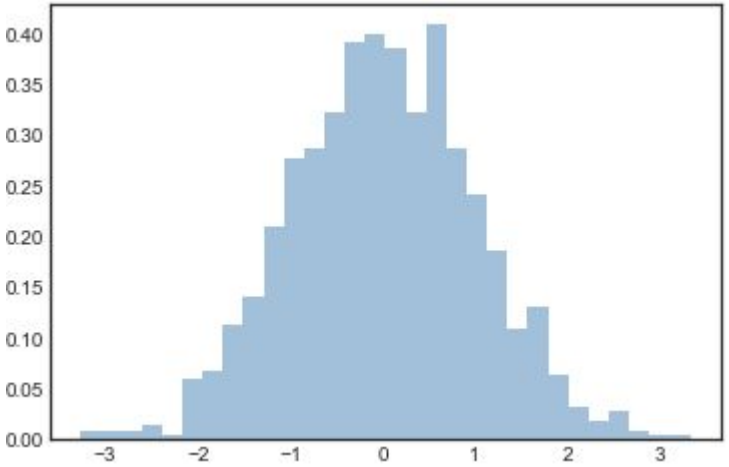
```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-v0_8-white')

rng = np.random.default_rng(1701)
data = rng.normal(size=1000)  # Generate normal distribution
plt.hist(data);
```



- **`plt.hist(data)`: Automatically computes bins and counts.**
- **Graph: Displays a simple histogram of the data distribution.**

Customized Histogram

```
plt.hist(data, bins=30, density=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='none');
```

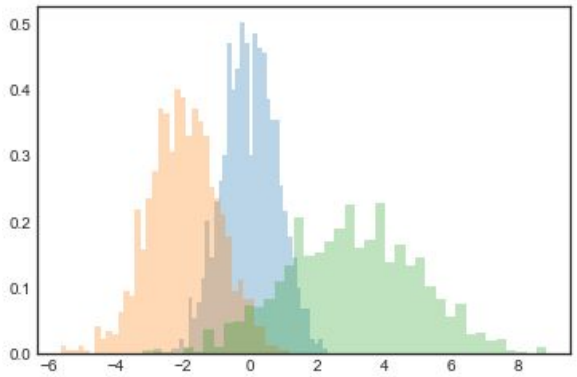


Key Customizations:

- `bins=30`: Specifies 30 equal-width bins.
- `density=True`: Normalizes the histogram.
- `alpha=0.5`: Adjusts transparency.
- `histtype='stepfilled'`: Creates a filled step plot.

Comparing Distributions

```
x1 = rng.normal(0, 0.8, 1000)
x2 = rng.normal(-2, 1, 1000)
x3 = rng.normal(3, 2, 1000)

kwargs = dict(histtype='stepfilled', alpha=0.3, density=True, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```



- **Multiple datasets are compared using transparency and a consistent style.**

Computing Histogram Values

```
counts, bin_edges = np.histogram(data, bins=5)
print(counts)  # Output: [ 23 241 491 224  21]
```

```
[ 23 241 491 224  21]
```

- `np.histogram`: Calculates bin counts without plotting.
- Output: Array of counts for each bin.

Two-Dimensional Histograms

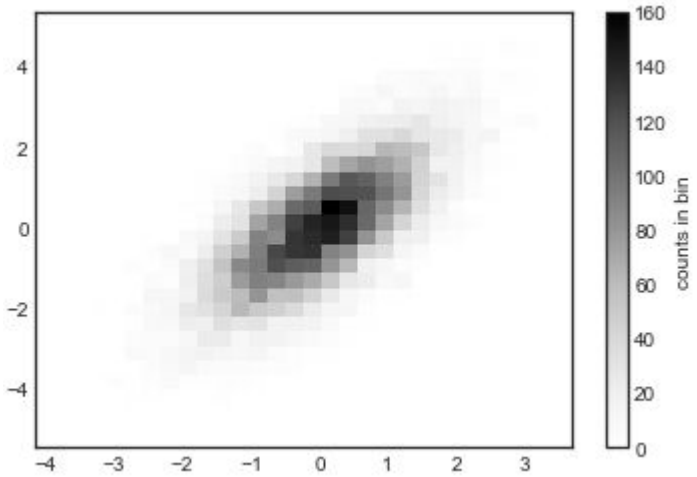
Histograms can be extended to two dimensions by dividing data into a 2D grid of bins.

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = rng.multivariate_normal(mean, cov, 10000).T
```

- **Data is drawn from a multivariate normal distribution.**

2D Histogram with `plt.hist2d`

```
plt.hist2d(x, y, bins=30)
cb = plt.colorbar()
cb.set_label('counts in bin')
```



- `plt.hist2d`: Creates a 2D histogram with square bins.
- `plt.colorbar`: Adds a color scale to represent bin counts.

Computing 2D Histogram Values

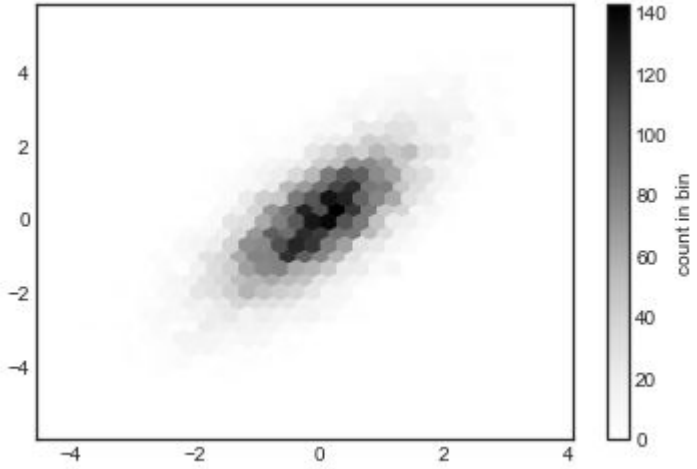
```
counts, xedges, yedges = np.histogram2d(x, y, bins=30)
print(counts.shape)  # Output: (30, 30)
```

```
(30, 30)
```

- `np.histogram2d`: **Computes counts for 2D bins.**

Hexagonal Binning with `plt.hexbin`

```
plt.hexbin(x, y, gridsize=30)
cb = plt.colorbar(label='count in bin')
```



- `plt.hexbin`: Uses hexagonal bins instead of squares.
- `gridsize=30`: Specifies the resolution of the hexagon grid.

Kernel Density Estimation (KDE)

KDE is a method for estimating data density by "smoothing" discrete points.

Simple KDE with `scipy.stats.gaussian_kde`

```
from scipy.stats import gaussian_kde

data = np.vstack([x, y])  # Combine x and y data
kde = gaussian_kde(data)  # Fit KDE model

# Evaluate KDE on a grid
xgrid = np.linspace(-3.5, 3.5, 40)
ygrid = np.linspace(-6, 6, 40)
Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))

# Plot the KDE
plt.imshow(Z.reshape(Xgrid.shape), origin='lower', aspect='auto',
           extent=[-3.5, 3.5, -6, 6])
cb = plt.colorbar()
cb.set_label("density")
```

- **`gaussian_kde`: Fits a KDE model to data.**
- **`Z`: KDE values evaluated on a 2D grid.**
- **Graph: Smooth density visualization with continuous color transitions**

Advantages of KDE

- **Provides a continuous density representation.**
- **Adjusting the smoothing length balances detail and smoothness.**

Choosing Between Methods

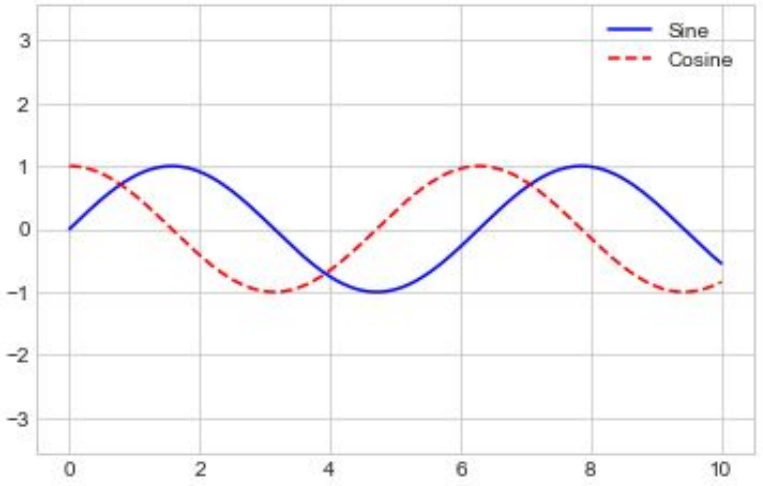
- **Histograms:**
  - **Best for discrete binning and direct count visualization.**
- **Hexbin:**
  - **Useful for large datasets; hexagons improve resolution.**
- **KDE**
  - **Ideal for smooth density estimation.**

Customizing Plot Legends in Matplotlib

1. Basic Plot Legend

Plot legends are essential for identifying plot elements. A basic legend can be created using the `plt.legend()` or `ax.legend()` command.

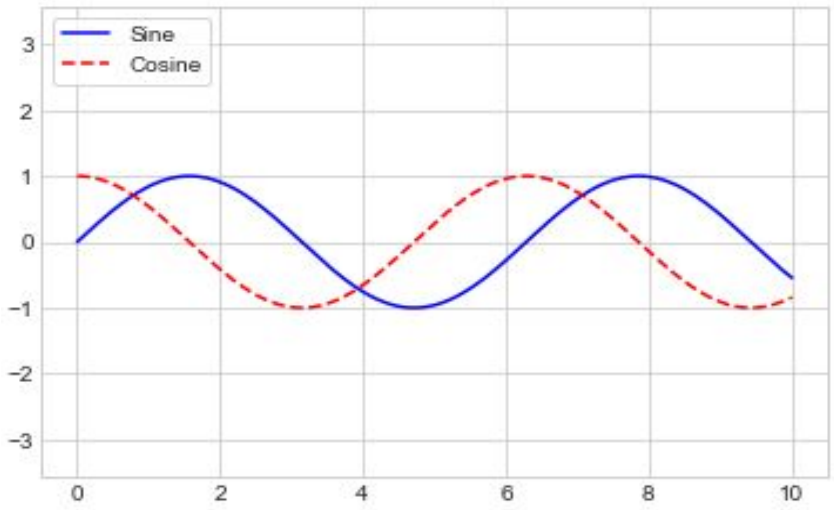
```
import matplotlib.pyplot as plt
import numpy as np
plt.style.use('seaborn-v0_8-white')

x = np.linspace(0, 10, 1000)
fig, ax = plt.subplots()

ax.plot(x, np.sin(x), '-b', label='Sine')  # Blue solid line
ax.plot(x, np.cos(x), '--r', label='Cosine')  # Red dashed line
ax.axis('equal')

leg = ax.legend()  # Automatically adds a legend for labeled elements
```



Customizing Legend Location and Appearance

Specifying Location

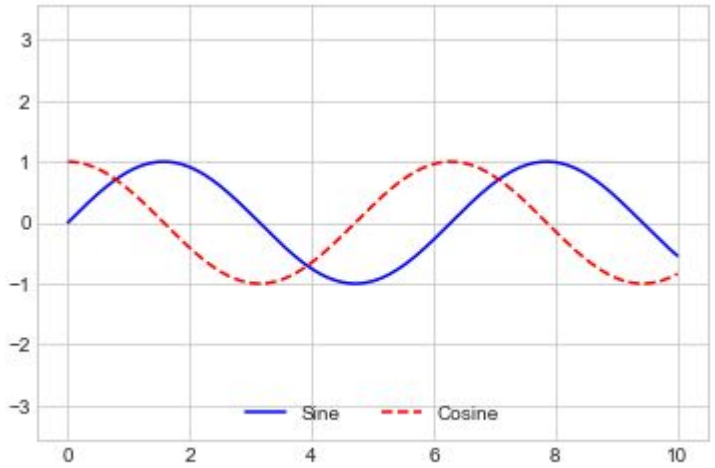
```
ax.legend(loc='upper left', frameon=True)
fig
```



- **loc='upper left': Places the legend in the upper-left corner.**
- **frameon=True: Adds a frame around the legend.**

Adjusting Columns

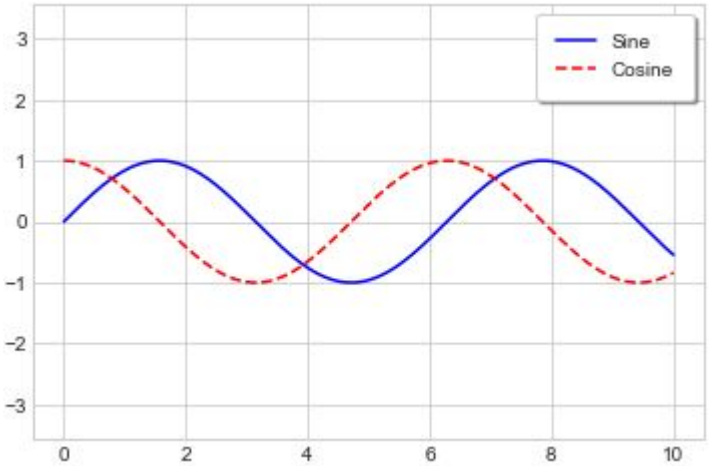
```
ax.legend(loc='lower center', ncol=2)
fig
```



- **ncol=2: Arranges legend items into two columns.**

Adding Rounded Frame and Shadows

```
ax.legend(frameon=True, fancybox=True, framealpha=1,
          shadow=True, borderpad=1)
fig
```



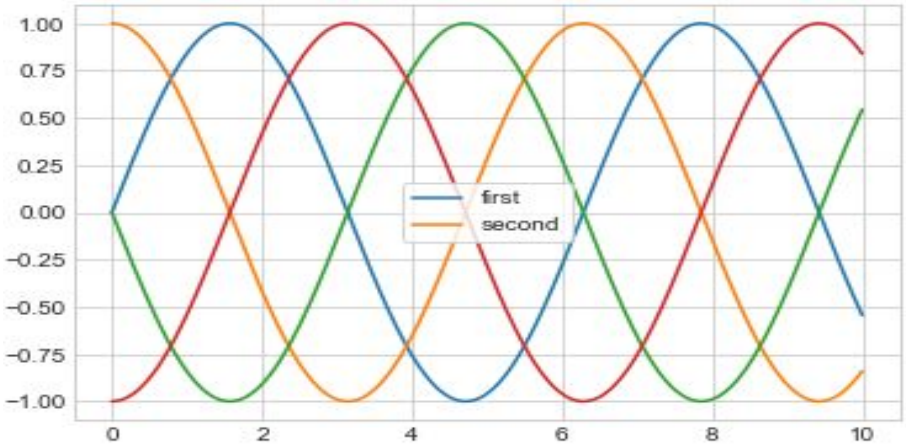
- **fancybox=True: Adds rounded corners to the frame.**
- **framealpha=1: Sets frame transparency (1 = opaque).**
- **shadow=True: Adds a shadow to the frame.**
- **borderpad=1: Increases padding around legend text.**

Controlling Legend Items

Manually Specifying Legend Items

```
y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.
lines = plt.plot(x, y)

# Specify specific lines for the legend
plt.legend(lines[:2], ['first', 'second'], frameon=True)
```

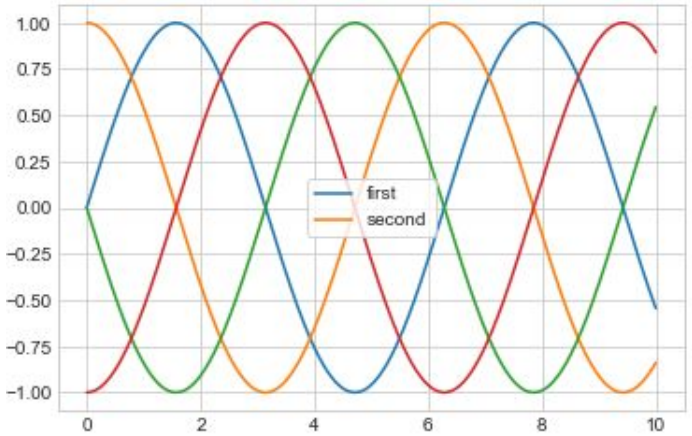
- **lines[:2]: Selects specific lines to include in the legend.**
- **Labels: Custom labels are provided as a list**



Using Labeled Plot Elements

```
plt.plot(x, y[:, 0], label='first')
plt.plot(x, y[:, 1], label='second')
plt.plot(x, y[:, 2:])

plt.legend(frameon=True)
```

```
<matplotlib.legend.Legend at 0x7e9d8bf02b00>
```



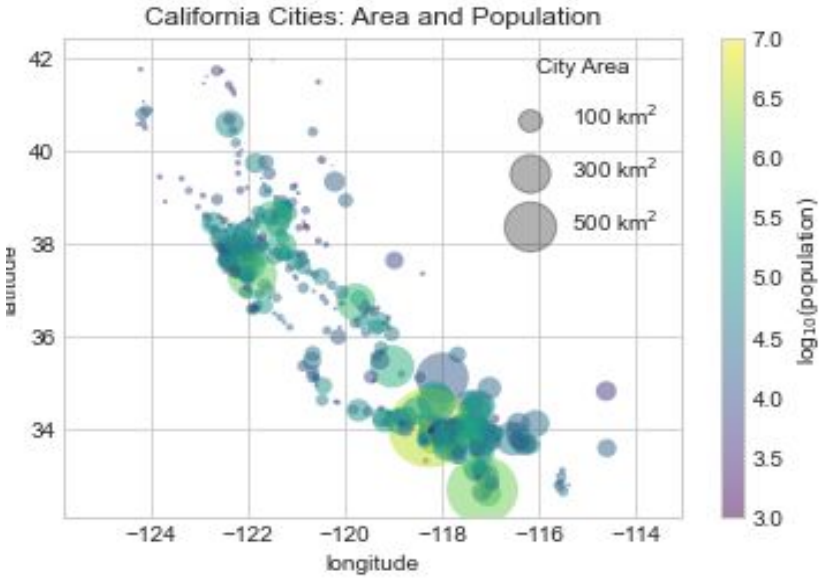
- **Only elements with a `label` attribute appear in the legend.**

Legends for Point Sizes

Sometimes point sizes represent features, like populations or areas. A legend can reflect this by plotting "fake" data points.

