Anwendungsbaustein - Auswertung von fds-Daten

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

2025-03-18

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 Einführung in Matplotlib

Matplotlib ist eine der bekanntesten Bibliotheken zur Datenvisualisierung in Python. Sie ermöglicht das Erstellen statischer, animierter und interaktiver Diagramme mit hoher Flexibilität.

1.1 Warum Matplotlib?

- **Breite Unterstützung:** Funktioniert mit NumPy, Pandas und SciPy.
- **Hohe Anpassbarkeit:** Vollständige Kontrolle über Diagramme.
- **Integration in Jupyter Notebooks:** Ideal für interaktive Datenanalyse.
- **Kompatibilität:** Unterstützt verschiedene Ausgabeformate (PNG, SVG, PDF etc.).

1.2 Alternativen zu Matplotlib

Während Matplotlib leistungsstark ist, gibt es Alternativen, die für bestimmte Zwecke besser geeignet sein können: - **Seaborn:** Basiert auf Matplotlib, erleichtert statistische Visualisierung. - **Plotly:** Erzeugt interaktive Plots, gut für Dashboards. - **Bokeh:** Ideal für Web-Anwendungen mit interaktiven Visualisierungen.

1.3 Erstes Beispiel: Einfache Linie plotten

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

# Beispiel-Daten
t = np.linspace(0, 10, 100)
y = np.sin(t)

# Erstellen des Plots
plt.plot(t, y, label='sin(t)')
plt.xlabel('Zeit (s)')
```

```
plt.ylabel('Amplitude')
plt.title('Einfaches Linien-Diagramm')
plt.legend()
plt.show()
```

Dieses einfache Beispiel zeigt, wie man mit Matplotlib eine **Sinuskurve** visualisieren kann.

1.4 Nächste Schritte

Im nächsten Kapitel werden wir uns mit den verschiedenen Diagrammtypen beschäftigen, die Matplotlib bietet.

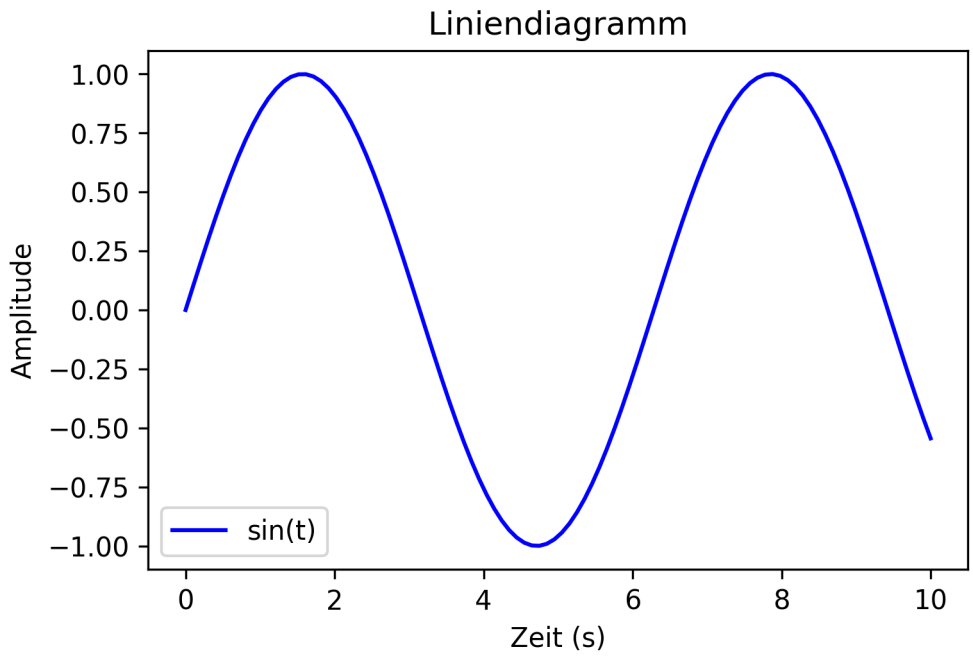
2 Diagrammtypen in Matplotlib

Matplotlib bietet eine Vielzahl von Diagrammtypen, die für unterschiedliche Zwecke geeignet sind. In diesem Kapitel werden die wichtigsten Diagrammtypen vorgestellt und ihre Anwendungsfälle erklärt.

2.1 1. Liniendiagramme (`plt.plot()`)

Liniendiagramme eignen sich hervorragend zur Darstellung von Trends über Zeit.

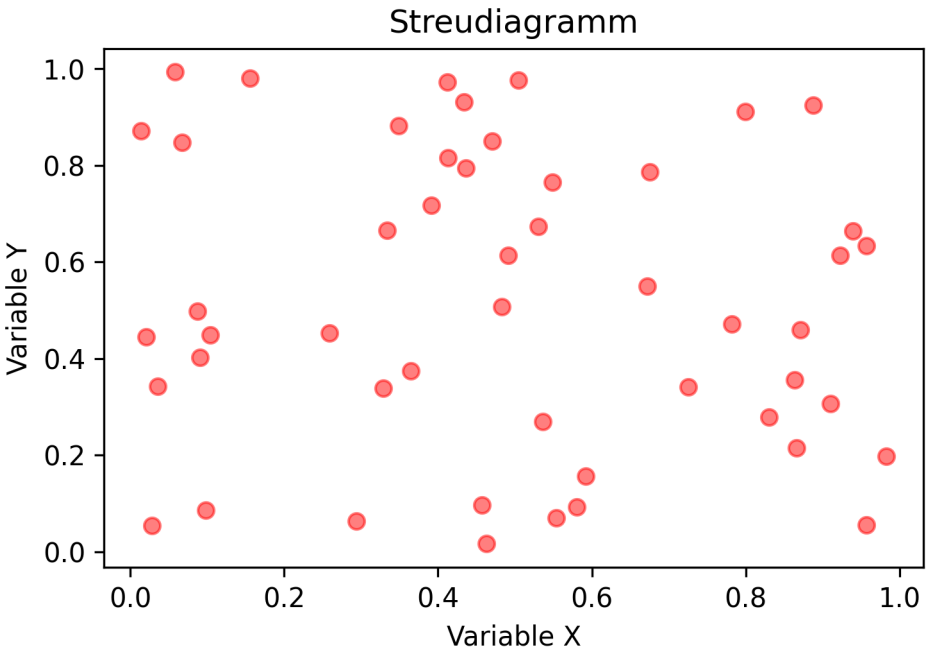
```
import matplotlib.pyplot as plt
import numpy as np

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

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



2.2 2. Streudiagramme (`plt.scatter()`)

Streudiagramme werden verwendet, um Zusammenhänge zwischen zwei Variablen darzustellen.

