



Kostenloses eBook

LERNEN

matplotlib

Free unaffiliated eBook created from
Stack Overflow contributors.

#matplotlib

Inhaltsverzeichnis

Über.....	1
Kapitel 1: Erste Schritte mit matplotlib.....	2
Bemerkungen.....	2
Überblick.....	2
Versionen.....	2
Examples.....	2
Installation und Einrichtung.....	2
Windows.....	2
OS X.....	2
Linux.....	3
Debian / Ubuntu.....	3
Fedora / Roter Hut.....	3
Fehlerbehebung.....	3
Anpassen eines Matplotlib-Diagramms.....	3
Imperative versus objektorientierte Syntax.....	5
Zweidimensionale (2D) Arrays.....	7
Kapitel 2: Animationen und interaktives Plotten.....	8
Einführung.....	8
Examples.....	8
Grundanimation mit FuncAnimation.....	8
Animation in GIF speichern.....	9
Interaktive Steuerelemente mit matplotlib.widgets.....	10
Live-Daten von Pipe mit Matplotlib darstellen.....	11
Kapitel 3: Bildbearbeitung.....	14
Examples.....	14
Bilder öffnen.....	14
Kapitel 4: Boxplots.....	16
Examples.....	16
Grundlegende Boxplots.....	16

Kapitel 5: Boxplots	18
Examples	18
Boxplot-Funktion	18
Kapitel 6: Colormaps	25
Examples	25
Grundlegende Verwendung	25
Benutzerdefinierte Colormaps verwenden	27
Wahrnehmungsmäßig einheitliche Farbkarten	29
Benutzerdefinierte diskrete Colormap	31
Kapitel 7: Dreidimensionale Diagramme	33
Bemerkungen	33
Examples	36
Dreidimensionale Achsen erstellen	36
Kapitel 8: Figuren und Äxte Objekte	38
Examples	38
Eine Figur erstellen	38
Achsen erstellen	38
Kapitel 9: Figurfenster schließen	40
Syntax	40
Examples	40
Schließung der aktuell aktiven Figur mittels Pyplot	40
Eine bestimmte Figur mit plt.close () schließen	40
Kapitel 10: Gitterlinien und Häkchen	41
Examples	41
Plot mit Gitternetzlinien	41
Plot mit Gitternetzlinien	41
Plot mit großen und kleinen Rasterlinien	42
Kapitel 11: Grundlegende Diagramme	44
Examples	44
Streudiagramme	44
Ein einfaches Streudiagramm	44

Ein Streudiagramm mit beschrifteten Punkten.....	45
Schattierte Pläne.....	46
Schattenbereich unter einer Linie.....	46
Schattierte Region zwischen zwei Zeilen.....	47
Liniendiagramme.....	48
Einfaches Liniendiagramm.....	48
Datenplot.....	50
Daten und Zeile.....	51
Heatmap.....	52
Kapitel 12: Histogramm.....	56
Examples.....	56
Einfaches Histogramm.....	56
Kapitel 13: Integration mit TeX / LaTeX.....	57
Bemerkungen.....	57
Examples.....	57
Einfügen von TeX-Formeln in Diagramme.....	57
Speichern und Exportieren von Plots, die TeX verwenden.....	59
Kapitel 14: Konturkarten.....	61
Examples.....	61
Einfache gefüllte Konturdarstellung.....	61
Einfache Konturdarstellung.....	62
Kapitel 15: Koordinatensysteme.....	63
Bemerkungen.....	63
Examples.....	64
Koordinatensysteme und Text.....	64
Kapitel 16: Legenden.....	67
Examples.....	67
Einfache Legende.....	67
Legende außerhalb des Grundstücks platziert.....	69
Einzelne Legende für mehrere Subplots freigegeben.....	71
Mehrere Legenden auf derselben Achse.....	72

Kapitel 17: LogLog-Darstellung	76
Einführung	76
Examples	76
LogLog-Darstellung	76
Kapitel 18: Mehrere Plots	79
Syntax	79
Examples	79
Raster von Subplots mit Subplot	79
Mehrere Linien / Kurven in derselben Zeichnung	80
Mehrere Plots mit Gitterspez	82
Eine Darstellung von 2 Funktionen auf der gemeinsamen x-Achse	83
Mehrere Plots und Multiple Plot-Funktionen	84
Credits	92



You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [matplotlib](#)

It is an unofficial and free matplotlib ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official matplotlib.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit matplotlib

Bemerkungen

Überblick

matplotlib ist eine *Plot*- Bibliothek für Python. Es bietet objektorientierte APIs zum Einbetten von Diagrammen in Anwendungen. Es ähnelt MATLAB in Kapazität und Syntax.

