Doch matplotlib

by Woche 2

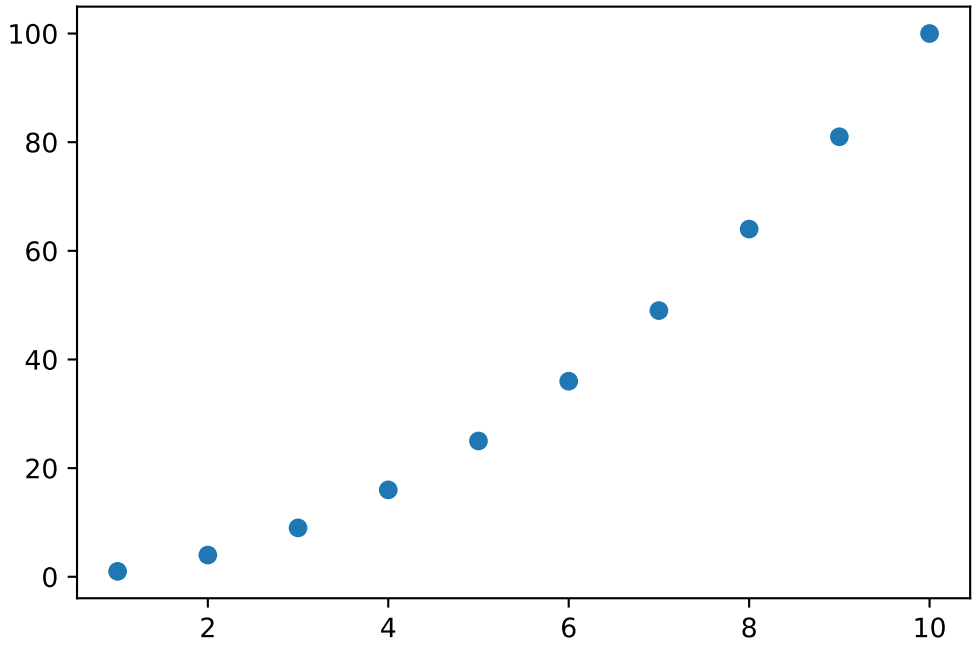
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

> **i** DataViz-Kapitel
>
> Kapitel zur Datenvisualisierung wie dieses werden verstreut über alle anderen Tehmenbereiche des Kurses auftauchen - eben immer dann wenn es sinnvoll ist, entsprechende Visualisierungen zu erstellen und damit einhergehend sich mit neuen Aspekte zu befassen. Aus diesem Grund folgen diese Kapitel auch nicht der normalen Kapitelnummerierung, sondern beginnen alle mit *DV*.

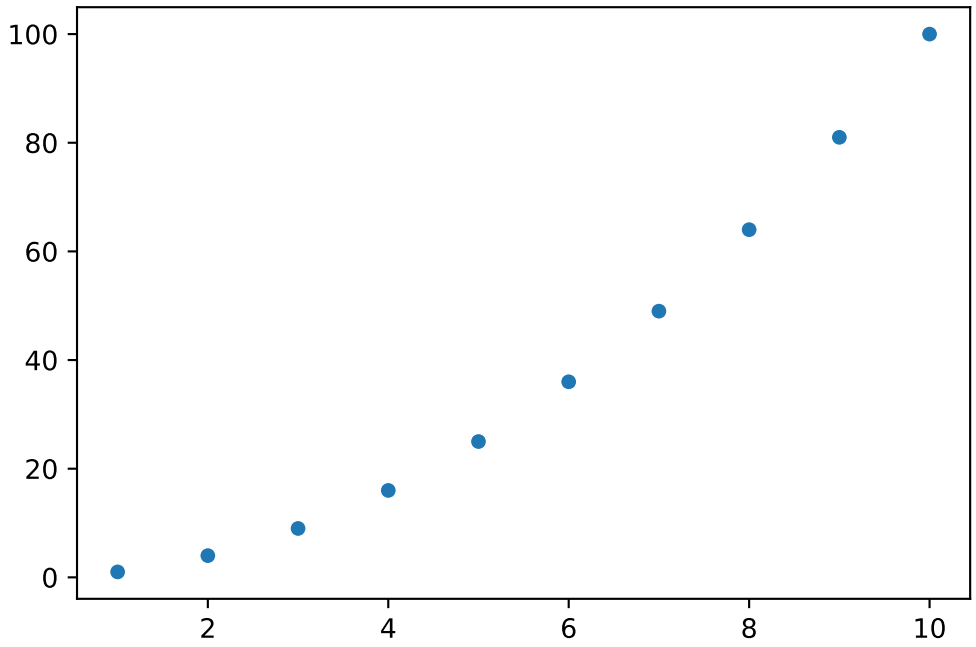
Im vorangegangenen Kurs haben wir bereits eine Reihe von Visualisierungen erstellt, oft mithilfe von Seaborn, das auf matplotlib aufbaut. Seaborn bietet einen einfachen und schnellen Weg, ästhetisch ansprechende Visualisierungen zu erzeugen - was es zu einem großartigen Einstiegspunkt macht. Hier nochmal der direkt Vergleich für beispielsweise einen Scatter-Plot, so wie wir ihn basierend auf dem bisherigen Wissen erzeugen würden:

```
x = np.arange(1, 11)
y = x ** 2
```

```
# nur matplotlib
plt.figure()
plt.scatter(x=x, y=y)
plt.show()
```