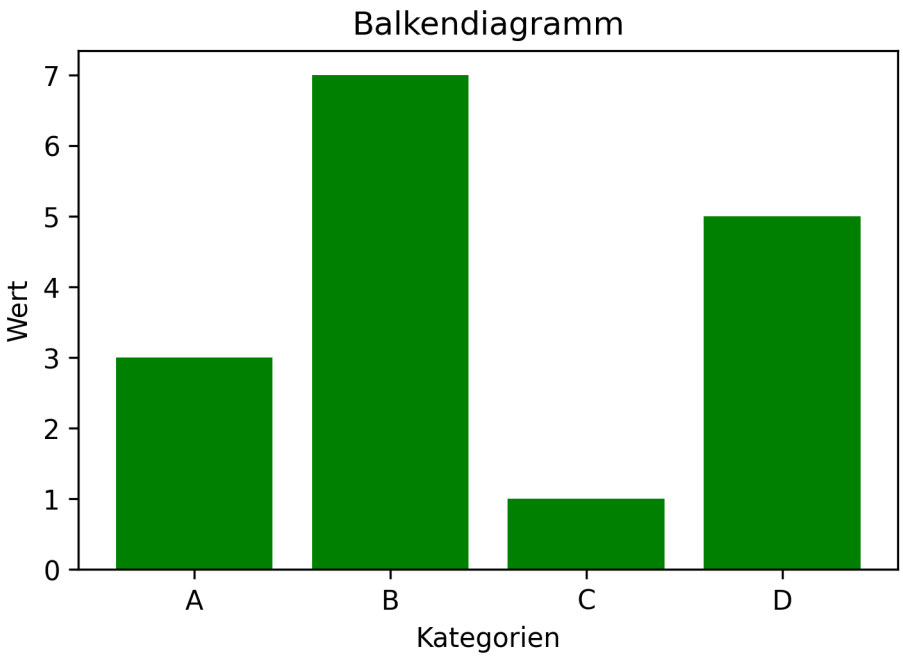
```
x = np.random.rand(50)
y = np.random.rand(50)

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



2.3 3. Balkendiagramme (`plt.bar()`)

Balkendiagramme eignen sich zur Darstellung kategorialer Daten.

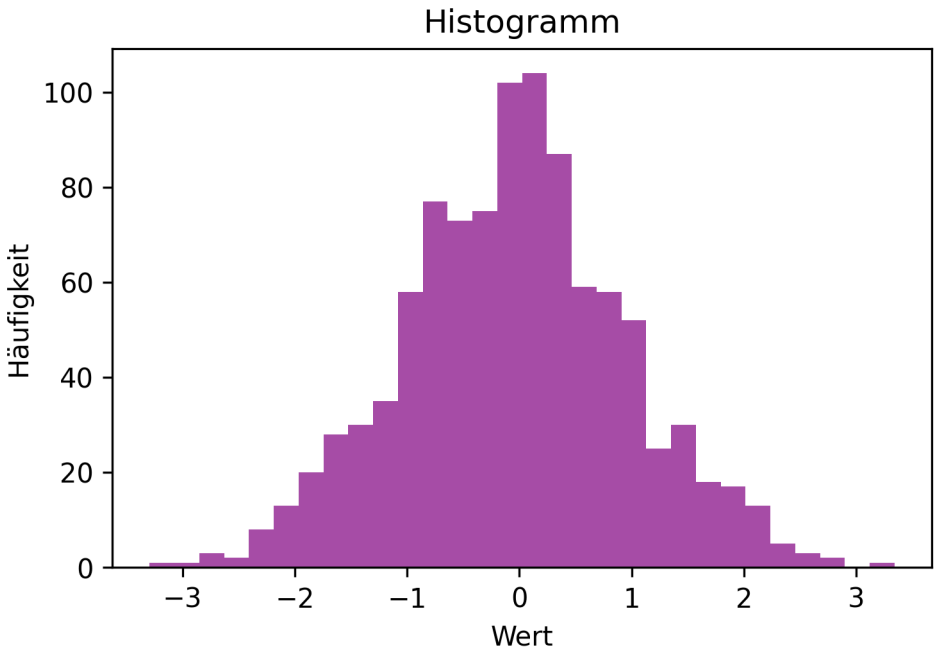
```
kategorien = ['A', 'B', 'C', 'D']
werte = [3, 7, 1, 5]

plt.bar(kategorien, werte, color='g')
plt.xlabel('Kategorien')
plt.ylabel('Wert')
plt.title('Balkendiagramm')
plt.show()
```



2.4 4. Histogramme (`plt.hist()`)

Histogramme zeigen die Verteilung numerischer Daten.

