Schriftarten und Schriftgrößen

by Woche 15

Einen entscheidenden Aspekt für professionelle Visualisierungen haben wir die ganze Zeit links liegen lassen: die **Anpassung von Schriftarten und -größen**.

Die Standardschrift von matplotlib - DejaVu Sans - ist zwar durchaus lesbar, wirkt aber oft generisch und wenig ansprechend. Besonders wenn Visualisierungen für Präsentationen, Berichte oder Publikationen bestimmt sind, kann die richtige Schrift einen enormen Unterschied für den professionellen Eindruck machen. Außerdem ist es gut möglich, dass es für eure Organisation oder Marke spezifische Schriftarten gibt, die verwendet werden sollten.

In diesem Kapitel lernen wir zunächst, wie wir **Schriftgrößen** für verschiedene Elemente unserer Plots anpassen können. Anschließend zeigen wir, wie wir mit dem `pyfonts`-Modul **eigene Schriftarten** einfach und reproduzierbar verwenden können. Abschließend behandeln wir **professionelle Annotationen mit Pfeilen** mithilfe dem `drawarrow`-Modul.

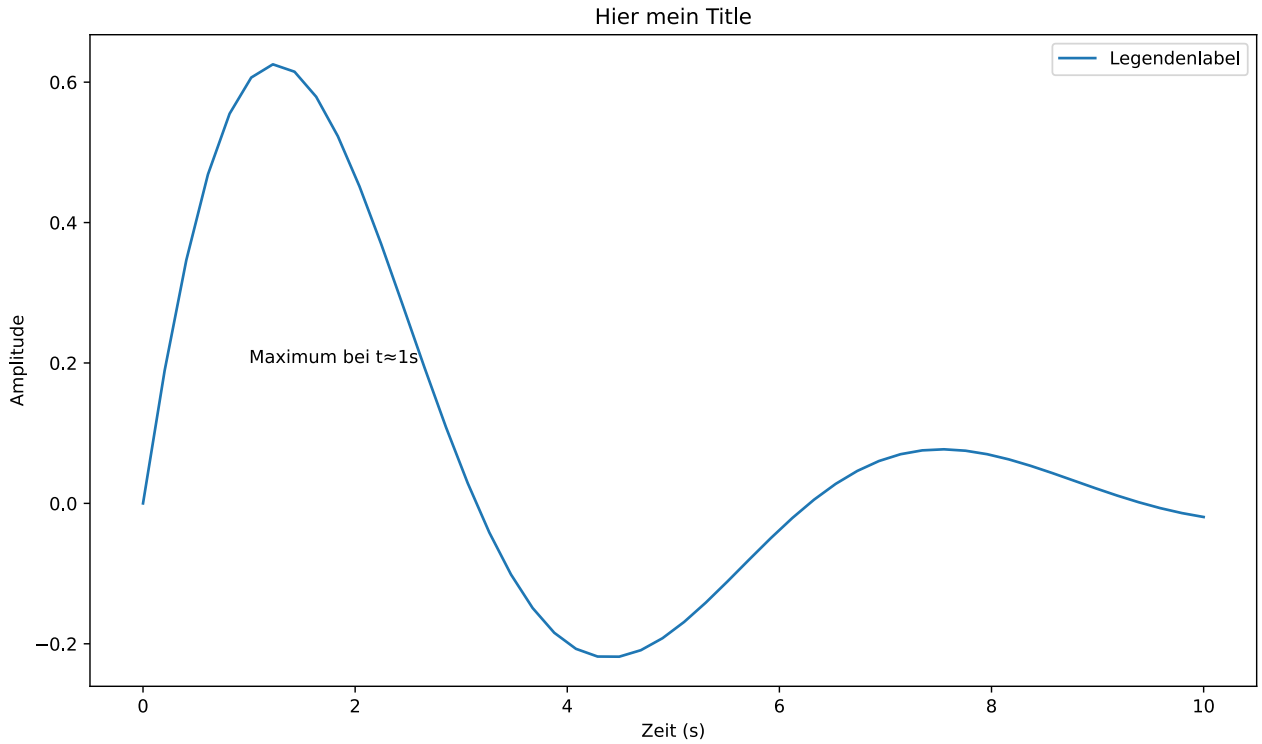
> ℹ Neue Pakete in diesem Kapitel
>
> Dieses Kapitel verwendet seit längerem mal wieder neue, externe Module, die ihr womöglich installieren müsst:
>
> - `pyfonts`: Für das einfache Laden von Schriftarten
> - `drawarrow`: Für professionelle Pfeil-Annotationen
> - `highlight_text`: Für erweiterte Textformatierung mit hervorgehobenen Wörtern

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from drawarrow import ax_arrow
from pyfonts import load_google_font
from highlight_text import ax_text
```

Schriftgrößen anpassen

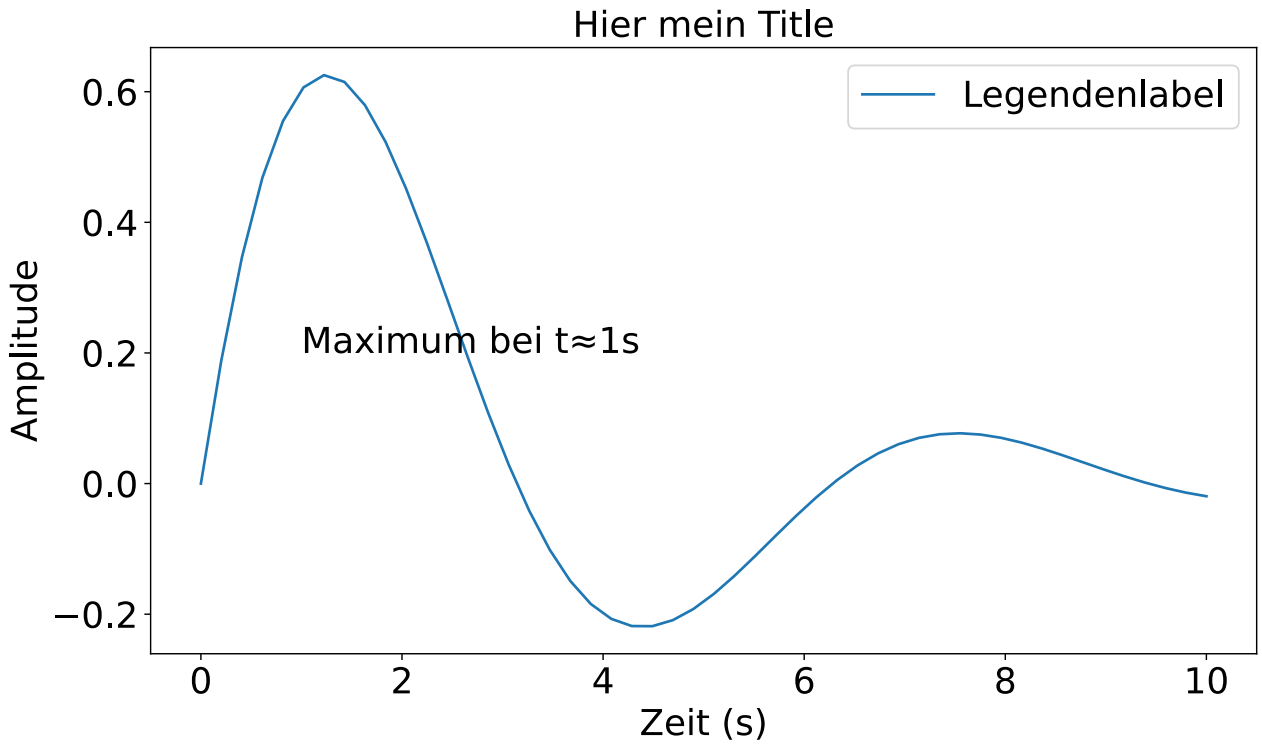
Matplotlib verwendet unterschiedliche Standardgrößen für Titel, Achsenbeschriftungen und andere Textelemente. Schauen wir uns zuerst an, wie ein Plot mit den Standardgrößen aussieht:

```
# Beispieldaten
x = np.linspace(0, 10, 50)
y = np.sin(x) * np.exp(-x/3)

# Plot mit Standardgrößen
fig, ax = plt.subplots(figsize=(10, 6), layout='tight')
ax.plot(x, y, label='Legendenlabel')
ax.set_title('Hier mein Title')
ax.set_xlabel('Zeit (s)')
ax.set_ylabel('Amplitude')
ax.legend()
ax.text(1, 0.2, 'Maximum bei t≈1s')
plt.show()
```



Jetzt passen wir die verschiedenen Schriftgrößen einzeln an - inklusive einer freien Text-Annotation direkt im Plot:

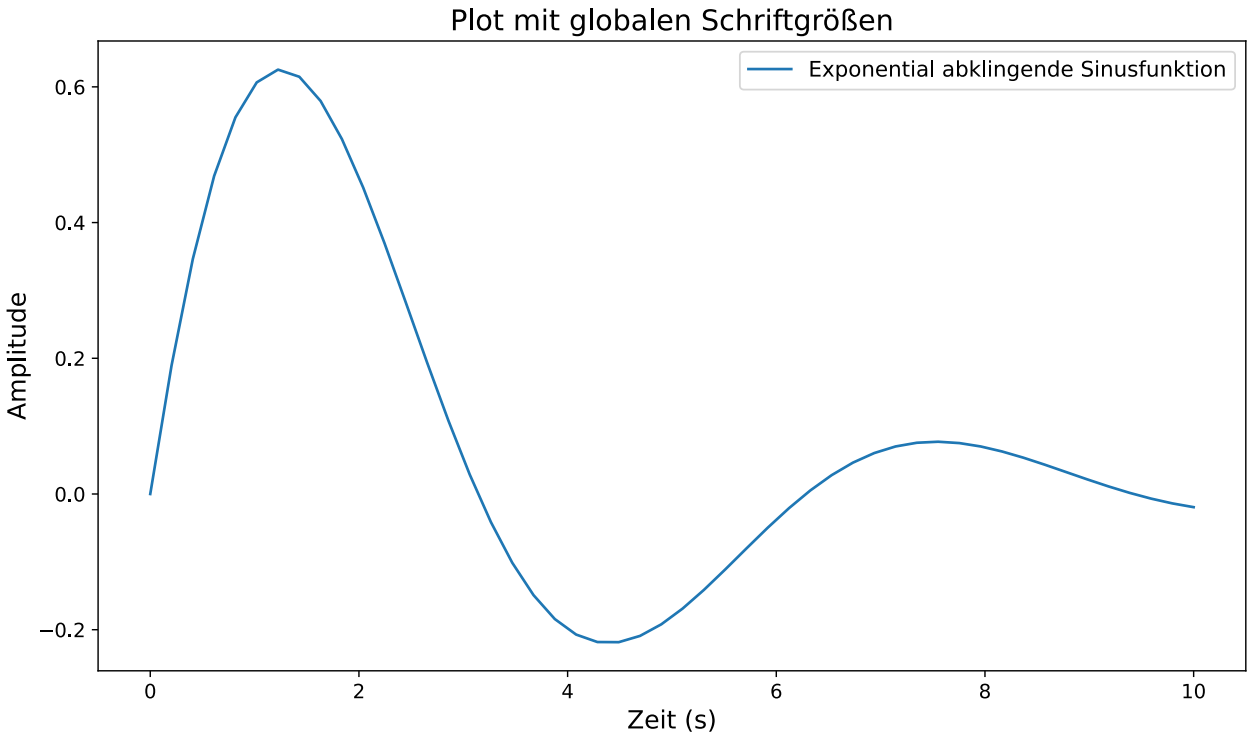
```
# Plot mit angepassten Schriftgrößen
fig, ax = plt.subplots(figsize=(10, 6), layout='tight')
ax.plot(x, y, label='Legendenlabel')
ax.set_title('Hier mein Title', fontsize=20)
ax.set_xlabel('Zeit (s)', fontsize=20)
ax.set_ylabel('Amplitude', fontsize=20)
ax.legend(fontsize=20)
ax.tick_params(axis='both', which='major', labelsize=20)
ax.text(1, 0.2, 'Maximum bei t≈1s', fontsize=20)
plt.show()
```



Globale Schriftgrößen mit rcParams

Für konsistente Schriftgrößen über mehrere Plots hinweg können wir auch die globalen matplotlib-Einstellungen anpassen:

```
# Globale Schriftgrößen setzen
plt.rcParams.update({
    'font.size': 12,          # Basis-Schriftgröße
    'axes.titlesize': 16,     # Titel-Größe
    'axes.labelsize': 14,     # Achsenbeschriftungen
    'legend.fontsize': 12,    # Legende
    'xtick.labelsize': 11,    # X-Achsen-Ticks
    'ytick.labelsize': 11     # Y-Achsen-Ticks
})

# Jetzt verwenden alle Plots diese Einstellungen
fig, ax = plt.subplots(figsize=(10, 6), layout='tight')
ax.plot(x, y, label='Exponential abklingende Sinusfunktion')
ax.set_title('Plot mit globalen Schriftgrößen')
ax.set_xlabel('Zeit (s)')
ax.set_ylabel('Amplitude')
ax.legend()
plt.show()

