# Anwendungsbaustein - Auswertung von fds-Daten

Lukas Arnold Simone Arnold Florian Bagemihl
Matthias Baitsch Marc Fehr Maik Poetzsch
Sebastian Seipel

2025-07-22

# Table of contents

# Preamble



Zitiervorschlag

Arnold, Lukas, Simone Arnold, Matthias Baitsch, Marc Fehr, Maik Poetzsch, und Sebastian Seipel. 2024. „Bausteine Computergestützter Datenanalyse. Werkzeugbaustein Plotting in Python“. https://github.com/bausteine-der-datenanalyse/w-python-plotting.

BibTeX-Vorlage

```
@misc{BCD-Styleguide-2024,
 title={Bausteine Computergestützter Datenanalyse. Werkzeugbaustein Plotting in Python},
 author={Arnold, Lukas and Arnold, Simone and Baitsch, Matthias and Fehr, Marc and Poetzsch,
 year={2024},
 url={https://github.com/bausteine-der-datenanalyse/w-python-plotting}}
```

# Intro

## Voraussetzungen

- Grundlagen Python
- Einbinden von zusätzlichen Paketen

## Verwendete Pakete und Datensätze

- matplotlib

## Bearbeitungszeit

Geschätzte Bearbeitungszeit: 1h

## Lernziele

- Einleitung: wie visualisiere ich Daten in Python
- Anpassen von Plots
- Do's & Dont's für wissenschaftliche Plots

# 1 Introduction to Matplotlib

Matplotlib is one of the most well-known libraries for data visualization in Python. It allows the creation of static, animated, and interactive plots with high flexibility.

## 1.1 Why Matplotlib?

- **Broad Support:** Works well with NumPy, Pandas, and SciPy.
- **High Customizability:** Full control over plots.
- **Integration with Jupyter Notebooks:** Ideal for interactive data analysis.
- **Compatibility:** Supports various output formats (PNG, SVG, PDF, etc.).

## 1.2 Alternatives to Matplotlib

While Matplotlib is powerful, there are alternatives that may be better suited for specific purposes: - **Seaborn:** Built on top of Matplotlib, simplifies statistical visualizations. - **Plotly:** Creates interactive plots, great for dashboards. - **Bokeh:** Ideal for web applications with interactive visualizations.

## 1.3 First Example: Plotting a Simple Line

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

# Example data
t = np.linspace(0, 10, 100)
y = np.sin(t)

# Create the plot
plt.plot(t, y, label='sin(t)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
```

```
plt.title('Simple Line Plot')
plt.legend()
plt.show()
```

This simple example demonstrates how to visualize a **sine curve** using Matplotlib.

## 1.4 Next Steps

In the next chapter, we will explore the different types of plots that Matplotlib offers.

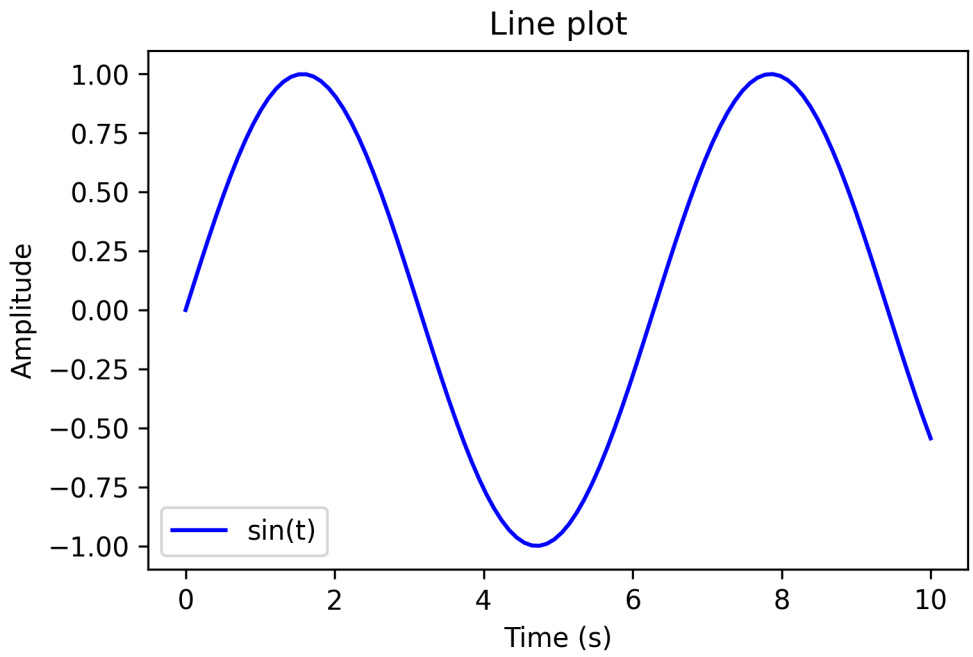
# 2 Plot Types in Matplotlib

Matplotlib offers a variety of plot types suitable for different purposes. In this chapter, we introduce the most important plot types and explain their use cases.

## 2.1 1. Line plots (`plt.plot()`)

Line plots are excellent for visualizing trends over time.

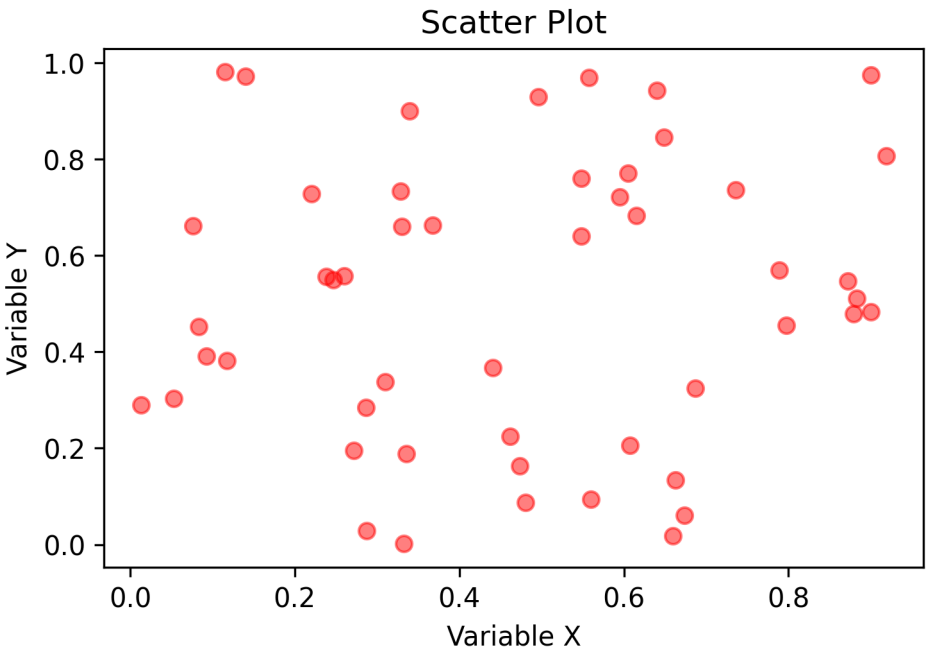
```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(0, 10, 100)
y = np.sin(t)

plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Line plot')
plt.legend()
plt.show()
```




## 2.2 2. Scatter Plots (`plt.scatter()`)

Scatter plots are used to show relationships between two variables.

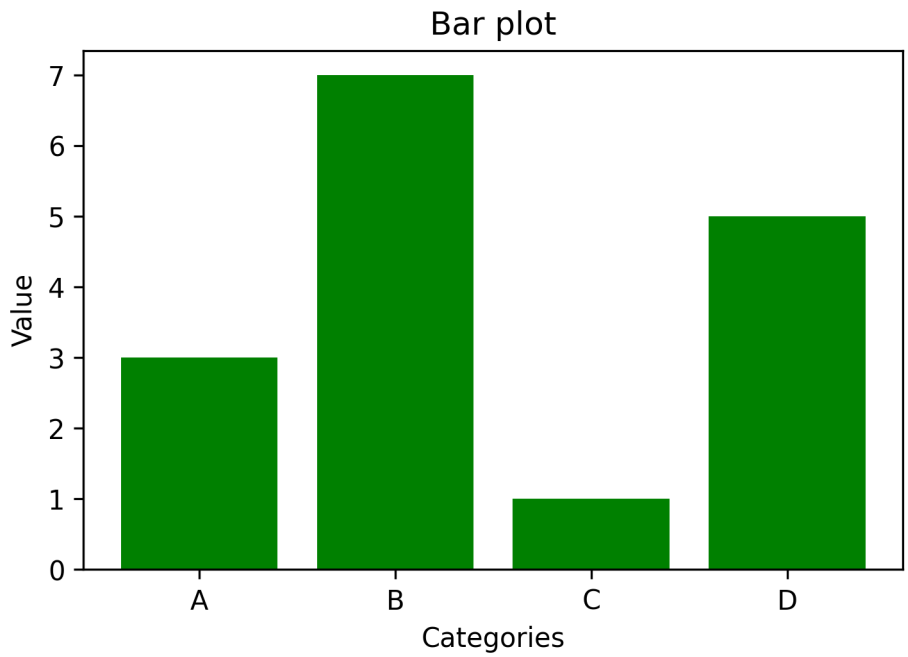
```
x = np.random.rand(50)
y = np.random.rand(50)

plt.scatter(x, y, color='r', alpha=0.5)
plt.xlabel('Variable X')
plt.ylabel('Variable Y')
plt.title('Scatter Plot')
plt.show()
```




## 2.3 3. Bar plots (`plt.bar()`)

Bar plots are suitable for representing categorical data.

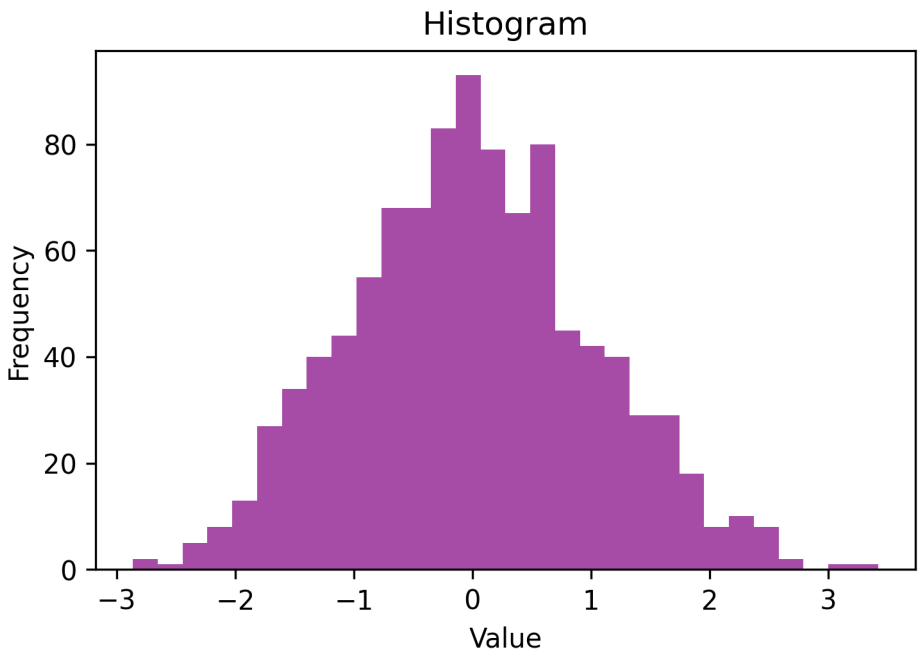
```
categories = ['A', 'B', 'C', 'D']
values = [3, 7, 1, 5]

plt.bar(categories, values, color='g')
plt.xlabel('Categories')
plt.ylabel('Value')
plt.title('Bar plot')
plt.show()
```




## 2.4 4. Histograms (`plt.hist()`)

Histograms show the distribution of numerical data.

```
data = np.random.randn(1000)
plt.hist(data, bins=30, color='purple', alpha=0.7)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram')
plt.show()
```

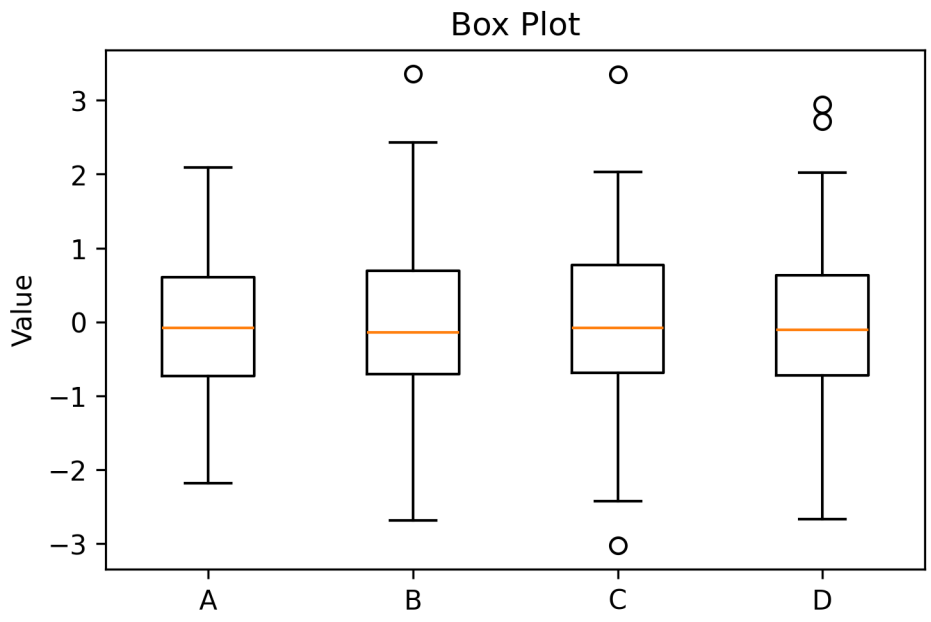



## 2.5 5. Box Plots (`plt.boxplot()`)

Box plots help visualize outliers and data distribution.