```
daten = np.random.randn(1000)
plt.hist(daten, bins=30, color='purple', alpha=0.7)
plt.xlabel('Wert')
plt.ylabel('Häufigkeit')
plt.title('Histogramm')
plt.show()
```

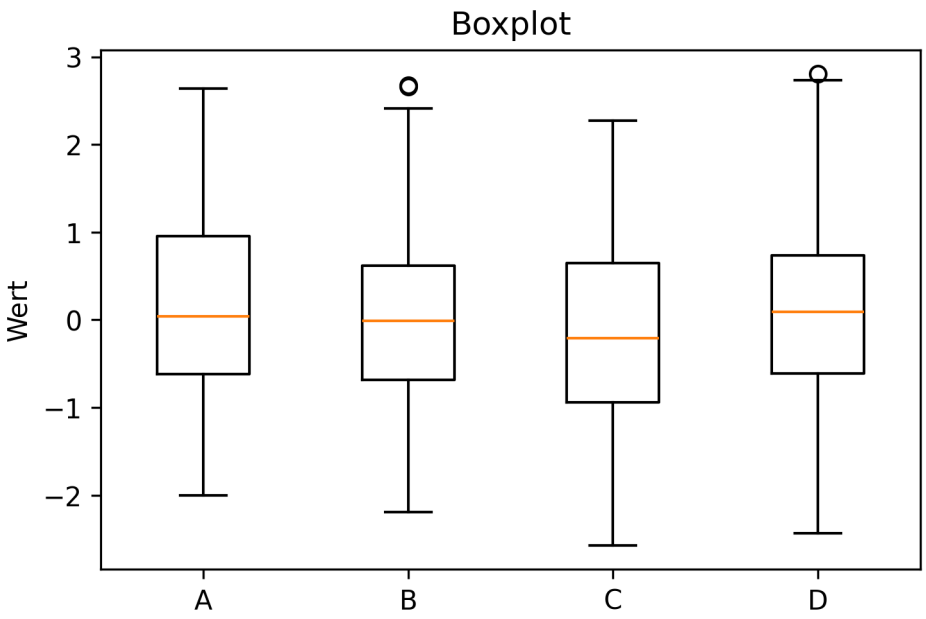


2.5 5. Boxplots (`plt.boxplot()`)

Boxplots helfen, Ausreißer und die Verteilung von Daten zu visualisieren.

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

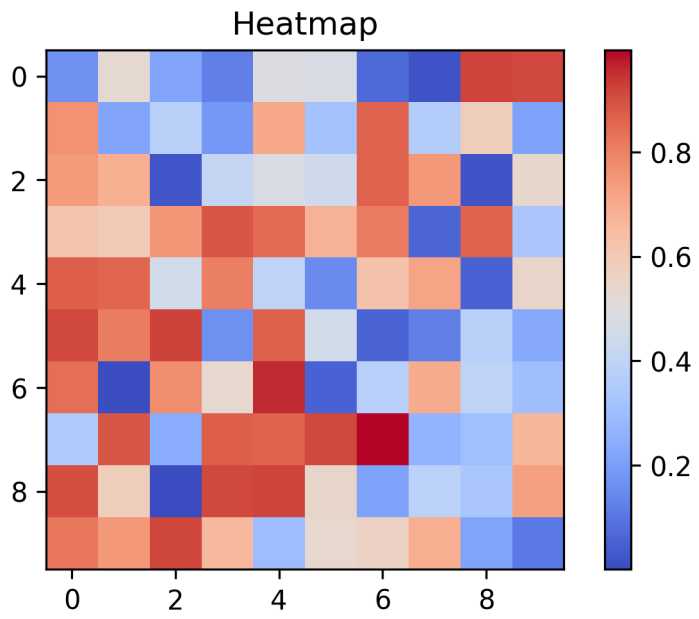
```
/var/folders/p_/ks3trxjx0jd839_g4g0vm4nc0000gn/T/ipykernel_7735/2728911591.py:2: MatplotlibD
  plt.boxplot(daten, labels=['A', 'B', 'C', 'D'])
```



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

Heatmaps eignen sich zur Darstellung von 2D-Daten.

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



2.7 Fazit

Die Wahl des richtigen Diagrammtyps hängt von der Art der Daten und der gewünschten Darstellung ab. Im nächsten Kapitel werden wir uns mit der Anpassung und Gestaltung von Plots beschäftigen.

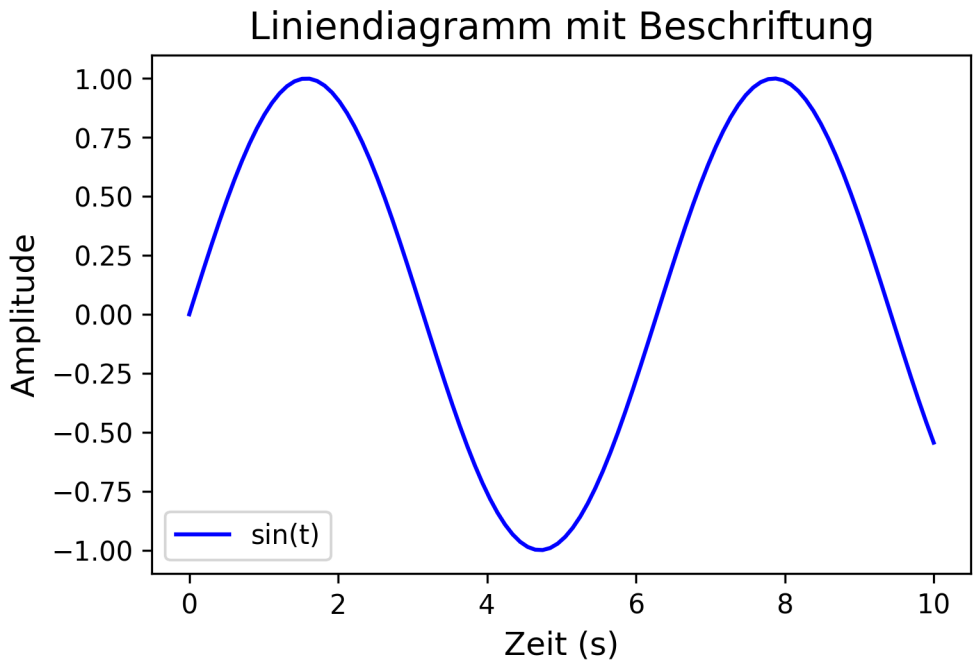
3 Anpassung und Gestaltung von Plots in Matplotlib

Ein gut gestaltetes Diagramm verbessert die Lesbarkeit und Verständlichkeit der dargestellten Daten. In diesem Kapitel werden wir verschiedene Möglichkeiten zur Anpassung und Gestaltung von Plots in Matplotlib erkunden.

3.1 1. Achsentitel und Diagrammtitel

Klare Achsen- und Diagrammtitel sind essenziell für die Verständlichkeit eines Plots.

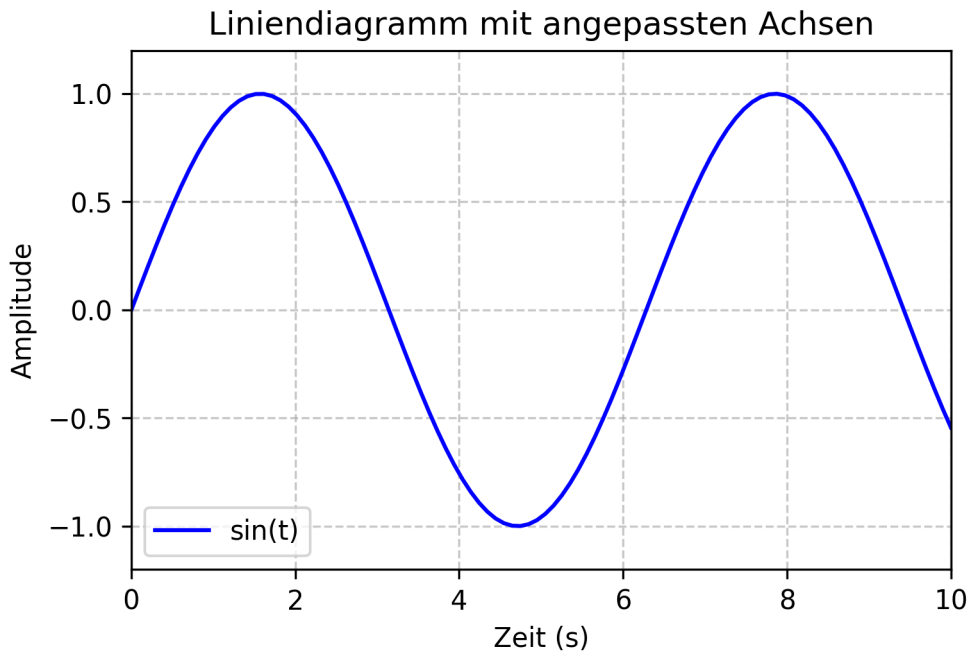
```
import matplotlib.pyplot as plt
import numpy as np

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

plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)', fontsize=12)
plt.ylabel('Amplitude', fontsize=12)
plt.title('Liniendiagramm mit Beschriftung', fontsize=14)
plt.legend()
plt.show()
```



3.2 2. Anpassung der Achsen

Die Skalierung der Achsen sollte sinnvoll gewählt werden, um die Daten bestmöglich darzustellen.