Es wurde ursprünglich von [JDHunter geschrieben](#) und wird aktiv weiterentwickelt. Es wird unter einer BSD-Style-Lizenz vertrieben.

Versionen

Ausführung	Unterstützte Python-Versionen	Bemerkungen	Veröffentlichungsdatum
1.3.1	2,6, 2,7, 3.x	Ältere stabile Version	2013-10-10
1.4.3	2,6, 2,7, 3.x	Vorherige stabile Version	2015-07-14
1.5.3	2,7, 3.x	Aktuelle stabile Version	2016-01-11
2.x	2,7, 3.x	Neueste Entwicklungsversion	2016-07-25

Examples

Installation und Einrichtung

Es gibt verschiedene Möglichkeiten, Matplotlib zu installieren. Einige davon hängen von dem verwendeten System ab. Wenn Sie Glück haben, können Sie mit einem Paketmanager das matplotlib-Modul und seine Abhängigkeiten einfach installieren.

Windows

Auf Windows-Computern können Sie versuchen, den Pip Package Manager zur Installation von matplotlib zu verwenden. Sehen Sie [hier](#) für Informationen über pip in einer Windows - Umgebung einrichten.

OS X

Es wird empfohlen, dass Sie den [pip](#) package manager verwenden, um matplotlib zu installieren. Wenn Sie einige Nicht-Python-Bibliotheken (z. B. `libfreetype`) auf Ihrem System installieren müssen, sollten Sie die Verwendung von [Homebrew in Betracht](#) ziehen.

Wenn Sie pip aus irgendeinem Grund nicht verwenden können, versuchen Sie, die Installation von der [Quelle](#) aus [durchzuführen](#).

Linux

Idealerweise sollte der Systempaket-Manager oder pip zur Installation von matplotlib verwendet werden, entweder durch Installieren des Pakets `python-matplotlib` oder durch Ausführen von `pip install matplotlib`.

Wenn dies nicht möglich ist (z. B. haben Sie keine Sudo-Berechtigungen auf dem von Ihnen verwendeten Computer), können Sie die Installation mit der Option `python setup.py install --user` von der [Quelle](#) aus `--user:python setup.py install --user`. Normalerweise wird matplotlib in `~/.local` installiert.

Debian / Ubuntu

```
sudo apt-get install python-matplotlib
```

Fedora / Roter Hut

```
sudo yum install python-matplotlib
```

Fehlerbehebung

Auf der [matplotlib-Website](#) finden Sie Hinweise, wie Sie eine kaputte matplotlib reparieren können.

