Legenden & direkte Beschriftung

by Woche 6

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

Weil die letzten zwei Kapitel recht viel trockene Theorie waren, soll dieses Kapitel mit ein bisschen mehr matplotlib für Abwechslung sorgen.

Hier werden wir uns mit einem weiteren wichtigen Element befassen: **Legenden**. Eine Legende ist entscheidend für das Verständnis eines Diagramms, insbesondere wenn verschiedene Farben, Formen oder Größen verwendet werden, um Informationen darzustellen. Wir werden jedoch auch lernen, wann Legenden möglicherweise nicht die beste Wahl sind und wie wir Alternativen einsetzen können, um unsere Visualisierungen besser zu gestalten.

Legenden in matplotlib

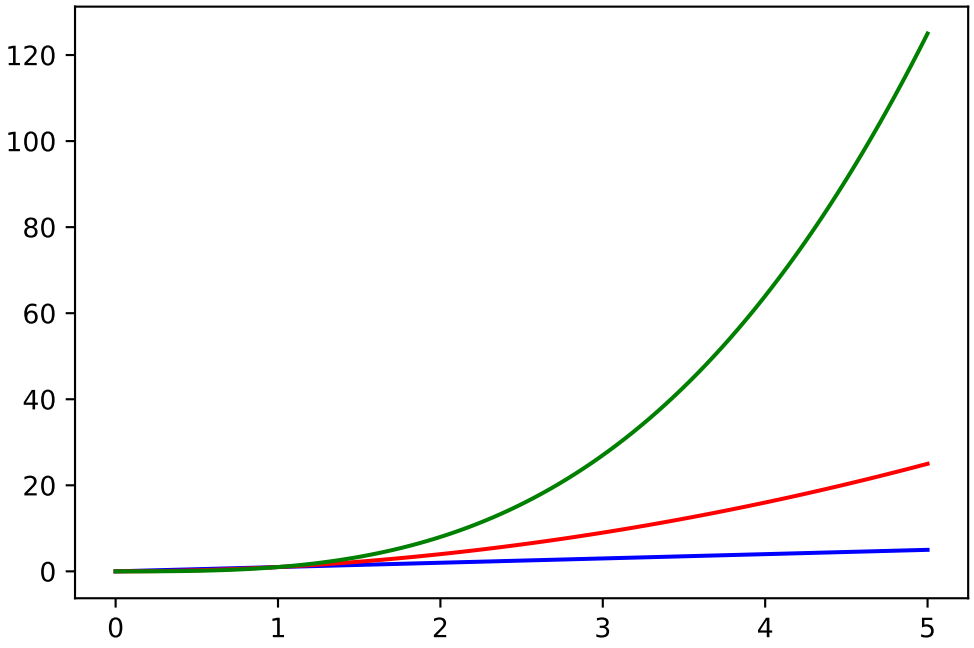
Eine Legende in matplotlib wird typischerweise durch die Funktion `ax.legend()` oder `plt.legend()` erstellt. Bevor wir die Legende hinzufügen, müssen wir jedoch Elemente im Plot erzeugen, die in der Legende erscheinen sollen. Dazu verwenden wir das `label`-Argument bei den entsprechenden Plotting-Funktionen.

Beginnen wir mit einem einfachen Beispiel: Wir erzeugen Linien, die verschiedene mathematische Funktionen darstellen und fügen dann eine Legende hinzu.

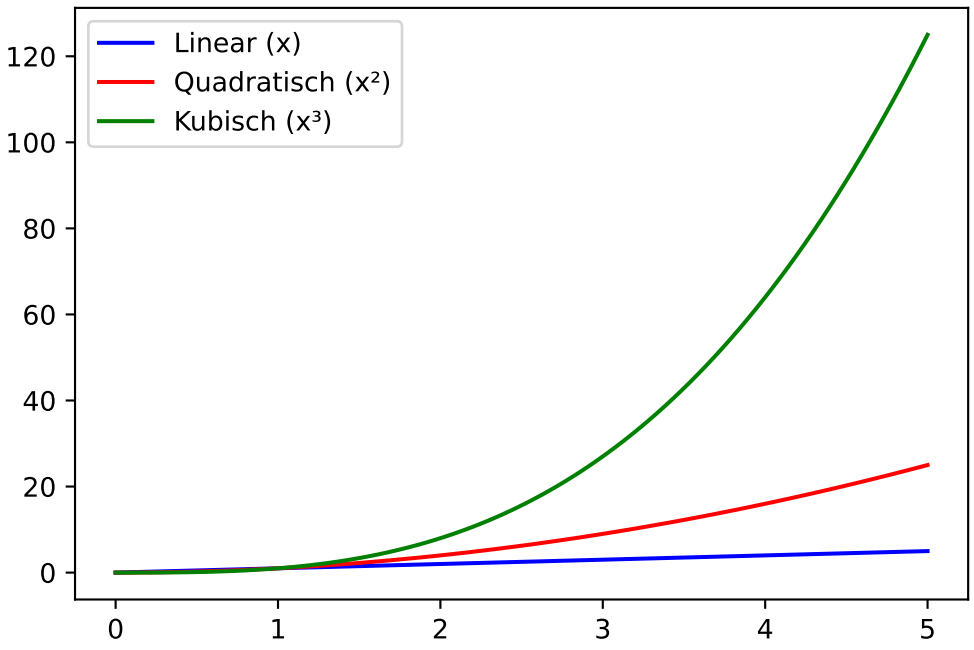
```
# Beispieldaten
x = np.linspace(0, 5, 100)
y_linear = x
y_squared = x**2
y_cubed = x**3
```

Hier eine direkt Gegenüberstellung was nötig ist, um eine sinnvolle Legende zu erhalten:

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

# Linien plotten mit Labels für die Legende
ax.plot(x, y_linear, color='blue')
ax.plot(x, y_squared, color='red')
ax.plot(x, y_cubed, color='green')




plt.show()
```




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

# Linien plotten mit Labels für die Legende
ax.plot(x, y_linear, color='blue', label='Linear (x)')
ax.plot(x, y_squared, color='red', label='Quadratisch (x²)')
ax.plot(x, y_cubed, color='green', label='Kubisch (x³)')

# Legende hinzufügen
ax.legend()

plt.show()
```



Man erkennt also wie wir die Legende quasi vorbereiten müssen, indem wir den Plots Labels mitgeben. Diese Labels werden dann in der Legende angezeigt.

Anpassung der Legende

Die Standardlegende ist funktional, aber möglicherweise nicht immer optimal positioniert oder gestaltet. Glücklicherweise bietet `ax.legend()` viele Parameter, um das Aussehen und die Position der Legende anzupassen.