```
plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.xlim(0, 10)
plt.ylim(-1.2, 1.2)
plt.grid(True, linestyle='--', alpha=0.7)
plt.title('Liniendiagramm mit angepassten Achsen')
plt.legend()
plt.show()
```



3.3 3. Farben und Linienstile

Farben und Linienstile helfen dabei, wichtige Informationen im Plot hervorzuheben.

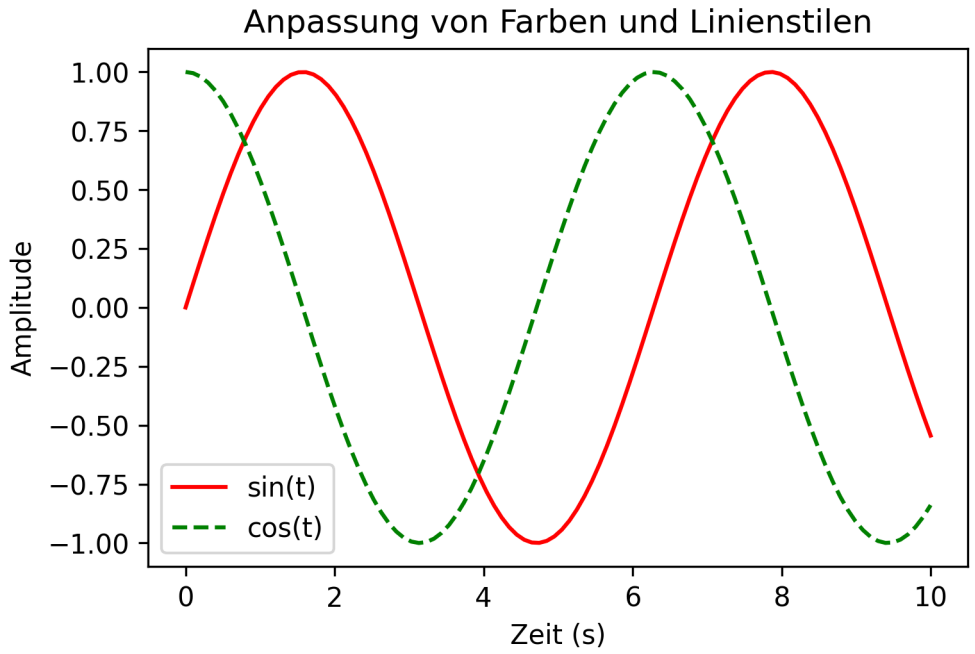
3.3.1 Wichtige Farben (Standardfarben in Matplotlib)

Farbe	Kürzel	Beschreibung
Blau	‘b’	blue
Grün	‘g’	green
Rot	‘r’	red
Cyan	‘c’	cyan
Magenta	‘m’	magenta
Gelb	‘y’	yellow
Schwarz	‘k’	black
Weiß	‘w’	white

3.3.2 Wichtige Linienstile

Linienstil	Kürzel	Beschreibung
Durchgezogen	'-'	Standardlinie
Gestrichelt	'–'	lange Striche
Gepunktet	':'	nur Punkte
Strich-Punkt	'-.'	abwechselnd Strich-Punkt

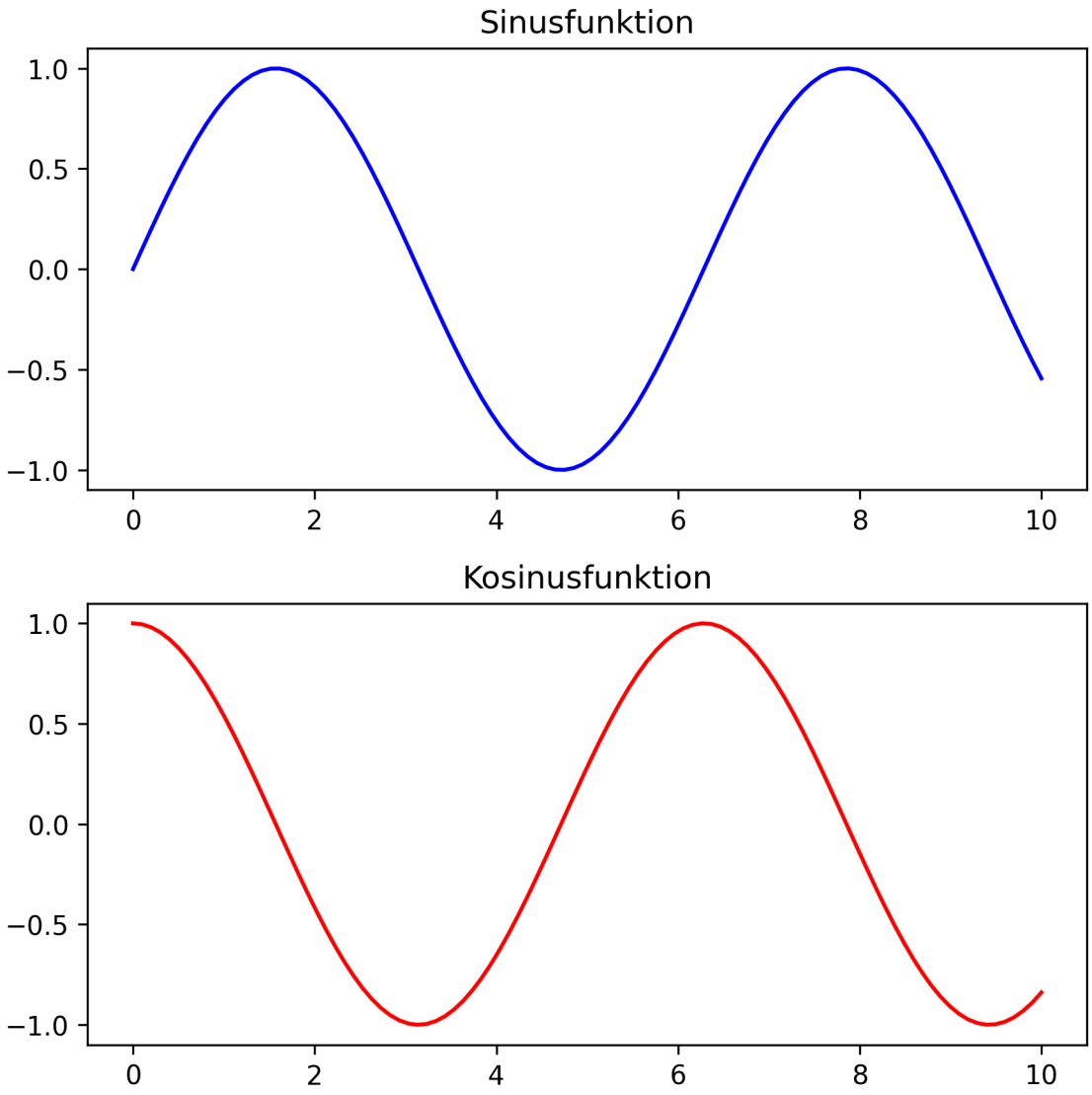
```
plt.plot(t, np.sin(t), linestyle='-', color='r', label='sin(t)')
plt.plot(t, np.cos(t), linestyle='--', color='g', label='cos(t)')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.title('Anpassung von Farben und Linienstilen')
plt.legend()
plt.show()
```



3.4 4. Mehrere Plots mit Subplots

Manchmal ist es sinnvoll, mehrere Diagramme in einer Abbildung darzustellen.

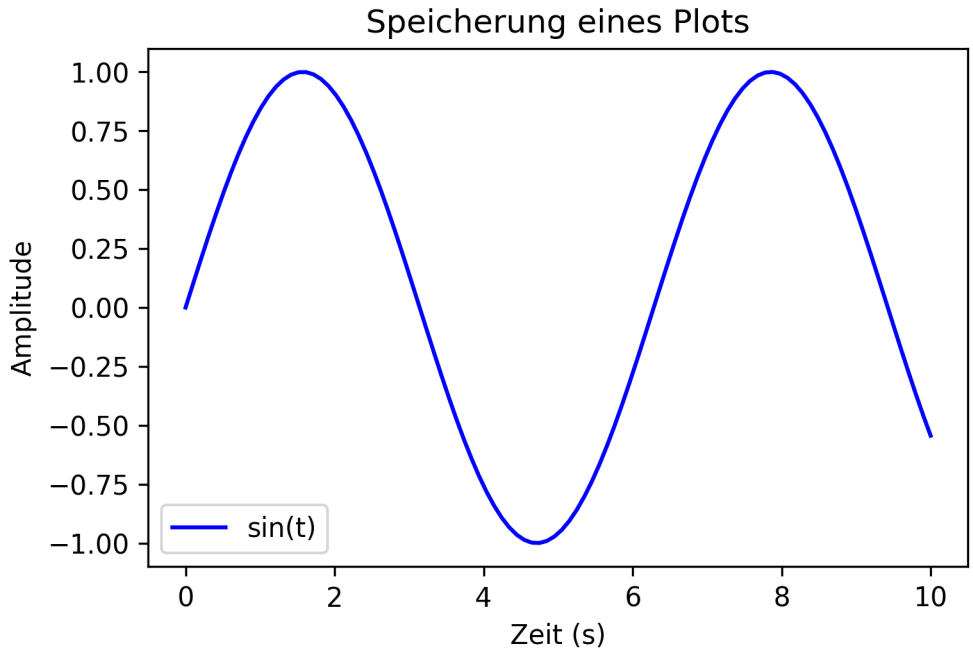
```
fig, axs = plt.subplots(2, 1, figsize=(6, 6))
axs[0].plot(t, np.sin(t), color='b')
axs[0].set_title('Sinusfunktion')
axs[1].plot(t, np.cos(t), color='r')
axs[1].set_title('Kosinusfunktion')
plt.tight_layout()
plt.show()
```



3.5 5. Speichern von Plots

Man kann Diagramme in verschiedenen Formaten speichern.