# Zurück zu den Standardwerten
plt.rcParams.update(plt.rcParamsDefault)
```



Diese Methode ist besonders praktisch, wenn man viele Plots mit einheitlichem Design erstellen möchte.

Eigene Schriftarten mit pyfonts

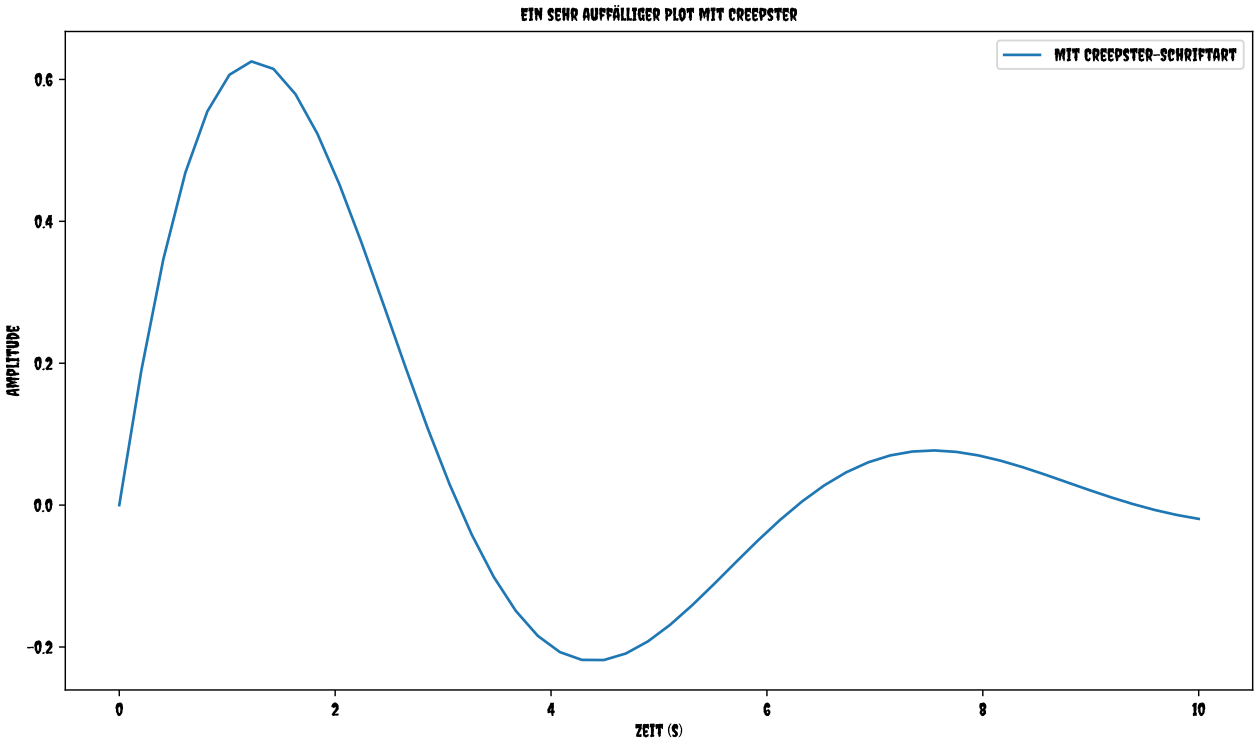
Während die Anpassung der Schriftgrößen bereits einen großen Unterschied macht, können wir mit eigenen Schriftarten noch professionellere und markantere Visualisierungen erstellen. Hier kommt das `pyfonts`-Modul ins Spiel.

Das Modul löst das größte Problem bei der Verwendung eigener Schriftarten in matplotlib: die **Reproduzierbarkeit**. Während man normalerweise Schriftdateien lokal installieren müsste (was bei der Weitergabe von Code zu Problemen führt), lädt pyfonts die Schriftarten automatisch aus dem Internet herunter.

Grundlegende Verwendung

Hier ist ein einfaches Beispiel, wie pyfonts funktioniert. Wir verwenden eine sehr auffällige Schriftart, um den Unterschied deutlich zu machen:

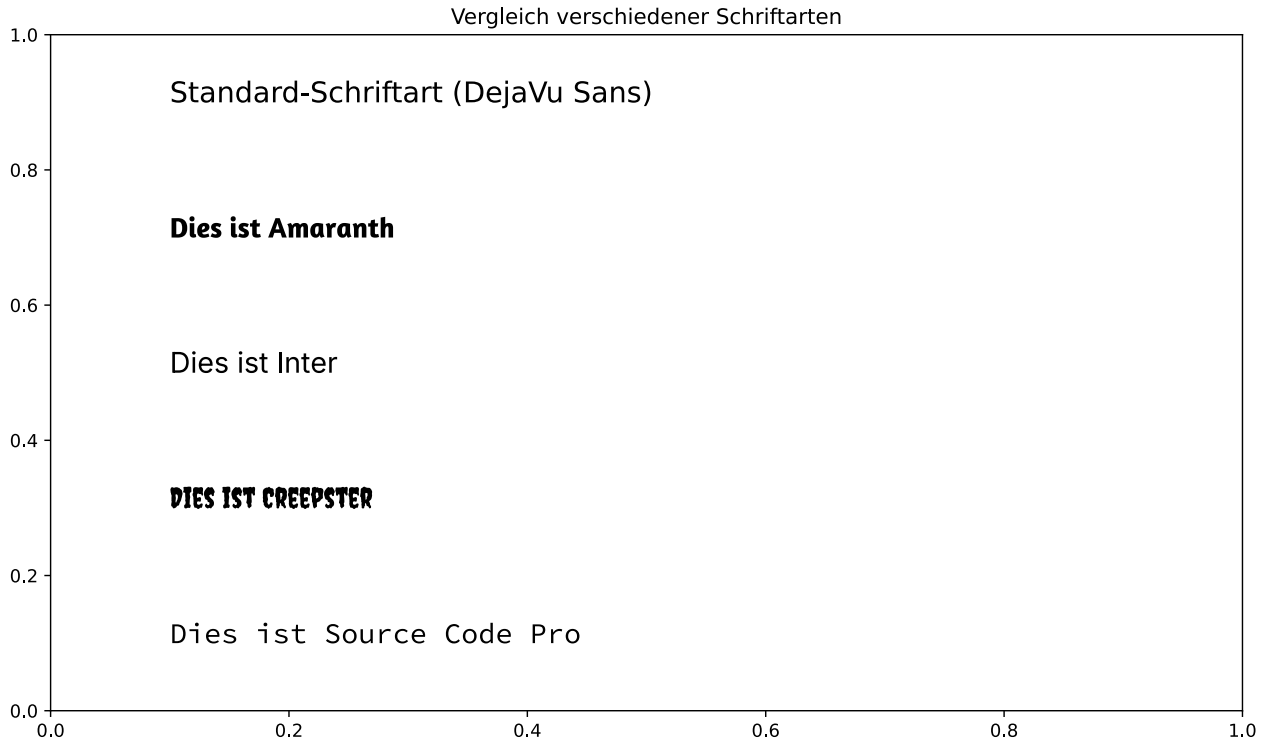
```
# Auffällige Schriftart direkt von Google Fonts laden
creepster_font = load_google_font("Creepster")

# Plot mit sehr auffälliger Schriftart
fig, ax = plt.subplots(figsize=(10, 6), layout='tight')
ax.plot(x, y, label='Mit Creepster-Schriftart')
ax.set_title('Ein sehr auffälliger Plot mit Creepster', font=creepster_font)
ax.set_xlabel('Zeit (s)', font=creepster_font)
ax.set_ylabel('Amplitude', font=creepster_font)
ax.legend(prop=creepster_font)
for label in ax.get_xticklabels():
    label.set_fontproperties(creepster_font)
for label in ax.get_yticklabels():
    label.set_fontproperties(creepster_font)
plt.show()
```



Und wir können auch verschiedene Schriftarten gleichzeitig nutzen:

```
# Verschiedene Fonts laden - alle von Google Fonts
amaranth_font = load_google_font("Amaranth", weight="bold")
inter_font = load_google_font("Inter")
creepster_font = load_google_font("Creepster")
source_code_pro = load_google_font("Source Code Pro")

fig, ax = plt.subplots(figsize=(10, 6), layout='tight')