Example: City Data

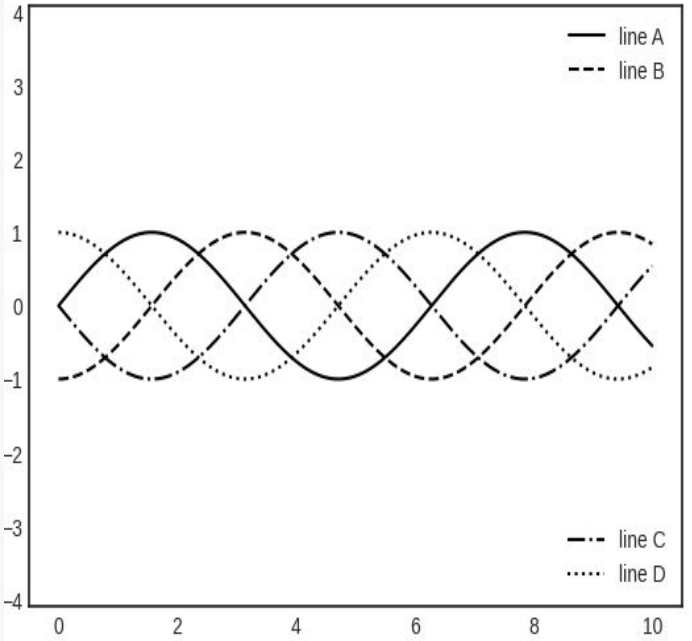
```
import pandas as pd
cities = pd.read_csv('data/california_cities.csv')

# Extract the data we're interested in
lat, lon = cities['latd'], cities['longd']
population, area = cities['population_total'], cities['area_total_km2']

# Scatter the points, using size and color but no label
plt.scatter(lon, lat, label=None,
            c=np.log10(population), cmap='viridis',
            s=area, linewidth=0, alpha=0.5)
plt.axis('equal')
plt.xlabel('longitude')
plt.ylabel('latitude')
plt.colorbar(label='log$_{10}$(population)')
plt.clim(3, 7)

# Here we create a legend:
# we'll plot empty lists with the desired size and label
for area in [100, 300, 500]:
    plt.scatter([], [], c='k', alpha=0.3, s=area,
                label=str(area) + ' km$^2$')
plt.legend(scatterpoints=1, frameon=False, labelspacing=1, title='City Area')

plt.title('California Cities: Area and Population');
```


California Cities: Area and Population
City Area
100 km²
300 km²
500 km²
latitude
longitude
log10(population)
42
40
38
36
34
−124
−122
−120
−118
−116
−114
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0

Adding Multiple Legends

Matplotlib only supports one legend per plot by default. To add multiple legends, you can use the `ax.add_artist()` method.

```
fig, ax = plt.subplots()

styles = ['-', '--', '-.', ':']
x = np.linspace(0, 10, 1000)
lines = []

# Create multiple lines
for i in range(4):
    lines += ax.plot(x, np.sin(x - i * np.pi / 2),
                     styles[i], color='black')
ax.axis('equal')

# First legend
ax.legend(lines[:2], ['line A', 'line B'], loc='upper right')

# Second legend
from matplotlib.legend import Legend
leg = Legend(ax, lines[2:], ['line C', 'line D'], loc='lower right')
ax.add_artist(leg)
```



Key Customization Tips

- **loc: Specifies position of the legend (e.g., `'upper left'`, `'center'`).**
- **frameon: Toggles the frame around the legend.**
- **fancybox: Adds rounded corners.**
- **ncol: Specifies number of columns in the legend.**
- **scatterpoints: Controls the number of points in a scatter legend.**

Customizing Colorbars

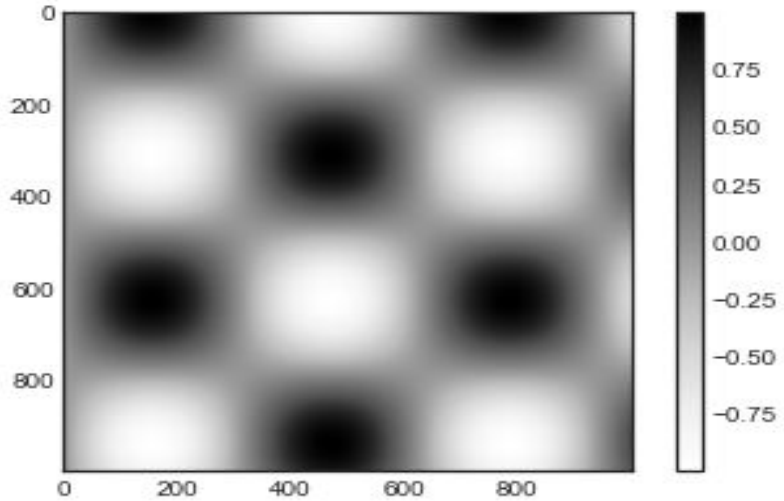
Colorbars are essential in visualizations for understanding the mapping of data values to colors. Below are various ways to create and customize them:

1. Basic Colorbar

A simple colorbar can be added to a plot using `plt.colorbar()`:

```
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])

plt.imshow(I)
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7e9d7fa5ae90>
```



Customizing Colormaps

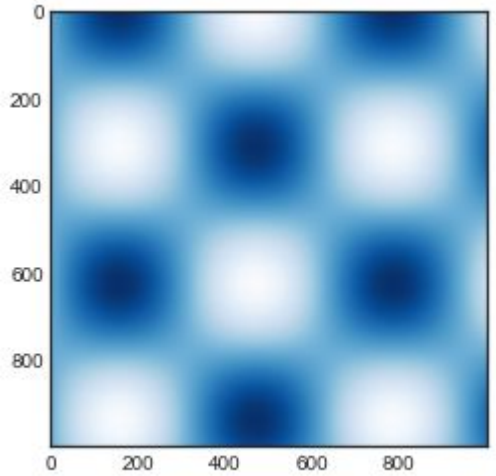
You can specify a colormap using the `cmap` argument. For example, to use a blue colormap:

```
plt.imshow(I, cmap='Blues')
```

Choosing the Right Colormap

Colormaps fall into three main categories:

- Sequential: A single, continuous range of colors. Examples: `viridis`, `Blues`.
- Divergent: Two contrasting colors representing deviations around a midpoint. Examples: `RdBu`, `PuOr`.
- Qualitative: Distinct colors with no inherent order. Example: `tab10`, `Set1`



Comparing Colormaps

The `view_colormap` function can visualize the brightness variation of a colormap and its grayscale equivalent.

Example:

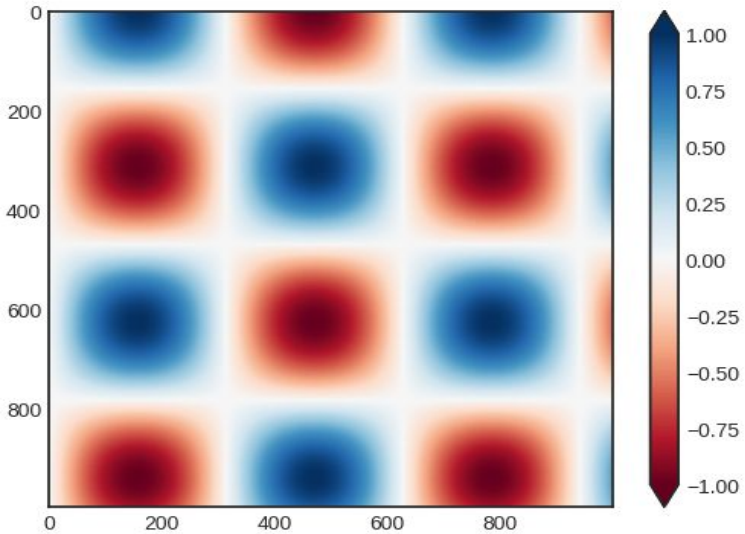
```
def view_colormap(cmap_name):
    """
    Visualize a colormap by displaying a gradient.

    Parameters:
        cmap_name (str): The name of the colormap to display.
    """
    gradient = np.linspace(0, 1, 256).reshape(1, -1)
    plt.figure(figsize=(8, 2))
    plt.imshow(gradient, aspect='auto', cmap=cmap_name)
    plt.title(f"Colormap: {cmap_name}", fontsize=12)
    plt.axis('off')
    plt.show()

view_colormap('jet')  # Shows uneven brightness
view_colormap('viridis')  # More uniform brightness
```

Adjusting Color Limits

The `extend` property allows you to indicate out-of-bound values in the colorbar:

```
plt.imshow(I, cmap='RdBu')
plt.colorbar(extend='both')
plt.clim(-1, 1)
```

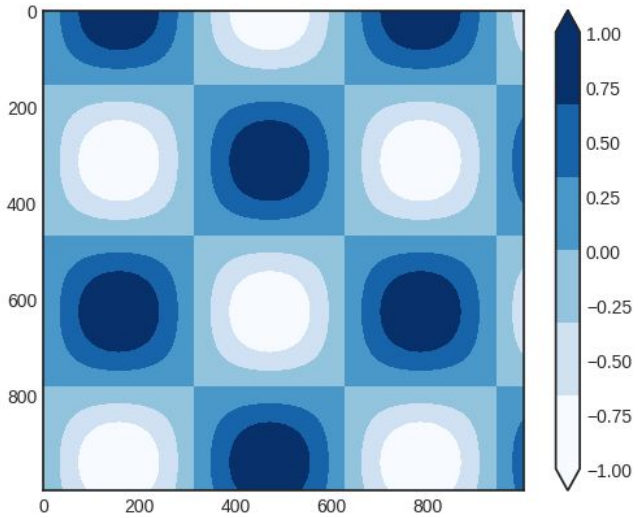
- **`extend='both'`: Adds arrows at both ends of the colorbar.**
- **`plt.clim(-1, 1)`: Restricts the displayed range of values.**



Discrete Colorbars

For discrete values, you can use a discrete colormap:

```
plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))
plt.colorbar(extend='both')
plt.clim(-1, 1)
```

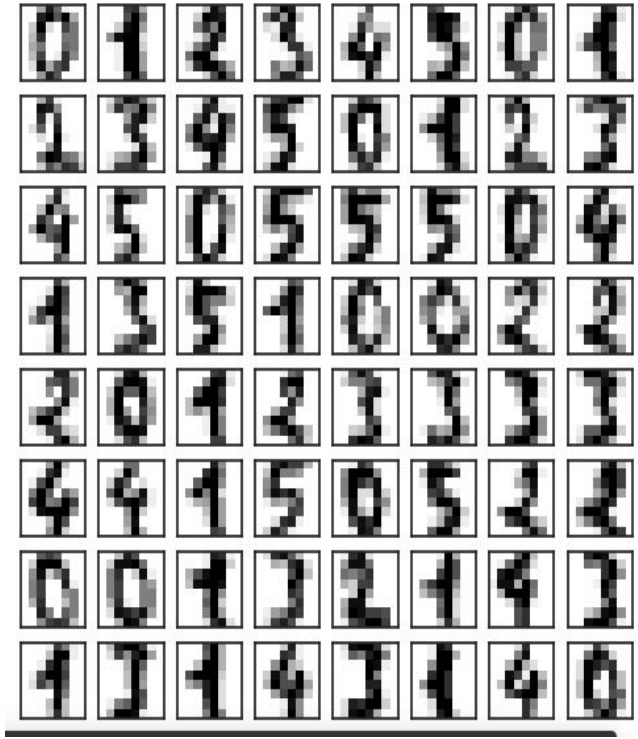


Example: Handwritten Digits

Dataset Visualization

To visualize the digits dataset from Scikit-Learn:

```
from sklearn.datasets import load_digits
digits = load_digits(n_class=6)

fig, ax = plt.subplots(8, 8, figsize=(6, 6))
for i, axi in enumerate(ax.flat):
    axi.imshow(digits.images[i], cmap='binary')
    axi.set(xticks=[], yticks=[])
```



Dimensionality Reduction

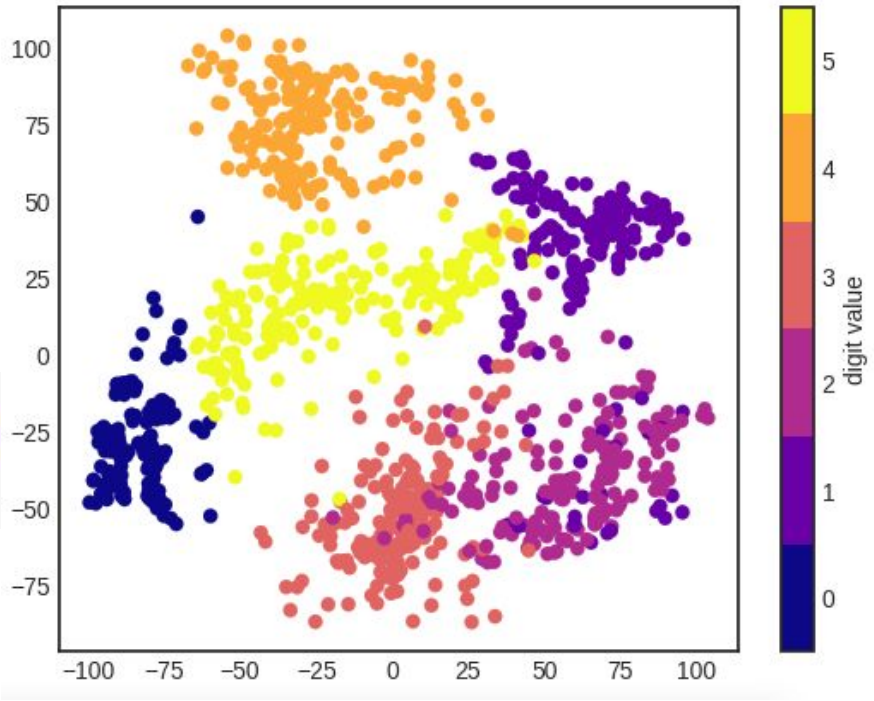
To reduce the dataset to 2D for visualization:

```
from sklearn.manifold import Isomap
iso = Isomap(n_components=2, n_neighbors=15)
projection = iso.fit_transform(digits.data)
```