```
plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.title('Speicherung eines Plots')
plt.legend()
plt.savefig('mein_plot.png', dpi=300)
plt.show()
```



3.6 Fazit

Durch geschickte Anpassungen lassen sich wissenschaftliche Plots deutlich verbessern. Im nächsten Kapitel werden wir uns mit erweiterten Techniken wie logarithmischen Skalen und Annotationen beschäftigen.

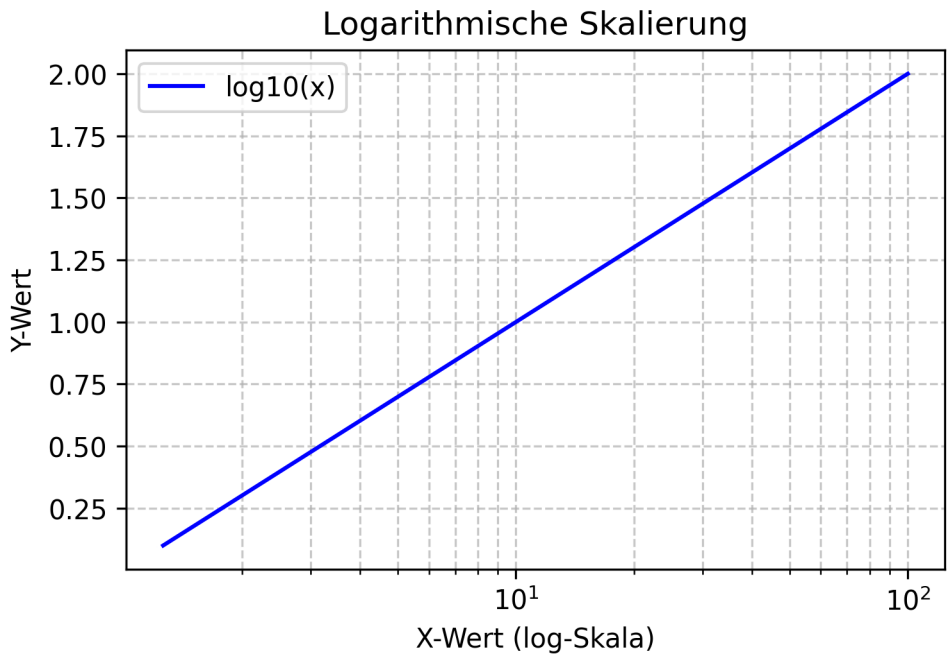
4 Erweiterte Techniken in Matplotlib

In diesem Kapitel betrachten wir einige fortgeschrittene Funktionen von Matplotlib, die für die wissenschaftliche Datenvisualisierung besonders nützlich sind.

4.1 1. Logarithmische Skalen

Logarithmische Skalen werden oft verwendet, wenn Werte große Größenordnungen umfassen.

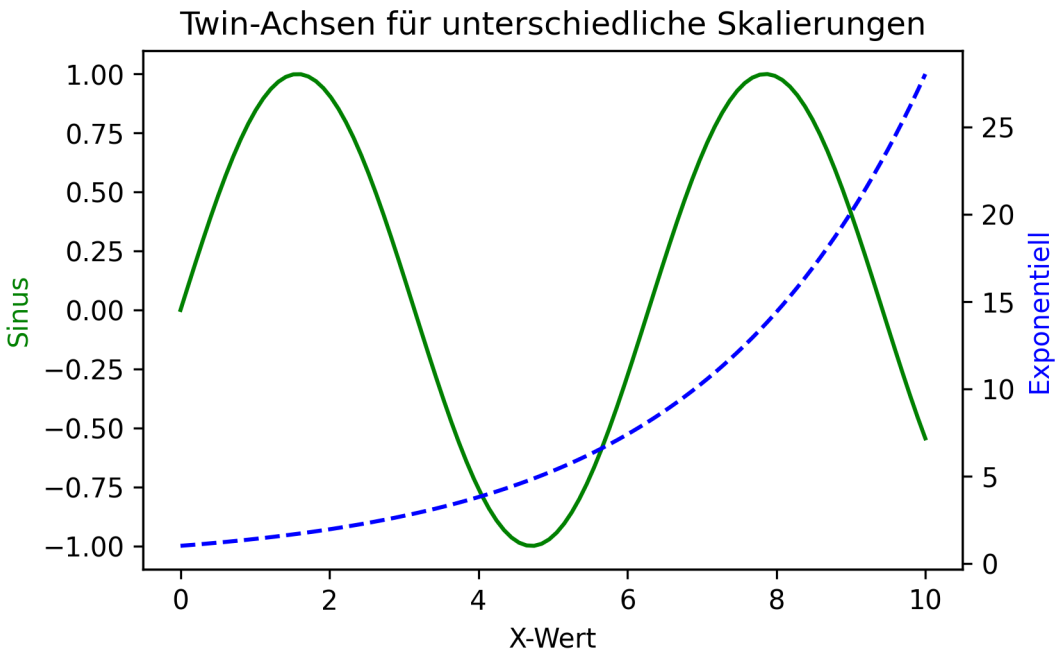
```
import matplotlib.pyplot as plt
import numpy as np