```
data = [np.random.randn(100) for _ in range(4)]
plt.boxplot(data, labels=['A', 'B', 'C', 'D'])
plt.ylabel('Value')
plt.title('Box Plot')
plt.show()
```

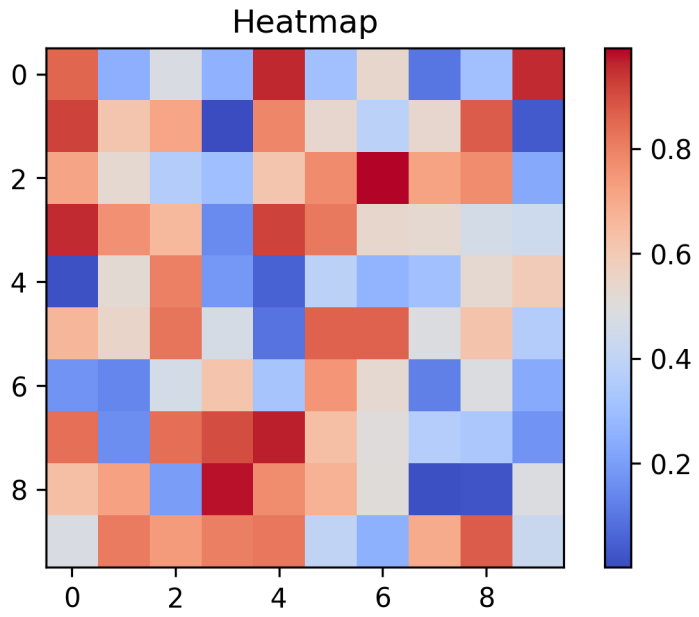
```
/tmp/ipykernel_3895/3930898833.py:2: MatplotlibDeprecationWarning: The 'labels' parameter of
  plt.boxplot(data, labels=['A', 'B', 'C', 'D'])
```




## 2.6 6. Heatmaps (`plt.imshow()`)

Heatmaps are useful for displaying 2D data.

```
data = np.random.rand(10, 10)
plt.imshow(data, cmap='coolwarm', interpolation='nearest')
plt.colorbar()
plt.title('Heatmap')
plt.show()
```




## 2.7 Conclusion

The choice of plot type depends on the nature of the data and the intended representation. In the next chapter, we will explore customizing and adapting plots.

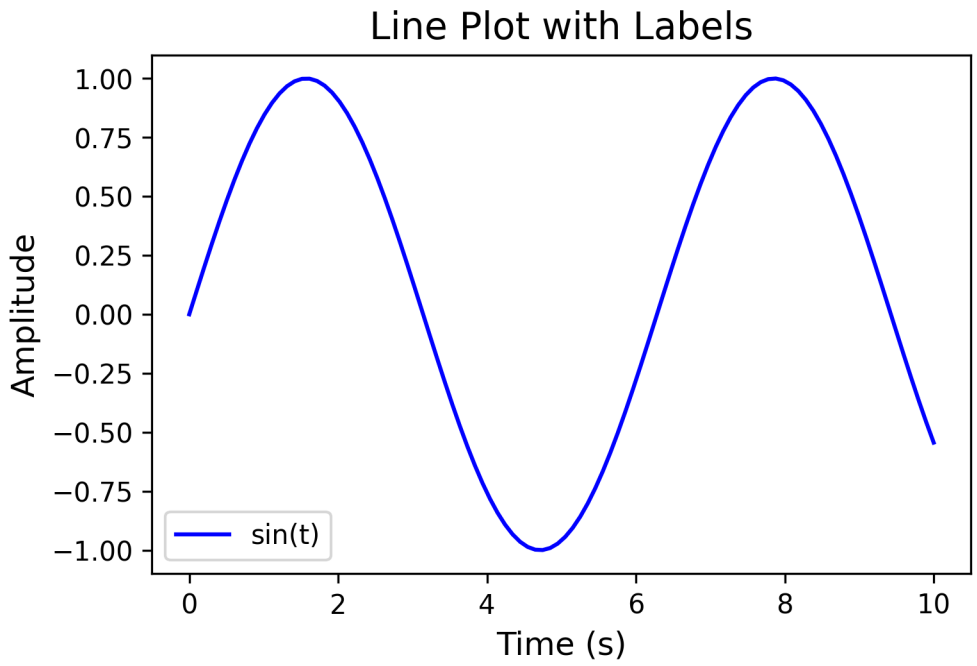
# 3 Customization and Adapting of Plots in Matplotlib

A well-designed plot improves the readability and comprehension of the data presented. In this chapter, we will explore various ways to customize and adapt plots using Matplotlib.

## 3.1 1. Axis Labels and Plot Title

Clear axis and plot titles are essential for understanding a plot.

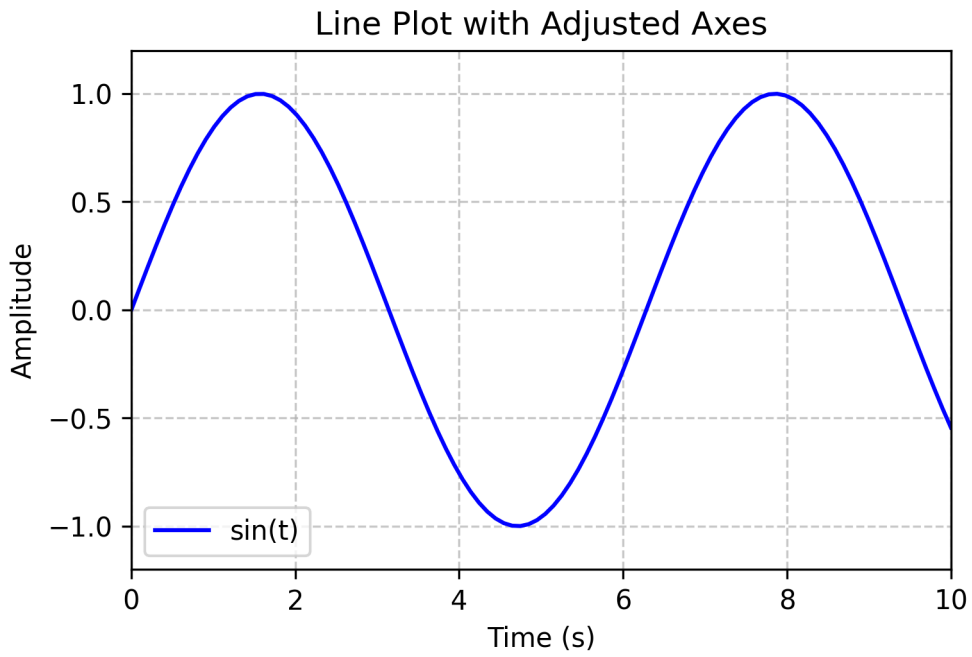
```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(0, 10, 100)
y = np.sin(t)

plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Time (s)', fontsize=12)
plt.ylabel('Amplitude', fontsize=12)
plt.title('Line Plot with Labels', fontsize=14)
plt.legend()
plt.show()
```




## 3.2 2. Adjusting Axes

The scaling of the axes should be chosen appropriately to best represent the data.