Scatter Plot with Discrete Colorbar

Using the reduced dimensions:

```
plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,
            c=digits.target, cmap=plt.cm.get_cmap('plasma', 6))
plt.colorbar(ticks=range(6), label='digit value')
plt.clim(-0.5, 5.5)
```



This reveals relationships in the dataset:

- **Digits like 2 and 3 overlap, indicating potential classification confusion.**
- **Distinct digits like 0 and 1 are well-separated.**

Takeaways

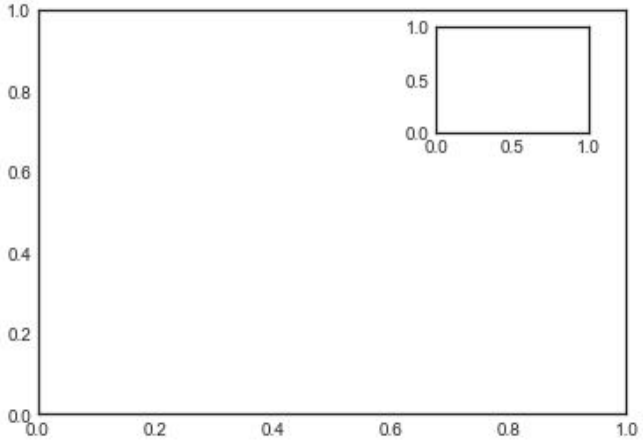
1. **Use sequential colormaps for continuous data and divergent colormaps for deviation-based visualizations.**
2. **Customize color limits and add extensions to improve clarity.**
3. **Use discrete colormaps and tick adjustments for categorical or grouped data.**
4. **Advanced visualizations, such as embedding data in 2D, require combining Matplotlib's colormaps with machine learning techniques like manifold learning.**

Multiple Subplots

plt.axes: Subplots by Hand

The `plt.axes()` function allows you to create custom axes with manual positioning. You specify the position and size using a list of four values: `[left, bottom, width, height]`, where the coordinates range from 0 to 1 (relative to the figure's dimensions). You can use this method to create inset axes or custom-shaped subplots.

```
import matplotlib.pyplot as plt  # Make sure to import matplotlib.pyplot
import numpy as np

# Create a figure and axes
ax1 = plt.axes()  # standard axes
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])  # inset axes
```

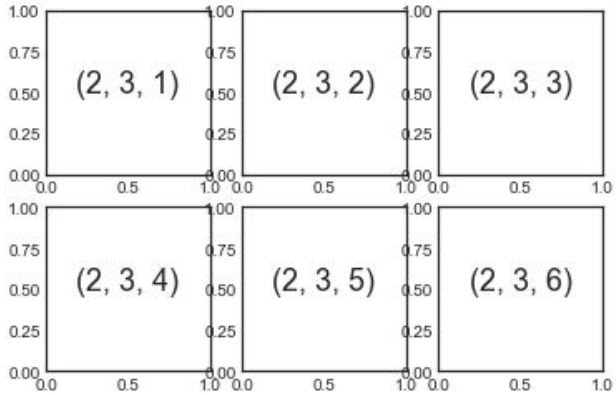


This approach is flexible and lets you control the exact position of each subplot.

plt.subplot: Simple Grids of Subplots

- For creating simple grids of subplots, `plt.subplot` can be used. It takes three arguments: the number of rows, columns, and the index of the subplot (starting from 1 at the top-left).

```
for i in range(1, 7):
    plt.subplot(2, 3, i)
    plt.text(0.5, 0.5, str((2, 3, i)), fontsize=18, ha='center')
```

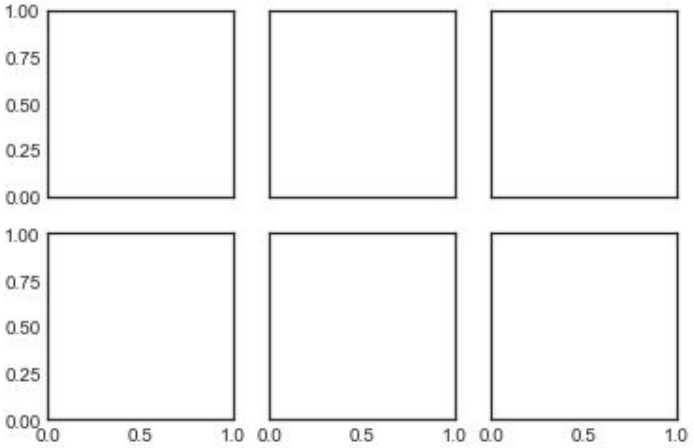


- You can adjust the spacing between subplots using `plt.subplots_adjust` to control `hspace` (vertical spacing) and `wspace` (horizontal spacing).

plt.subplots: The Whole Grid in One Go

- The `plt.subplots` function is ideal for creating a whole grid of subplots in one call. It returns a NumPy array of axes, and you can share axes across rows or columns using the `sharex` and `sharey` parameters. This method simplifies the creation of consistent grid layouts.

```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```



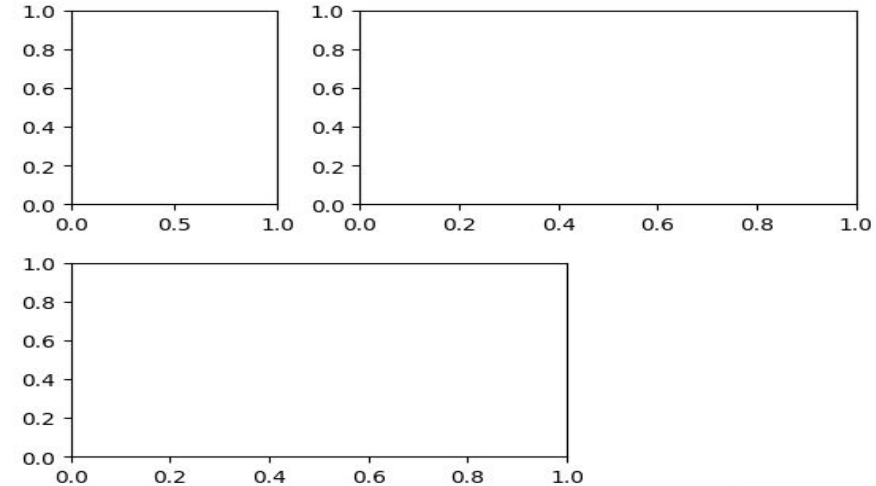
- You can then access individual subplots using array indexing.

plt.GridSpec: More Complicated Arrangements

- **`plt.GridSpec` is used for more complex subplot arrangements where you want to span multiple rows or columns. You can specify the grid layout and then slice it to assign axes to different regions. This provides flexibility in aligning subplots.**

```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
plt.subplot(grid[0, 0])
plt.subplot(grid[0, 1:])
plt.subplot(grid[1, :2])
```

```
<Axes: >
```

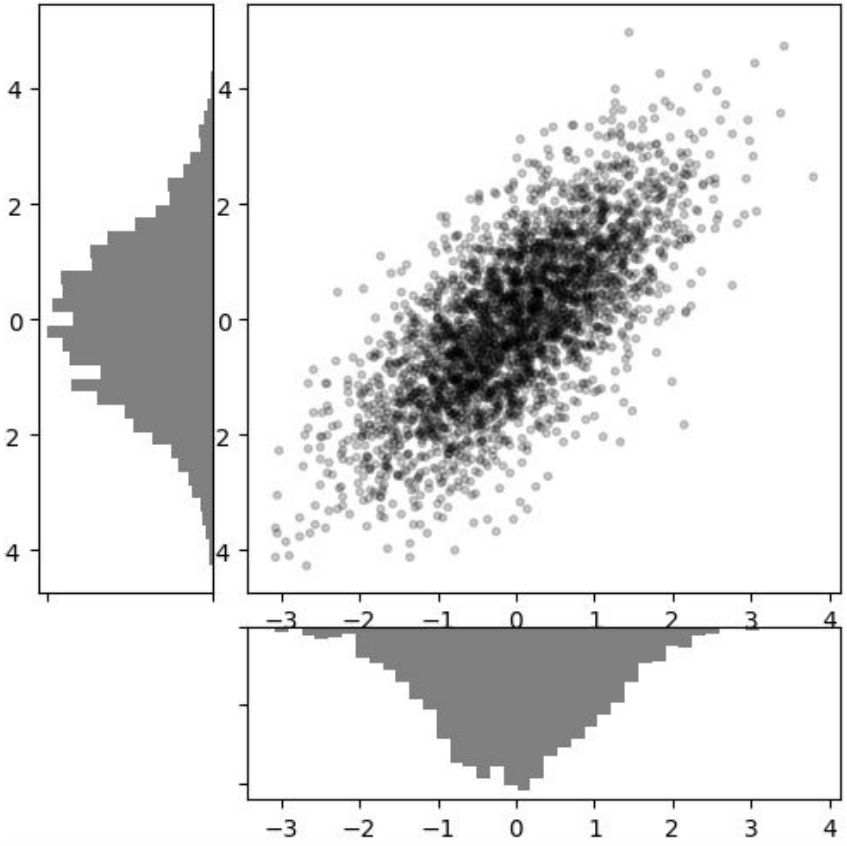


- **You can use `GridSpec` for advanced plots like scatter plots with marginal histograms (common in Seaborn visualizations), where you arrange multiple axes in a non-standard grid.**

Example: Creating a Scatter Plot with Marginal Histograms

In this example, a scatter plot with histograms on the margins is created using `GridSpec`:

```
# Create some normally distributed data
mean = [0, 0]
cov = [[1, 1], [1, 2]]
rng = np.random.default_rng(1701)
x, y = rng.multivariate_normal(mean, cov, 3000).T

# Set up the axes with GridSpec
fig = plt.figure(figsize=(6, 6))
grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)
main_ax = fig.add_subplot(grid[:-1, 1:])
y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=main_ax)
x_hist = fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=main_ax)

# Scatter points on the main axes
main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)

# Histograms on the attached axes
x_hist.hist(x, 40, histtype='stepfilled', orientation='vertical', color='gray')
x_hist.invert_yaxis()
y_hist.hist(y, 40, histtype='stepfilled', orientation='horizontal', color='gray')
y_hist.invert_xaxis()
```



- **This setup creates a central scatter plot with histograms on the top and right margins, showing the distribution of x and y.**

Summary

- **plt.axes: For precise control over subplot placement.**
- **plt.subplot: Simple grids of subplots.**
- **plt.subplots: Convenient for creating a grid of subplots at once, with optional axis sharing.**
- **plt.GridSpec: For more flexible layouts, such as spanning multiple rows or columns.**

Enhancing Visualization with Text

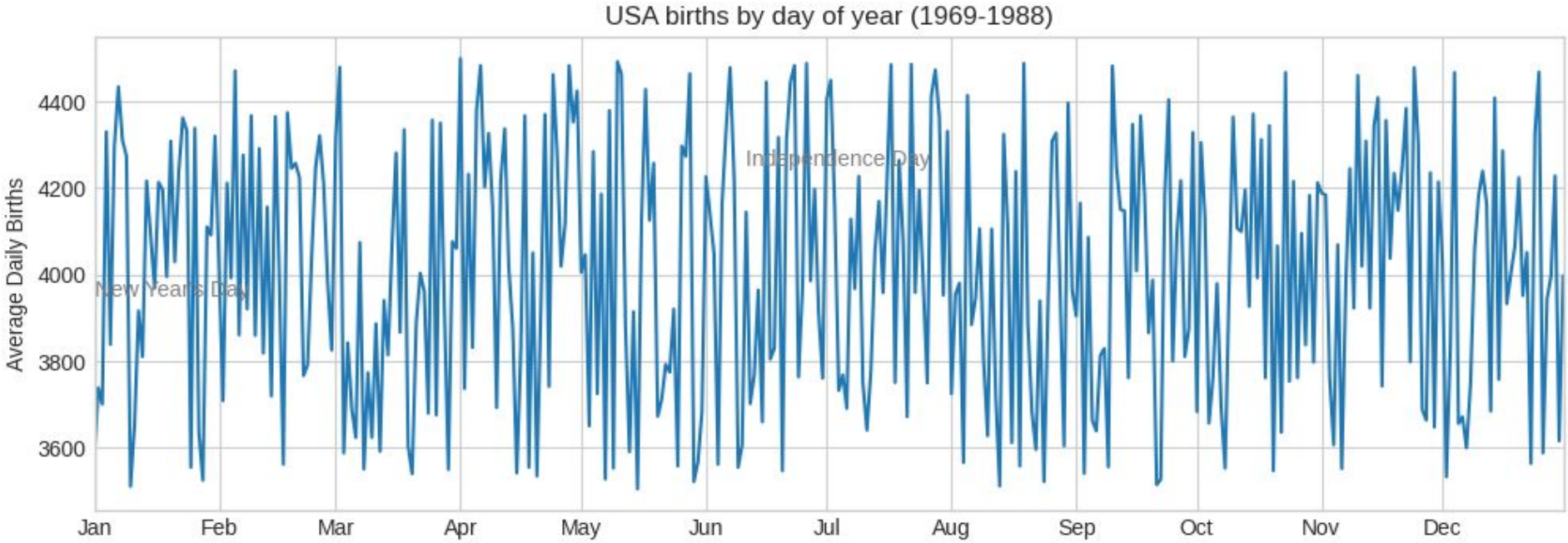
Annotations and labels help make plots more informative and guide the reader through the data. Commonly used annotations include:

- **Axes labels and titles for basic descriptions.**
- **Custom text labels for highlighting specific data points.**

Example: Annotating a Plot of US Births

We use cleaned birthrate data to plot daily average births in the US. Annotations are added to mark notable holidays:

```
# Example data creation (replace this with your actual data)
# Simulated data for daily births
date_range = pd.date_range(start='2012-01-01', end='2012-12-31')
average_daily_births = np.random.randint(3500, 4500, size=len(date_range))
births_by_date = pd.Series(average_daily_births, index=date_range)

# Plotting
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax)

# Adding text annotations
style = dict(size=10, color='gray')
ax.text(pd.Timestamp('2012-01-01'), 3950, "New Year's Day", **style)
ax.text(pd.Timestamp('2012-07-04'), 4250, "Independence Day", ha='center', **style)

# Axes labels
ax.set(title='USA births by day of year (1969-1988)',
       ylabel='Average Daily Births')

# Formatting x-axis for month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_major_formatter(mpl.dates.DateFormatter('%b'))  # Format as short month names
plt.show()
```



- **The `ax.text()` function positions text at a given data coordinate.**
- **Parameters like `ha` (horizontal alignment) and `style` control the text's appearance.**

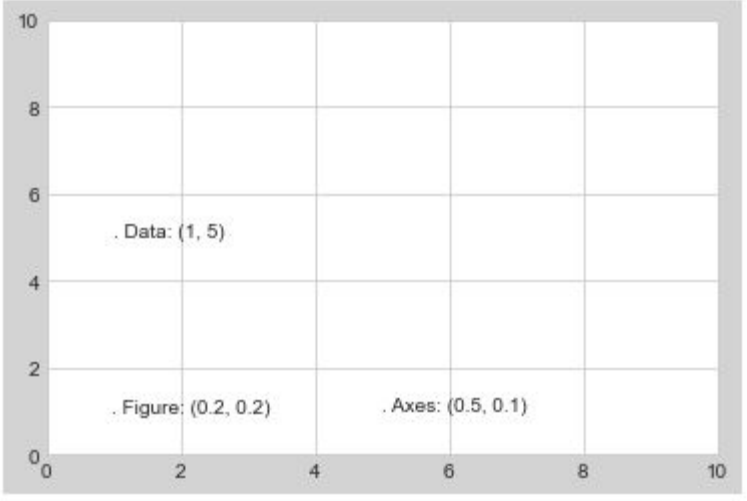
Coordinate Transforms for Flexible Placement

Matplotlib uses multiple coordinate systems:

- **Data coordinates (`transData`): Position text relative to data values.**
- **Axes coordinates (`transAxes`): Position text relative to the plot axes (e.g., fractions like `(0.5, 0.1)`).**
- **Figure coordinates (`transFigure`): Position text relative to the figure dimensions.**