Anpassen eines Matplotlib-Diagramms

```
import pylab as plt
import numpy as np

plt.style.use('ggplot')

fig = plt.figure(1)
ax = plt.gca()

# make some testing data
x = np.linspace( 0, np.pi, 1000 )
test_f = lambda x: np.sin(x)*3 + np.cos(2*x)
```



```

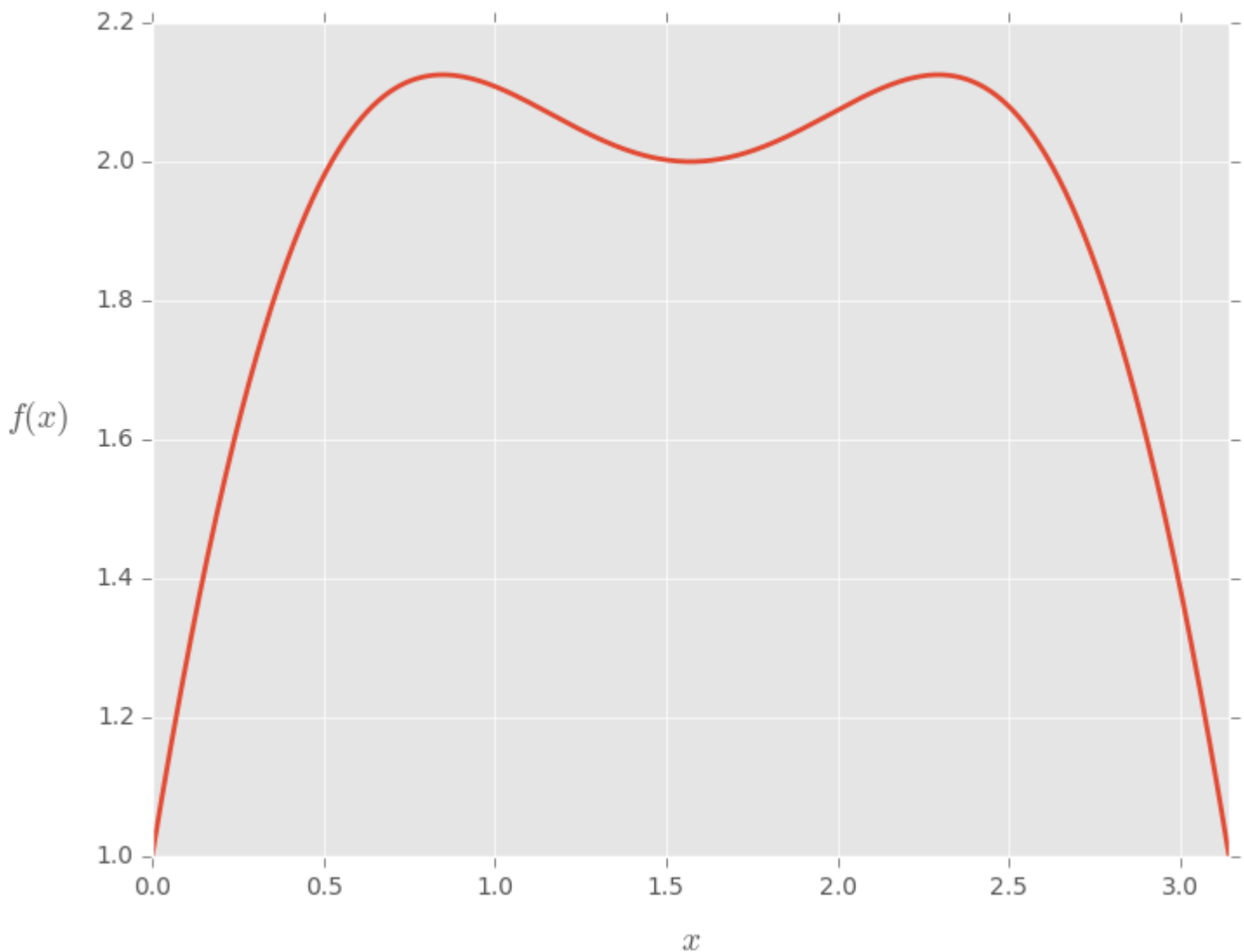
# plot the test data
ax.plot( x, test_f(x) , lw = 2)

# set the axis labels
ax.set_xlabel(r'$x$', fontsize=14, labelpad=10)