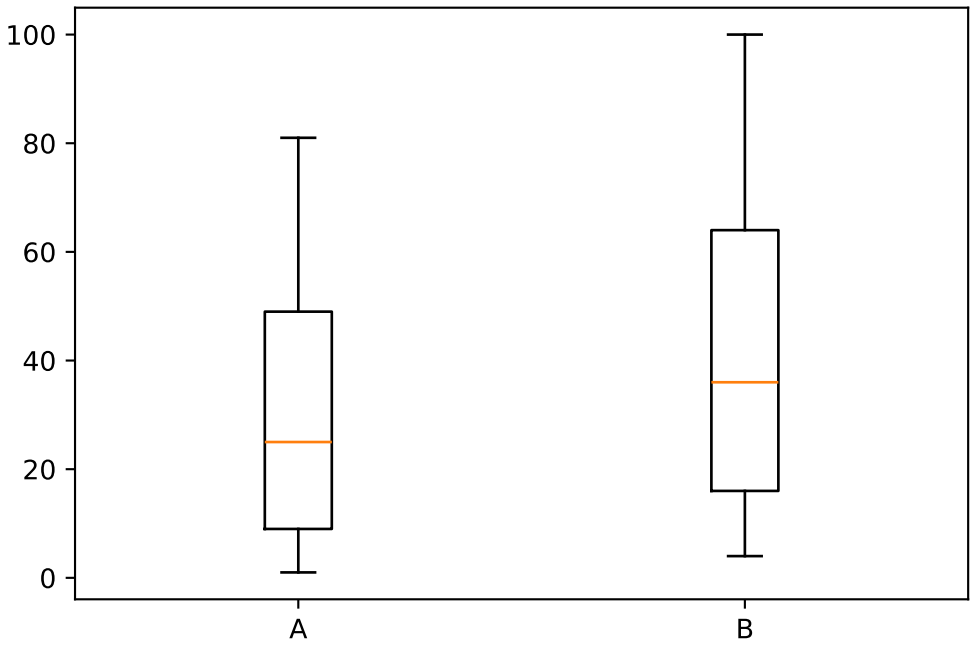

```
# mit seaborn
plt.figure()
sns.scatterplot(x=x, y=y)
plt.show()
```



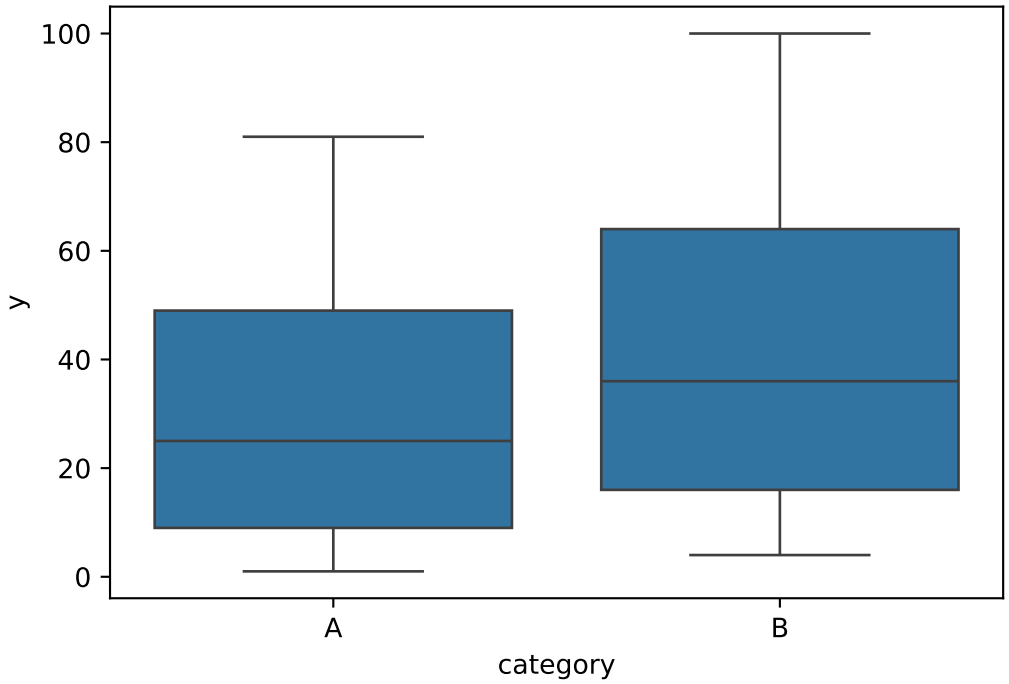
Speziell beim Scatterplot sehen wir kaum einen Unterschied und demnach auch kaum einen Vorteil. Bei einem multiplen Boxplot wird es jedoch schon deutlicher: Nicht nur wirkt der Code umständlicher, auch sieht der Plot von matplotlib nicht ganz so schön aus als der von seaborn - auch wenn das natürlich subjektiv ist.

```
z = np.array(['A', 'B'] * 5)
df = pd.DataFrame({'y': y, 'category': z})
```

```
# nur matplotlib
plt.figure()
plt.boxplot(x=[df[df['category']=='A']['y'],
               df[df['category']=='B']['y']],
            tick_labels=['A', 'B']);
plt.show()
```




```
# mit seaborn
plt.figure()
sns.boxplot(x='category', y='y', data=df)
#
#
plt.show()
```