```
plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.xlim(0, 10)
plt.ylim(-1.2, 1.2)
plt.grid(True, linestyle='--', alpha=0.7)
plt.title('Line Plot with Adjusted Axes')
plt.legend()
plt.show()
```




## 3.3 3. Colors and Line Styles

Colors and line styles help to highlight important information in the plot.

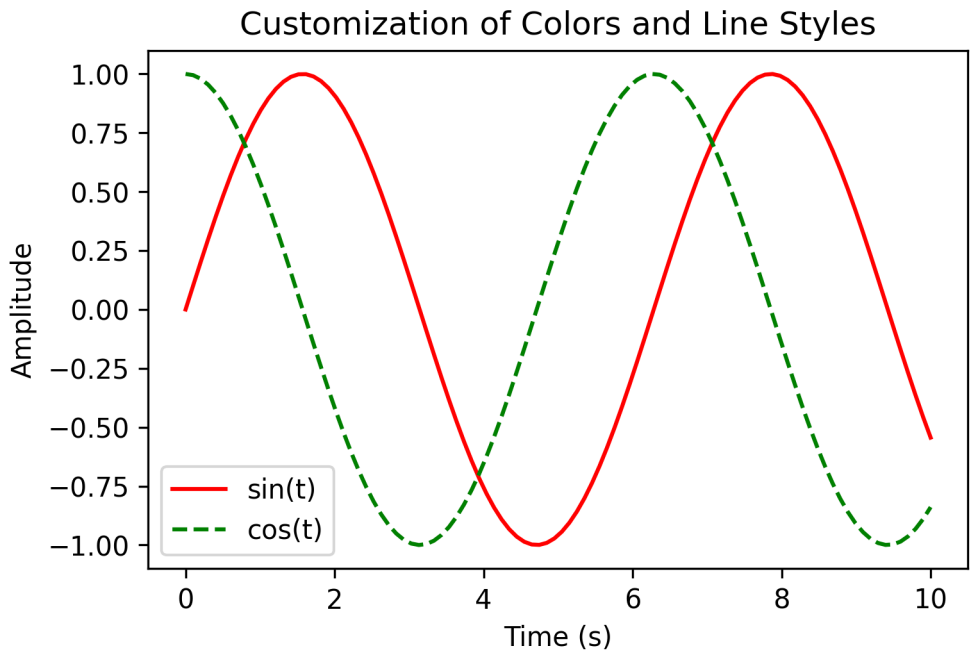
### 3.3.1 Common Colors (Default Colors in Matplotlib)

| Color | Code | Description |
|---|---|---|
| Blue | ‘b’ | blue |
| Green | ‘g’ | green |
| Red | ‘r’ | red |
| Cyan | ‘c’ | cyan |
| Magenta | ‘m’ | magenta |
| Yellow | ‘y’ | yellow |
| Black | ‘k’ | black |
| White | ‘w’ | white |

### 3.3.2 Common Line Styles

| Line Style | Code | Description |
|---|---|---|
| Solid | '-' | default line |
| Dashed | '–' | long dashes |
| Dotted | ':' | only dots |
| Dash-dot | '-.' | alternating dash-dot |

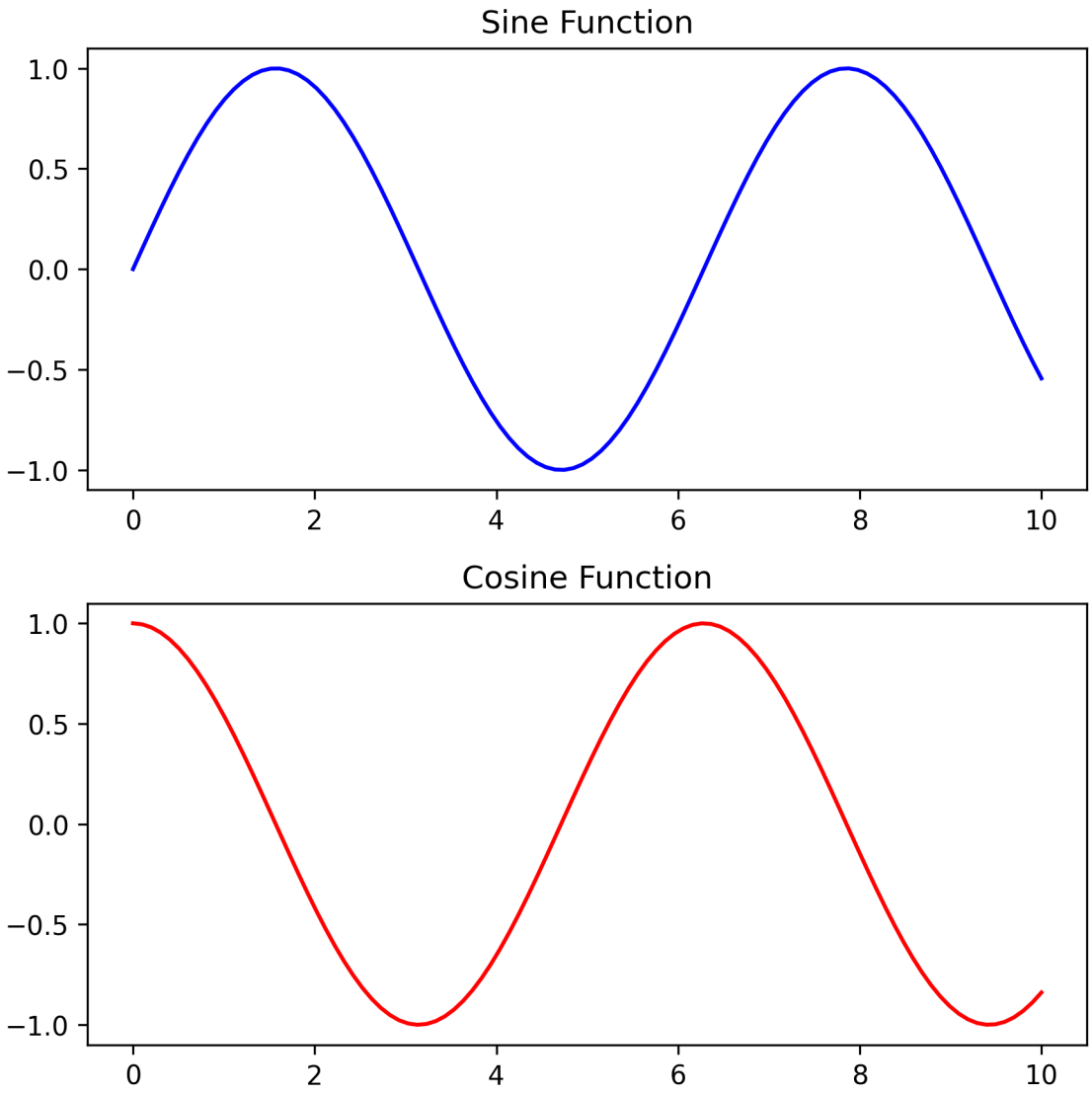
```
plt.plot(t, np.sin(t), linestyle='-', color='r', label='sin(t)')
plt.plot(t, np.cos(t), linestyle='--', color='g', label='cos(t)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Customization of Colors and Line Styles')
plt.legend()
plt.show()
```




## 3.4 4. Multiple Plots with Subplots

Sometimes it's useful to display multiple plots in a single figure.

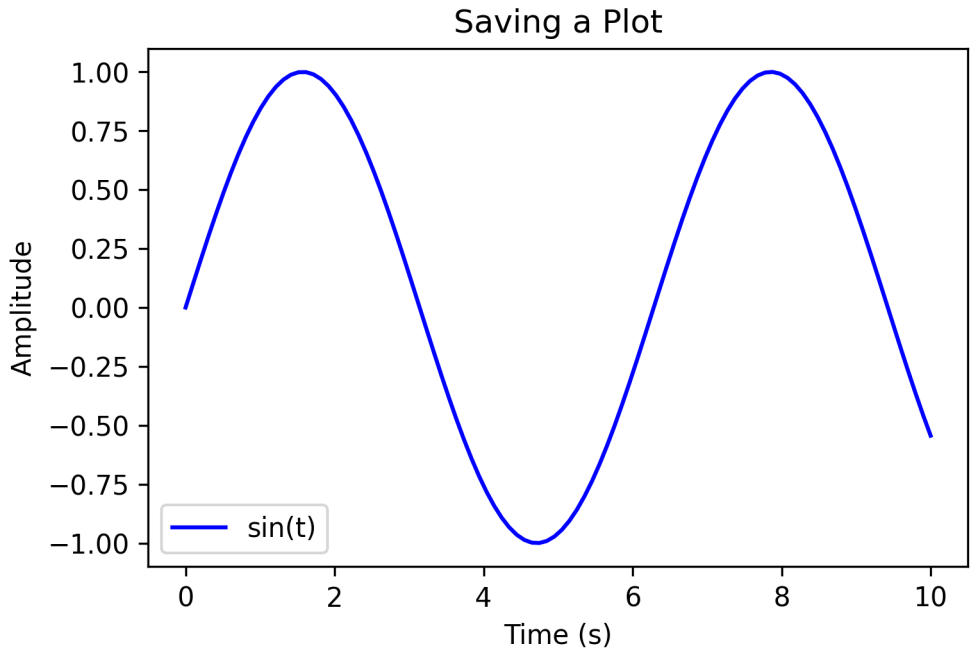
```
fig, axs = plt.subplots(2, 1, figsize=(6, 6))
axs[0].plot(t, np.sin(t), color='b')
axs[0].set_title('Sine Function')
axs[1].plot(t, np.cos(t), color='r')
axs[1].set_title('Cosine Function')
plt.tight_layout()
plt.show()
```



## 3.5 5. Saving Plots

Plots can be saved in various formats.