# Drei verschiedene Texte mit verschiedenen Schriftarten
ax.text(x=0.1, y=0.9, size=16, s="Standard-Schriftart (DejaVu Sans)")
ax.text(x=0.1, y=0.7, size=16, s="Dies ist Amaranth", font=amaranth_font)
ax.text(x=0.1, y=0.5, size=16, s="Dies ist Inter", font=inter_font)
ax.text(x=0.1, y=0.3, size=16, s="Dies ist Creepster", font=creepster_font)
ax.text(x=0.1, y=0.1, size=16, s="Dies ist Source Code Pro", font=source_code_pro)

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_title('Vergleich verschiedener Schriftarten')
plt.show()
```

```
(0.0, 1.0)
(0.0, 1.0)
```



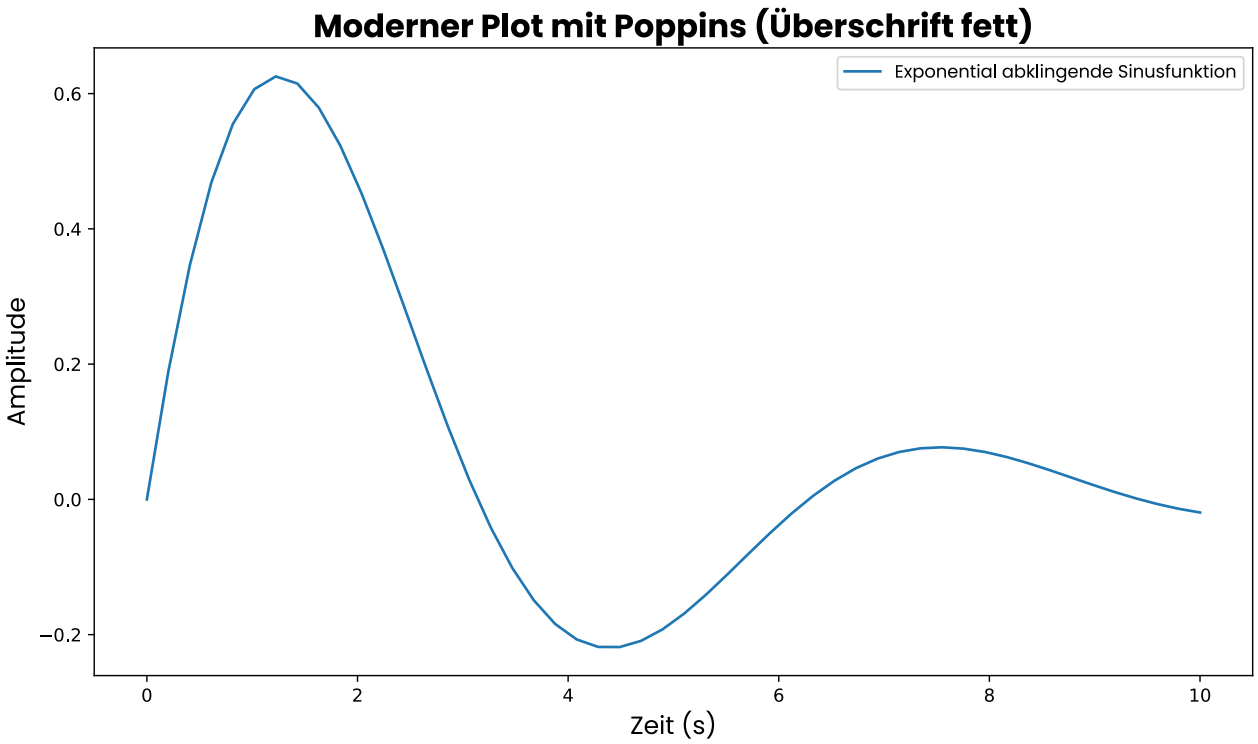
Der entscheidende Vorteil: Der Code läuft auf jedem Computer identisch, da die Schriftart automatisch heruntergeladen wird.

> **Weitere pyfonts-Funktionen**
>
> Neben `load_google_font()` bietet pyfonts auch andere Funktionen:
>
> - `load_font()`: Für das Laden von Schriftarten über direkte URLs
> - Unterstützung für lokale Schriftdateien: Für Unternehmens-Fonts oder spezielle Schriftarten, die nicht online verfügbar sind
>
> Für die meisten Anwendungsfälle ist `load_google_font()` jedoch die einfachste und zuverlässigste Option.

Verschiedene Schriftschnitte

Für professionelle Typografie benötigen wir oft verschiedene Schriftschnitte (fett, normal, etc.). Da pyfonts immer nur eine einzelne Schriftdatei lädt, müssen wir für verschiedene Gewichtungen separate Fonts laden:

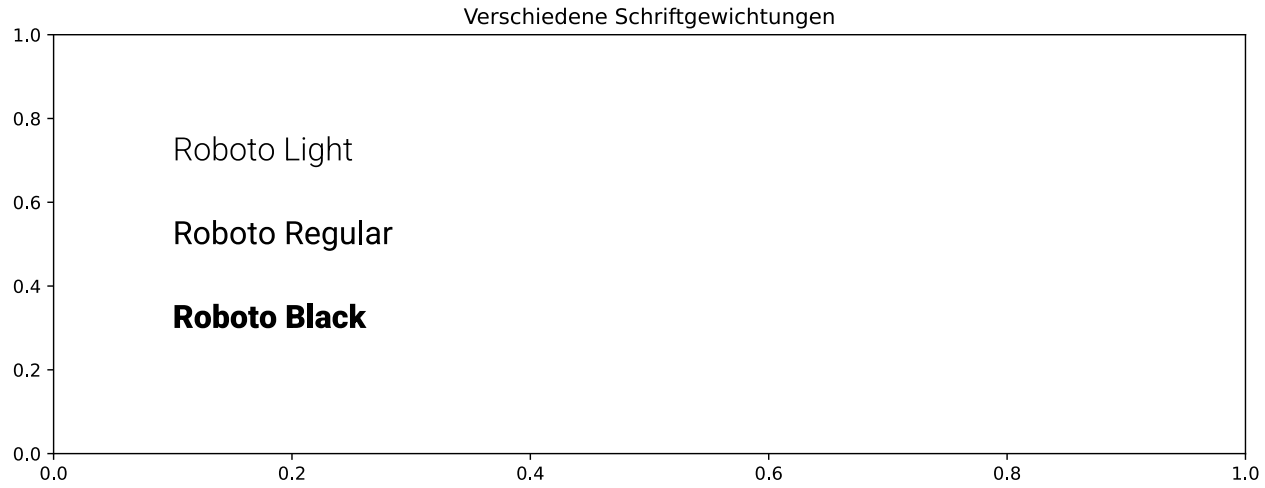
```
# Verschiedene Schriftschnitte mit load_google_font laden
poppins_regular = load_google_font("Poppins")
poppins_bold = load_google_font("Poppins", weight="bold")

# Plot mit verschiedenen Schriftgewichtungen
fig, ax = plt.subplots(figsize=(10, 6), layout='tight')
ax.plot(x, y, label='Exponential abklingende Sinusfunktion')
ax.set_title('Moderner Plot mit Poppins (Überschrift fett)', font=poppins_bold,
fontsize=18)
ax.set_xlabel('Zeit (s)', font=poppins_regular, fontsize=14)
ax.set_ylabel('Amplitude', font=poppins_regular, fontsize=14)
ax.legend(prop=poppins_regular)
plt.show()
```



Alternativ können wir auch verschiedene Gewichtungen vergleichen:

```
# Beispiel: Weitere Schriftarten laden
roboto_font = load_google_font("Roboto")
roboto_light = load_google_font("Roboto", weight="light")
roboto_black = load_google_font("Roboto", weight="black")

# Vergleich verschiedener Gewichtungen
fig, ax = plt.subplots(figsize=(10, 4), layout='tight')
ax.text(0.1, 0.7, 'Roboto Light', font=roboto_light, fontsize=18)
ax.text(0.1, 0.5, 'Roboto Regular', font=roboto_font, fontsize=18)
ax.text(0.1, 0.3, 'Roboto Black', font=roboto_black, fontsize=18)
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.set_title('Verschiedene Schriftgewichtungen')
plt.show()
```

```
(0.0, 1.0)
(0.0, 1.0)
```



Schriftarten finden und verwenden

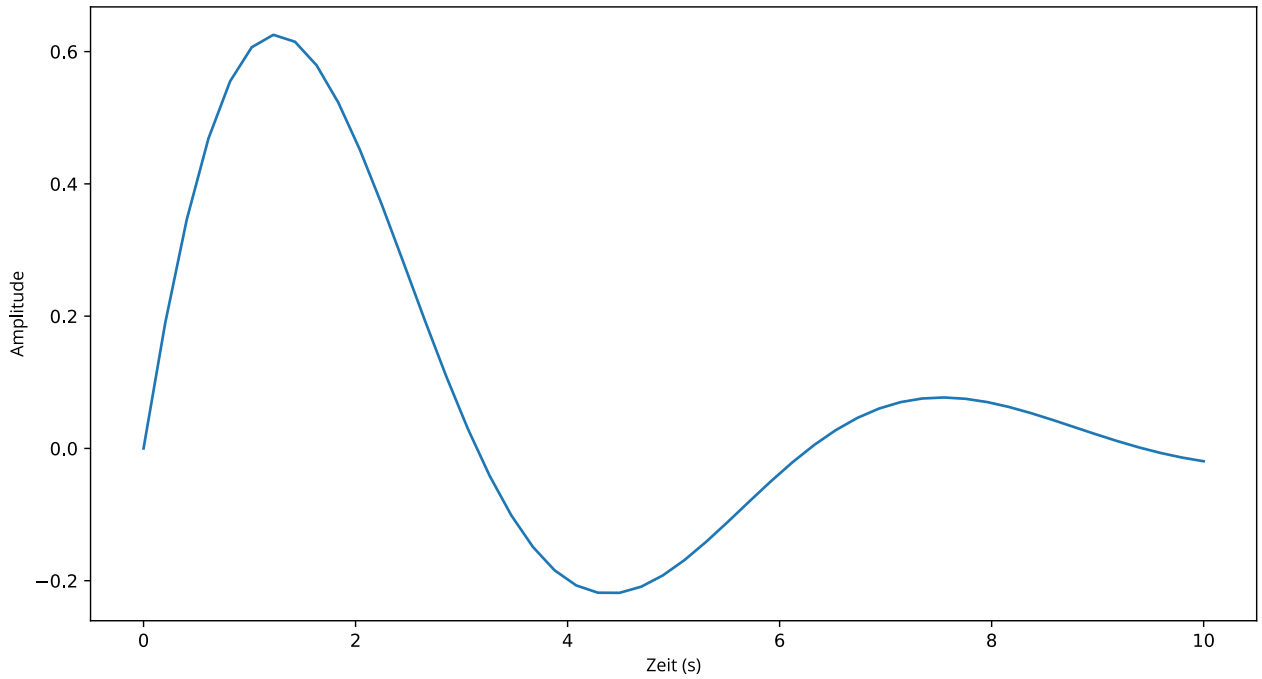
Google Fonts bietet eine riesige Auswahl kostenloser Schriftarten. Die `load_google_font()` Funktion macht es sehr einfach, diese zu verwenden. Schaut euch am besten dieses Video an um zu verstehen wie ihr die Schriftarten findet und welche Optionen es gibt.

Kombination mit highlight_text

Im DV3-Kapitel hatten wir schon kurz gesehen, dass wir mit dem `highlight_text`-Modul Text in matplotlib formatieren können. Diese Funktionalität lässt sich problemlos mit pyfonts kombinieren, um besonders ansprechende Titel zu erstellen:

```
regular_font = load_google_font("Open Sans")
bold_font = load_google_font("Open Sans", weight="bold")

fig, ax = plt.subplots(figsize=(10, 6), layout='tight')
ax.plot(x, y)

# Titel mit hervorgehobenem Wort
ax_text(
    x=0, y=0.75, transform=ax.transAxes,
    s="Die <exponential> abklingende Sinusfunktion",
    font=regular_font, fontsize=16,
    highlight_textprops=[{"font": bold_font, "color": "red"}]
);