ax.set_ylabel(r'$f(x)$', fontsize=14, labelpad=25, rotation=0)

# set axis limits
ax.set_xlim(0,np.pi)

plt.draw()

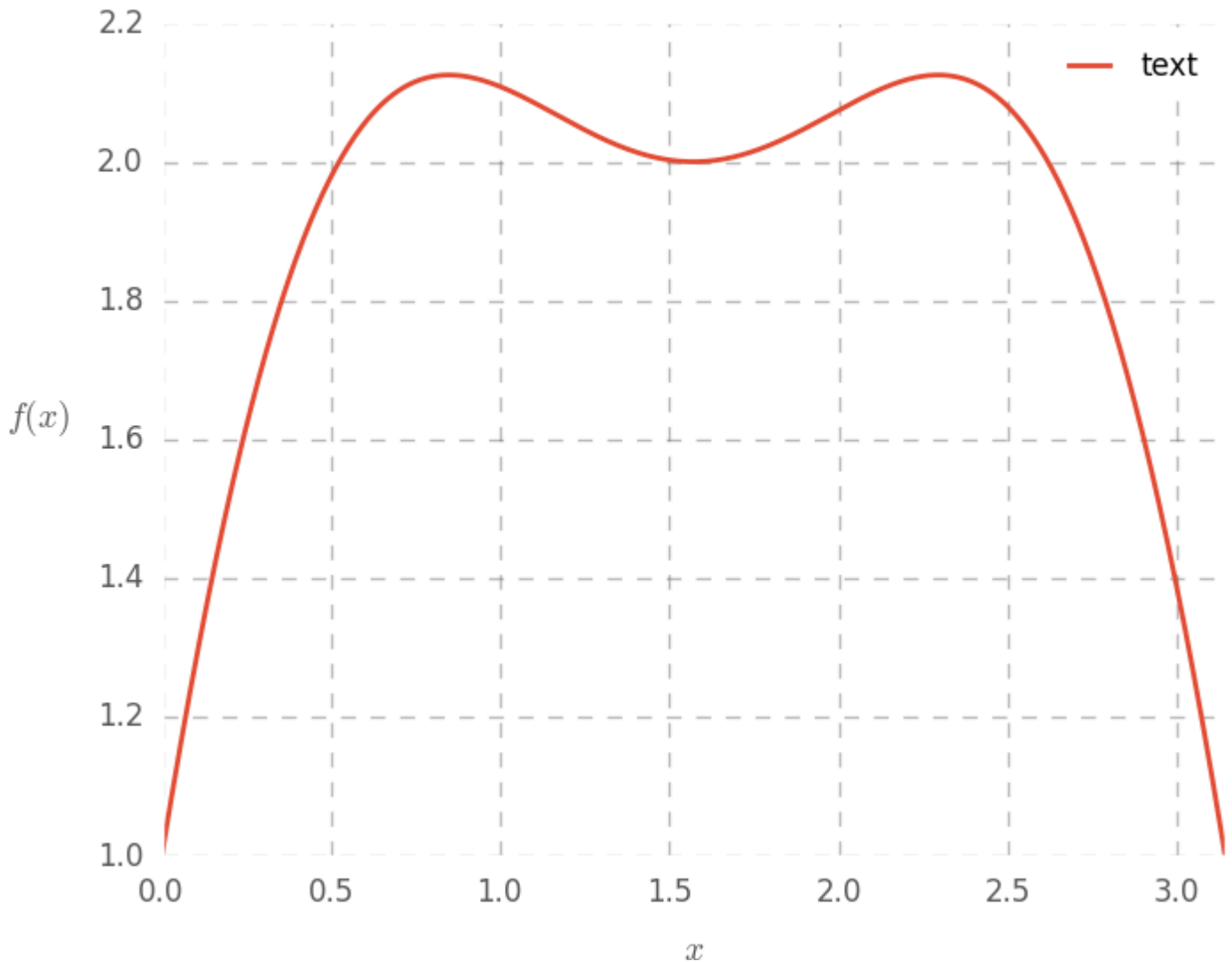
```



```

# Customize the plot
ax.grid(1, ls='--', color='#777777', alpha=0.5, lw=1)
ax.tick_params(labelsize=12, length=0)
ax.set_axis_bgcolor('w')
# add a legend
leg = plt.legend( ['text'], loc=1 )
fr = leg.get_frame()
fr.set_facecolor('w')
fr.set_alpha(.7)
plt.draw()