```
plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Saving a Plot')
plt.legend()
plt.savefig('my_plot.png', dpi=300)
plt.show()
```




## 3.6 Conclusion

With careful customization, scientific plots can be significantly improved. In the next chapter, we will explore advanced techniques such as logarithmic scales and annotations.

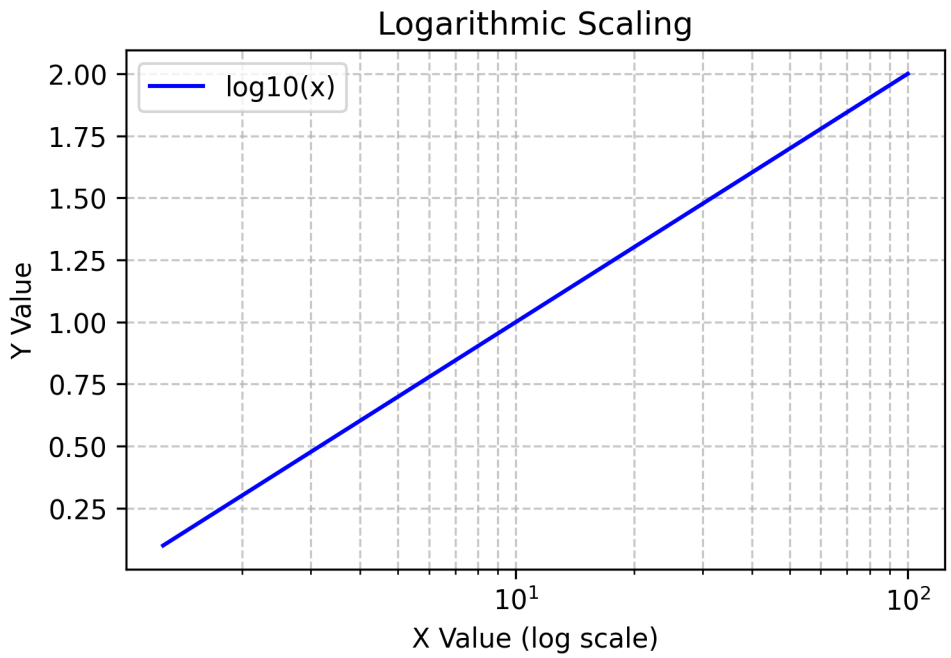
# 4 Advanced Techniques in Matplotlib

In this chapter, we explore some advanced features of Matplotlib that are especially useful for scientific data visualization.

## 4.1 1. Logarithmic Scales

Logarithmic scales are often used when values span several orders of magnitude.

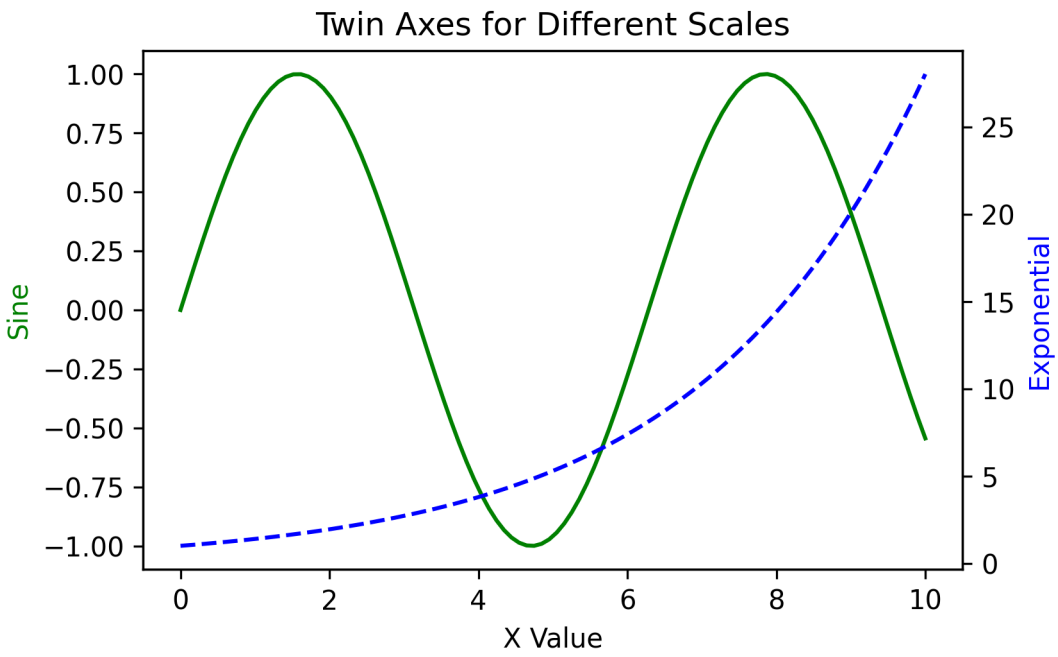
```
import matplotlib.pyplot as plt
import numpy as np