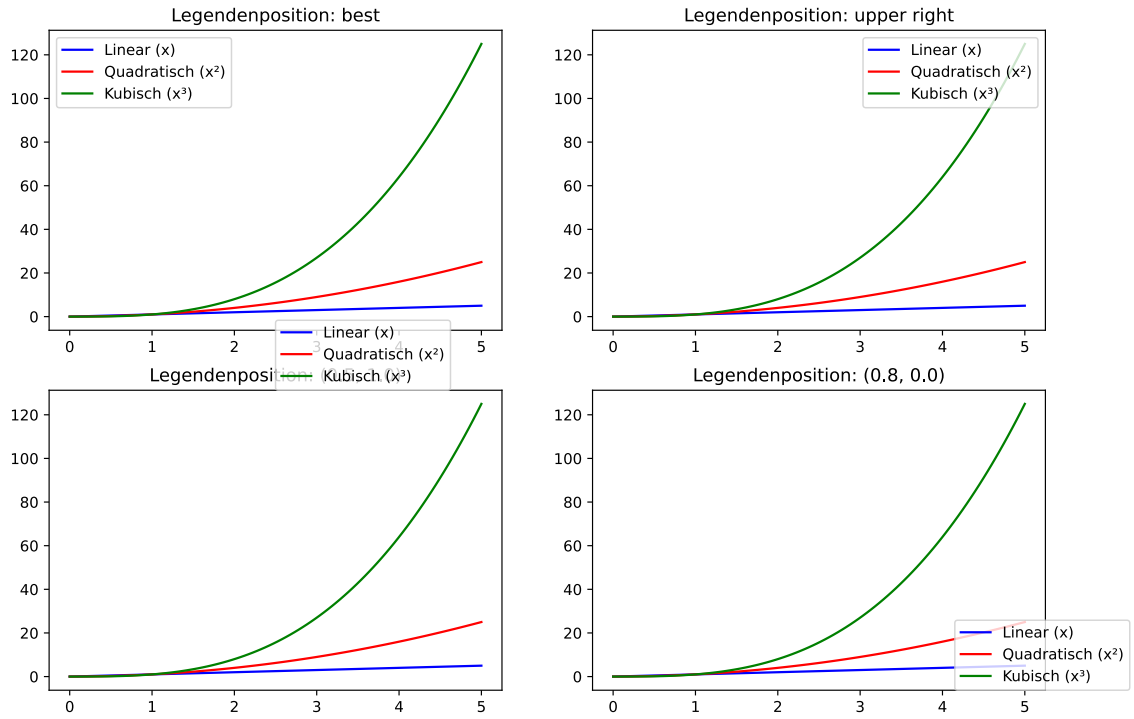
Position der Legende

Die Position der Legende kann über das Argument `loc` gesteuert werden. Dabei kann entweder einer dieser vorgegebene Strings genutzt werden:

- `'best'`
- `'upper left'`, `'upper center'`,`'upper right'`,
- `'center left'`, `'center'`, `'center right'`,
- `'lower left'`, `'lower center'`, `'lower right'`

oder aber eine relative Position in Form eines Tupels `(x, y)` angegeben werden. Hierbei ist `x` und `y` der relative Abstand von der linken unteren Ecke des Plots (0,0) bis zur rechten oberen Ecke (1,1). Die Option `'best'` ist standardmäßig ausgewählt und versucht, die Legende an einer Stelle zu platzieren, wo sie am wenigsten mit den Daten im Plot überlappt.

```
fig, axs = plt.subplots(2, 2, figsize=(12, 8))
axs = axs.flatten() # siehe Info Box unten

# Verschiedene Legendenpositionen demonstrieren
legend_locations = ['best', 'upper right', (0.5, 1.0), (0.8, 0.0)]
location_names = ['best', 'upper right', '(0.5, 1.0)', '(0.8, 0.0)']

for i in range(len(legend_locations)):
    # Linien plotten
    axs[i].plot(x, y_linear, color='blue', label='Linear (x)')
    axs[i].plot(x, y_squared, color='red', label='Quadratisch (x²)')
    axs[i].plot(x, y_cubed, color='green', label='Kubisch (x³)')

    # Legende mit unterschiedlicher Position
    axs[i].legend(loc=legend_locations[i])

    # Titel
    axs[i].set_title(f'Legendenposition: {location_names[i]}')

plt.show()
```



axs.flatten()

Der Befehl `axs.flatten()` wandelt ein mehrdimensionales Array von Achsenobjekten in ein eindimensionales Array um. In unserem Beispiel erzeugt `plt.subplots(2, 2)` ein 2×2-Array von Achsen (also ein 2D-Array mit der Form `(2,2)`). Mit `axs.flatten()` wird dieses in ein 1D-Array mit 4 Elementen umgewandelt, sodass wir mit einer einfachen Schleife von 0 bis 3 durch alle Achsen iterieren können. Ohne das Abflachen müssten wir verschachtelte Schleifen verwenden, um auf jede Achse zuzugreifen (z.B. `axs[0,0]`, `axs[0,1]`, `axs[1,0]`, `axs[1,1]`).

```
fig, axs = plt.subplots(2, 2)

axs.shape
```

```
(2, 2)
```

```
fig, axs = plt.subplots(2, 2)
axs = axs.flatten()
axs.shape
```

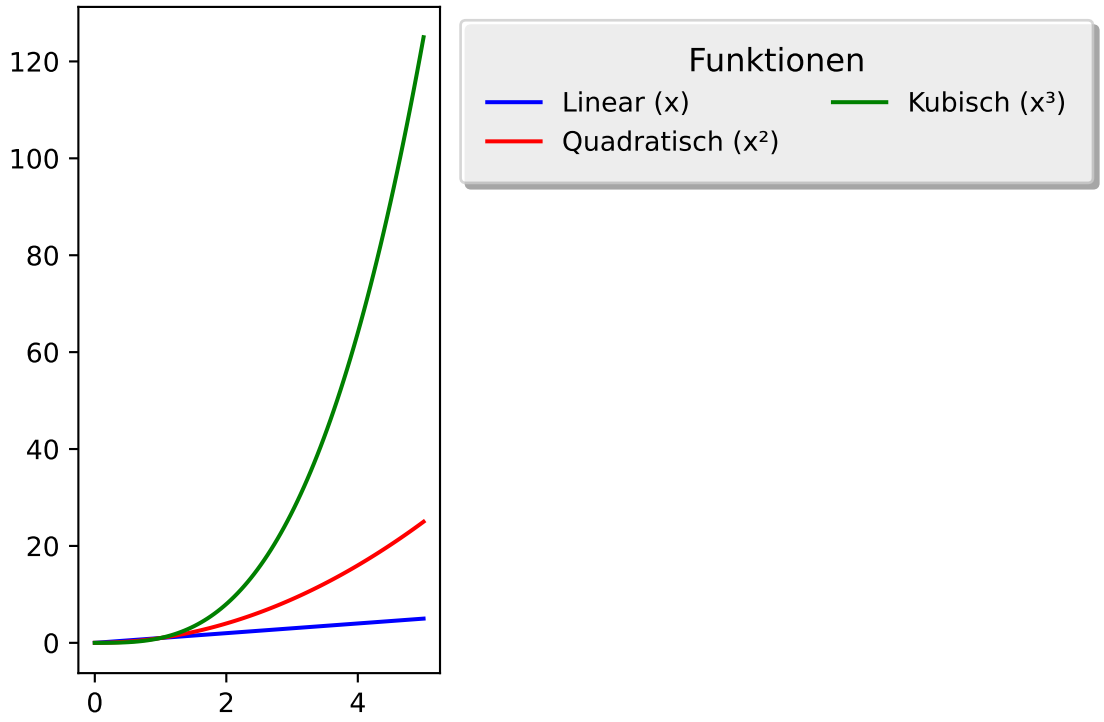
```
(4,)
```

Weitere Anpassungen der Legende

Neben der Position können wir viele weitere Aspekte der Legende anpassen:

```
fig, ax = plt.subplots(layout="tight")
```

```
# Linien plotten
ax.plot(x, y_linear, color='blue', label='Linear (x)')
ax.plot(x, y_squared, color='red', label='Quadratisch (x²)')
ax.plot(x, y_cubed, color='green', label='Kubisch (x³)')

# Angepasste Legende
ax.legend(
    loc='upper left',               # Ankerpunkt der Legende
    bbox_to_anchor=(1.02, 1.0),     # Position relativ zum Plot
    title='Funktionen',             # Titel für die Legende
    frameon=True,                   # Rahmen um die Legende
    framealpha=0.8,                 # Transparenz des Rahmens
    shadow=True,                    # Schatten hinter der Legende
    fontsize=10,                    # Schriftgröße
    title_fontsize=12,              # Schriftgröße des Titels
    ncol=2,                         # Anzahl der Spalten
    borderpad=1                     # Padding zwischen Rahmen und Inhalt
)

plt.show()
```