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-Wert (log-Skala)')
plt.ylabel('Y-Wert')
plt.title('Logarithmische Skalierung')
plt.legend()
plt.grid(True, which='both', linestyle='--', alpha=0.7)
plt.show()
```



4.2 2. Twin-Achsen für verschiedene Skalierungen

Manchmal möchte man zwei verschiedene y-Achsen in einem Plot darstellen.

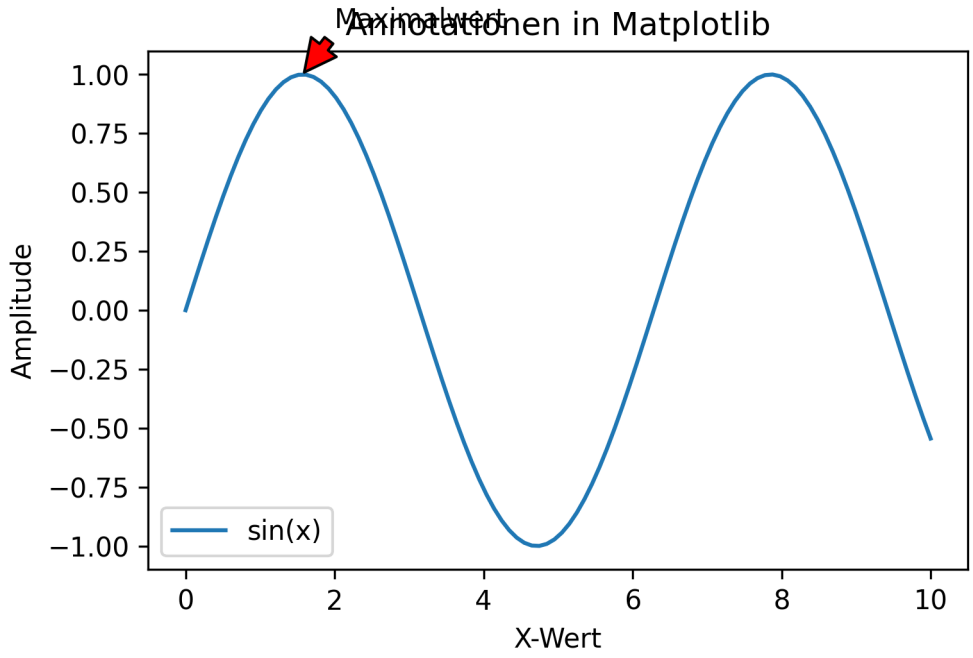
```
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.exp(x / 3)