```



Imperative versus objektorientierte Syntax

Matplotlib unterstützt sowohl die objektorientierte als auch die imperative Syntax für das Plotten. Die imperative Syntax wurde bewusst so konzipiert, dass sie der Matlab-Syntax sehr nahe kommt.

Die imperative Syntax (manchmal auch als "State-Machine" -Syntax bezeichnet) gibt eine Befehlsfolge aus, die alle auf die neueste Figur oder Achse wirken (wie Matlab). Die objektorientierte Syntax wirkt dagegen explizit auf die interessierenden Objekte (Figur, Achse usw.) ein. Ein Schlüsselpunkt im [Zen von Python](#) besagt, dass explizit besser als implizit ist, sodass die objektorientierte Syntax pythonischer ist. Die imperative Syntax eignet sich jedoch für neue Konvertierungen aus Matlab und für das Schreiben kleiner "wegwerfender" Plot-Skripts. Im Folgenden finden Sie ein Beispiel für die beiden verschiedenen Stile.

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

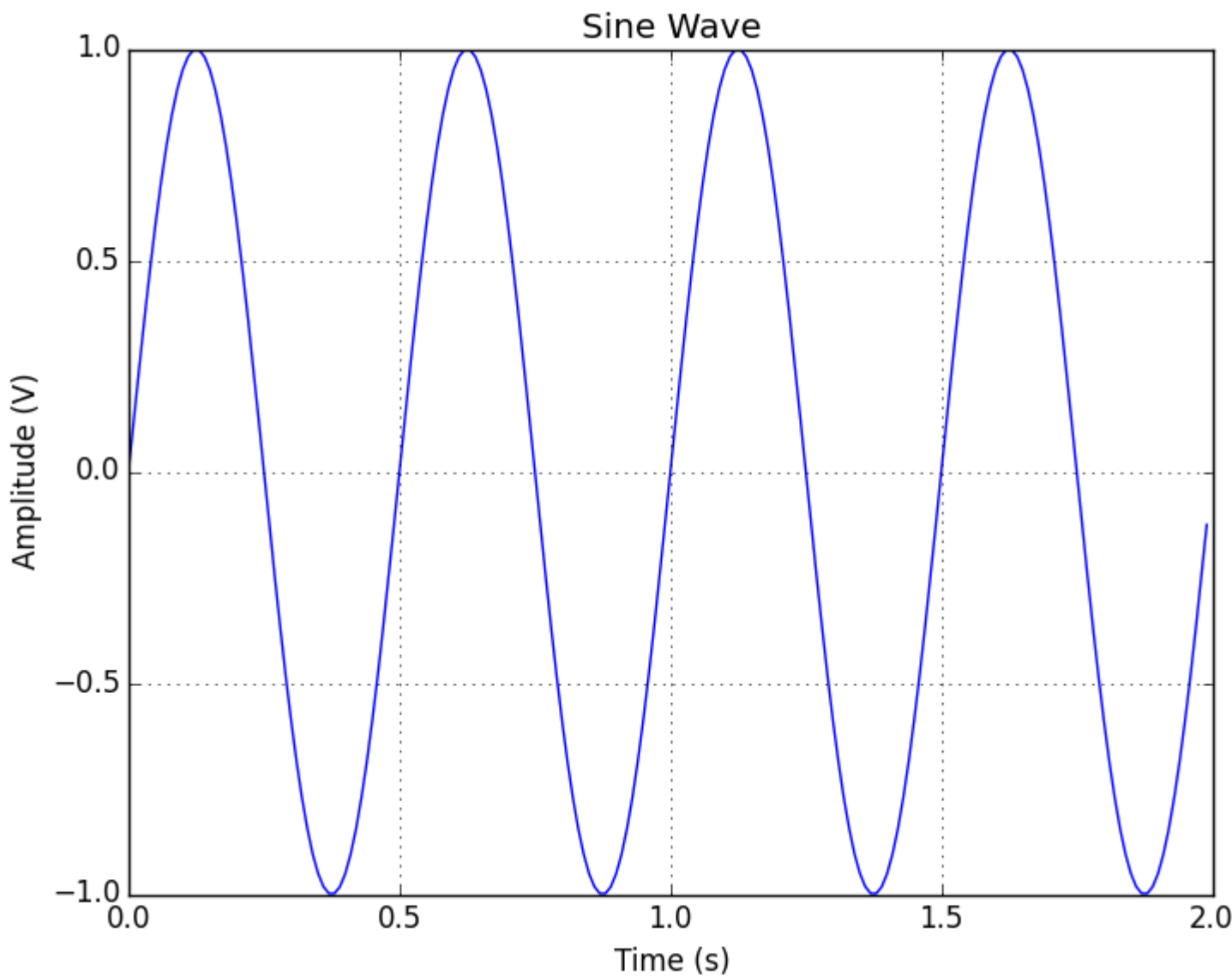
t = np.arange(0, 2, 0.01)
y = np.sin(4 * np.pi * t)