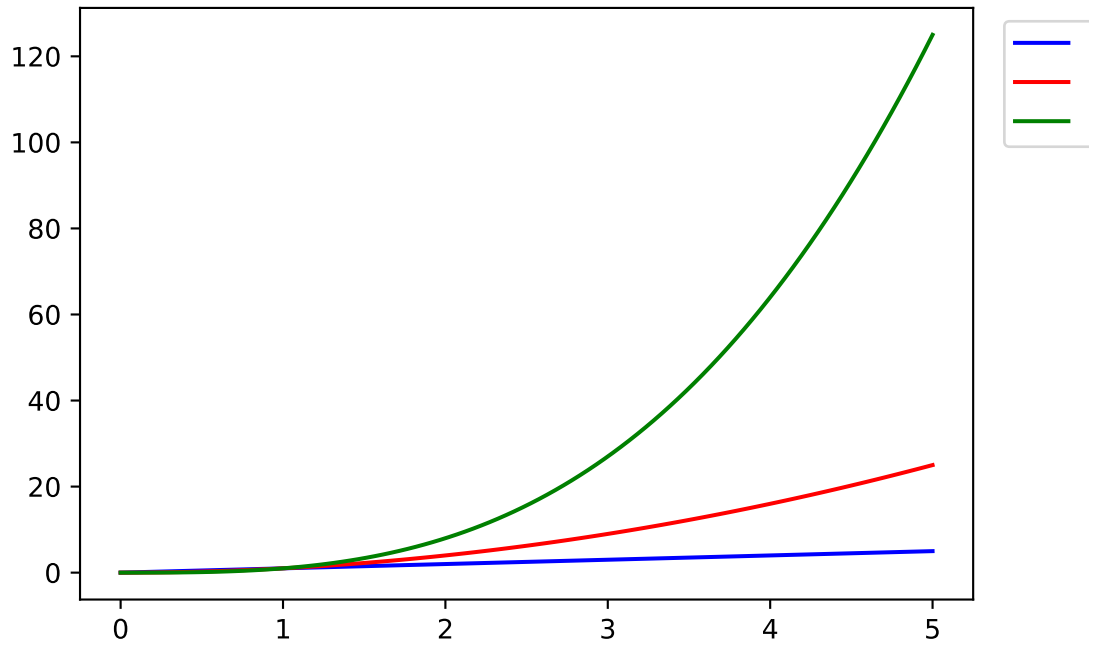
Ein besonders nützlicher Parameter ist `bbox_to_anchor`, der die Positionierung der Legende relativ zu einem bestimmten Punkt im Plot ermöglicht. Dies ist besonders hilfreich, wenn man die Legende außerhalb des eigentlichen Plotbereichs platzieren möchte, wie oben. Wir haben also zwar `loc='upper left'`, doch der Punkt, auf den sich das bezieht, ist nicht die obere linke Ecke des Plots, sondern nun `bbox_to_anchor=(1.02, 1.0)`, also ein Punkt leicht rechts außerhalb des Plots. Vereinfacht gesagt:

- `loc` bestimmt, welcher Punkt der Legende als Ankerpunkt dient (z.B. ‘upper left’ = die obere linke Ecke der Legende)
- `bbox_to_anchor` gibt die Position dieses Ankerpunkts relativ zu den Achsen an

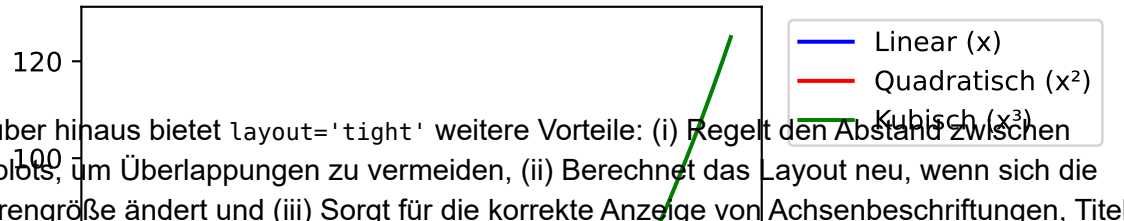
layout='tight'

Spätestens beim Platzieren der Legende außerhalb des Plots kann man sehen warum `layout='tight'` verwendet wird: Es sorgt beispielsweise dafür, dass alle Elemente innerhalb der Grenzen des letztendlich abgebildeten Plots bleiben:

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

# Linien plotten
ax.plot(x, y_linear, color='blue', label='Linear (x)')
ax.plot(x, y_squared, color='red', label='Quadratisch (x²)')
ax.plot(x, y_cubed, color='green', label='Kubisch (x³)')

# Angepasste Legende
ax.legend(loc='upper left', bbox_to_anchor=(1.02, 1.0))

plt.show()
```




```
fig, ax = plt.subplots(layout="tight")

# Linien plotten
ax.plot(x, y_linear, color='blue', label='Linear (x)')
ax.plot(x, y_squared, color='red', label='Quadratisch (x²)')
ax.plot(x, y_cubed, color='green', label='Kubisch (x³)')