Es muss aber klar sein, dass seaborn ein matplotlib-wrapper ist, also auf matplotlib aufbaut und im Endeffekt lediglich andere, vermeindlich schönere Standardwerte für die Darstellung, sowie eine benutzerfreundlichere Syntax anbietet.

Wir wollen ab diesem Punkt aber sozusagen zurück zu den Wurzeln und uns eingehender mit matplotlib beschäftigen. Warum? Weil matplotlib uns eine unvergleichliche Flexibilität bietet, wenn wir unsere Diagramme anpassen möchten. Hier ist eine Sammlung besonders gut aufbereiteter Diagramme, die mit matplotlib erstellt wurden, in der mal einige durch runterscrollen betrachtet werden sollen: Python Graph Gallery.

Warum matplotlib wichtig ist

Matplotlib ist die älteste und am weitesten verbreitete Visualisierungsbibliothek in Python und bietet mehrere entscheidende Vorteile:

1. **Vollständige Kontrolle**: Mit matplotlib können wir jedes einzelne Element eines Diagramms präzise steuern.
2. **Weit verbreitet**: Es gibt eine riesige Community, umfangreiche Dokumentation und zahllose Beispiele.
3. **Basis für andere Bibliotheken**: Seaborn, Pandas-Plotting, Plotly und viele andere bauen auf matplotlib auf.
4. **Branchenstandard**: In der Data Science-Welt ist matplotlib der De-facto-Standard für statische Visualisierungen.

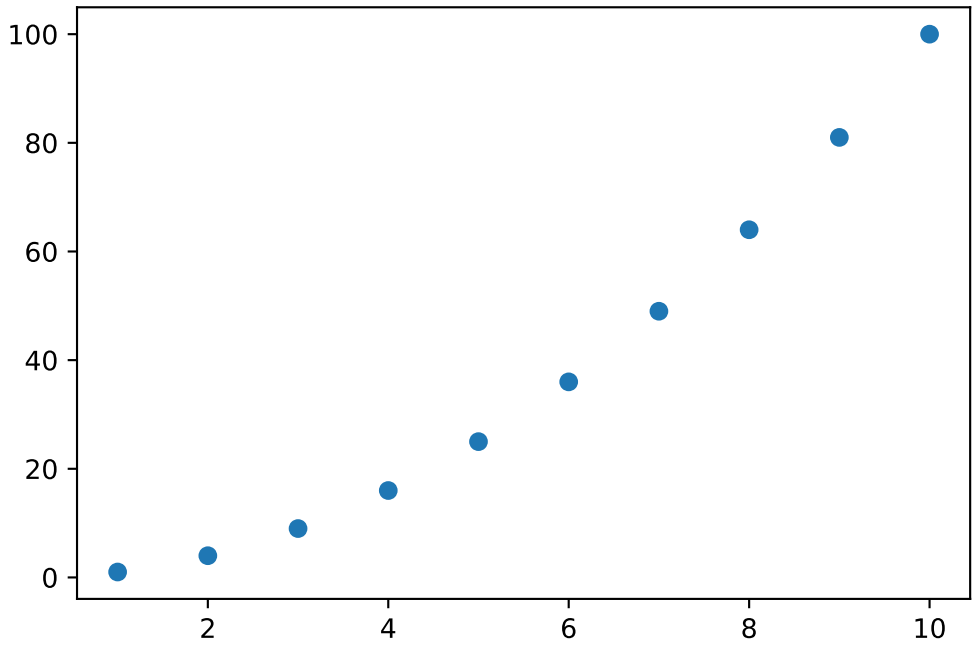
Im Grundkurs haben wir häufig Abkürzungen genommen, um schnell zu ansprechenden Visualisierungen zu gelangen. Das ist für den Einstieg völlig in Ordnung und es sei auch vorweggenommen, dass die Handhabung von matplotlib wie wir sie ab jetzt kennenlernen auch wirklich nicht unbedingt intuitiv ist. Im Gegenteil, es kann vor allem zu Beginn umständlich und kompliziert wirken - selbst wenn man die zugehörige Dokumentation liest.

Als fortgeschrittene Datenanalysten und angehende Data Scientists müssen wir jedoch ein tieferes Verständnis der Grundlagen entwickeln um überdurchschnittliche Diagramme erzeugen zu können - und das bedeutet, matplotlib im Detail zu beherrschen.