ax.set_xlabel('Zeit (s)', font=regular_font)
ax.set_ylabel('Amplitude', font=regular_font)
plt.show()
```

Die **exponential** abklingende Sinusfunktion



Warum pyfonts verwenden?

Die Vorteile von pyfonts gegenüber herkömmlichen Methoden:

- **100% Reproduzierbarkeit**: Code läuft überall identisch
- **Keine lokale Installation**: Schriftarten werden automatisch heruntergeladen
- **Einfache Verwendung**: Ein Funktionsaufruf genügt
- **Große Auswahl**: Zugriff auf alle Google Fonts

Typografie-Grundsätze für Datenvisualisierung

Bei der Auswahl von Schriftarten für Datenvisualisierungen sollten einige grundlegende Prinzipien beachtet werden:

Lesbarkeit steht an erster Stelle

Befasst man sich mehr mit dem Thema Text (in Datenvisualisierungen), wird schnell klar, dass **Lesbarkeit** das wichtigste Kriterium ist. Eine gut lesbare Schriftart sorgt dafür, dass die Betrachter die Informationen schnell erfassen können, ohne sich mit der Schriftart selbst auseinandersetzen zu müssen. Dabei stößt man dann auf “Serifen” und “Sans-Serifen”:

Quelle: logo.com

Sans-serif Schriftarten (ohne Serifen) sind in der Regel die beste Wahl für Datenvisualisierungen, da sie auch bei kleinen Größen gut lesbar bleiben. Beispiele: Roboto, Open Sans, Inter, Arial.

Serif-Schriftarten (mit Serifen) können für Titel verwendet werden, um Eleganz und Tradition zu vermitteln, sind aber für Achsenbeschriftungen oft weniger geeignet.

Konsistenz bewahren

Verwendet nicht mehr als zwei verschiedene Schriftarten pro Visualisierung - eine für Überschriften und eine für den restlichen Text. Zu viele verschiedene Schriftarten wirken unprofessionell.

Größenverhältnisse beachten

Die Hierarchie der Textelemente sollte durch die Schriftgröße klar erkennbar sein:

- **Haupttitel**: 16-20pt
- **Achsenbeschriftungen**: 12-14pt
- **Tick-Labels**: 10-12pt
- **Legende**: 10-12pt

Brand Alignment

Wenn die Visualisierung für eine Organisation erstellt wird, sollte die Schriftart zu den Corporate Design Guidelines passen. Viele Unternehmen haben spezifische Schriftarten als Teil ihrer Markenidentität.

Professionelle Annotationen mit Pfeilen

Neben der richtigen Typografie gibt es noch einen weiteren Aspekt, der unsere Visualisierungen deutlich professioneller wirken lassen kann: **gezielte Annotationen mit Pfeilen**. Die meisten Menschen betrachten eine Grafik nur wenige Sekunden lang - umso wichtiger ist es, ihre Aufmerksamkeit gezielt auf 1-2 Schlüsselelemente zu lenken.

Warum Pfeile verwenden?

Pfeile sind perfekt geeignet, um **Aufmerksamkeit zu lenken** und wichtige Details hervorzuheben. Sie helfen dabei:

- **Fokus zu schaffen**: Die wichtigsten Punkte werden sofort sichtbar
- **Zusätzliche Details anzubieten**: Für Leser, die tiefer ins Detail gehen möchten
- **Professionellen Eindruck zu vermitteln**: Gut platzierte Annotationen wirken durchdacht und poliert

Das Problem mit matplotlib-Pfeilen

Leider ist das Erstellen von Pfeilen in reinem matplotlib eine **frustrierende Erfahrung**. Es gibt nicht weniger als 5 verschiedene Funktionen für Pfeile:

- `ax.arrow()`
- `ax.annotate()`
- `patches.Arrow()`
- `patches.FancyArrow()`
- `patches.FancyArrowPatch()`

Jede hat ihre eigene Syntax und Eigenarten, was die Verwendung umständlich und verwirrend macht.

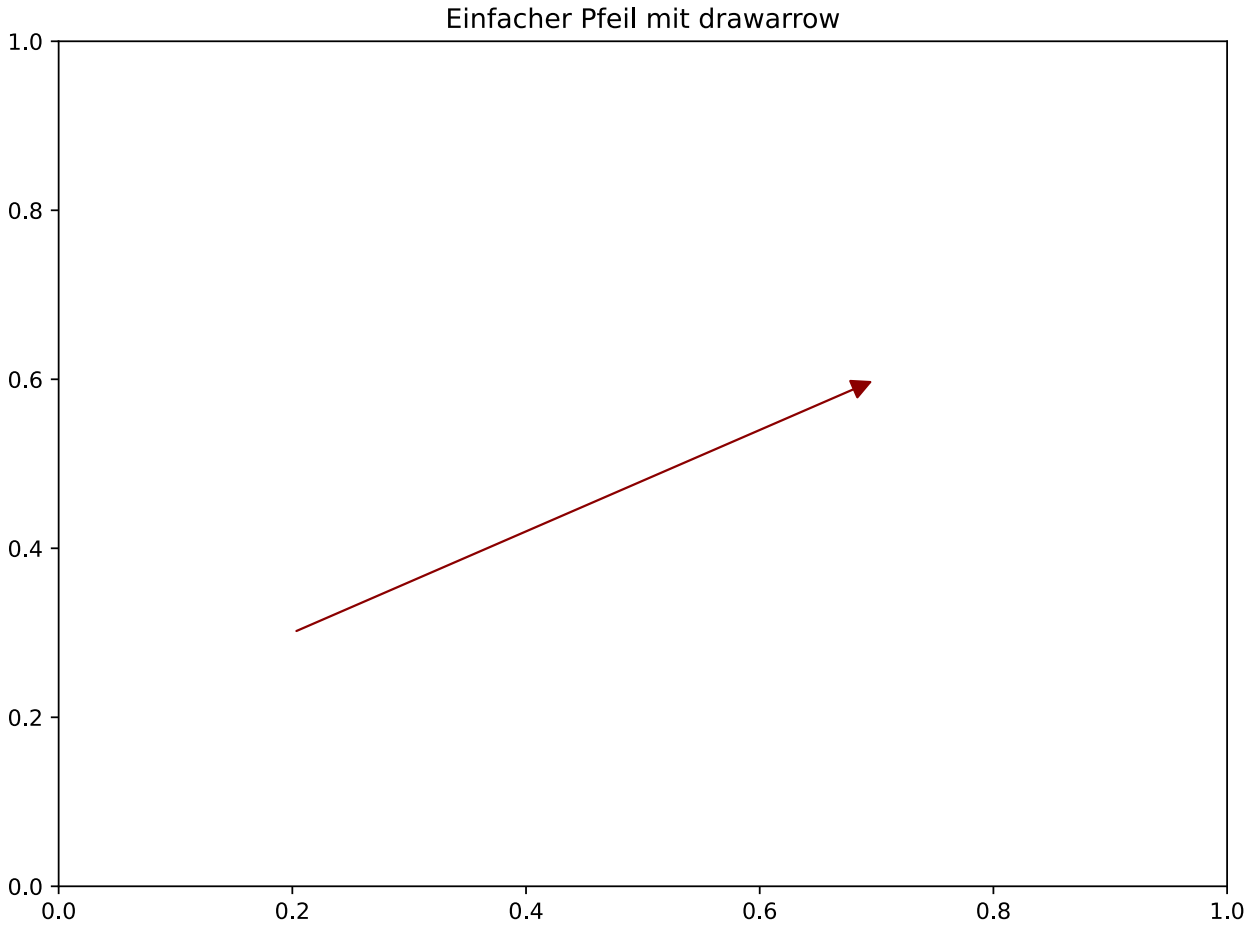
Die Lösung: drawarrow

Zum Glück gibt es `drawarrow` - ein Modul, die all diese Komplexität in eine einzige, intuitive Funktion verpackt.

Grundlegende Verwendung

Die `ax_arrow()` Funktion ist erstaunlich einfach zu verwenden:

```
fig, ax = plt.subplots(figsize=(8, 6), layout='tight')