# Imperative syntax
```

```
plt.figure(1)
plt.clf()
plt.plot(t, y)
plt.xlabel('Time (s)')
plt.ylabel('Amplitude (V)')
plt.title('Sine Wave')
plt.grid(True)

# Object oriented syntax
fig = plt.figure(2)
fig.clf()
ax = fig.add_subplot(1,1,1)
ax.plot(t, y)
ax.set_xlabel('Time (s)')
ax.set_ylabel('Amplitude (V)')
ax.set_title('Sine Wave')
ax.grid(True)
```

Beide Beispiele erzeugen dieselbe Darstellung, die unten gezeigt wird.

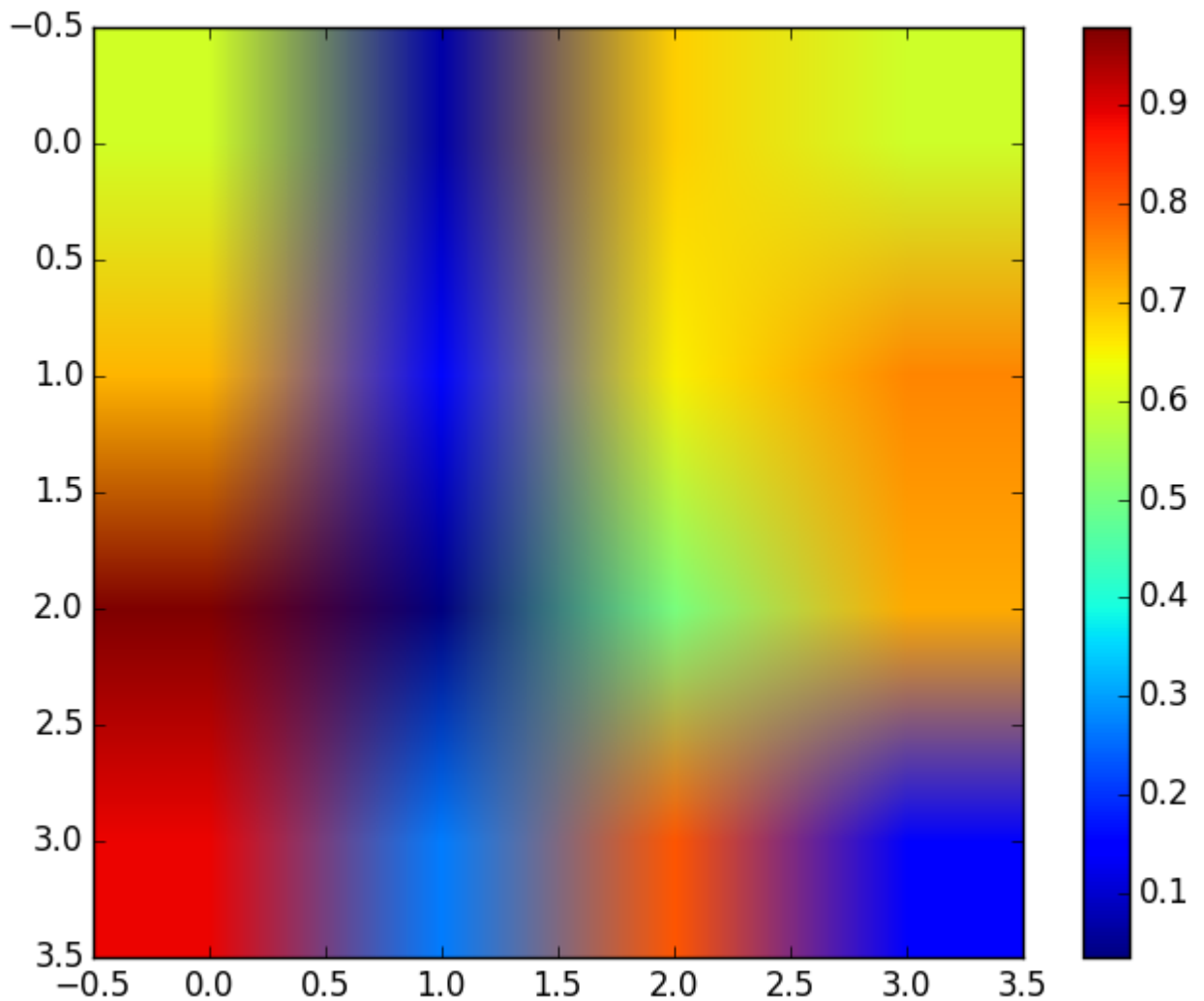


Zweidimensionale (2D) Arrays

Zeigen Sie ein zweidimensionales (2D) Array auf den Achsen an.