```
fig, ax = plt.subplots()
ax.axis([0, 10, 0, 10])

ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure)
```

- **Adjusting the axes limits (`ax.set_xlim(0, 2)`) demonstrates how only `transData` coordinates change with the axis.**

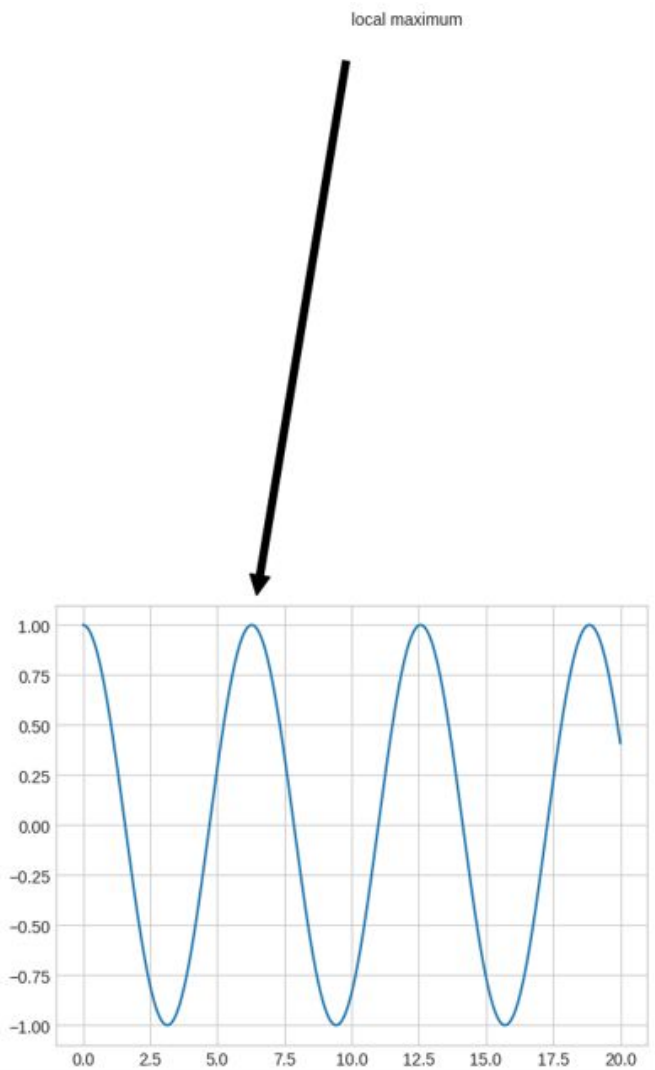


Adding Arrows and Annotations

For arrows and labels:

- **Use `plt.annotate()` instead of `plt.arrow()` for better flexibility and style control.**
- **Parameters include:**
  - **`xy`: Data point where the annotation points.**
  - **`xytext`: Position of the annotation text.**
  - **`arrowprops`: Style of the arrow.**

```
fig, ax = plt.subplots()
x = np.linspace(0, 20, 1000)
ax.plot(x, np.cos(x))

ax.annotate('local maximum', xy=(6.28, 1), xytext=(10, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))
```

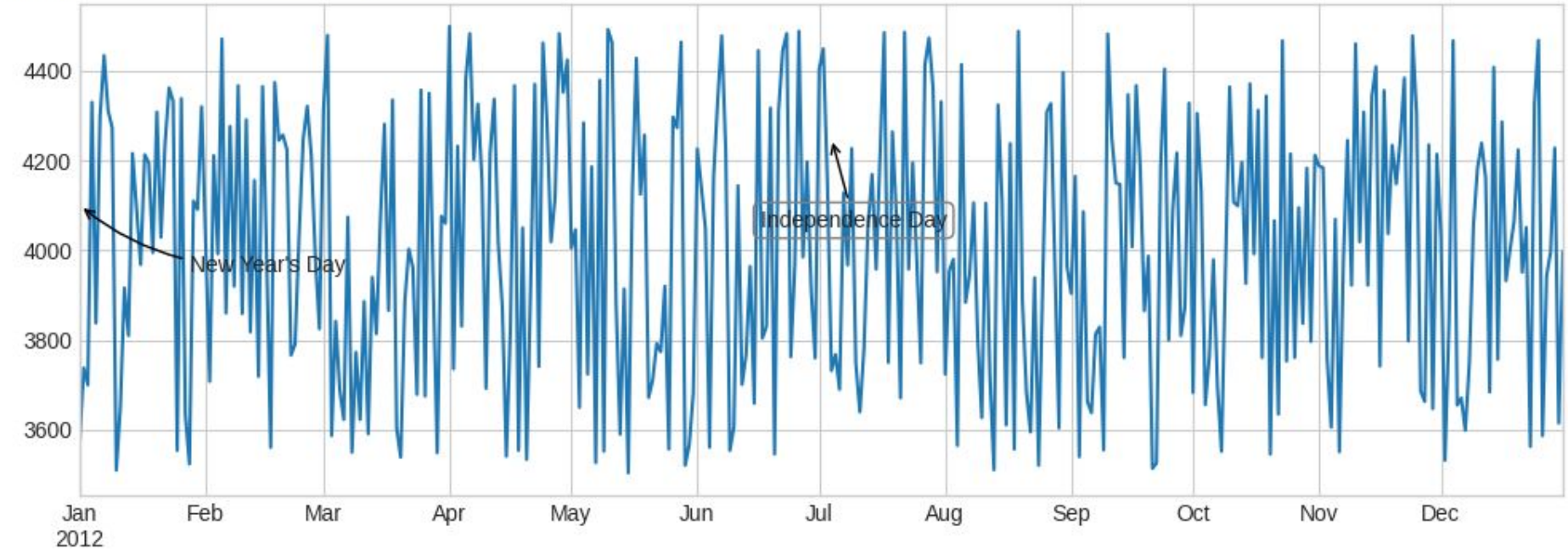


Complex Annotation Styles

Annotations can use bounding boxes, fancy arrow styles, and custom alignment.

```
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax)

# Annotating holidays
ax.annotate("New Year's Day", xy=('2012-1-1', 4100), xycoords='data',
            xytext=(50, -30), textcoords='offset points',
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=-0.2"))

ax.annotate("Independence Day", xy=('2012-7-4', 4250), xycoords='data',
            bbox=dict(boxstyle="round", fc="none", ec="gray"),
            xytext=(10, -40), textcoords='offset points', ha='center',
            arrowprops=dict(arrowstyle="->"))
```

```
Text(10, -40, 'Independence Day')
```



Key Takeaways

- Text and arrows help highlight important features in a plot.
- Choose the coordinate system (`transData`, `transAxes`, or `transFigure`) based on whether the text should move with the data, axes, or remain fixed.
- Use `plt.annotate()` for flexible, styled arrows and text.

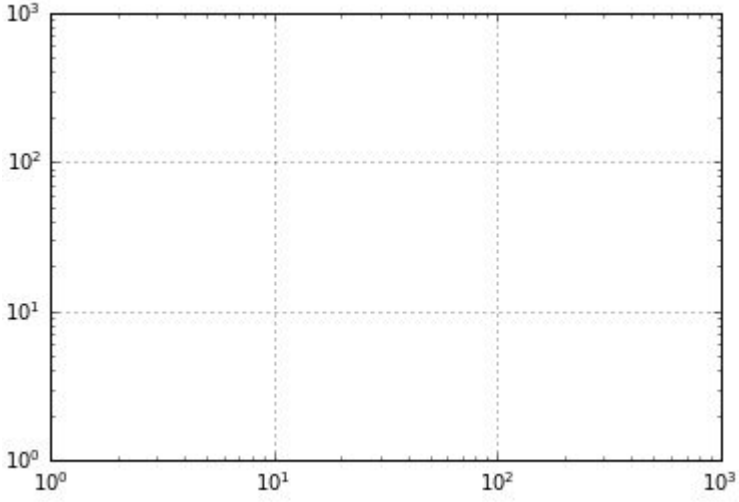
Customizing Ticks

Tick customization allows greater control over the appearance, placement, and formatting of ticks in plots. By fine-tuning ticks, you can enhance the readability and aesthetics of your visualizations. Key components include:

1. **Tick Locations: Where the ticks appear.**
2. **Tick Labels: The text associated with the ticks.**

Example: Logarithmic Scale

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

plt.style.use('classic')
ax = plt.axes(xscale='log', yscale='log')
ax.set(xlim=(1, 1E3), ylim=(1, 1E3))
ax.grid(True)
```



- **Major ticks display prominent gridlines and labels.**
- **Minor ticks only show small tick marks without labels.**

Accessing Tick Locators and Formatters:

```
print(ax.xaxis.get_major_locator())  # <matplotlib.ticker.LogLocator>
print(ax.xaxis.get_minor_locator())  # <matplotlib.ticker.LogLocator>
print(ax.xaxis.get_major_formatter())  # <matplotlib.ticker.LogFormatterSciNotation>
print(ax.xaxis.get_minor_formatter())  # <matplotlib.ticker.NullFormatter>
```

```
<matplotlib.ticker.LogLocator object at 0x7efd6eba1d50>
<matplotlib.ticker.LogLocator object at 0x7efd6eba3b80>
<matplotlib.ticker.LogFormatterSciNotation object at 0x7efd6eba39d0>
<matplotlib.ticker.LogFormatterSciNotation object at 0x7efd6eba16c0>
```

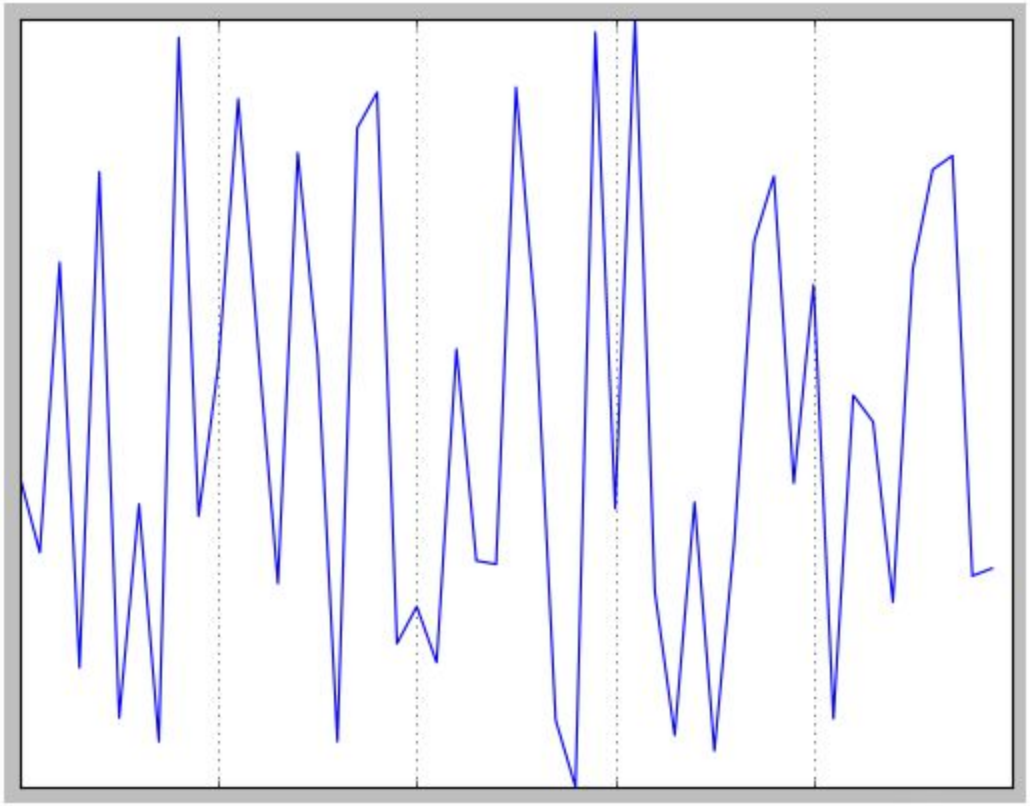
- **Here, major tick labels use scientific notation, while minor ticks are unlabeled (`NullFormatter`).**

Hiding Ticks or Labels

To hide ticks or labels, use `NullLocator` and `NullFormatter`.

Example: Hiding Specific Ticks

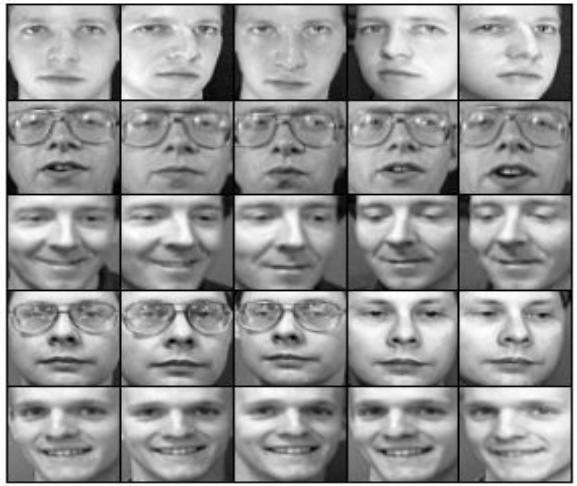
```
ax = plt.axes()
rng = np.random.default_rng(1701)
ax.plot(rng.random(50))
ax.grid()

ax.yaxis.set_major_locator(plt.NullLocator())  # Hide y-axis ticks
ax.xaxis.set_major_formatter(plt.NullFormatter())  # Hide x-axis labels
```



Example: Image Grid

For images, tick labels often add unnecessary clutter:

```
from sklearn.datasets import fetch_olivetti_faces
faces = fetch_olivetti_faces().images

for i in range(5):
    for j in range(5):
        ax[i, j].xaxis.set_major_locator(plt.NullLocator())
        ax[i, j].yaxis.set_major_locator(plt.NullLocator())
        ax[i, j].imshow(faces[10 * i + j], cmap='binary_r')
```



All tick labels are hidden for better clarity.

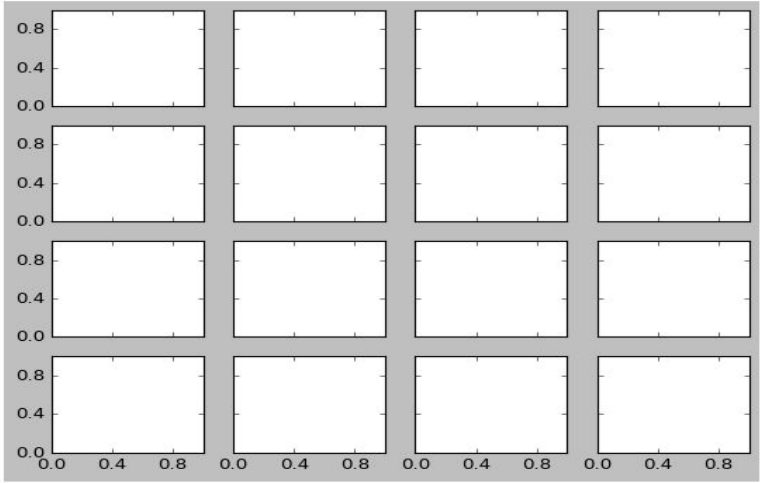
Reducing or Increasing the Number of Ticks

Crowded tick labels can be mitigated using `MaxNLocator`.