x = np.logspace(0.1, 2, 100)
y = np.log10(x)

plt.plot(x, y, label='log10(x)', color='b')
plt.xscale('log')
plt.xlabel('X Value (log scale)')
plt.ylabel('Y Value')
plt.title('Logarithmic Scaling')
plt.legend()
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.show()
```




## 4.2 2. Twin Axes for Different Scales

Sometimes you may want to display two different y-axes in one plot.

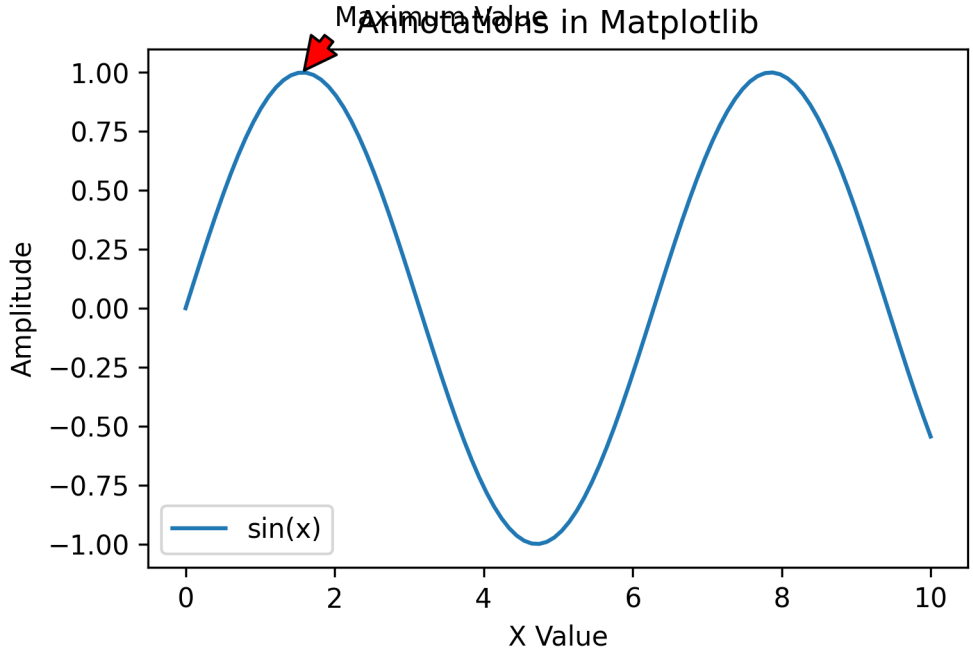
```
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.exp(x / 3)

fig, ax1 = plt.subplots()
ax2 = ax1.twinx()
ax1.plot(x, y1, 'g-', label='sin(x)')
ax2.plot(x, y2, 'b--', label='exp(x/3)')

ax1.set_xlabel('X Value')
ax1.set_ylabel('Sine', color='g')
ax2.set_ylabel('Exponential', color='b')
ax1.set_title('Twin Axes for Different Scales')
plt.show()
```




## 4.3 3. Annotations in Plots

Important points or values in a plot can be highlighted using annotations.

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y, label='sin(x)')
plt.xlabel('X Value')
plt.ylabel('Amplitude')
plt.title('Annotations in Matplotlib')
plt.annotate('Maximum Value', xy=(np.pi/2, 1), xytext=(2, 1.2),
             arrowprops=dict(facecolor='red', shrink=0.05))
plt.legend()
plt.show()
```




## 4.4 Conclusion

These advanced features help make scientific plots more informative. In the next chapter, we will look at best practices and common mistakes in scientific visualization.

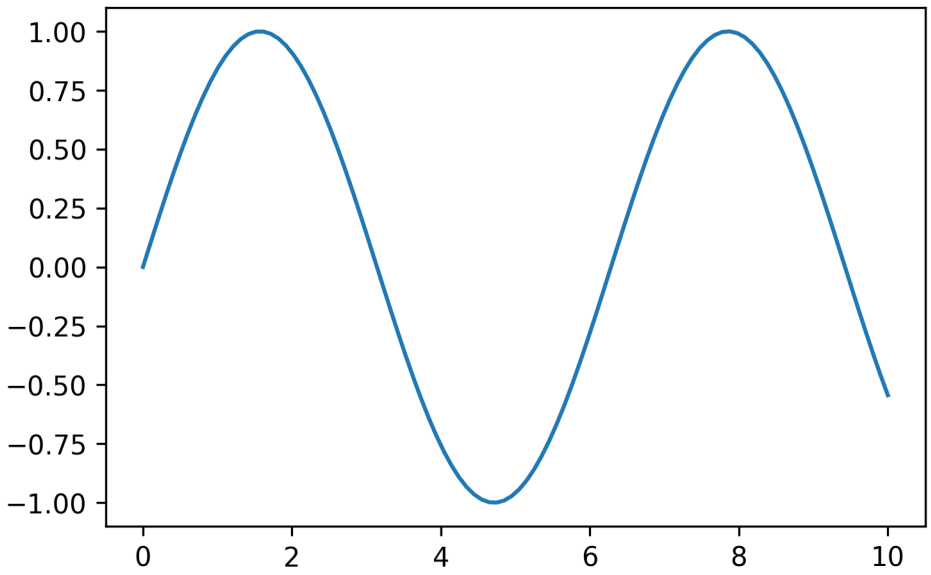
# 5 Best Practices in Matplotlib: Common Mistakes and Improvements

In this chapter, we demonstrate a poor and an improved example for each common issue encountered when plotting with Matplotlib.