# Angepasste Legende
ax.legend(loc='upper left', bbox_to_anchor=(1.02, 1.0))

plt.show()
```

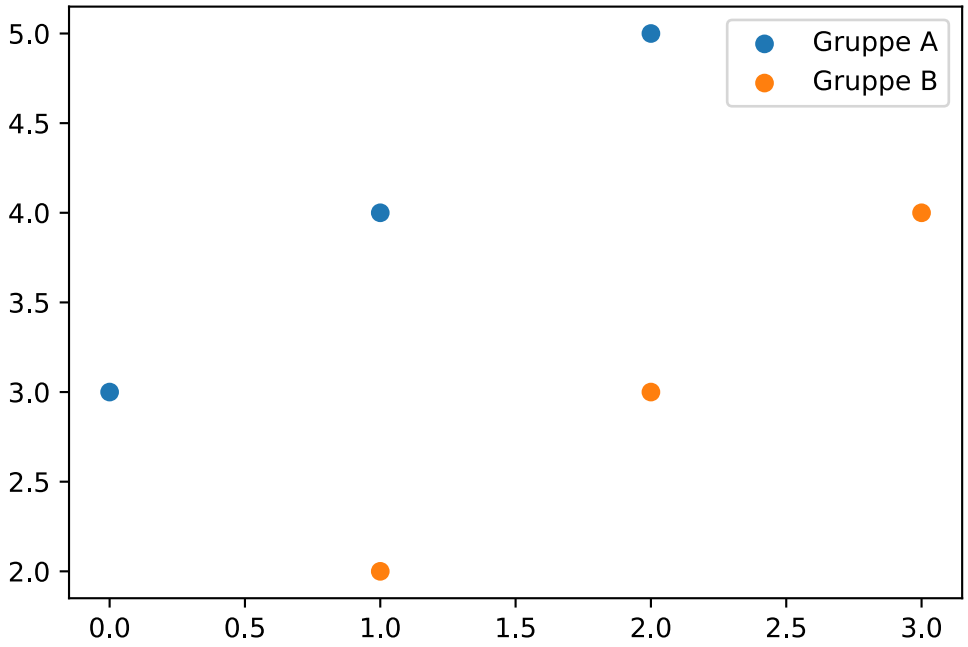


Darüber hinaus bietet `layout='tight'` weitere Vorteile: (i) Regelt den Abstand zwischen Subplots, um Überlappungen zu vermeiden, (ii) Berechnet das Layout neu, wenn sich die Figurengröße ändert und (iii) Sorgt für die korrekte Anzeige von Achsenbeschriftungen, Titeln

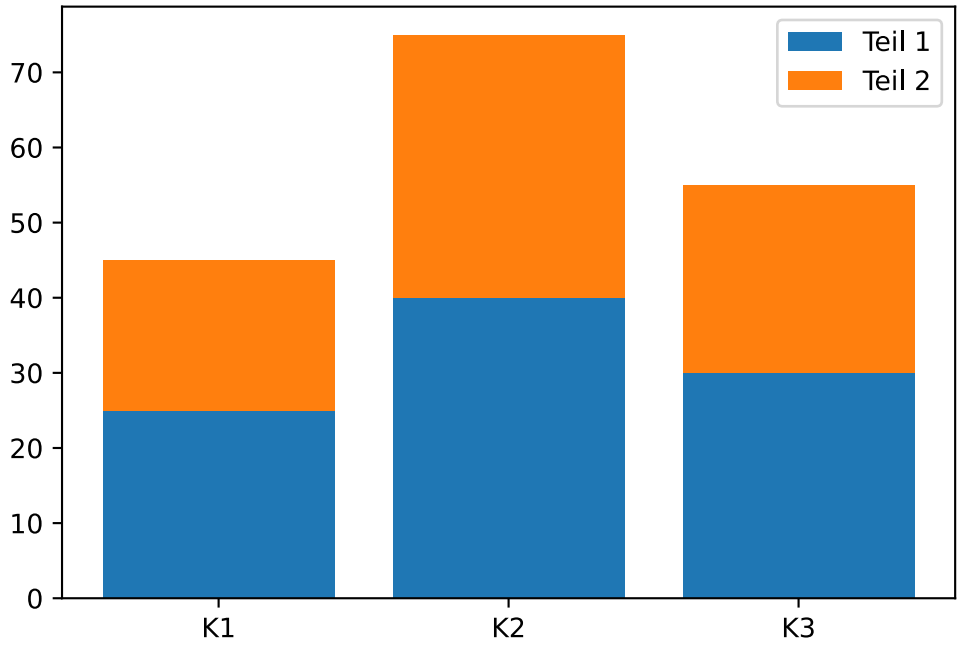
Legende für verschiedene Arten von Plots

Natürlich funktioniert die Legende nicht nur für Linienplots, sondern auch für andere Diagrammtypen wie beispielsweise Balkendiagramme oder Scatterplots:

```
fig, ax = plt.subplots()
ax.scatter([0, 1, 2], [3, 4, 5], label='Gruppe A')
ax.scatter([1, 2, 3], [2, 3, 4], label='Gruppe B')
ax.legend()
plt.show()
```

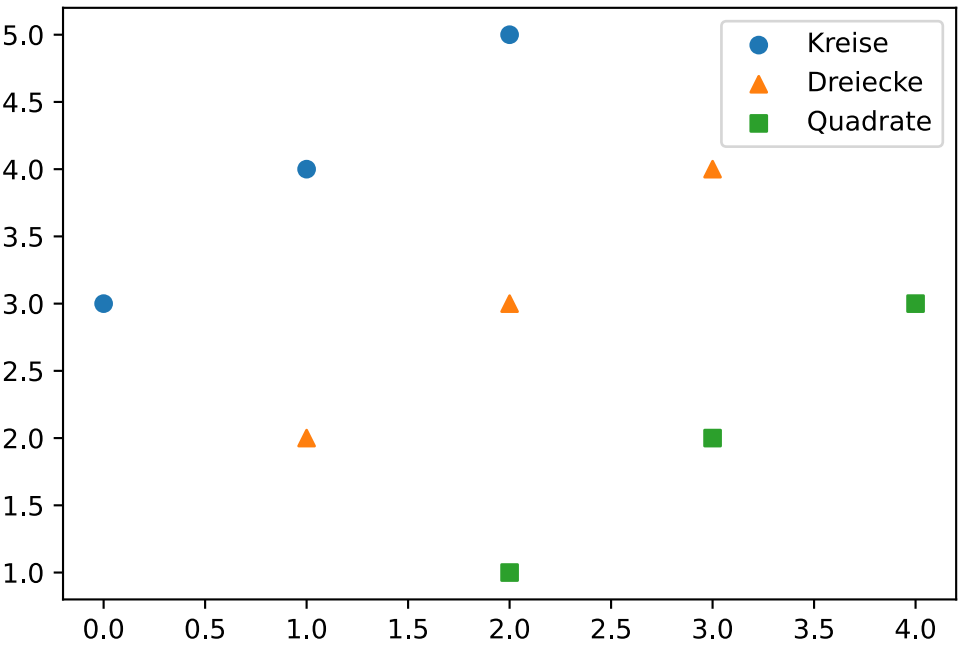



```
fig, ax = plt.subplots()
ax.bar(['K1', 'K2', 'K3'], [25, 40, 30], label='Teil 1')
ax.bar(['K1', 'K2', 'K3'], [20, 35, 25], bottom=[25, 40, 30], label='Teil 2')
ax.legend()
plt.show()
```

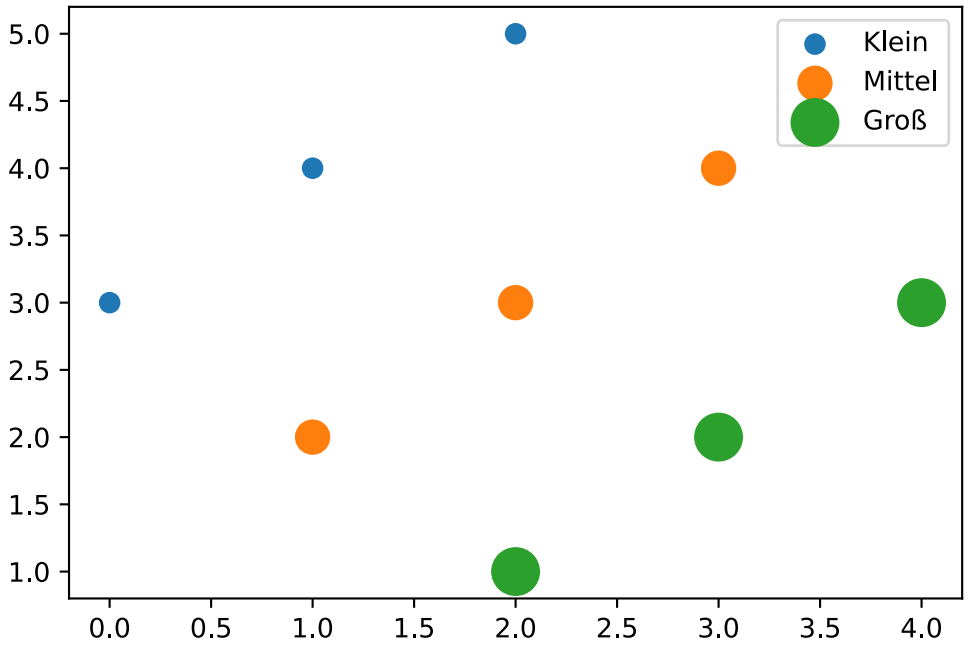


Ebenso können die Legenden auch andere Informationen als nur die Farbe enthalten. Beim Scatter Plot können wir beispielsweise auch verschiedene Marker-Formen oder Punktgrößen verwenden:

```
fig, ax = plt.subplots()
ax.scatter([0, 1, 2], [3, 4, 5], marker='o', label='Kreise')
ax.scatter([1, 2, 3], [2, 3, 4], marker='^', label='Dreiecke')
ax.scatter([2, 3, 4], [1, 2, 3], marker='s', label='Quadrate')
ax.legend()
plt.show()
```

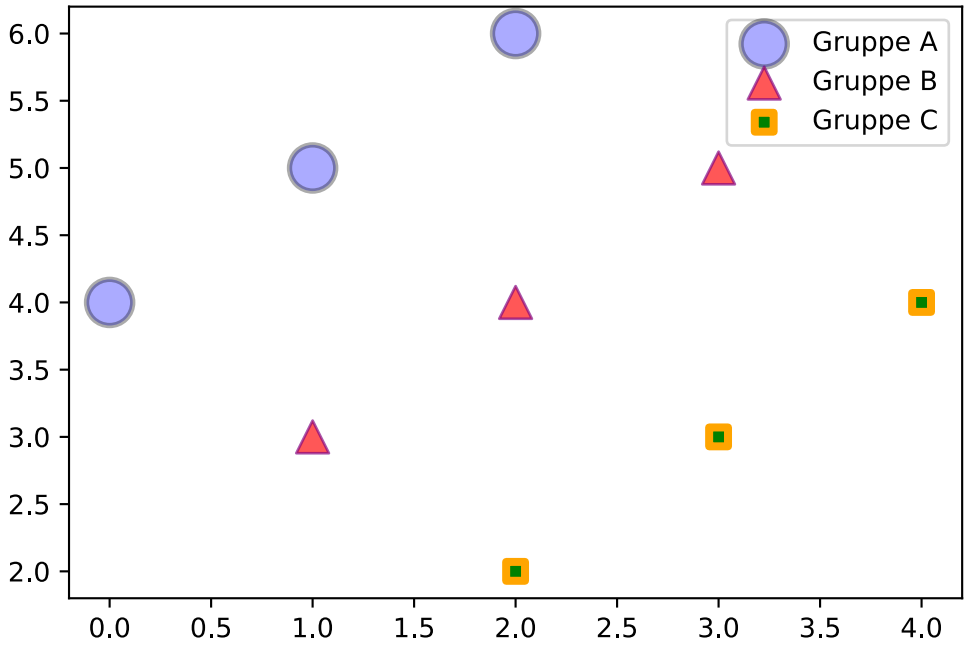


```
fig, ax = plt.subplots()
ax.scatter([0, 1, 2], [3, 4, 5], s=50,  label='Klein')
ax.scatter([1, 2, 3], [2, 3, 4], s=150, label='Mittel')
ax.scatter([2, 3, 4], [1, 2, 3], s=300, label='Groß')
ax.legend()
plt.show()
```



Und auch die Kombinationen sind ohne Probleme möglich:

```
fig, ax = plt.subplots()
ax.scatter([0, 1, 2], [4, 5, 6], label='Gruppe A',
           color='blue', marker='o', s=300, alpha=0.33, edgecolor='black', linewidth=2)
ax.scatter([1, 2, 3], [3, 4, 5], label='Gruppe B',
           color='red', marker='^', s=150, alpha=0.66, edgecolor='purple', linewidth=1)
ax.scatter([2, 3, 4], [2, 3, 4], label='Gruppe C',
           color='green', marker='s', s=50, alpha=1, edgecolor='orange', linewidth=3)
ax.legend()
plt.show()
```

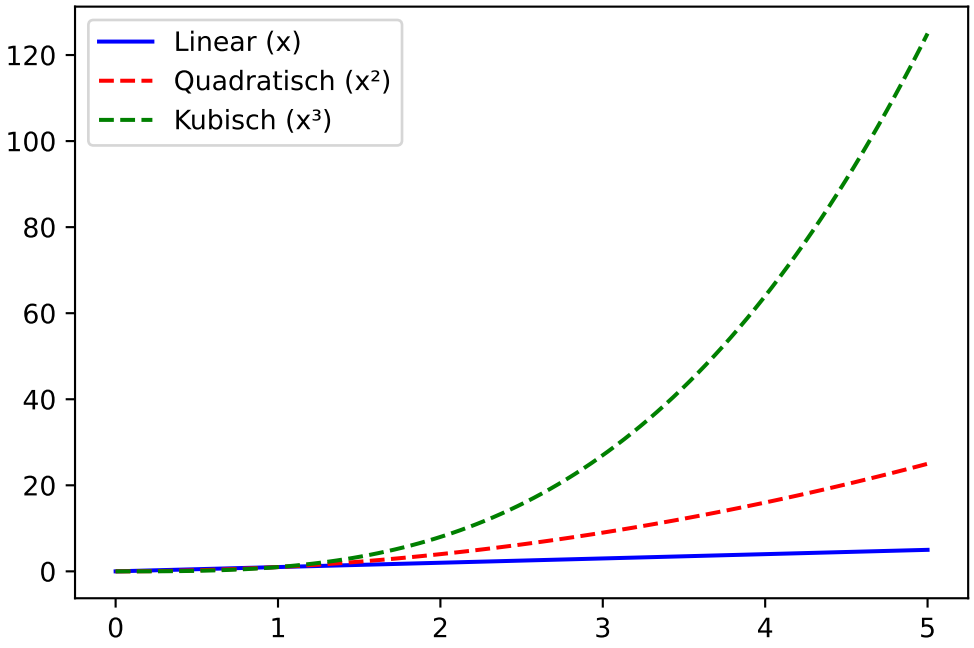


Legende manuell erzeugen

Falls die Legende anders aussehen soll, als matplotlib es standardmäßig gestaltet, müssen wir sie manuell erstellen. Das ist durchaus aufwändiger als nur `ax.legend()` zu verwenden, nachdem den einzelnen Elementen Labels zugewiesen wurden, bietet aber deutlich mehr Flexibilität, um die Legende genau nach unseren Vorstellungen zu gestalten.

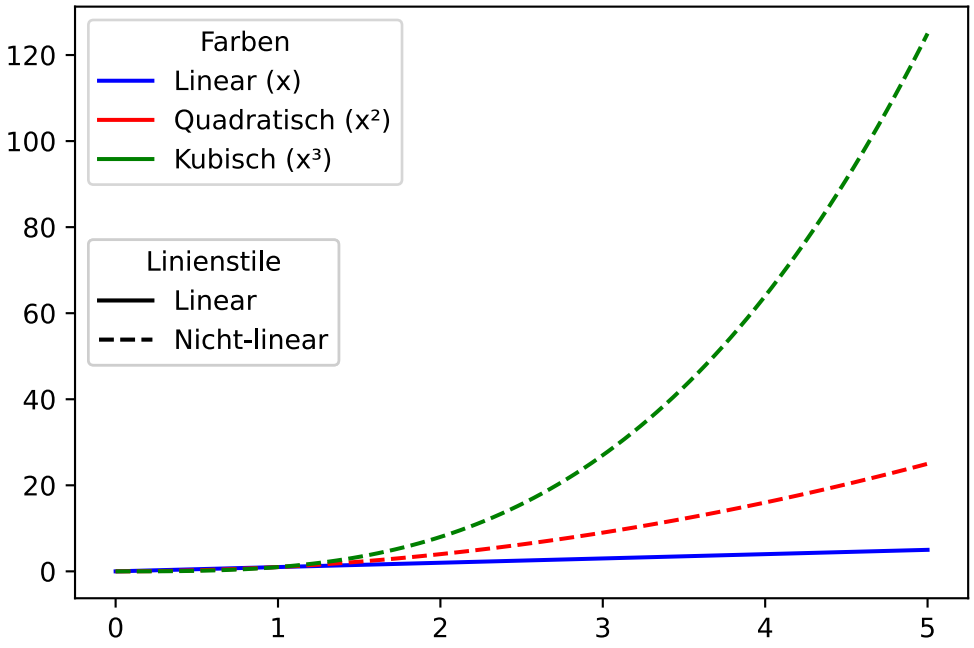
Nehmen wir nochmals das Liniendiagramm der drei Funktionen zur Hand und kennzeichnen nun nicht nur die Linienfarbe, sondern auch durch den Linienstil, ob es sich um eine lineare Funktion handelt. Die quadratische und kubische Funktion sollen also gestrichelt gezeichnet werden. Hier zunächst mit dem einfachen automatischen Ansatz:

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

# Linien plotten mit Labels für die Legende
ax.plot(x, y_linear, color='blue', label='Linear (x)')
ax.plot(x, y_squared, color='red', linestyle='--', label='Quadratisch (x²)')
ax.plot(x, y_cubed, color='green', linestyle='--', label='Kubisch (x³)')

# Legende hinzufügen
ax.legend()

plt.show()
```