Example: Adjusting Tick Density

```
fig, ax = plt.subplots(4, 4, sharex=True, sharey=True)

for axi in ax.flat:
    axi.xaxis.set_major_locator(plt.MaxNLocator(3))
    axi.yaxis.set_major_locator(plt.MaxNLocator(3))
```

- **This reduces the number of ticks to three per axis, improving readability.**

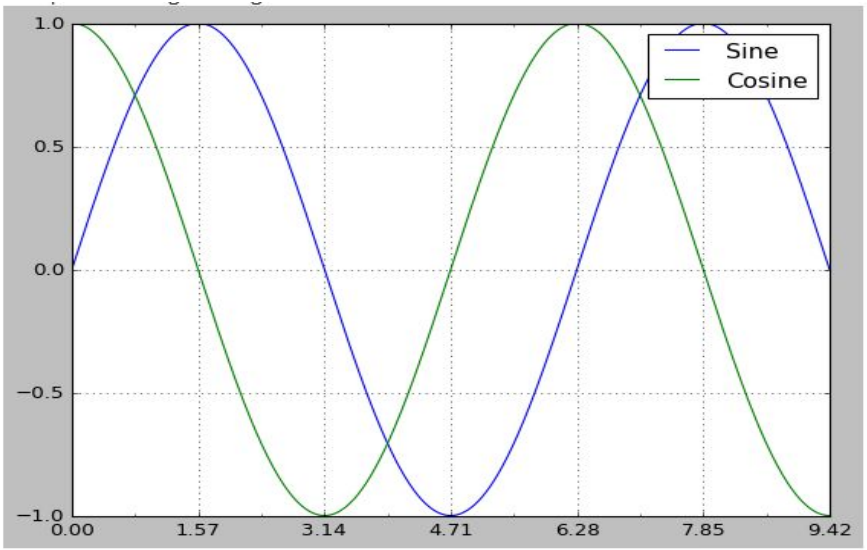


Custom Tick Formats

For more control over tick formatting, use locators like `MultipleLocator` and custom formatters like `FuncFormatter`.

Example: Customizing Tick Locations

```
fig, ax = plt.subplots()
x = np.linspace(0, 3 * np.pi, 1000)
ax.plot(x, np.sin(x), label='Sine')
ax.plot(x, np.cos(x), label='Cosine')

# Major ticks every π/2, minor ticks every π/4
ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
ax.xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 4))
ax.grid(True)
ax.legend()
```



Example: Custom Tick Labels

```
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter
import numpy as np

# Generate data
x = np.linspace(0, 2 * np.pi, 100)
y = np.sin(x)

# Create a plot
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(x, y, label='sin(x)')

# Define custom tick formatter function
def format_func(value, tick_number):
    N = int(np.round(2 * value / np.pi))
    if N == 0:
        return "0"
    elif N == 1:
        return r"$\pi/2$"
    elif N == 2:
        return r"$\pi$"
```

```
 elif N % 2 > 0:
        return rf"${N}\pi/2$"
    else:
        return rf"${N // 2}\pi$"

# Apply custom tick formatter to x-axis
ax.xaxis.set_major_formatter(FuncFormatter(format_func))

# Add labels, legend, and title
ax.set_xlabel('x-axis (radians)')
ax.set_ylabel('y-axis')
ax.legend()
ax.set_title('Sine Wave with Custom Tick Labels')

# Show the plot
plt.show()
```

- **Ticks are labeled in terms of multiples of π\piπ, enhancing interpretability.**

Summary of Locators and Formatters

Locators

Class	Description
`NullLocator`	No ticks
`FixedLocator`	Fixed tick locations
`IndexLocator`	Locator for index plots
`LinearLocator`	Evenly spaced ticks
`LogLocator`	Logarithmic ticks
`MultipleLocator`	Ticks at multiples of a base
`MaxNLocator`	Limits the number of ticks
`AutoLocator`	Default locator
`AutoMinorLocator`	Locator for minor ticks

Formatters

Class	Description
`NullFormatter`	No labels
`IndexFormatter`	Labels from a list
`FixedFormatter`	Manually set labels
`FuncFormatter`	User-defined function for labels
`FormatStrFormatter`	Format string for labels
`ScalarFormatter`	Default scalar formatter
`LogFormatter`	Formatter for logarithmic scales

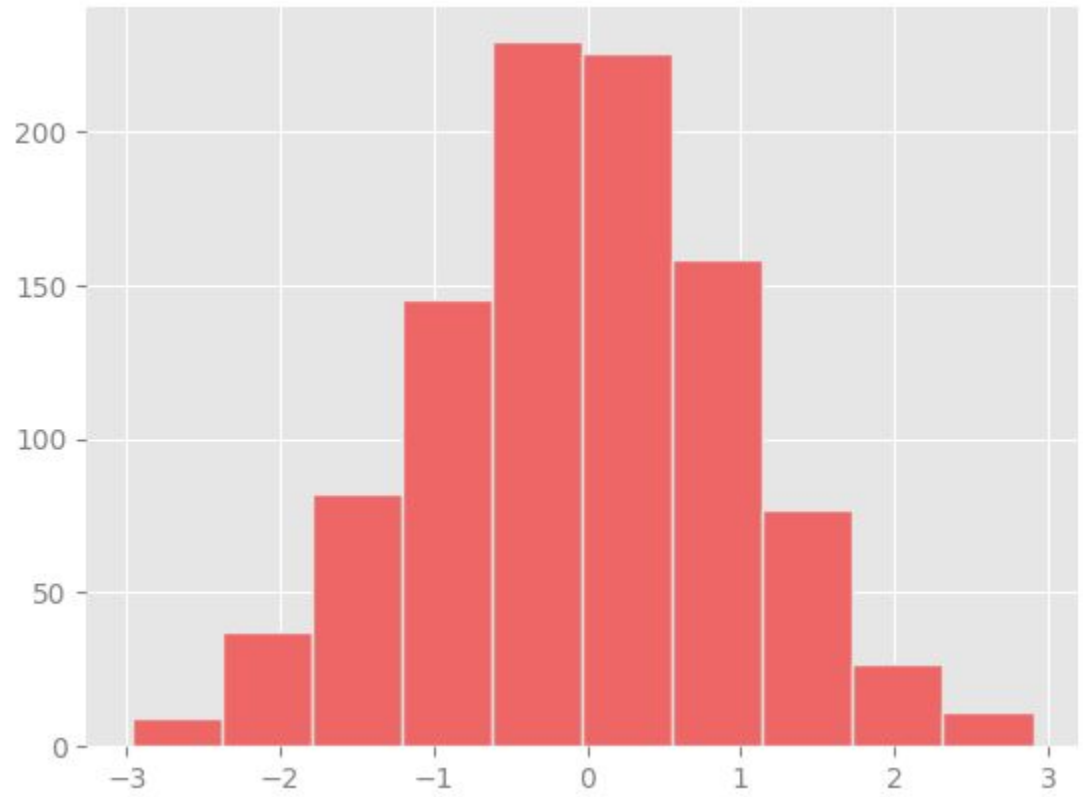
For more details, refer to the Matplotlib documentation.

Customizing Matplotlib: Configurations and Stylesheets

1. Plot Customization by Hand

- **You can tweak individual plot elements for improved visual appeal.**
- **Example: Adjusting the background, gridlines, and axis spines to create a more polished histogram.**

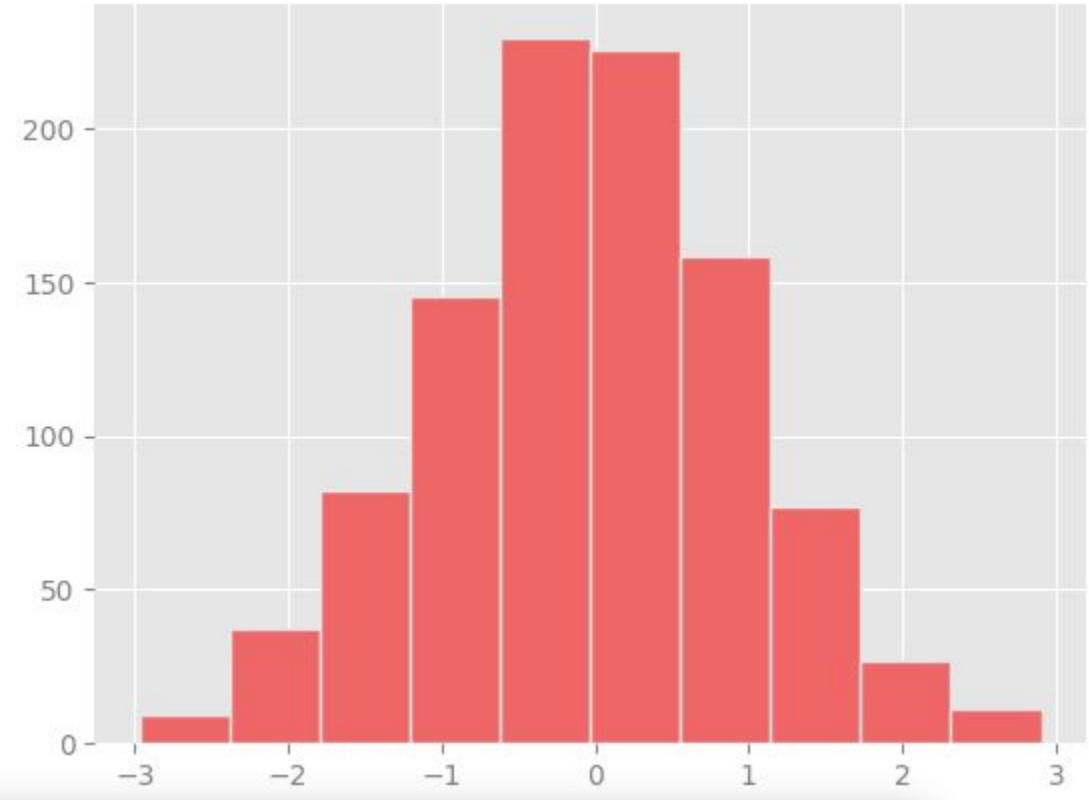
```
import matplotlib.pyplot as plt
import numpy as np

# Data
x = np.random.randn(1000)

# Customized histogram
fig = plt.figure(facecolor='white')
ax = plt.axes(facecolor='#E6E6E6')
ax.set_axisbelow(True)
plt.grid(color='w', linestyle='solid')

for spine in ax.spines.values():
    spine.set_visible(False)
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()
ax.tick_params(colors='gray', direction='out')
ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');
```


200
150
100
50
0
−3
−2
−1
0
1
2
3

2. Changing Defaults with `rcParams`

- Adjust runtime configuration settings for consistent styling across plots.
- Use `plt.rc()` to customize settings like colors, gridlines, tick marks, etc.

```
from matplotlib import cycler

# Custom colors
colors = cycler('color', ['#EE6666', '#3388BB', '#9988DD', '#EECC55', '#88BB44', '#FFBBBB'])

plt.rc('figure', facecolor='white')
plt.rc('axes', facecolor='#E6E6E6', edgecolor='none', axisbelow=True, grid=True, prop_cycle=colors)
plt.rc('grid', color='w', linestyle='solid')
plt.rc('xtick', direction='out', color='gray')
plt.rc('ytick', direction='out', color='gray')
plt.rc('patch', edgecolor='#E6E6E6')
plt.rc('lines', linewidth=2)

# Apply rcParams
plt.hist(x);
```


200
150
100
50
0
−3
−2
−1
0
1
2
3

Stylesheets

- Matplotlib offers built-in stylesheets to easily change the overall appearance of plots.
- Use `plt.style.use('stylename')` to apply a style globally or within a context manager.

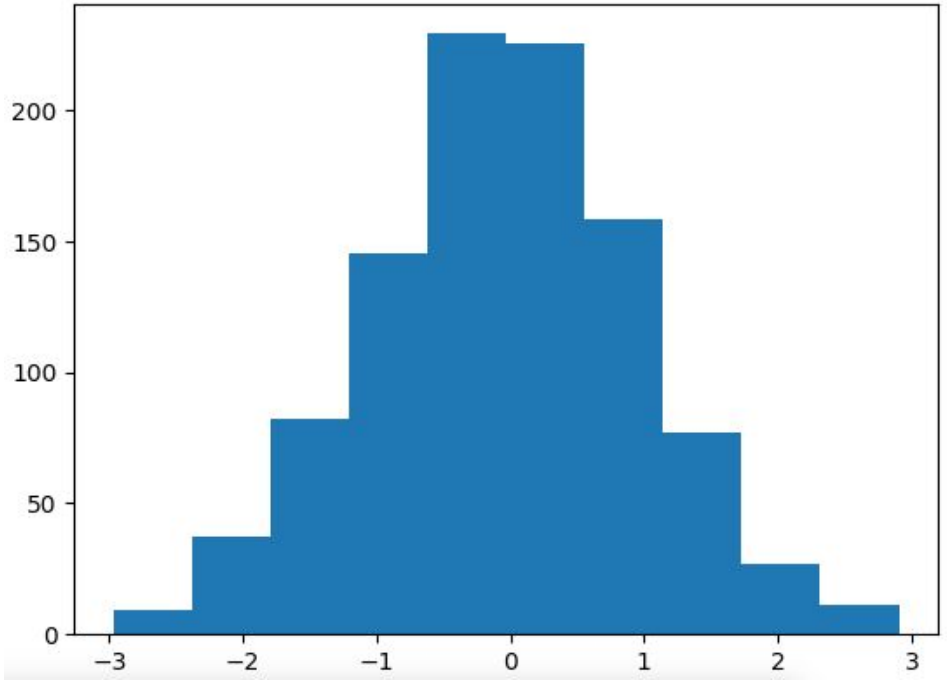
Exploring Built-in Styles

- Check available styles: `plt.style.available`.