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-Wert')
ax1.set_ylabel('Sinus', color='g')
ax2.set_ylabel('Exponentiell', color='b')
ax1.set_title('Twin-Achsen für unterschiedliche Skalierungen')
plt.show()
```



4.3 3. Annotationen in Diagrammen

Wichtige Punkte oder Werte in einem Diagramm können mit Annotationen hervorgehoben werden.

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

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



4.4 Fazit

Diese erweiterten Funktionen helfen dabei, wissenschaftliche Plots noch informativer zu gestalten. Im nächsten Kapitel werden wir Best Practices und typische Fehler in der wissenschaftlichen Visualisierung betrachten.

5 Best Practices und häufige Fehler in der wissenschaftlichen Visualisierung

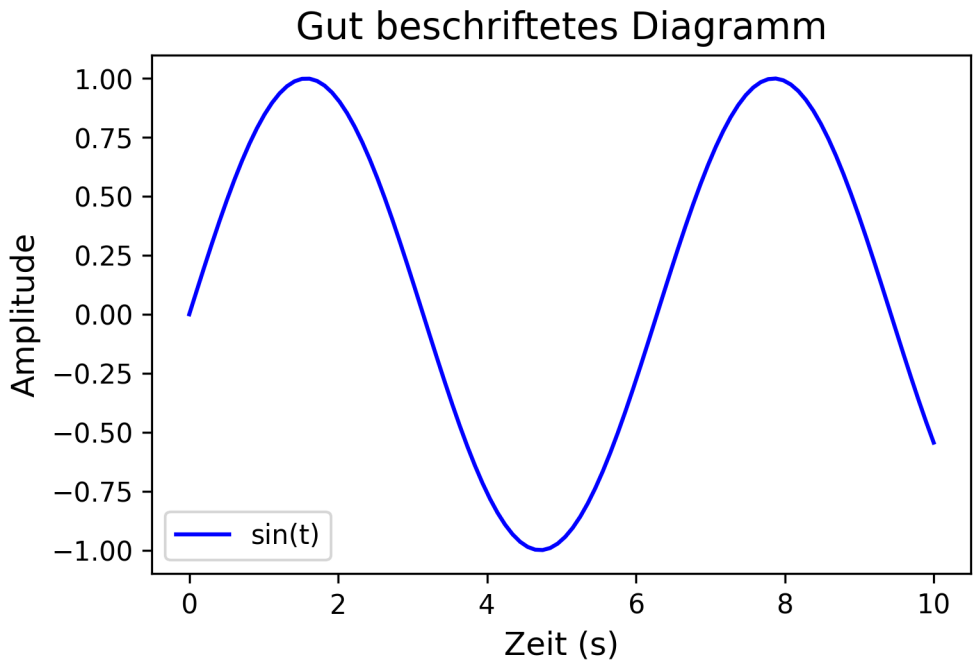
Eine gute wissenschaftliche Visualisierung ist klar, informativ und vermeidet irreführende Darstellungen. In diesem Kapitel betrachten wir bewährte Praktiken sowie typische Fehler, die bei der Nutzung von Matplotlib auftreten können.

5.1 1. Best Practices für wissenschaftliche Plots

5.1.1 Klare Achsenbeschriftungen und Titel verwenden

Ein Diagramm sollte immer gut beschriftet sein, um Missverständnisse zu vermeiden.

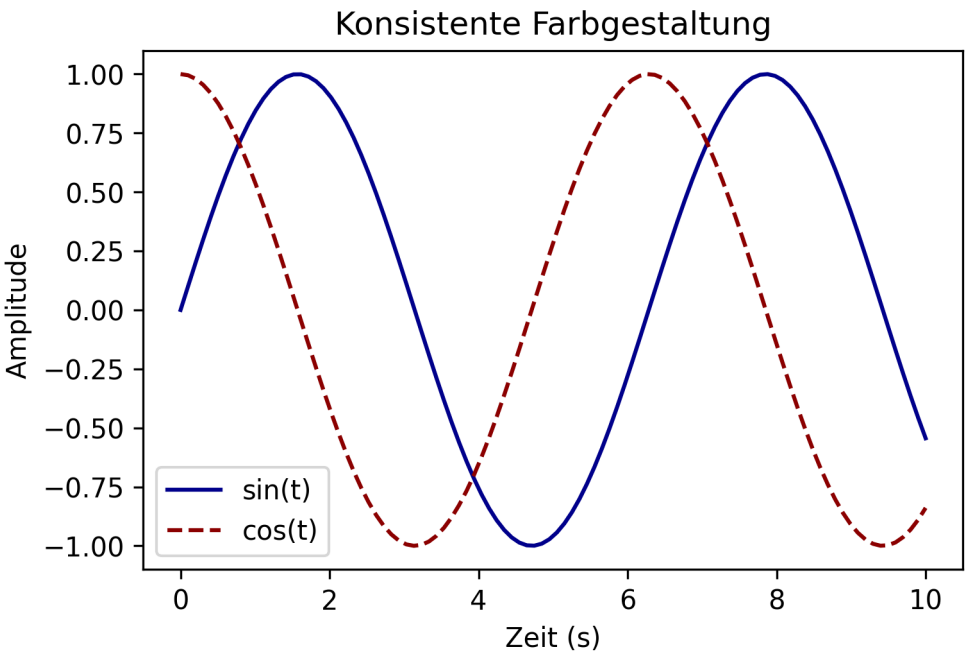
```
import matplotlib.pyplot as plt
import numpy as np

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

plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)', fontsize=12)
plt.ylabel('Amplitude', fontsize=12)
plt.title('Gut beschriftetes Diagramm', fontsize=14)
plt.legend()
plt.show()
```



5.1.2 Konsistente Farbgebung und Kontraste

Farbwahl sollte sinnvoll sein und auf gute Lesbarkeit achten.

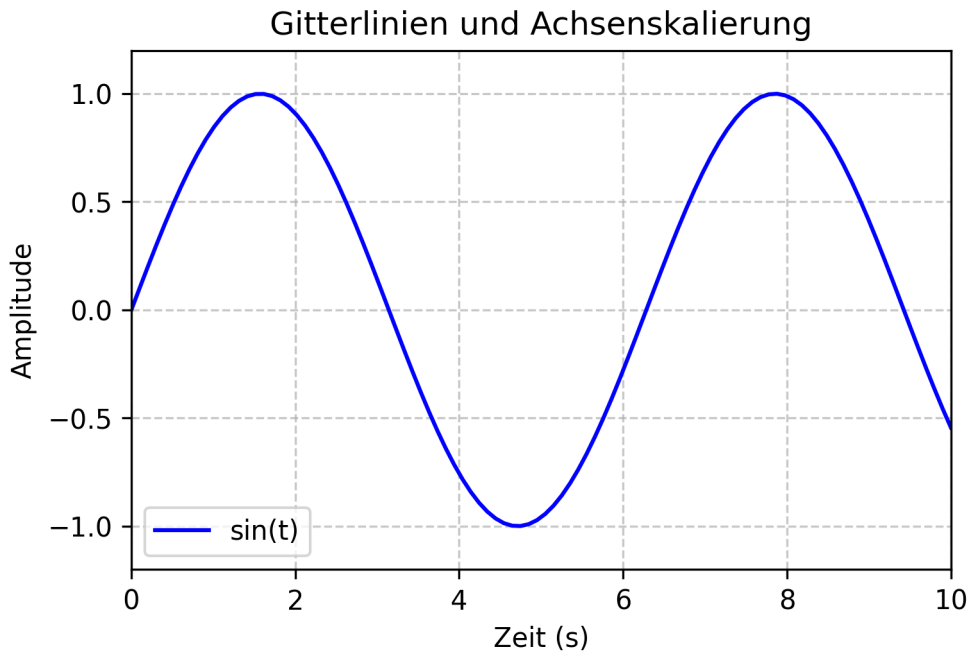
```
plt.plot(t, np.sin(t), color='darkblue', linestyle='-', label='sin(t)')
plt.plot(t, np.cos(t), color='darkred', linestyle='--', label='cos(t)')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.title('Konsistente Farbgestaltung')
plt.legend()
plt.show()
```



5.1.3 Richtige Skalierung und Gitterlinien nutzen

Eine gute Skalierung verbessert die Lesbarkeit von Diagrammen.