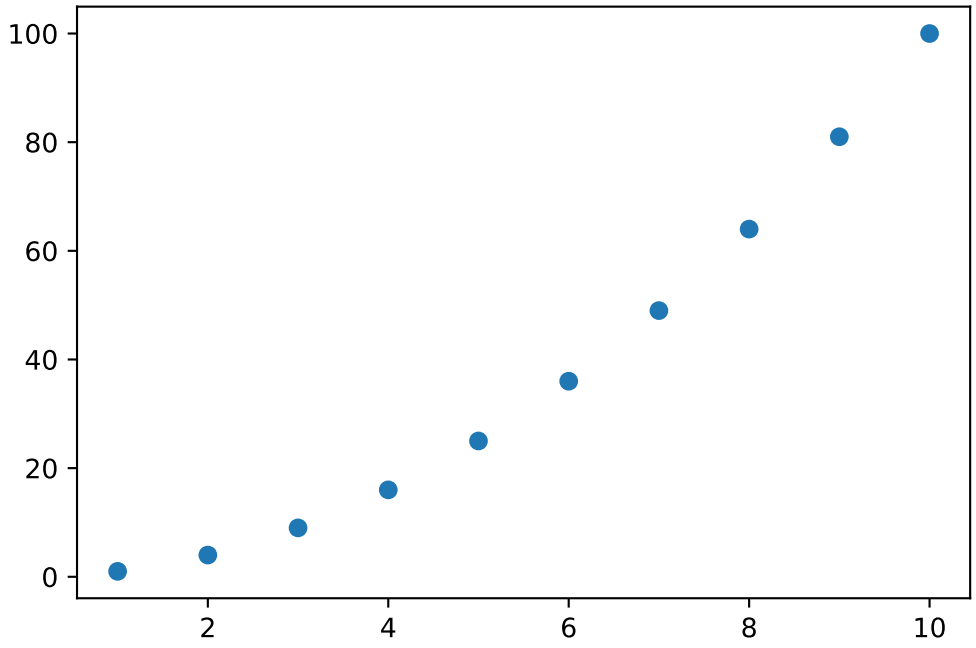
Die zweite Schnittstellen von matplotlib

Gleich zu Beginn kommt ein Bruch mit dem bisherigen Nutzen von matplotlib. Es ist nämlich so, dass matplotlib auf zwei verschiedene Arten genutzt werden kann. Die erste (links) ist die, die wir bisher verwendet haben und die objektorientierte (rechts) ist die, die wir ab jetzt verwenden werden.

```
plt.figure()
plt.scatter(x=x, y=y)
plt.show()
```




```
fig, ax = plt.subplots()
ax.scatter(x=x, y=y)
plt.show()
```



Obwohl wir also bisher gut gefahren sind, wollen wir ab nun stets diesen objektorientierten Ansatz verwenden:

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

Allein zu wissen, dass es diese zwei Ansätze gibt, sollte euch beim zukünftigen googlen und lesen von Dokumentationen oder Code anderer helfen. Der objektorientierte Ansatz ist schlichtweg mächtiger und flexibler wenn es hin zu komplexeren Visualisierungen geht, auch wenn er anfangs umständlicher erscheinen mag.

plt.subplots()

Die Funktion `plt.subplots()` ist der Einstiegspunkt für den objektorientierten Ansatz von matplotlib. Sie erzeugt sowohl eine Figure (`fig`) als auch eine oder mehrere Axes (`ax`). Der Begriff *subplots* selbst ist eventuell schon irritierend, weil es impliziert, dass wir mehrere Diagramme in einem Bild erstellen werden. Das ist auch manchmal so, aber wir können auch einfach nur ein Diagramm erstellen und tatsächlich sind die ersten beiden Argumente der Funktion `plt.subplots()` die Anzahl der Zeilen und Spalten von Diagrammen, die wir erstellen wollen - beide sind standardmäßig 1.

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

fig

Das `fig`-Objekt repräsentiert die gesamte Abbildung - es ist sozusagen die Leinwand, auf der alle Elemente platziert werden. Die Figure enthält alles, was wir sehen: den weißen Hintergrund, alle Achsen, Titel, Legenden und sonstige Elemente. Über das `fig`-Objekt können wir Eigenschaften steuern, die die gesamte Abbildung betreffen:

```
fig.suptitle('Haupttitel der gesamten Figure') # Gesamttitel setzen
fig.set_size_inches(8, 6)                      # Größe anpassen
fig.set_facecolor('gold')                      # Hintergrundfarbe ändern
```

Die Figure ist also der Container für alles andere - wie ein Bilderrahmen mit Leinwand[1].