Example Styles:

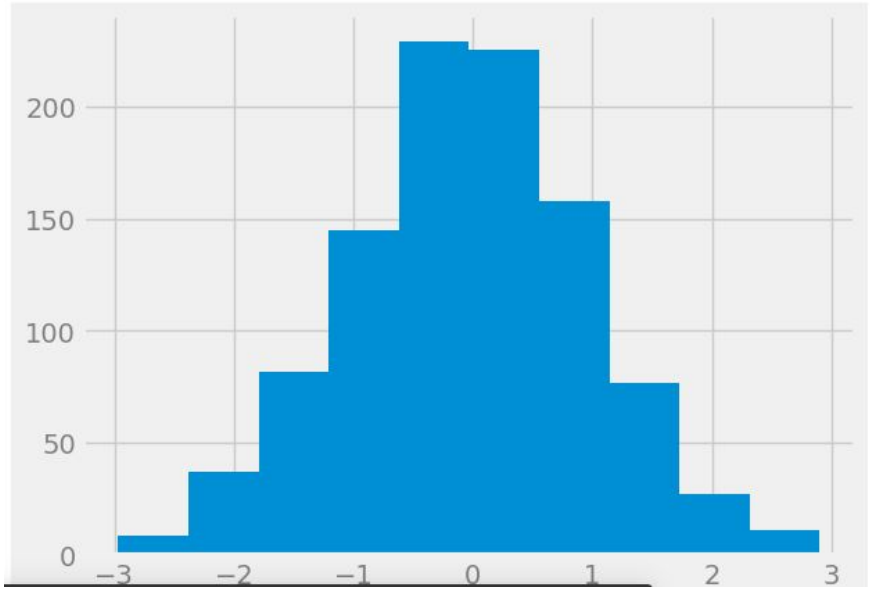
1. Default Style

```
with plt.style.context('default'):
    plt.hist(x);
```



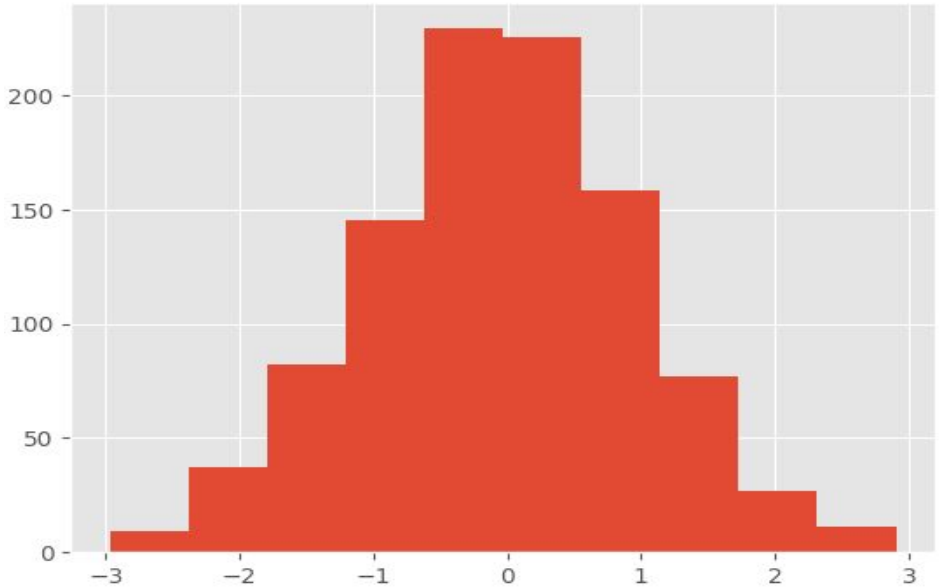
FiveThirtyEight Style

```
with plt.style.context('fivethirtyeight'):
    plt.hist(x);
```



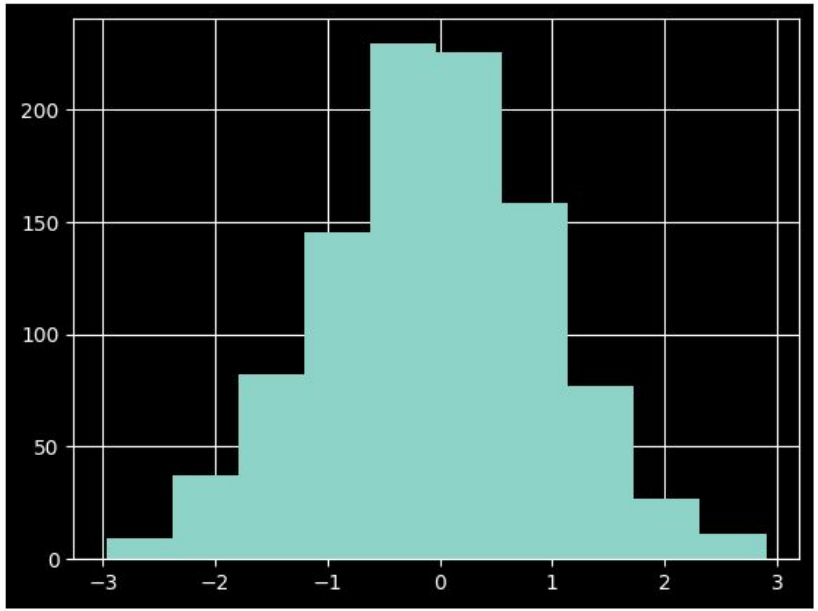
ggplot Style

```
with plt.style.context('ggplot'):
    plt.hist(x);
```



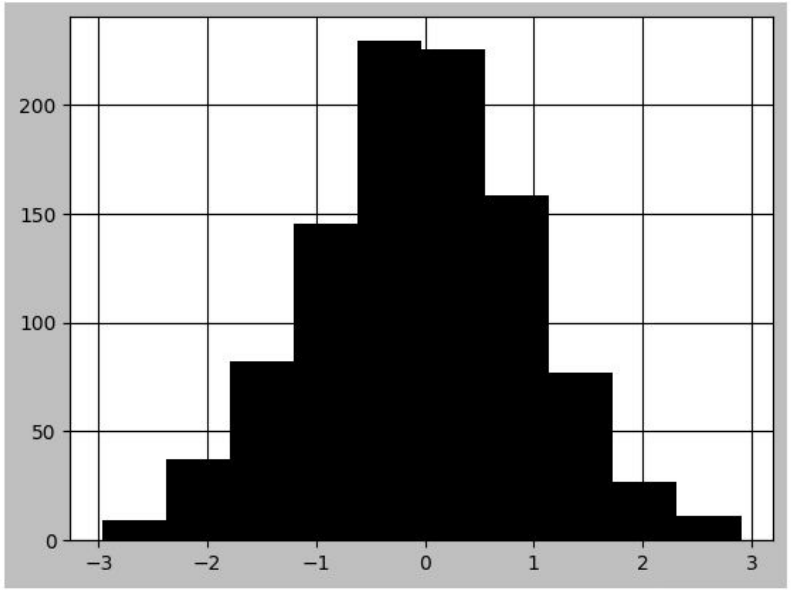
Dark Background Style

```
with plt.style.context('dark_background'):
    plt.hist(x);
```



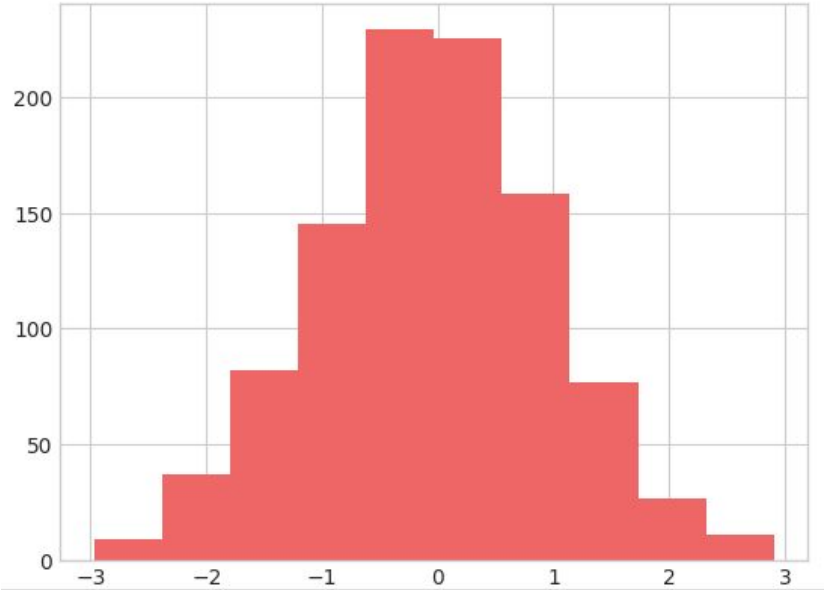
Grayscale Style

```
with plt.style.context('grayscale'):
    plt.hist(x);
```



Seaborn Style

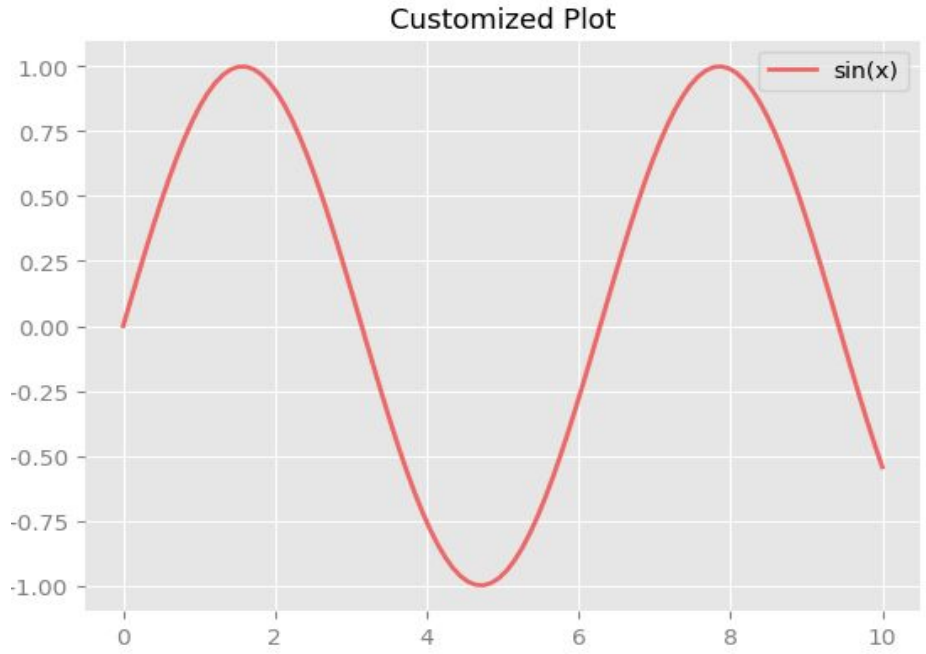
```
with plt.style.context('seaborn-v0_8-whitegrid'):
    plt.hist(x);
```



Create Custom Styles

- Save your configurations in `.mplstyle` files for reusable custom styles.
- Example content of a custom `.mplstyle` file:

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

# Apply style settings
plt.rcParams.update({
    'axes.facecolor': '#E6E6E6',
    'axes.edgecolor': 'none',
    'grid.color': 'white',
    'lines.linewidth': 2,
    'xtick.color': 'gray',
    'ytick.color': 'gray'
})

# Example plot
x = np.linspace(0, 10, 100)
y = np.sin(x)

fig, ax = plt.subplots()
ax.plot(x, y, label='sin(x)')
ax.set_title('Customized Plot')
ax.legend()
plt.show()
```



Summary

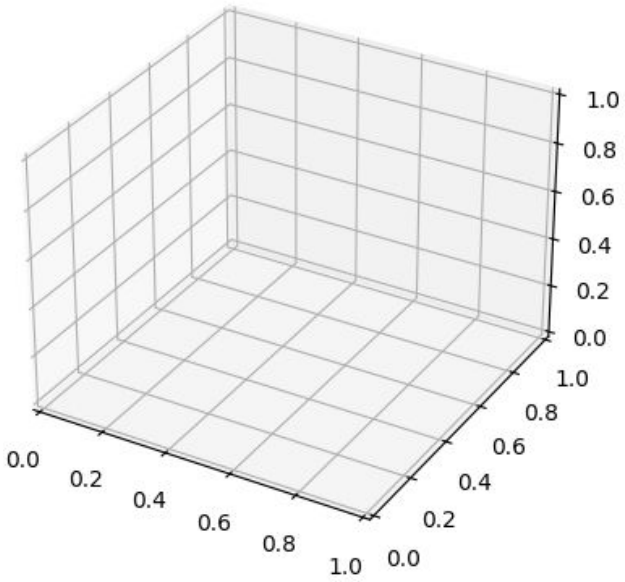
- **rcParams: Fine-grained control over plot appearance via runtime configurations.**
- **Stylesheets: Predefined themes for easy application or creation of custom styles.**
- **Experiment with styles to find what works best for your presentation or analysis needs!**

Three-Dimensional Plotting in Matplotlib

1. Enabling 3D Axes

- **Import the mplot3d toolkit to enable 3D plotting:**

```
from mpl_toolkits import mplot3
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = plt.axes(projection='3d')
```

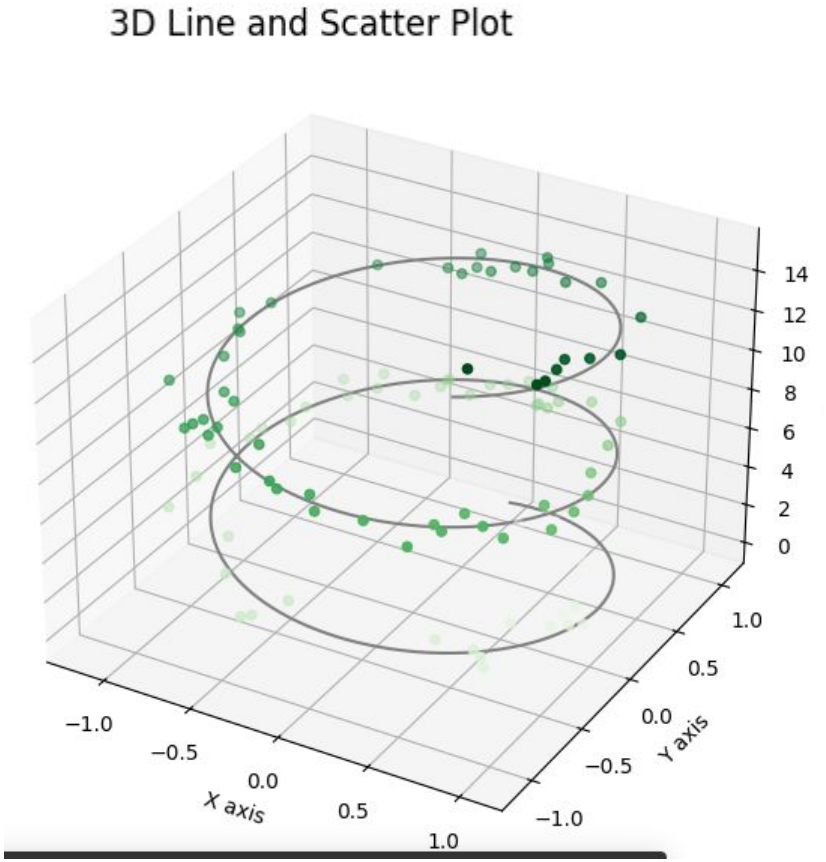


Three-Dimensional Points and Lines

- **Line Plot: Use `ax.plot3D` for 3D line plotting.**
- **Scatter Plot: Use `ax.scatter3D` to visualize points.**

Example

```
# Create a 3D plot
fig = plt.figure(figsize=(10, 7))
ax = plt.axes(projection='3d')  # Define 3D axes

# Line data
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')  # Plot the line

# Scatter data
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens')  # Scatter points

# Add labels for clarity
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('3D Line and Scatter Plot')

# Show the plot
plt.show()
```

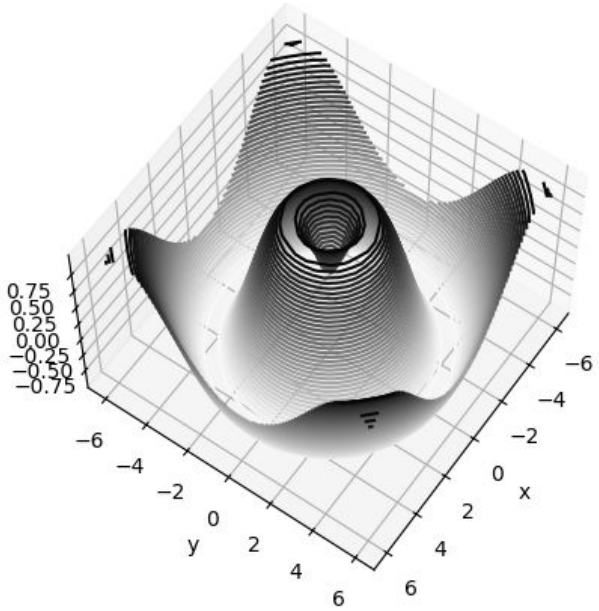

3D Line and Scatter Plot
14
12
10
8
6
4
2
0
1.0
0.5
0.0
−0.5
−1.0
Y axis
−1.0
−0.5
0.0
0.5
1.0
X axis

Three-Dimensional Contour Plots

- **Use `ax.contour3D` for 3D contours.**

  Example:

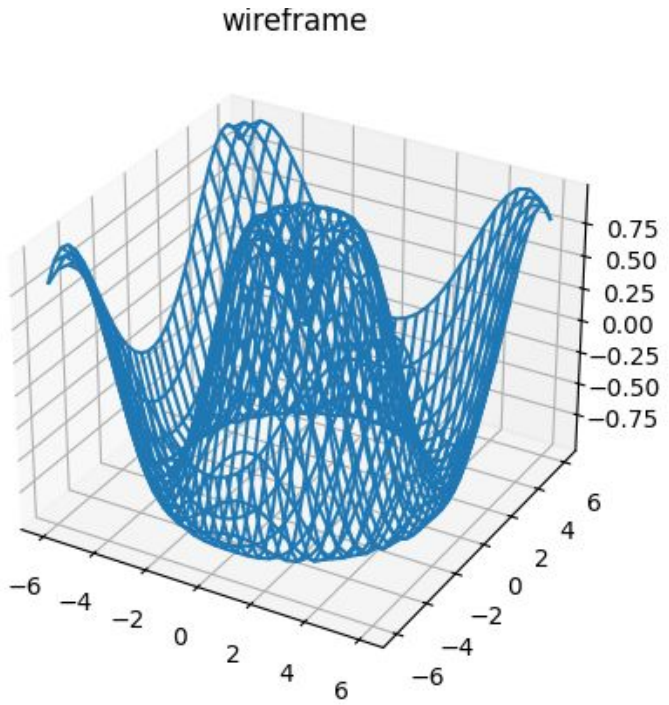
```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))

x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 40, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.view_init(60, 35)  # Adjust view
```



Wireframes and Surface Plots

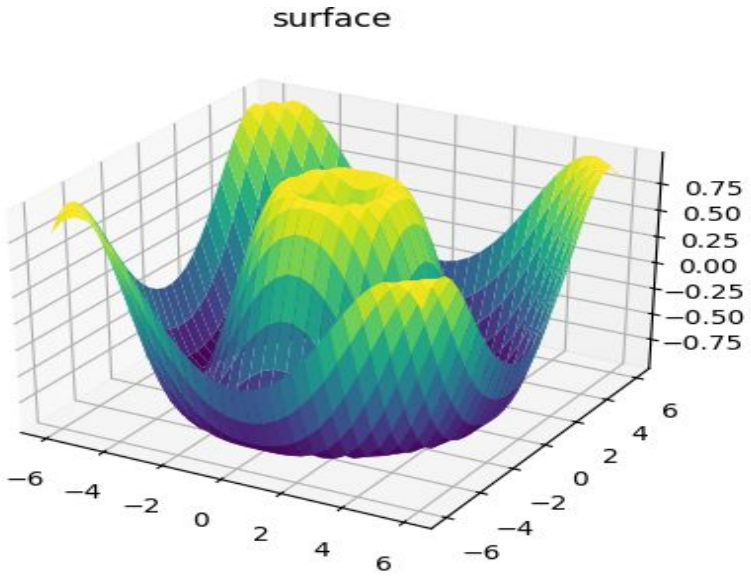
- **Wireframe Plot: `ax.plot_wireframe`**

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z)
ax.set_title('wireframe');
```