## 5.1 1. Missing Labels

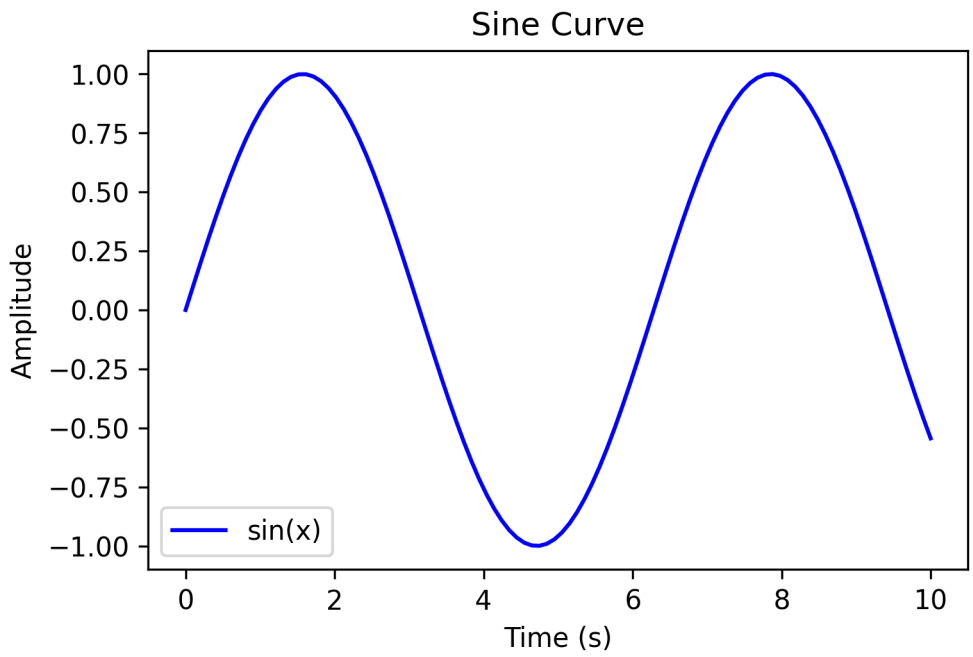
### 5.1.1 Bad Example

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

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

plt.plot(x, y)
plt.show()
```




### 5.1.2 Improved Example

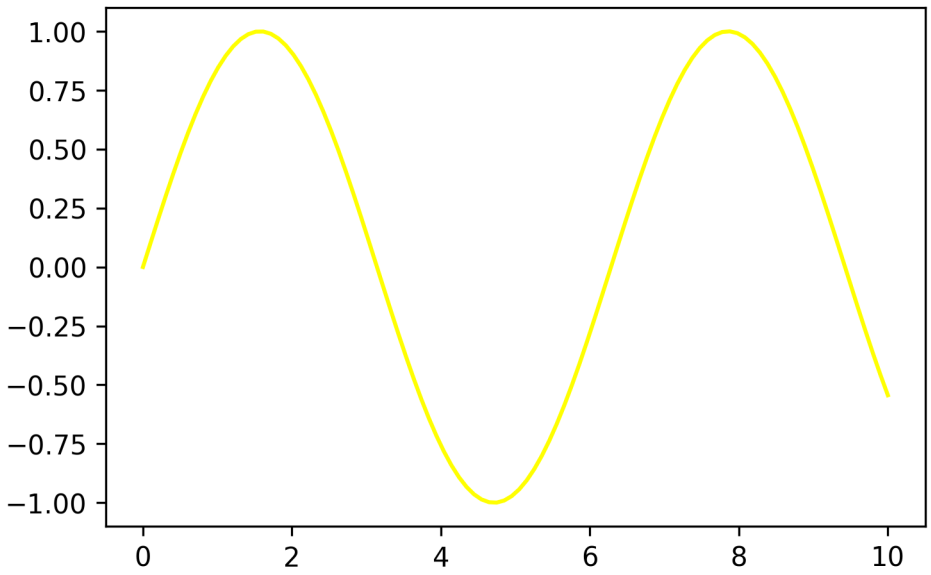
```
plt.plot(x, y, label='sin(x)', color='b')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Sine Curve')
plt.legend()
plt.show()
```




## 5.2 2. Poor Color Choice

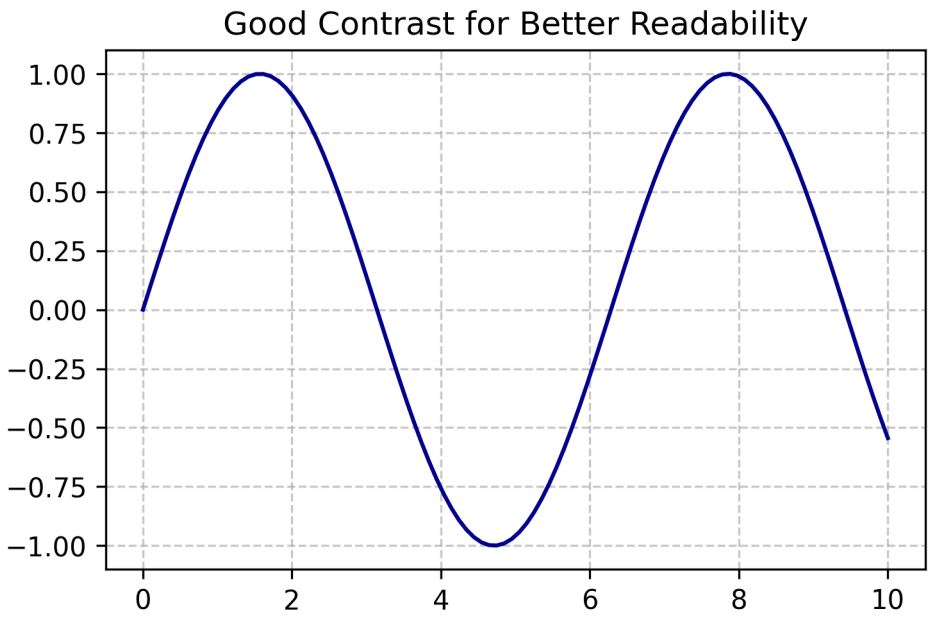
### 5.2.1 Bad Example