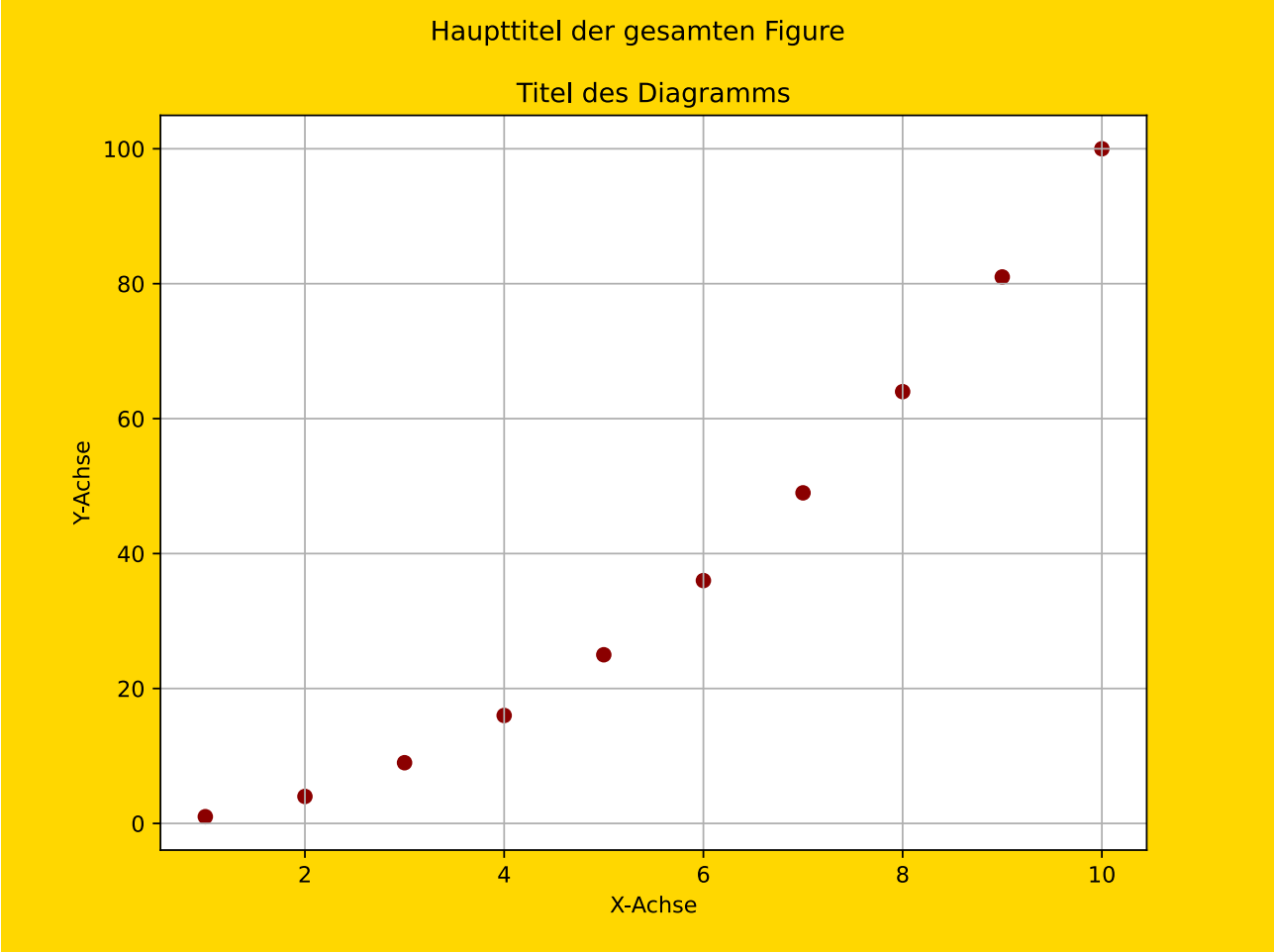
ax

Das `ax`-Objekt repräsentiert eine einzelne Achsenregion innerhalb der Figure, also den eigentlichen Bereich, in dem Daten dargestellt werden - mit x- und y-Achsen, Gitternetzlinien und natürlich den Datenpunkten selbst. Die meisten Plotting-Methoden werden auf das `ax`-Objekt angewendet:

```
ax.scatter(x=x, y=y, c='darkred')   # Scatter-Plot erstellen
ax.set_title('Titel des Diagramms') # Titel für dieses Diagramm setzen
ax.set_xlabel('X-Achse')            # X-Achsenbeschriftung
ax.set_ylabel('Y-Achse')            # Y-Achsenbeschriftung
ax.grid(True)                       # Gitternetzlinien einschalten
```

Bei mehreren Subplots enthält `ax` mehrere solcher Achsenobjekte, und wir können auf jedes einzeln zugreifen (z.B. `ax[0, 1]` für die Achse in der ersten Zeile, zweite Spalte).

```
plt.show()
```