Surface Plot: `ax.plot_surface`

```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_title('surface');
```

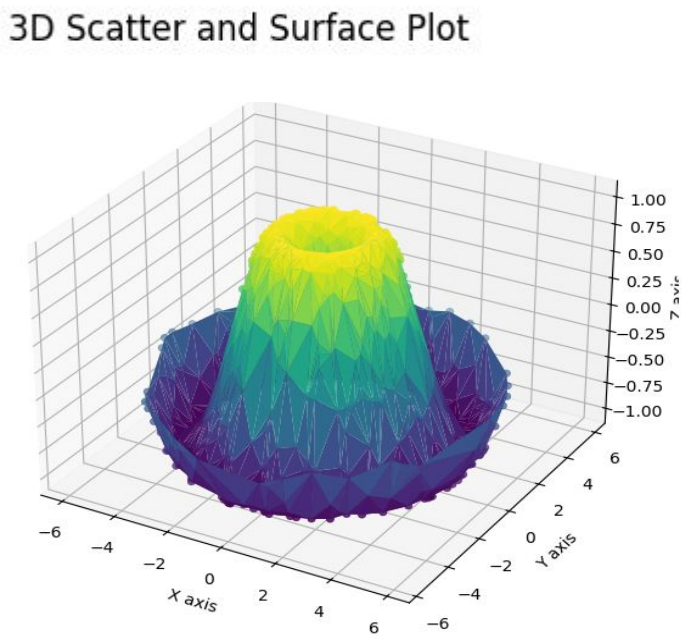


Surface Triangulations

- **For irregular data, use `ax.plot_trisurf`.**

```
def f(x, y):
    return np.sin(np.sqrt(x**2 + y**2))

# Generate data
theta = 2 * np.pi * np.random.random(1000)  # Angles
r = 6 * np.random.random(1000)  # Radius
x = r * np.sin(theta)
y = r * np.cos(theta)
z = f(x, y)  # Compute z values

# Create 3D plot
fig = plt.figure(figsize=(10, 7))
ax = plt.axes(projection='3d')  # Define 3D axes

# Scatter plot
scatter = ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5)

# Surface plot
surface = ax.plot_trisurf(x, y, z, cmap='viridis', edgecolor='none')
```

```python
# Labels and title
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.set_title('3D Scatter and Surface Plot')

# Show the plot
plt.show()
```

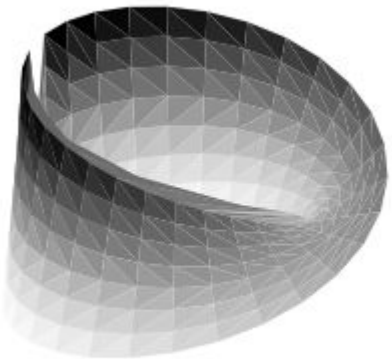


Example: Möbius Strip

- **Parametrize and plot a Möbius strip.**

```
from matplotlib.tri import Triangulation

# Generate the data
theta = np.linspace(0, 2 * np.pi, 30)
w = np.linspace(-0.25, 0.25, 8)
w, theta = np.meshgrid(w, theta)

phi = 0.5 * theta
r = 1 + w * np.cos(phi)

x = r * np.cos(theta)
y = r * np.sin(theta)
z = w * np.sin(phi)

# Flatten x, y, z for triangulation
x_flat = x.ravel()
y_flat = y.ravel()
z_flat = z.ravel()

# Create a triangulation object based on flattened arrays
tri = Triangulation(x_flat, y_flat)

# Create the 3D plot
fig = plt.figure(figsize=(10, 7))
ax = plt.axes(projection='3d')
```

```
# Plot the surface
ax.plot_trisurf(x_flat, y_flat, z_flat, triangles=tri.triangles, cmap='Greys', linewidths=0.2)

# Set limits and turn off axes
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-0.5, 0.5)
ax.axis('off')

# Show the plot
plt.show()
```



Summary

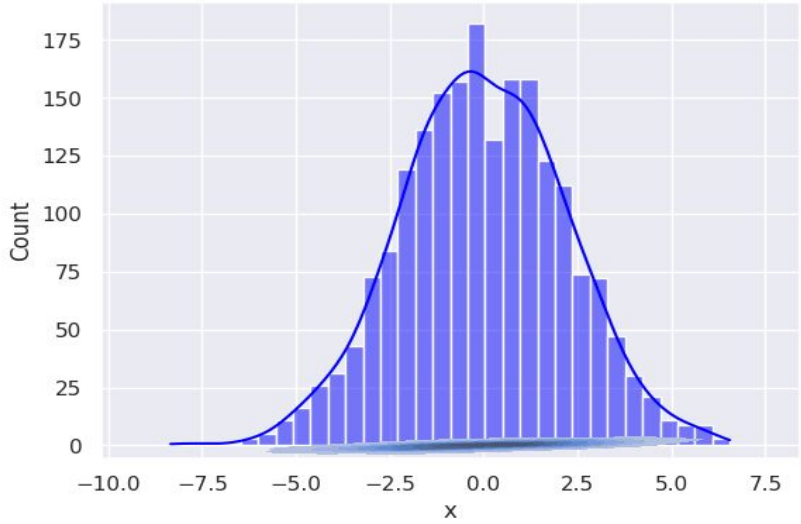
- Use `projection='3d'` to enable 3D plotting.
- Choose appropriate plot types (line, scatter, contour, surface, etc.).
- Explore interactive features for enhanced visualization (e.g., `ax.view_init`).

Visualization with Seaborn

Histograms and Density Plots

- **Histogram: Visualize the frequency distribution.**
- **Kernel Density Estimation (KDE): A smooth estimate of the distribution.**
- **Joint Density: Shows the relationship between two variables.**

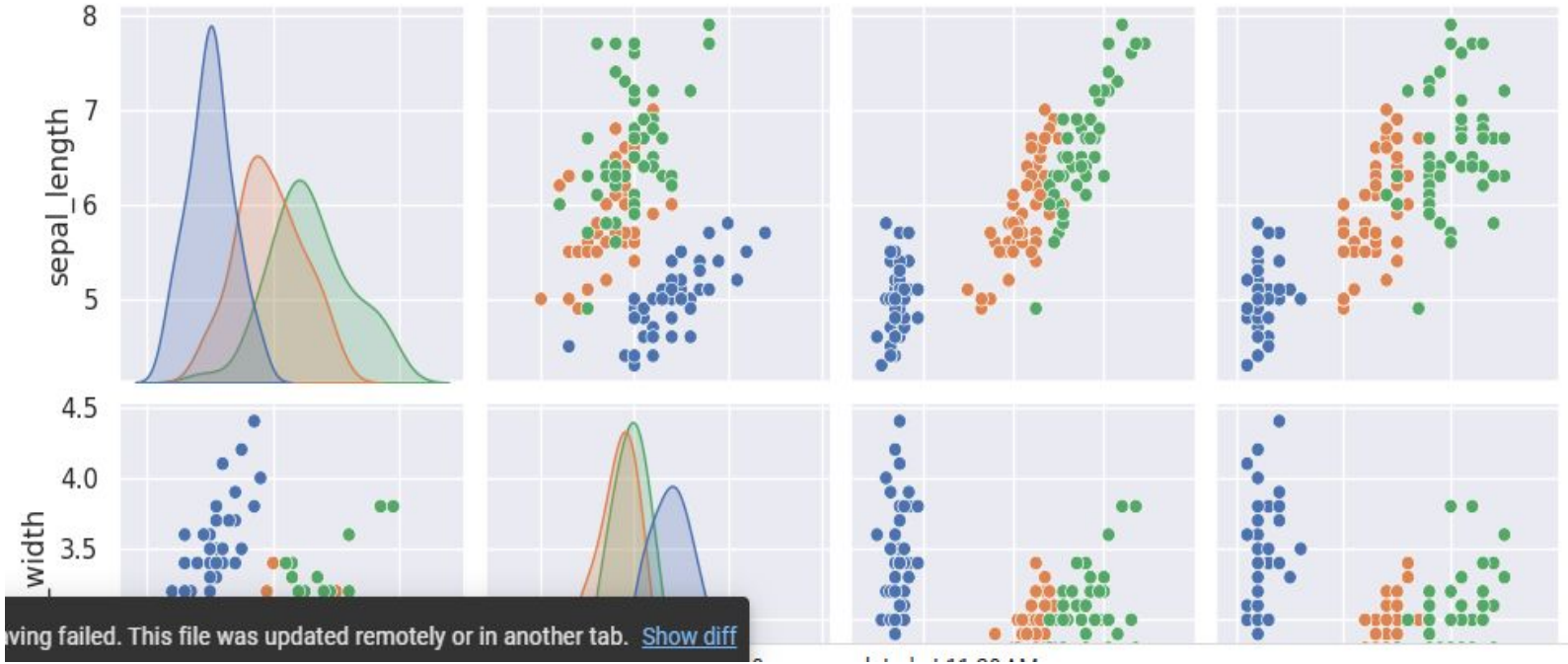
```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

sns.set()  # Set Seaborn's default style
# Example Data
data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
data = pd.DataFrame(data, columns=['x', 'y'])

# Histogram
sns.histplot(data['x'], kde=True, color="blue", alpha=0.5)

# KDE Plot
sns.kdeplot(data=data, x="x", y="y", shade=True)
```



Pair Plots

Generalize joint plots to visualize relationships across all variable pairs.

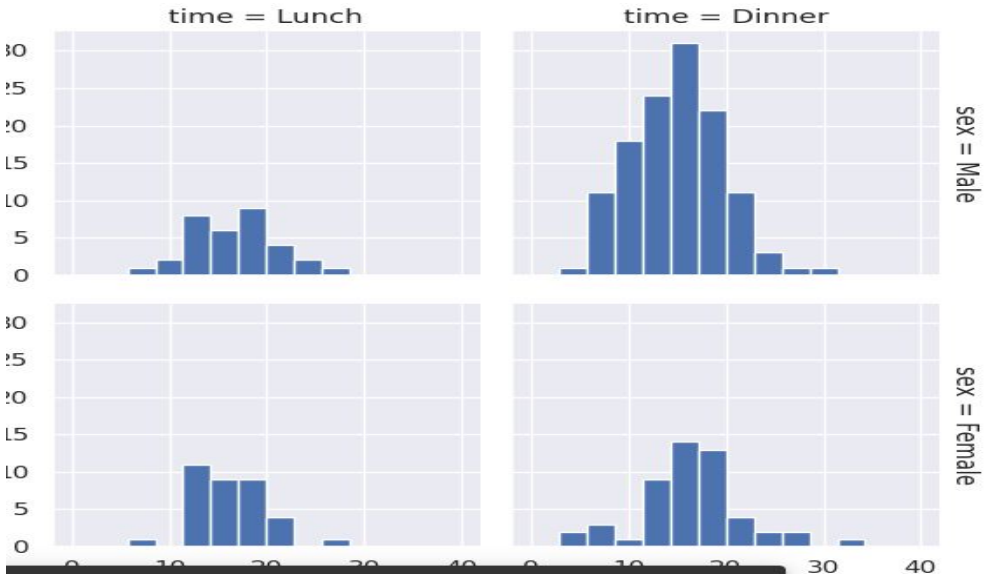
```
iris = sns.load_dataset("iris")
sns.pairplot(iris, hue='species', height=2.5)
```



Faceted Histograms

Split data into subsets and visualize them in a grid.

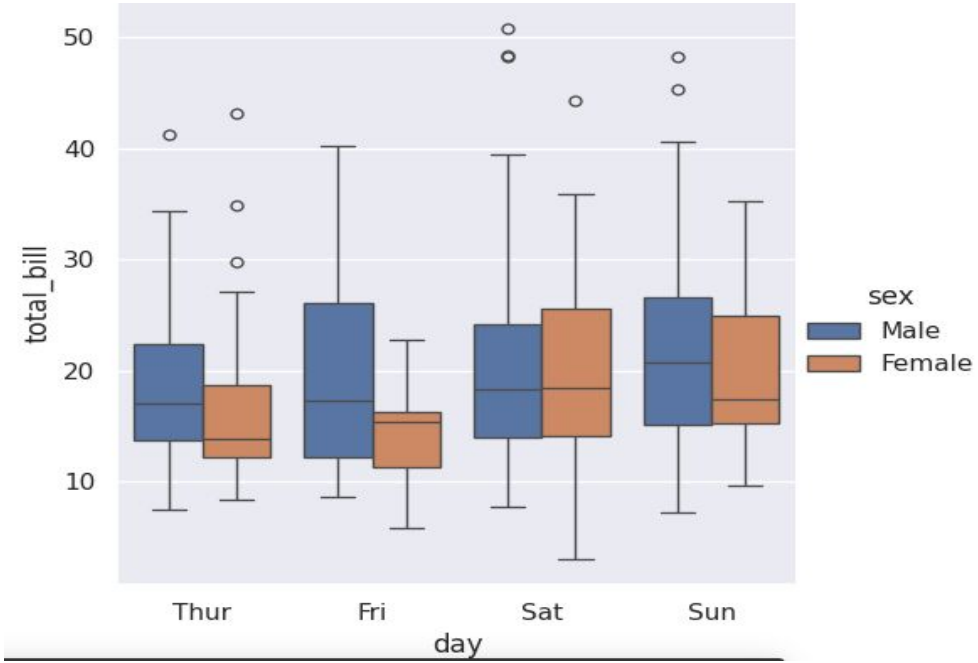
```
tips = sns.load_dataset('tips')
tips['tip_pct'] = 100 * tips['tip'] / tips['total_bill']

# Facet Grid by Gender and Time
grid = sns.FacetGrid(tips, row="sex", col="time", margin_titles=True)
grid.map(plt.hist, "tip_pct", bins=np.linspace(0, 40, 15))
```



Categorical Plots

Visualize data grouped by categories.