```
plt.plot(x, y, color='yellow')
plt.show()
```




### 5.2.2 Improved Example

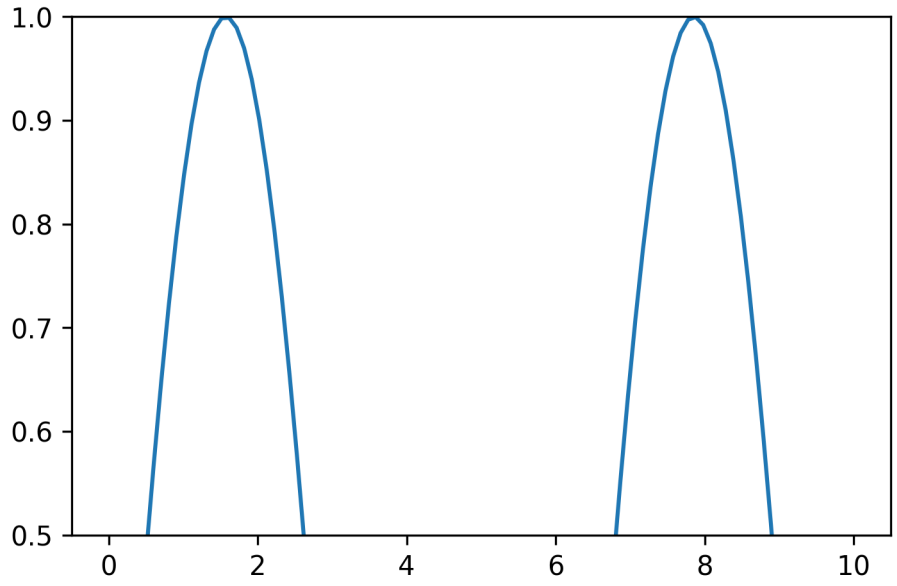
```
plt.plot(x, y, color='darkblue')
plt.grid(True, linestyle='--', alpha=0.7)
plt.title('Good Contrast for Better Readability')
plt.show()
```




## 5.3 3. Unreasonable Axis Scaling

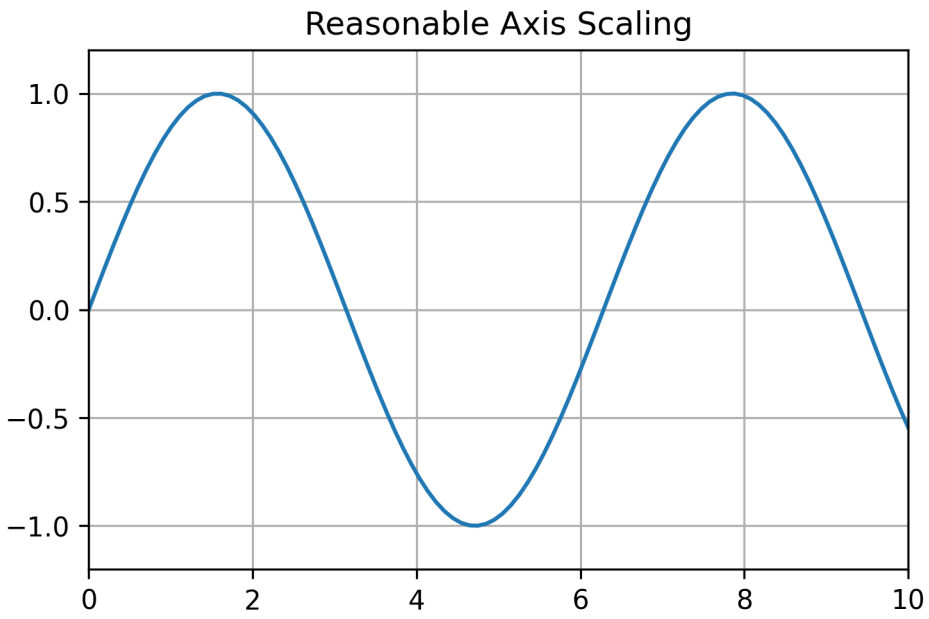
### 5.3.1 Bad Example

```
plt.plot(x, y)
plt.ylim(0.5, 1)
plt.show()
```




### 5.3.2 Improved Example

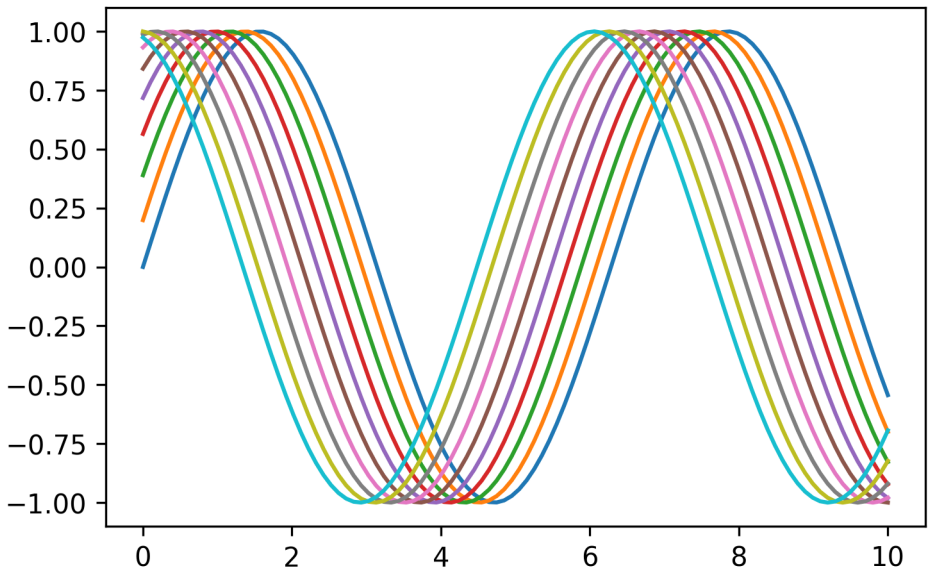
```
plt.plot(x, y)
plt.ylim(-1.2, 1.2)
plt.xlim(0, 10)
plt.grid(True)
plt.title('Reasonable Axis Scaling')
plt.show()
```




## 5.4 4. Overcrowded with Too Many Lines

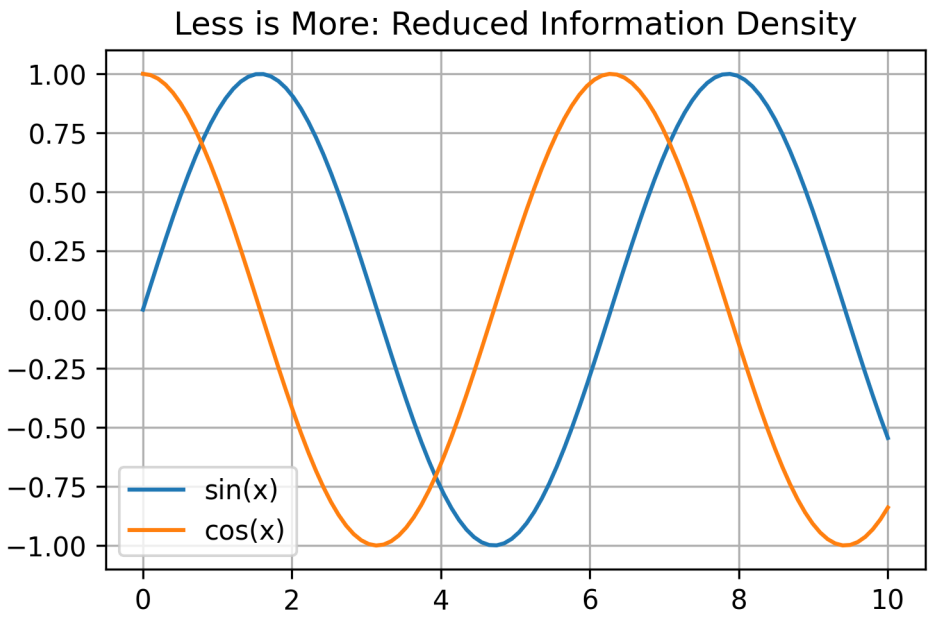
### 5.4.1 Bad Example

```
for i in range(10):
    plt.plot(x, np.sin(x + i * 0.2))
plt.show()
```




### 5.4.2 Improved Example

```
plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
plt.legend()
plt.title('Less is More: Reduced Information Density')
plt.grid(True)
plt.show()
```




## 5.5 Conclusion

Good plots are characterized by clear labels, good readability, and a reasonable amount of information.