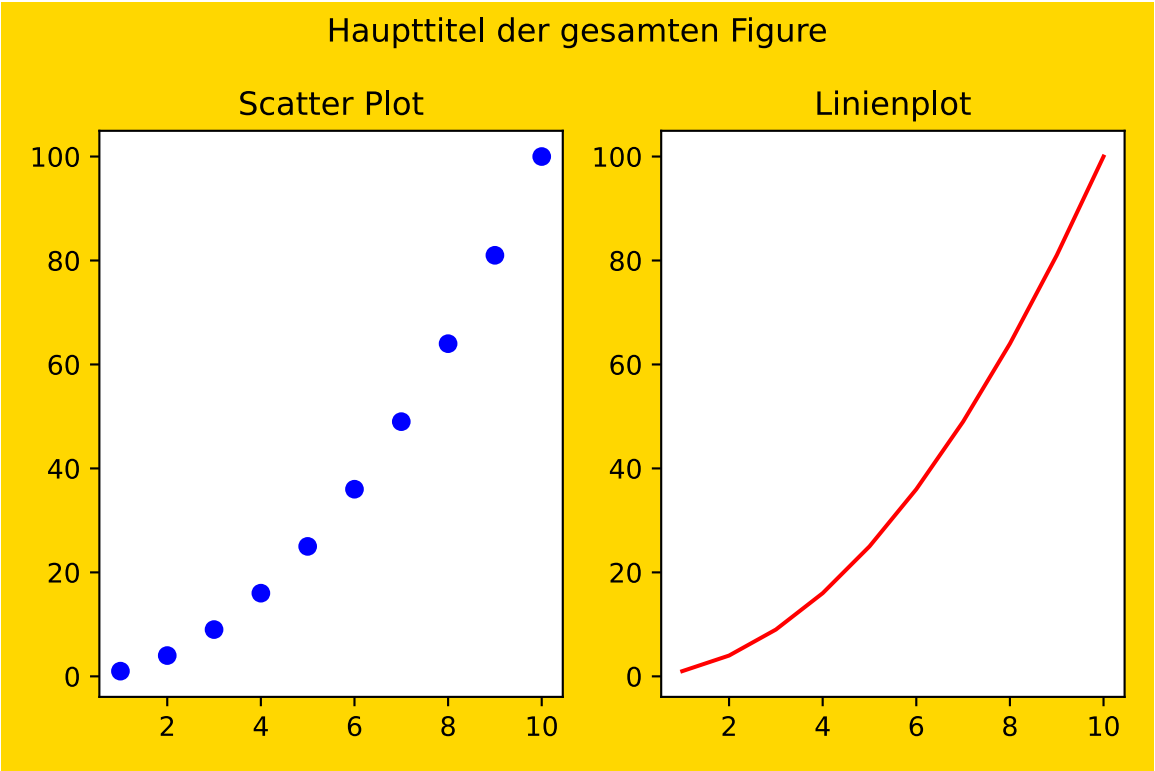


[1] Übrigens hatte unser bisheriger code `plt.figure()` auch immer ein `fig` erstellt, aber wir haben es nie explizit benutzt.

Mehrere Subplots

Hier noch ein einfaches Beispiel mit zwei Subplots, was zu zwei Axes-Objekten führt:

```
fig, axs = plt.subplots(ncols=2, layout="tight")

fig.suptitle('Haupttitel der gesamten Figure')
fig.set_facecolor('gold')

# 1. Diagramm: Scatter Plot
axs[0].scatter(x, y, color='blue')
axs[0].set_title('Scatter Plot')

# 2. Diagramm: Linienplot
axs[1].plot(x, y, color='red')
axs[1].set_title('Linienplot')

plt.show()
```



Wir haben also folgendes Änderungen vorgenommen:

1. `plt.subplots(ncols=2)` erzeugt direkt zwei `ax`, welche innerhalb der `fig` nebeneinander, also in zwei Spalten, angeordnet sind. Aus diesem Grund haben wir das Objekt auch `axs` statt `ax` genannt um uns daran zu erinnern, dass es mehrere sind - aber weiterhin in nur einem Objekt.
2. `plt.subplots(layout="tight")` sorgt dafür, dass die Diagramme eng beieinander stehen und keine unnötigen Abstände entstehen. Alternativ kann man denselben Effekt erreichen, indem man nachträglich `plt.tight_layout()` aufruft.
3. Mit dem Index `axs[0]` und `axs[1]` greifen wir auf den jeweiligen Plot zu und können ihn individuell wie gewohnt anpassen.

Da wir nun mehrere `ax` haben, wird auch der Unterschied zur `fig` deutlicher. So ist der Titel der `fig` der Haupttitel der gesamten Abbildung, während der Titel der `ax` nur für das jeweilige Diagramm gilt. Das gleiche gilt für die Größe, Hintergrundfarbe und andere Eigenschaften.