# Einfacher Pfeil
ax_arrow(
    tail_position=[0.2, 0.3],
    head_position=[0.7, 0.6],
    color="darkred",
    ax=ax
)

ax.set_xlim(0, 1);
ax.set_ylim(0, 1);
ax.set_title('Einfacher Pfeil mit drawarrow')
plt.show()
```

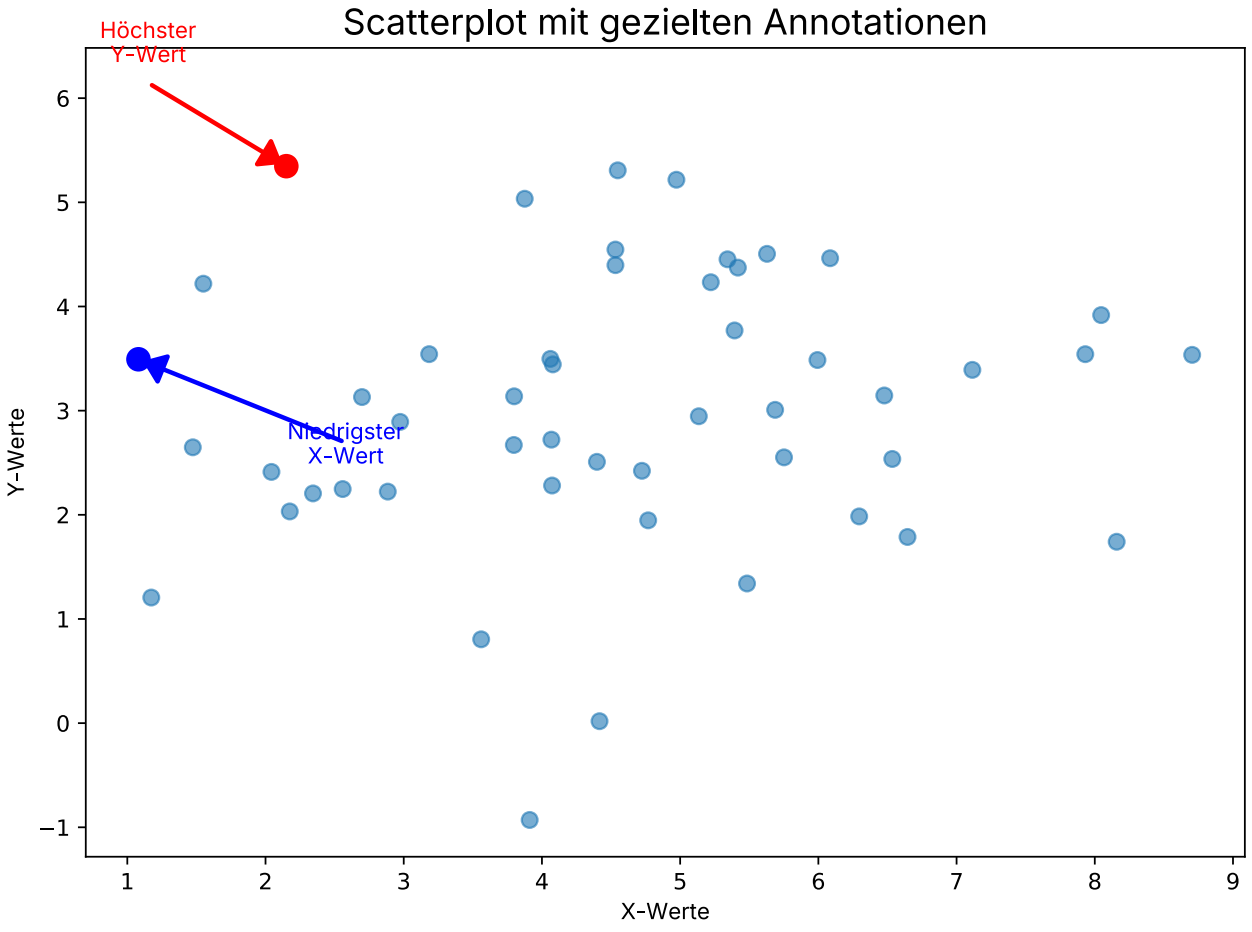


Ein nicht so optimales Beispiel

Hier ist ein Beispiel, wie Pfeile zur Annotation von Datenpunkten verwendet werden können. Man kann also jeden Pfeil und jede Text-Annotation einzeln setzen um z.B. Extremwerte hervorzuheben. (Eine Übungsaufgabe wird sein die Position der Pfeile und Länge etwas zu optimieren.)

```
# Schriftart laden
font = load_google_font("Inter")

# Beispieldaten
np.random.seed(42)
x_data = np.random.normal(5, 2, 50)
y_data = np.random.normal(3, 1.5, 50)

# Extremwerte identifizieren
max_idx = np.argmax(y_data)
min_idx = np.argmin(x_data)

fig, ax = plt.subplots(figsize=(8, 6), layout='tight')

# Scatterplot
ax.scatter(x_data, y_data, alpha=0.6, s=50)
ax.scatter(x_data[max_idx], y_data[max_idx], color='red', s=100, zorder=5)
ax.scatter(x_data[min_idx], y_data[min_idx], color='blue', s=100, zorder=5)

# Pfeile und Annotationen
ax_arrow(
    tail_position=[x_data[max_idx] - 1, y_data[max_idx] + 0.8],
```

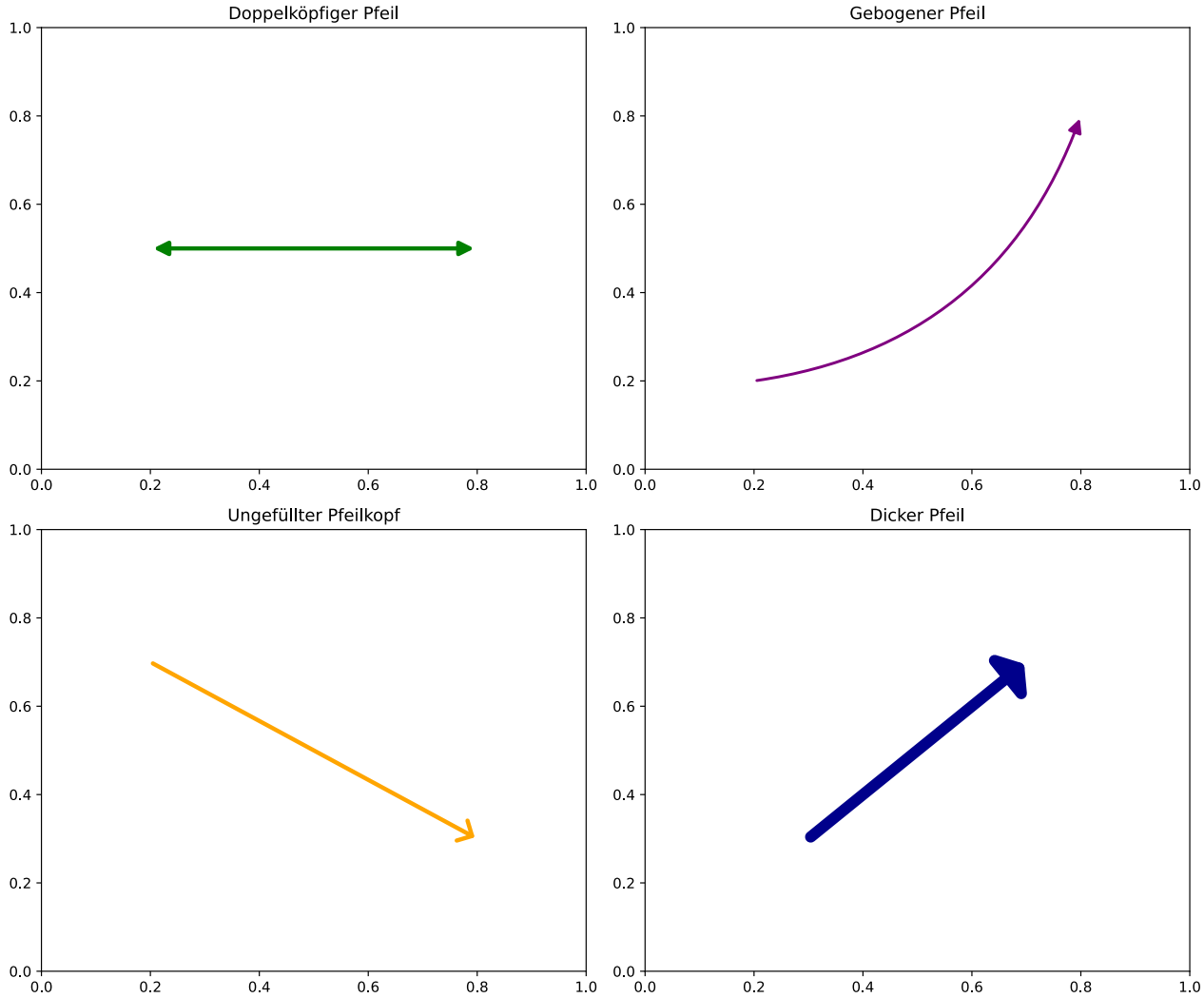
```
    head_position=[x_data[max_idx], y_data[max_idx]],
    color="red",
    width=2,
    head_width=6,
    ax=ax
)

ax_arrow(
    tail_position=[x_data[min_idx] + 1.5, y_data[min_idx] - 0.8],
    head_position=[x_data[min_idx], y_data[min_idx]],
    color="blue",
    width=2,
    head_width=6,
    ax=ax
)