```
import numpy as np
from matplotlib.pyplot import imshow, show, colorbar

image = np.random.rand(4,4)
imshow(image)
colorbar()
show()
```



Erste Schritte mit matplotlib online lesen: <https://riptutorial.com/de/matplotlib/topic/881/erste-schritte-mit-matplotlib>

Kapitel 2: Animationen und interaktives Plotten

Einführung

Mit Python Matplotlib können Sie animierte Grafiken erstellen.

Examples

Grundanimation mit FuncAnimation

Das Paket [matplotlib.animation](#) bietet einige Klassen zum Erstellen von Animationen.

[FuncAnimation](#) erstellt Animationen durch wiederholtes Aufrufen einer Funktion. Hier verwenden wir eine Funktion `animate()`, die die Koordinaten eines Punktes im Graphen einer Sinusfunktion ändert.

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

TWOPI = 2*np.pi

fig, ax = plt.subplots()

t = np.arange(0.0, TWOPI, 0.001)
s = np.sin(t)
l = plt.plot(t, s)

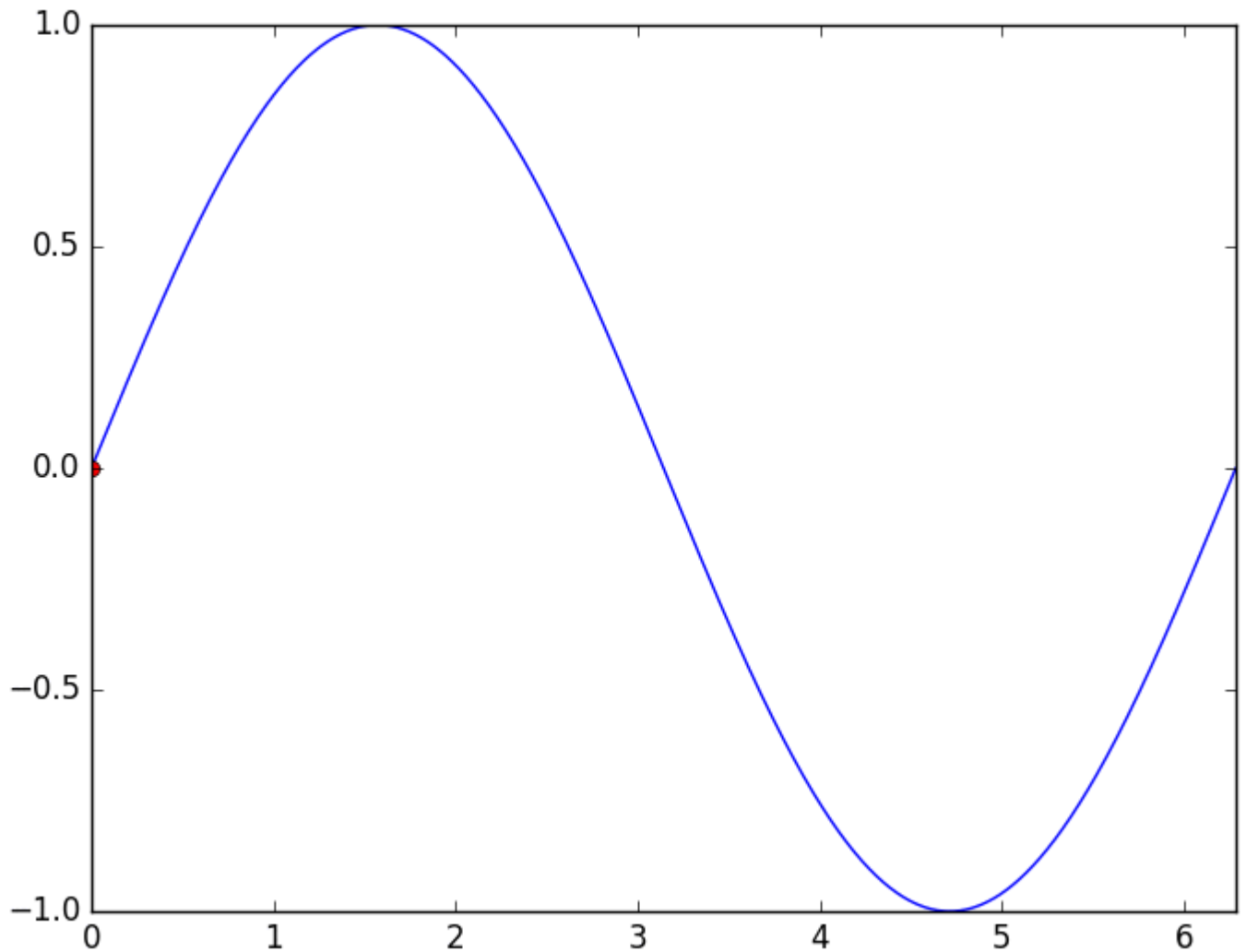
ax = plt.axis([0, TWOPI, -1, 1])

redDot, = plt.plot([0], [np.sin(0)], 'ro')

def animate(i):
    redDot.set_data(i, np.sin(i))
    return redDot,

# create animation using the animate() function
myAnimation = animation.FuncAnimation(fig, animate, frames=np.arange(0.0, TWOPI, 0.1), \
                                     interval=10, blit=True, repeat=True)

plt.show()
```