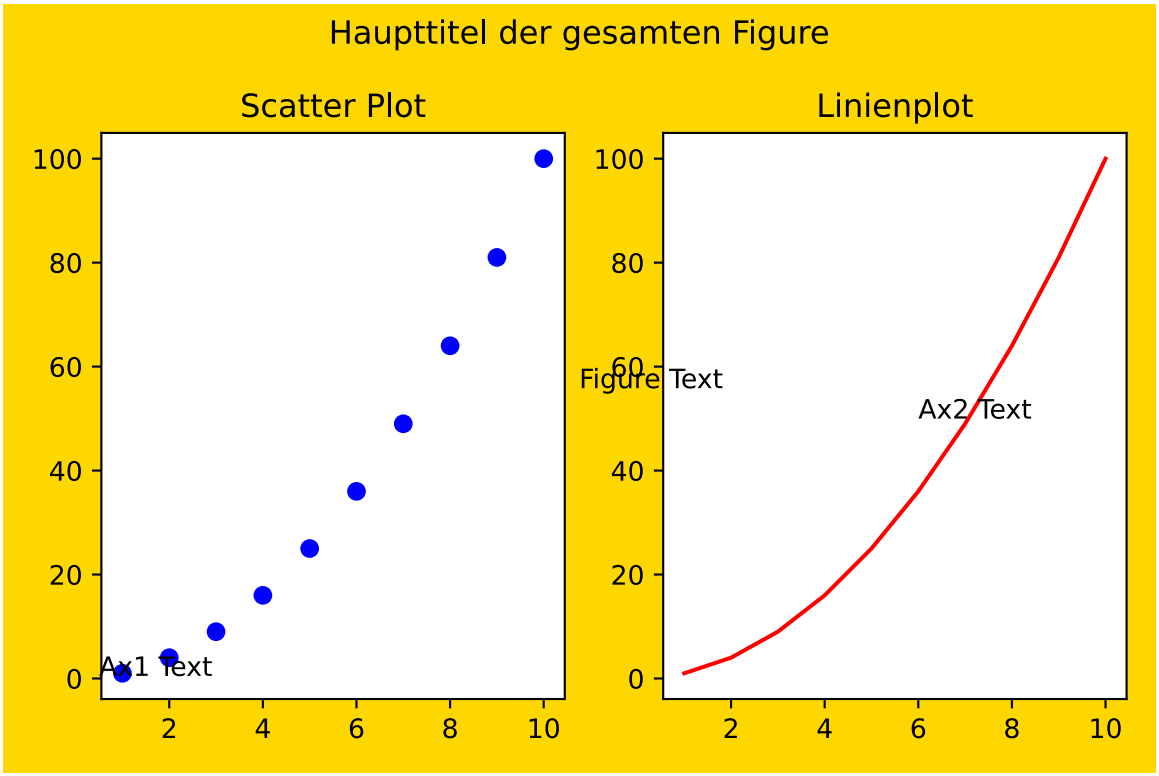
Passend dazu bietet es sich an direkt noch über einen letzten Punkt in diesem Kaptitel zu sprechen:

Koordinaten in `fig` vs. `ax`

Ein wichtiger Unterschied zwischen `fig` und `ax` ist die Art und Weise, wie Koordinaten behandelt werden. In der `fig`-Instanz sind die Koordinaten immer relativ zur gesamten Abbildung, während sie in der `ax`-Instanz relativ zur jeweiligen Achse sind. Das bedeutet, dass die Koordinaten in `fig` immer zwischen 0 und 1 liegen, während sie je `ax` eben von den jeweiligen Achsen bzw. Daten abhängen.

Am einfachsten lässt sich das zeigen, indem wir etwas Text in die Abbildung einfügen. Dies geht mit `.text(x, y, s)` und zwar sowohl in `fig` als auch in `ax`. Neben den x- und y-Koordinaten zur Bestimmung wohin der Text soll, steht `s` in dieser Funktion für den Text (String) selbst.

```
fig, axs = plt.subplots(ncols=2, layout="tight")

fig.suptitle('Haupttitel der gesamten Figure')
fig.set_facecolor('gold')

# 1. Diagramm: Scatter Plot
axs[0].scatter(x, y, color='blue')
axs[0].set_title('Scatter Plot')

# 2. Diagramm: Linienplot
axs[1].plot(x, y, color='red')
axs[1].set_title('Linienplot')

fig.text(0.5, 0.5, 'Figure Text')
axs[0].text(0.5, 0.5, 'Ax1 Text')
axs[1].text(6, 50, 'Ax2 Text')

plt.show()
```