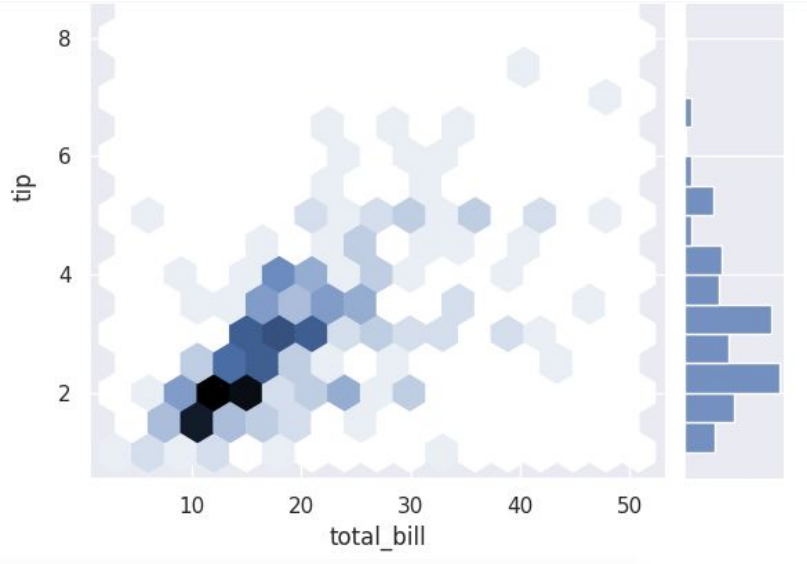
```
sns.catplot(x="day", y="total_bill", hue="sex", data=tips, kind="box")
```

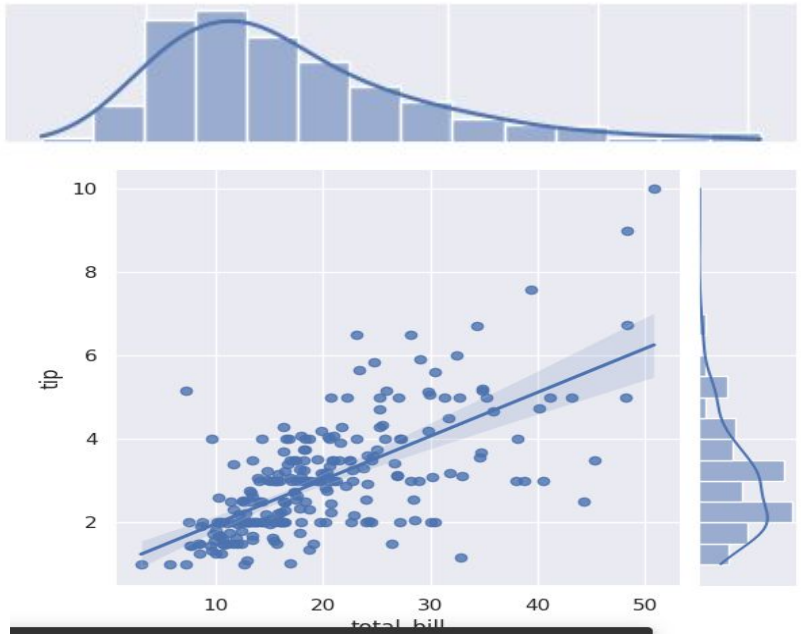


Joint Distributions

Show joint and marginal distributions in one figure.

```
sns.jointplot(x="total_bill", y="tip", data=tips, kind='hex')
sns.jointplot(x="total_bill", y="tip", data=tips, kind='reg')  # Regression
```

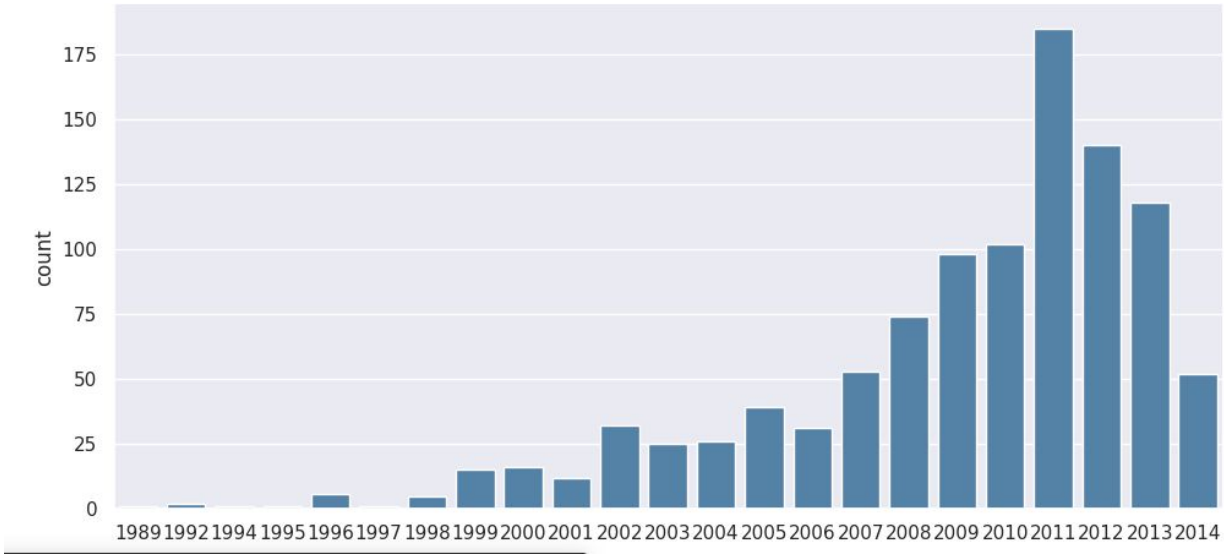




Bar Plots

Visualize time-series counts or distributions.

```
planets = sns.load_dataset('planets')

# Number of discoveries per year
sns.catplot(x="year", data=planets, kind="count", color='steelblue', aspect=2)
```



Example: Marathon Finishing Times

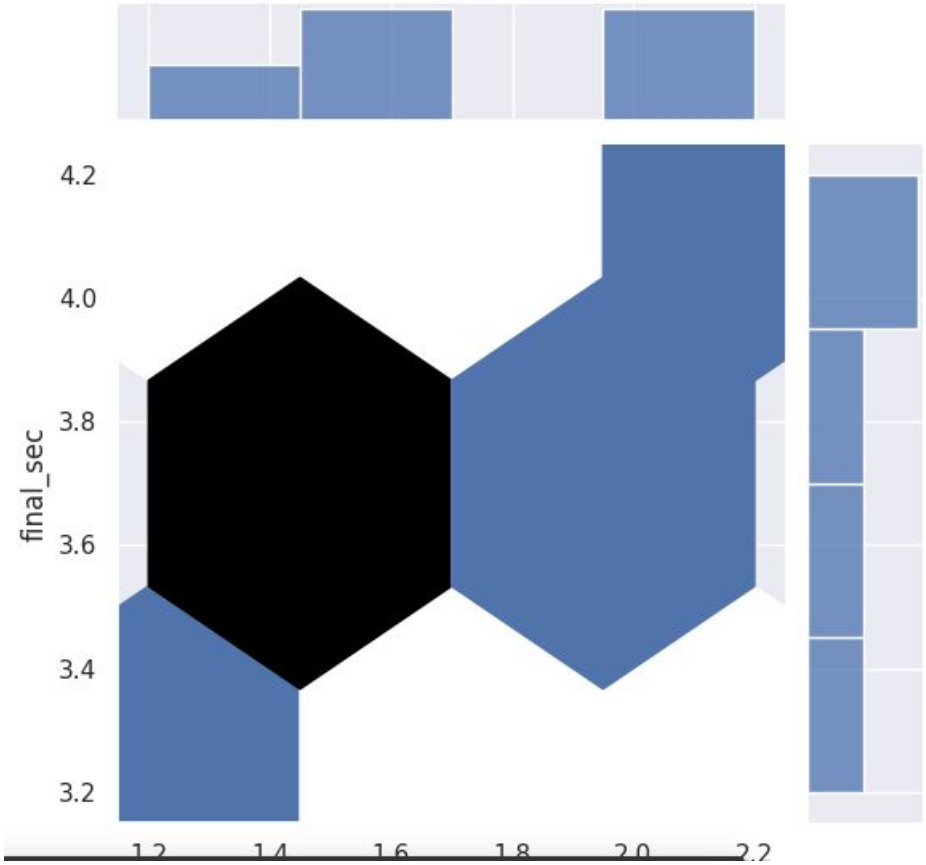
Data Preparation:

```
data = pd.read_csv('marathon-data.csv')

# Convert time strings to timedelta
def convert_time(s):
    h, m, s = map(int, s.split(':'))
    return datetime.timedelta(hours=h, minutes=m, seconds=s)

data = pd.read_csv('marathon-data.csv',
                   converters={'split': convert_time, 'final': convert_time})

# Add split and final times in seconds
data['split_sec'] = data['split'].dt.total_seconds()
data['final_sec'] = data['final'].dt.total_seconds()
```

Split vs. Final Time:

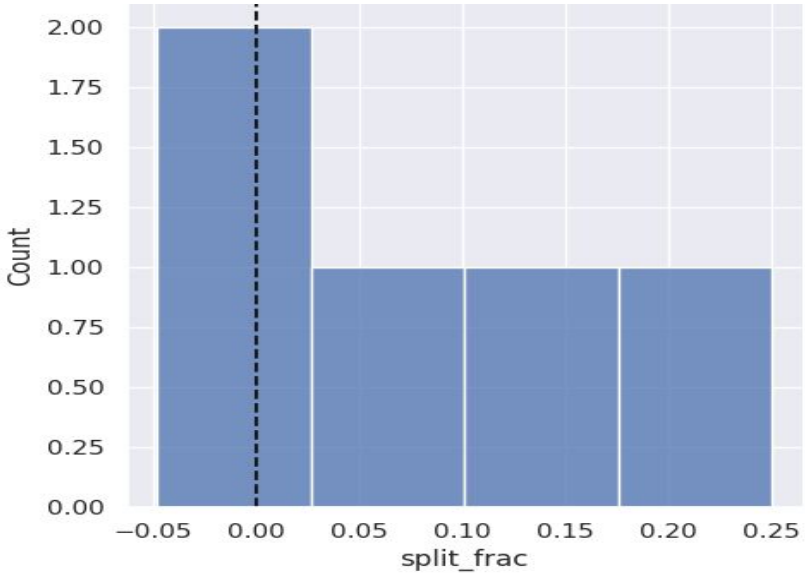
```
data = pd.DataFrame({
    'split_sec': [1.2, 1.5, 1.7, 2.0, 2.2],
    'final_sec': [3.2, 3.5, 3.7, 4.0, 4.2]
})

# Ensure columns exist and match
print(data.head())
print(data.columns)

# Create joint plot
sns.jointplot(x='split_sec', y='final_sec', data=data, kind='hex')
```

```
   split_sec  final_sec
0        1.2        3.2
1        1.5        3.5
2        1.7        3.7
3        2.0        4.0
4        2.2        4.2
Index(['split_sec', 'final_sec'], dtype='object')
<seaborn.axisgrid.JointGrid at 0x78bb774af820>
```


4.2
4.0
3.8
3.6
3.4
3.2
final_sec
1.2
1.4
1.6
1.8
2.0
2.2

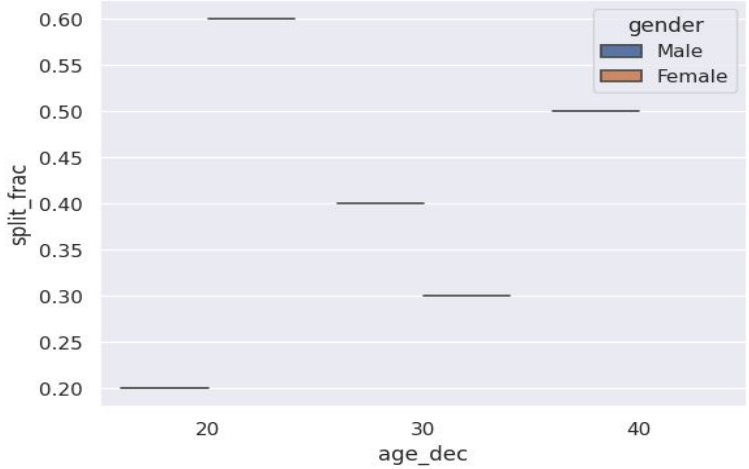
Split Fraction:

```
data['split_frac'] = 1 - 2 * data['split_sec'] / data['final_sec']
sns.displot(data['split_frac'], kde=False)
plt.axvline(0, color="k", linestyle="--")
```



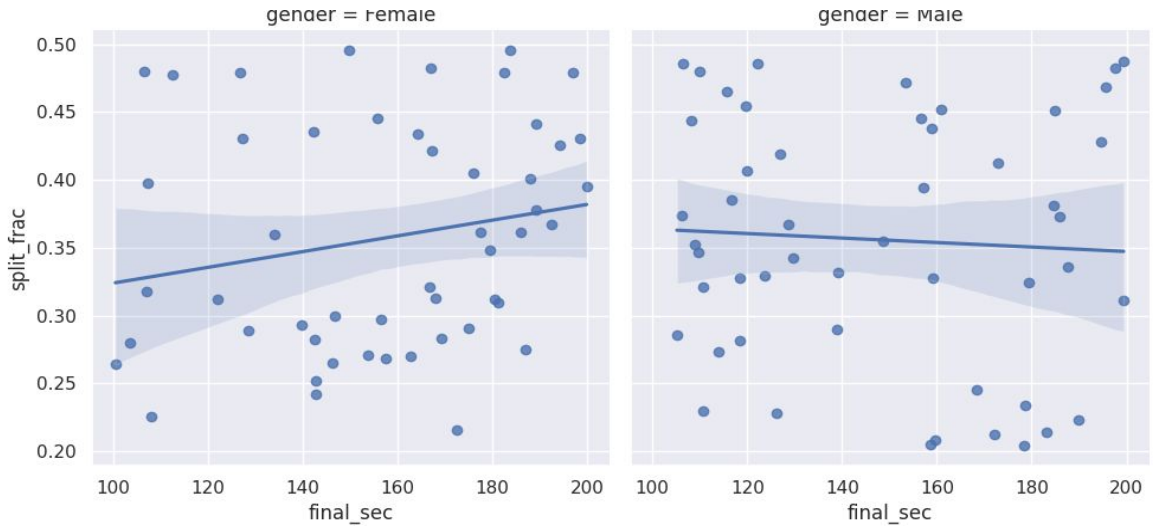
Violin Plots by Gender and Age:

```
data = pd.DataFrame({
    'age_dec': [20, 30, 40, 20, 30],
    'split_frac': [0.2, 0.3, 0.5, 0.6, 0.4],
    'gender': ['Male', 'Female', 'Male', 'Female', 'Male']
})

# Plot violin plot
sns.violinplot(x="age_dec", y="split_frac", hue="gender", data=data, split=True)
```



Linear Regression:

```
# Example DataFrame
data = pd.DataFrame({
    'final_sec': np.random.uniform(100, 200, 100),
    'split_frac': np.random.uniform(0.2, 0.5, 100),
    'gender': np.random.choice(['Male', 'Female'], 100)
})

# Plot with lmplot
sns.lmplot(x='final_sec', y='split_frac', col='gender', data=data)
```



Key Takeaways:

- **Negative Splits: Few runners achieve negative splits, and they tend to have faster overall times.**
- **Gender Comparison: Men are more likely to have even splits than women.**
- **Age Insights: The split fraction varies across age groups, with older women outperforming other groups in some cases.**

Seaborn is a versatile tool for statistical visualization, offering both simplicity and elegance for exploratory data analysis. For further examples, consult the Seaborn documentation.