Es funktioniert also wie erwartet, allerdings möchten wir die Legende hier nun so gestalten, dass die Farbinfo und die Linienstilinfo getrennt dargestellt werden. Also eine Legende, die nur erklärt was die Linienfarbe bedeutet und eine zweite Legende, die nur erklärt was die Linienstile bedeuten. Das ist mit dem automatischen Ansatz nicht möglich, also müssen wir die Legende manuell erstellen:

```
from matplotlib.lines import Line2D

fig, ax = plt.subplots()

ax.plot(x, y_linear, color='blue')
ax.plot(x, y_squared, color='red', linestyle='--')
ax.plot(x, y_cubed, color='green', linestyle='--')

# Erste Legende für Farben
legend_color = [
    Line2D([0], [0], color='blue', label='Linear (x)'),
    Line2D([0], [0], color='red', label='Quadratisch (x²)'),
    Line2D([0], [0], color='green', label='Kubisch (x³)')
]
legend1 = ax.legend(handles=legend_color, loc='upper left', title='Farben')
ax.add_artist(legend1)

# Zweite Legende für Linienstile
legend_linestyle = [
    Line2D([0], [0], color='black', linestyle='-', label='Linear'),
    Line2D([0], [0], color='black', linestyle='--', label='Nicht-linear')
]
legend2 = ax.legend(handles=legend_linestyle, loc='center left', title='Linienstile')
ax.add_artist(legend2)

plt.show()
```



Folgendes ist dabei also neu:

Wir müssen erstmals via `from matplotlib.lines import Line2D` die Klasse `Line2D` importieren. Diese Klasse wird hier erstmals verwendet, weil wir “künstliche” Linienelemente für die Legende erzeugen müssen. Diese Elemente existieren nicht im eigentlichen Plot, sondern dienen ausschließlich als visuelle Referenz in der Legende. Mit `Line2D` können wir genau definieren, wie diese Legendeneinträge aussehen sollen, unabhängig von den tatsächlich gezeichneten Plotlinien.

Der grundlegende Ablauf ist wie folgt: Wir erstellen zunächst den eigentlichen Plot mit `ax.plot()`, aber ohne automatische Labels. Dann definieren wir für jede Legende eine Liste von `Line2D`-Objekten, die die gewünschten visuellen Eigenschaften haben: Für die Farblegende: Verschiedene Farben, aber einheitliche Linienstile. Für die Linienstil-Legende: Einheitliche Farbe (schwarz), aber unterschiedliche Linienstile Jedes `Line2D`-Objekt bekommt ein Label, das in der Legende angezeigt wird. Die `handles`-Liste wird dann an `ax.legend()` übergeben, um die Legende zu erstellen. Im Grunde hat dieser Ansatz also Ähnlichkeit mit dem automatischen Ansatz von vorher: Wir geben jedem Element ein Label und übergeben diese dann an `ax.legend()`. Der Unterschied ist, dass wir hier aber separate Elemente, die ja eigentlich nicht im Plot auftauchen, selbst definieren. Das heißt selbstverständlich auch, dass wir in unserer manuellen Legende genau die Linienfarben und -stile definieren müssen, die aufgrund der vorangegangenen `ax.plot()`-Befehle im Plot verwendet wurden - ansonsten passt ja die Legende nicht zum Plot.

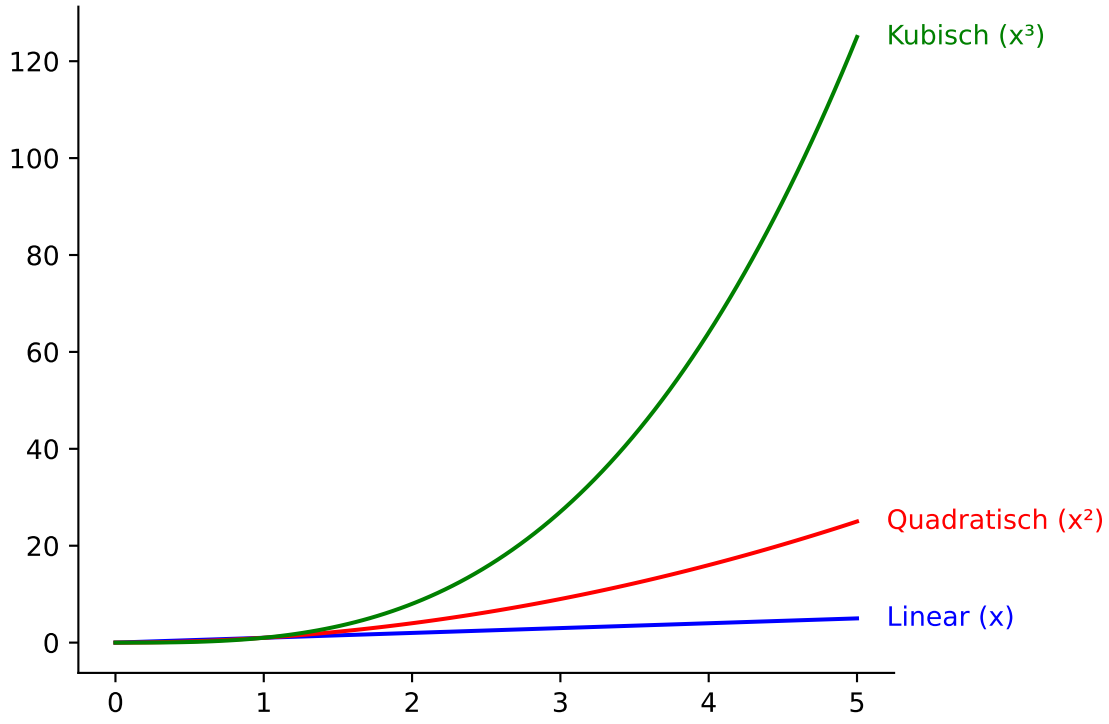
Warum `ax.add_artist()`? Der Befehl `ax.add_artist(legend1)` ist entscheidend, weil standardmäßig jeder Aufruf von `ax.legend()` die vorherige Legende überschreibt. Mit `add_artist()` teilen wir matplotlib mit, dass die erste Legende als eigenständiges Grafikelement behandelt und beibehalten werden soll, wenn die zweite Legende hinzugefügt wird. Das bedeutet übrigens, dass der zweite Aufruf von `ax.add_artist(legend2)` nicht unbedingt notwendig ist (da wir keine dritte Legende hinzufügen und der Aufruf `ax.legend(handles=legend_linestyle ...)` reichen würde um die zweite Legende hinzuzufügen). Allerdings ist der Code so klarer und außerdem bereit für z.B. eine zukünftige dritte Legende.