```
plt.plot(t, y, label='sin(t)', color='b')
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.xlim(0, 10)
plt.ylim(-1.2, 1.2)
plt.grid(True, linestyle='--', alpha=0.7)
plt.title('Gitterlinien und Achsenskalierung')
plt.legend()
plt.show()
```



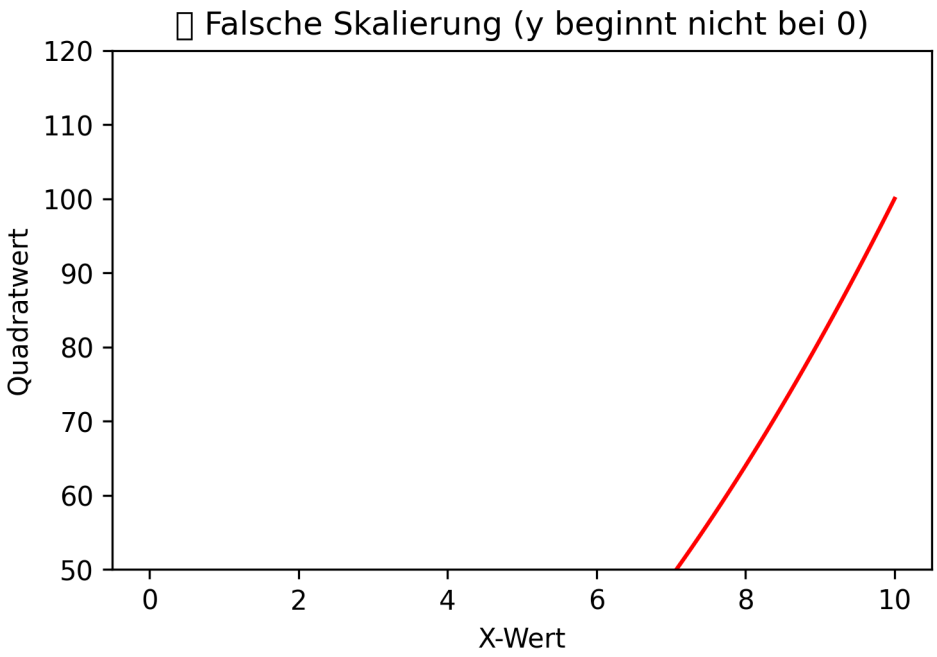
5.2 2. Häufige Fehler in wissenschaftlichen Plots

5.2.1 Irreführende Achsenskalierung

Wenn eine y-Achse nicht bei Null beginnt, kann dies die Daten verzerren.

```
x = np.linspace(0, 10, 100)
y = x**2

plt.plot(x, y, color='r')
plt.xlabel('X-Wert')
plt.ylabel('Quadratwert')
plt.title('  Falsche Skalierung (y beginnt nicht bei 0)')
plt.ylim(50, 120)
plt.show()
```

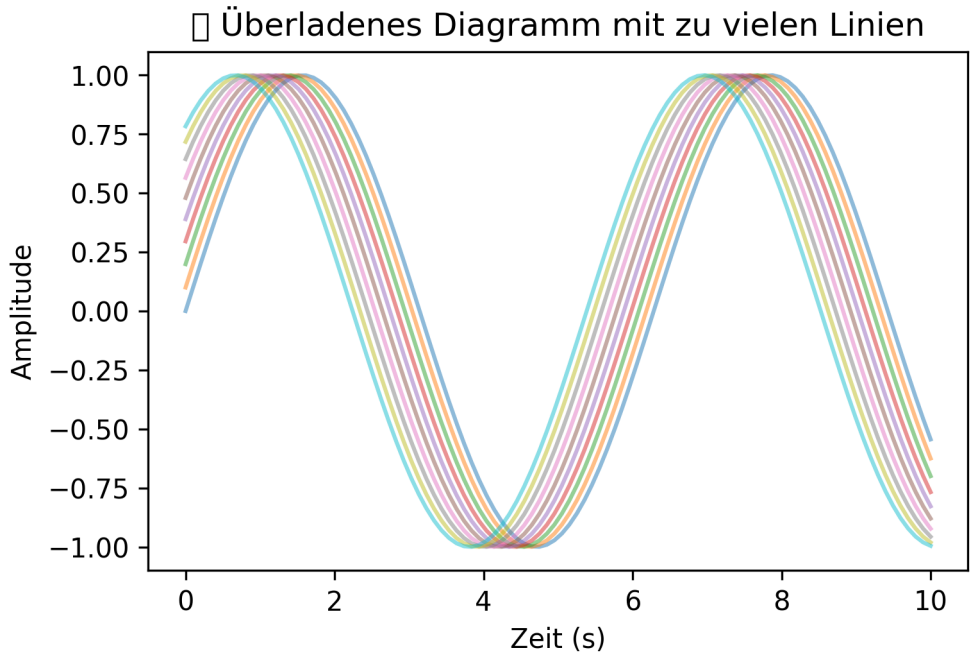
```
/Users/mfehr/Documents/BCD/w-python-plotting/.venv/lib/python3.12/site-packages/IPython/core/
  fig.canvas.print_figure(bytes_io, **kw)
```



5.2.2 Überladene Diagramme

Zu viele Linien oder Datenpunkte können ein Diagramm unübersichtlich machen.

```
for i in range(10):
    plt.plot(t, np.sin(t + i * 0.1), alpha=0.5)
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.title(' Überladenes Diagramm mit zu vielen Linien')
plt.show()
```

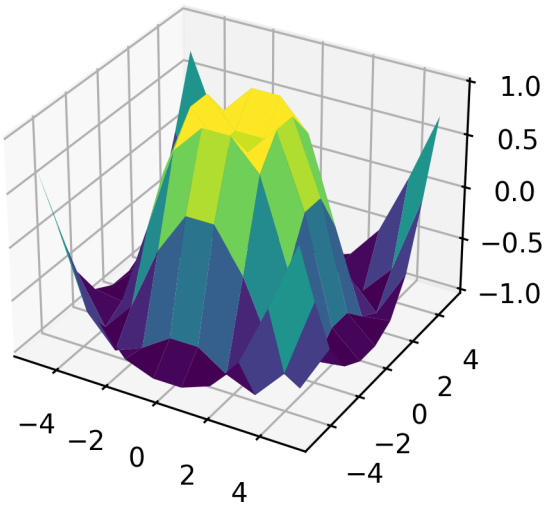


5.2.3 Unnötige 3D-Diagramme

Nicht jede Datenvisualisierung benötigt eine 3D-Darstellung.

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
X, Y = np.meshgrid(np.linspace(-5, 5, 10), np.linspace(-5, 5, 10))
Z = np.sin(np.sqrt(X**2 + Y**2))
ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_title('  Unnötige 3D-Darstellung')
plt.show()
```

 Unnötige 3D-Darstellung



5.3 Fazit

Durch die Anwendung dieser Best Practices und das Vermeiden häufiger Fehler können wissenschaftliche Diagramme klarer und informativer gestaltet werden. Im nächsten Kapitel werden wir interaktive Visualisierungen mit Matplotlib betrachten.