# Textannotationen
ax.text(x_data[max_idx] - 1, y_data[max_idx] + 1,
        'Höchster\nY-Wert', font=font, ha='center', color='red')
ax.text(x_data[min_idx] + 1.5, y_data[min_idx] - 1,
        'Niedrigster\nX-Wert', font=font, ha='center', color='blue')

ax.set_xlabel('X-Werte', font=font)
ax.set_ylabel('Y-Werte', font=font)
ax.set_title('Scatterplot mit gezielten Annotationen', font=font, fontsize=16)
plt.show()
```



Erweiterte Pfeil-Optionen

`drawarrow` bietet viele Anpassungsmöglichkeiten:

```
fig, axes = plt.subplots(2, 2, figsize=(12, 10), layout='tight')

# Doppelköpfiger Pfeil
ax_arrow(
    tail_position=[0.2, 0.5],
    head_position=[0.8, 0.5],
    double_headed=True,
    color="green",
    width=3,
    ax=axes[0, 0]
)
axes[0, 0].set_title('Doppelköpfiger Pfeil')

# Gebogener Pfeil
ax_arrow(
    tail_position=[0.2, 0.2],
    head_position=[0.8, 0.8],
    radius=0.3,
    color="purple",
    width=2,
    ax=axes[0, 1]
)
axes[0, 1].set_title('Gebogener Pfeil')

# Ungefüllter Pfeilkopf
ax_arrow(
    tail_position=[0.2, 0.7],
    head_position=[0.8, 0.3],
    fill_head=False,
    color="orange",
    width=3,
    head_width=8,
    ax=axes[1, 0]
)
axes[1, 0].set_title('Ungefüllter Pfeilkopf')

# Dicker Pfeil
ax_arrow(
    tail_position=[0.3, 0.3],
    head_position=[0.7, 0.7],
    width=8,
    head_width=15,
    head_length=10,
    color="darkblue",
    ax=axes[1, 1]
)
axes[1, 1].set_title('Dicker Pfeil')

for ax in axes.flat:
    ax.set_xlim(0, 1);
    ax.set_ylim(0, 1);

plt.show()
```



Das `drawarrow`-Modul macht es endlich einfach, professionelle Annotationen zu erstellen, ohne sich mit der komplexen matplotlib-Syntax herumschlagen zu müssen.

Übungen

Übung 1: Pfeil-Positionen optimieren

Im Kapitel haben wir ein Beispiel mit einem Scatterplot gesehen, bei dem zwei Extremwerte durch Pfeile hervorgehoben wurden. Dort war angemerkt, dass die Positionierung der Pfeile noch optimiert werden könnte.

Kopiere den folgenden Code aus dem Kapitel und verbessere durch manuelles Ausprobieren die Positionen der Pfeile und Textannotationen, sodass sie professioneller und übersichtlicher aussehen:

```
# Schriftart laden
font = load_google_font("Inter")

# Beispieldaten
np.random.seed(42)
x_data = np.random.normal(5, 2, 50)
y_data = np.random.normal(3, 1.5, 50)

# Extremwerte identifizieren
max_idx = np.argmax(y_data)
```

```
min_idx = np.argmin(x_data)

fig, ax = plt.subplots(figsize=(8, 6), layout='tight')

# Scatterplot
ax.scatter(x_data, y_data, alpha=0.6, s=50)
ax.scatter(x_data[max_idx], y_data[max_idx], color='red', s=100, zorder=5)
ax.scatter(x_data[min_idx], y_data[min_idx], color='blue', s=100, zorder=5)

# Pfeile und Annotationen - DIESE SOLLEN VERBESSERT WERDEN
ax_arrow(
    tail_position=[x_data[max_idx] - 1, y_data[max_idx] + 0.8],
    head_position=[x_data[max_idx], y_data[max_idx]],
    color="red",
    width=2,
    head_width=6,
    ax=ax
)

ax_arrow(
    tail_position=[x_data[min_idx] + 1.5, y_data[min_idx] - 0.8],
    head_position=[x_data[min_idx], y_data[min_idx]],
    color="blue",
    width=2,
    head_width=6,
    ax=ax
)

# Textannotationen
ax.text(x_data[max_idx] - 1, y_data[max_idx] + 1,
        'Höchster\nY-Wert', font=font, ha='center', color='red')
ax.text(x_data[min_idx] + 1.5, y_data[min_idx] - 1,
        'Niedrigster\nX-Wert', font=font, ha='center', color='blue')

ax.set_xlabel('X-Werte', font=font)
ax.set_ylabel('Y-Werte', font=font)
ax.set_title('Scatterplot mit gezielten Annotationen', font=font, fontsize=16)
plt.show()
```

Ziel ist es, dass die Pfeile, Texte und Punkte nicht überlappen und die Aufmerksamkeit klar und professionell auf die hervorgehobenen Punkte lenken.

- (A) Geschafft

Übung 2: Pinguine mit systematischen Annotationen

Erstelle einen Plot mit dem Palmer Penguins Datensatz, der folgende Anforderungen erfüllt:

1. **Grundplot**: Lade den Datensatz und erstelle einen Scatterplot mit Körpergewicht (`body_mass_g`) auf der y-Achse und den drei Pinguinarten (`species`) auf der x-Achse. Die Punkte sollen **nicht** jittern (alle Punkte einer Art sollen exakt auf derselben x-Position liegen).
2. **Extremwerte identifizieren**: Finde für jede Pinguinart das jeweils schwerste Männchen und das jeweils schwerste Weibchen (verwende die Spalte `sex`).
3. **Systematische Annotationen**: Verwende eine Schleife, um für alle 6 identifizierten Extremwerte (3 Arten × 2 Geschlechter) folgende Elemente hinzuzufügen:
   - Einen Pfeil, der auf den entsprechenden Punkt zeigt

    - Einen Text, der das Extremum beschreibt (z.B. “Größtes männliches Adelie-Individuum”)
4. **Einheitliches Design**: Alle Pfeile sollen parallel zueinander verlaufen, gleich lang sein und in dieselbe Richtung zeigen.

Verwende für den Plot eine Schriftart deiner Wahl aus dem `pyfonts`-Modul und achte auf eine professionelle Darstellung.

- (A) Geschafft