Alternativen zu Legenden

Direkte Beschriftung

Obwohl Legenden nützlich sein können, führen sie oft dazu, dass die Leser ihre Augen zwischen dem Plot und der Legende hin und her bewegen müssen, was die Interpretation erschweren kann. Eine häufig bessere Alternative ist die **direkte Beschriftung**, bei der wir die Bezeichnungen direkt neben die relevanten Elemente im Plot setzen. Hier direkt das Beispiel:

```
fig, ax = plt.subplots(layout='tight')

line1, = ax.plot(x, y_linear, color='blue')
line2, = ax.plot(x, y_squared, color='red')
line3, = ax.plot(x, y_cubed, color='green')

# Direkte Beschriftungen am Ende der Linien
x_pos = 5.2  # Position entlang der x-Achse für die Beschriftungen
ax.text(x_pos, y_linear[-1], 'Linear (x)', color='blue', va='center')
ax.text(x_pos, y_squared[-1], 'Quadratisch (x²)', color='red', va='center')
ax.text(x_pos, y_cubed[-1], 'Kubisch (x³)', color='green', va='center')

ax.spines[["top", "right"]].set_visible(False)

plt.show()
```



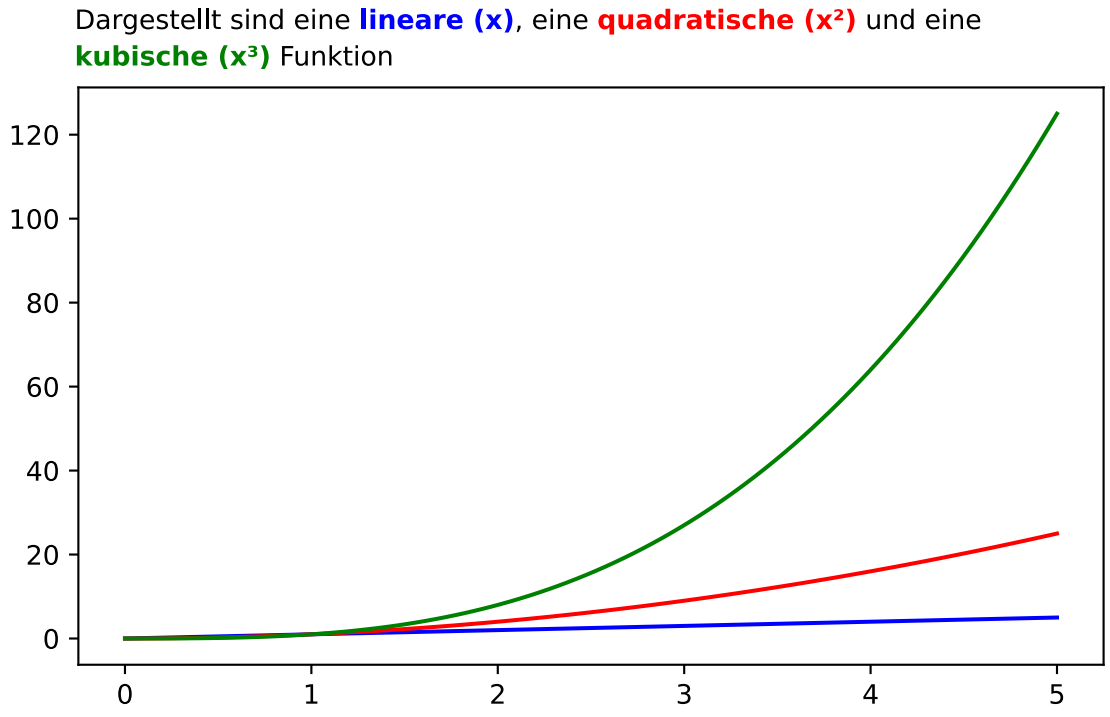
Hier haben wir die Beschriftungen direkt neben die entsprechenden Linien am Ende des Plots gesetzt. Die Farbe der Beschriftung stimmt mit der jeweiligen Linie überein, was die Zuordnung noch intuitiver macht. So können direkte Beschriftungen die kognitive Belastung reduzieren, da die Lesenden nicht mit den Augen zwischen Plot und Legende wechseln müssen. Sie können auch mehr Kontext bieten als eine einfache Legende.

Formatierter Text

Eine weitere Alternative zu Legendenist die Verwendung von formatiertem Text z.B. im Titel oder Untertitel des Plots. Vor allem wenn sowieso etwas Fließtext zur Erläuterung nötig ist, kann z.B. wie folgt bei der Nennung der daragestellten Elemente direkt die Farbe hervorgehoben werden.

Wie man sieht nutzen wir hier ein neues Modul `highlight_text`. Dieses Beispiel ist aber nur kurz ohne weitere Erläuterungen vorgegeben, da wir dies in einem zukünftigen Kapitel vertiefen werden.