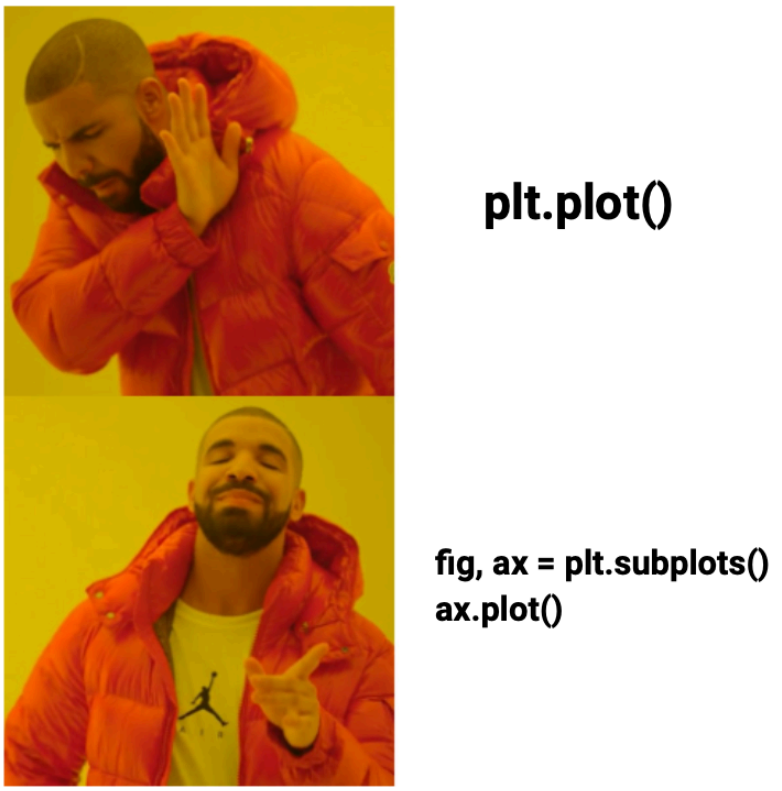
Wir sehen, dass in der Tat alle drei Texte erscheinen. Bei genauerem Hinsehen wird klar, was auch eben schon gesagt wurde:

- ‘Figure Text’ erscheint in der Mitte der gesamten Abbildung. Genauer gesagt beginnt der String dort - ist also links-zentriert am Mittelpunkt der Abbildung. Dies entspricht den Koordinaten (0.5, 0.5) in der `fig` - also 50% von links und 50% von oben.
- Dieselben Koordinaten führen im ersten Diagramm aber dazu, dass ‘Ax1 Text’ nicht in der Mitte des Diagramms erscheint, sondern da, wo der x-Wert 0.5 und der y-Wert 0.5 des Diagramms liegen.
- Für das zweite Diagramm wurden dann Koordinaten gewählt, die schon eher dem Mittelpunkt der Daten entsprechen - eben relativ zu den tatsächlichen Werten auf der Achse. Aber auch wieder links-zentriert.

In der Praxis sind beide Arten von Koordinaten nützlich, sogar auch direkt für das Anbringen von Labels wie diesen. Man sollte also den Unterschied kennen und je nach Bedarf die passende Art von Koordinaten verwenden.

Zusammenfassung

Wir wollen ab jetzt matplotlib im objektorientierten Stil verwenden, um die volle Kontrolle über unsere Diagramme zu haben. Dazu erstellen wir eine `fig`- und eine `ax`-Instanz mit `fig, ax = plt.subplots()`. `fig` ist der gesamte “Rahmen”, während `ax` der Bereich ist, in dem die eigentlichen Diagramme gezeichnet werden.



Quelle: www.matplotlib-journey.com

Weitere Ressourcen

- Dataviz Inspiration - Weitere beeindruckende Abbildungen, welche mit matplotlib erzeugt wurden

Übungen

Übung 1

Beim genaueren Betrachten der Abbildung oben mit `ax.grid(True)` fällt auf, dass die Gitternetzlinien nicht hinter, sondern vor den Punkten gezeichnet wurden. Versuch dich zu erinnern (Stichwort: Data Analytics Kurs) wie man die Reihenfolge der gezeichneten Elemente ändern kann und setze die Gitternetzlinien hinter die Punkte.

- (A) Geschafft

Übung 2

Erstelle ein Figure-Objekt mit vier verschiedenen Diagrammtypen. Verwende neben `.scatter()` und `.plot()` auch `.bar()` und `.hist()`. Verwende die folgenden Daten. Achtung, `.bar()` und `.hist()` werden `x` nicht benötigen.

```
x = np.arange(1, 11)
y = x ** 2
kategorien = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

- (A) Geschafft: vier Subplots in einer Reihe (alle nebeneinander)
- (A) Geschafft: vier Subplots in einem 2x2-Raster

Übung 3

Der Parameter `ha` (horizontale Ausrichtung) steuert die horizontale Ausrichtung des Textes und kann folgende Werte annehmen:

- `'left'`: Text ist links-ausgerichtet (Standardwert)
- `'center'`: Text ist mittig ausgerichtet
- `'right'`: Text ist rechts-ausgerichtet

Mit dem Parameter `va` (vertikale Ausrichtung) kann man außerdem die vertikale Ausrichtung steuern:

- `'bottom'`: Text ist unten ausgerichtet
- `'center'`: Text ist mittig ausgerichtet (vertikal)
- `'top'`: Text ist oben ausgerichtet

Erstelle einen Scatter-Plot mit den oben definierten Daten (x und y) und setze dann Text möglichst genau neben einen Punkt deiner Wahl:

- (A) Geschafft via ax.text()
- (A) Geschafft via fig.text()

Übung 4

Gehe zurück zu Kapitel 4.2 MW & Histogramm aus dem Data Analytics Kurs und sorge dafür, dass die beiden Histogramme, die dort ganz am Ende erzeugt wurden, nun in einer einzigen Abbildung dargestellt werden.

- (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.