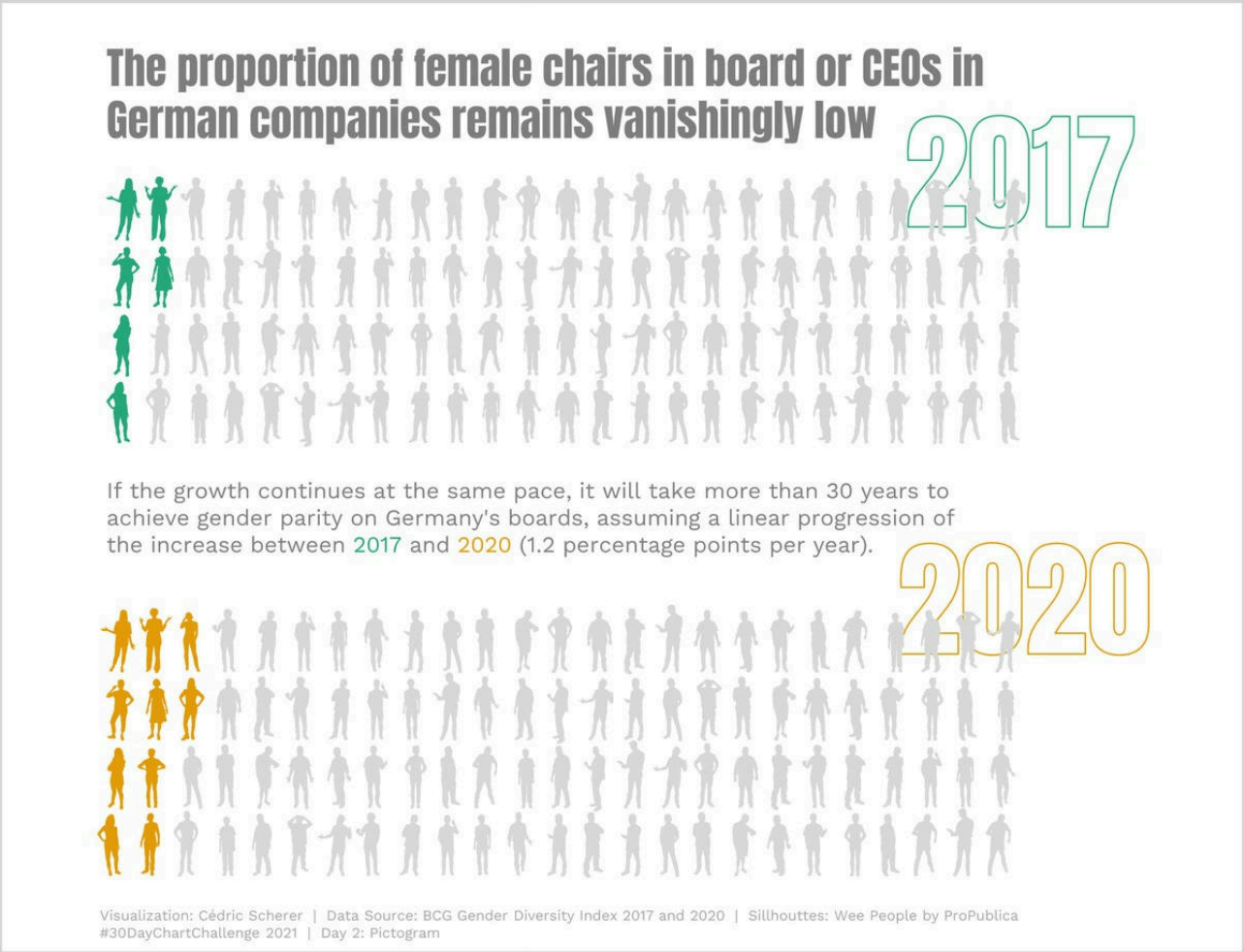
```
from highlight_text import ax_text

fig, ax = plt.subplots(layout='tight')

# Linien plotten ohne Labels
ax.plot(x, y_linear, color='blue')
ax.plot(x, y_squared, color='red')
ax.plot(x, y_cubed, color='green')

# Titel mit farbigen Hervorhebungen
title_text = "Dargestellt sind eine <lineare (x)>, eine <quadratische (x²)> und
eine\n<kubische (x³)> Funktion"
ax_text(
    -0.27, 150,
    title_text,
    fontsize=10,
    transform=ax.transAxes,
    highlight_textprops=[
        {"color": 'blue', "fontweight": 'bold'},
        {"color": 'red', "fontweight": 'bold'},
        {"color": 'green', "fontweight": 'bold'}
    ]
);

plt.show()
```



Fazit

Dieses Kapitel behandelte verschiedene Methoden zur Kennzeichnung von Visualisierungselementen. Legenden mit `ax.legend()` bieten einen strukturierten Ansatz, besonders nach Verwendung des `label`-Attributs. Mit Parametern wie z.B. `loc`, `bbox_to_anchor` und `title` lassen sich Legenden gezielt positionieren und anpassen. Für komplexere Anforderungen ermöglichen manuelle Legenden mit z.B. Line2D-Objekten maximale Kontrolle.

Als Alternative können direkte Beschriftungen oder farbiger Text die kognitive Belastung reduzieren, indem sie den Blickwechsel zwischen Plot und Legende vermeiden. Die optimale Wahl zwischen diesen Methoden hängt von der Datenkomplexität, dem verfügbaren Platz und den Bedürfnissen der Zielgruppe ab. In vielen Fällen ist eine Kombination verschiedener Ansätze die effektivste Lösung. Zum Schluss ein Beispiel, in dem sowohl eine direkte Beschriftung, als auch formatierter Text im Titel verwendet werden - aber eben keine Legende und das ist auch nicht nötig:



Quelle: Cédric Scherer

Weitere Ressourcen

- Matplotlib Legend Tutorial || matplotlib legend outside of graph || Matplotlib Tips
- Matplotlib Legend Guide -> Hier reicht es die Beispiele zu überfliegen!
- Post by Yan Holtz

Übungen

Übung 1

Erstelle einen Plot mit 9 Subplots, angeordnet in einer 3×3-Matrix. In jedem Subplot soll dieselbe Funktion y = sin(x) als rote Linie dargestellt werden und zwar für x-Werte 0 bis 2*Pi. Positioniere in jedem Subplot die Legende an einer unterschiedlichen Position, sodass die Legendenposition dem Ort des Subplots in der Matrix entspricht. Verwende dafür die Positionsparameter ‘upper left’, ‘upper center’, ‘upper right’, ‘center left’, ‘center’, ‘center right’, ‘lower left’, ‘lower center’ und ‘lower right’. Beispiel: Im Subplot oben links soll auch die Legende oben links sein.

- (A) Geschafft

Übung 2

Der Iris-Datensatz ist ein bekannter Datensatz in der Datenanalyse, der Messungen von Schwertlilien enthält.

a) Erstelle einen Scatterplot, der für jede der drei Arten (setosa, versicolor, virginica) mit unterschiedlichen Farben die Beziehung zwischen der Breite und Länge der Kelchblätter (sepal_width und sepal_length) darstellt. Füge eine passende Legende hinzu.

b) Erstelle denselben Plot noch einmal, aber entferne die Legende und verwende stattdessen die direkte Beschriftung. Überlege dir, wo die Beschriftungen am besten platziert werden können. Tipp: Für bessere Sichtbarkeit der Beschriftung, kann man weiße Hintergrundboxen um den Text einfügen, indem man folgendes Argument innerhalb `ax.text()` einfügt: `bbox=dict(facecolor='white', alpha=0.7, edgecolor='none', boxstyle='round,pad=0.3')`

```
# Iris-Datensatz laden
from sklearn.datasets import load_iris
import pandas as pd

iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
iris_df['species'] = [iris.target_names[i] for i in iris.target]
```

- (A) Geschafft

Übung 3

Nimm das Beispiel mit der manuellen Legende für Farben und Linienstile von oben und erstelle nicht wie dort zwei separate, sondern nur eine eine einzige Legende, die aber dieselben Informationen enthält - also die 3 Farben und die 2 Linientypen. Verwende dabei weiterhin Line2D, aber auch um Pseudo-Elemente wie einen Titel oder gar eine leere Zeile zu erzeugen.

Tipp: Du kannst mit `Line2D([0], [0], color='none', label=' ')` Leerzeilen in die Legende einfügen.

- (A) Geschafft

Hinweis: Lösungen zu den meisten Übungsaufgaben findet ihr, indem ihr ganz oben rechts im Kapitel auf den `</> Code` Button klickt und dann entsprechend nach ganz unten scrollt. Es ist Absicht, dass dies etwas umständlich ist.