Animation in GIF speichern

In diesem Beispiel verwenden wir die `save`, um ein Animation mit ImageMagick zu speichern.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import rcParams

# make sure the full paths for ImageMagick and ffmpeg are configured
rcParams['animation.convert_path'] = r'C:\Program Files\ImageMagick\convert'
rcParams['animation.ffmpeg_path'] = r'C:\Program Files\ffmpeg\bin\ffmpeg.exe'

TWOPI = 2*np.pi

fig, ax = plt.subplots()

t = np.arange(0.0, TWOPI, 0.001)
s = np.sin(t)
l = plt.plot(t, s)
```

```

ax = plt.axis([0,TWOPI,-1,1])

redDot, = plt.plot([0], [np.sin(0)], 'ro')

def animate(i):
    redDot.set_data(i, np.sin(i))
    return redDot,

# create animation using the animate() function with no repeat
myAnimation = animation.FuncAnimation(fig, animate, frames=np.arange(0.0, TWOPI, 0.1), \
                                     interval=10, blit=True, repeat=False)

# save animation at 30 frames per second
myAnimation.save('myAnimation.gif', writer='imagemagick', fps=30)

```

Interaktive Steuerelemente mit matplotlib.widgets

Für die Interaktion mit Plots bietet Matplotlib GUI-neutrale [Widget](#) . Widgets erfordern ein `matplotlib.axes.Axes` Objekt.

Hier ist eine Slider-Widget-Demo, die die Amplitude einer Sinuskurve ändert. Die Aktualisierungsfunktion wird durch das Ereignis `on_changed()` des Sliders ausgelöst.

```

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.widgets import Slider

TWOPI = 2*np.pi

fig, ax = plt.subplots()

t = np.arange(0.0, TWOPI, 0.001)
initial_amp = .5
s = initial_amp*np.sin(t)
l, = plt.plot(t, s, lw=2)

ax = plt.axis([0,TWOPI,-1,1])

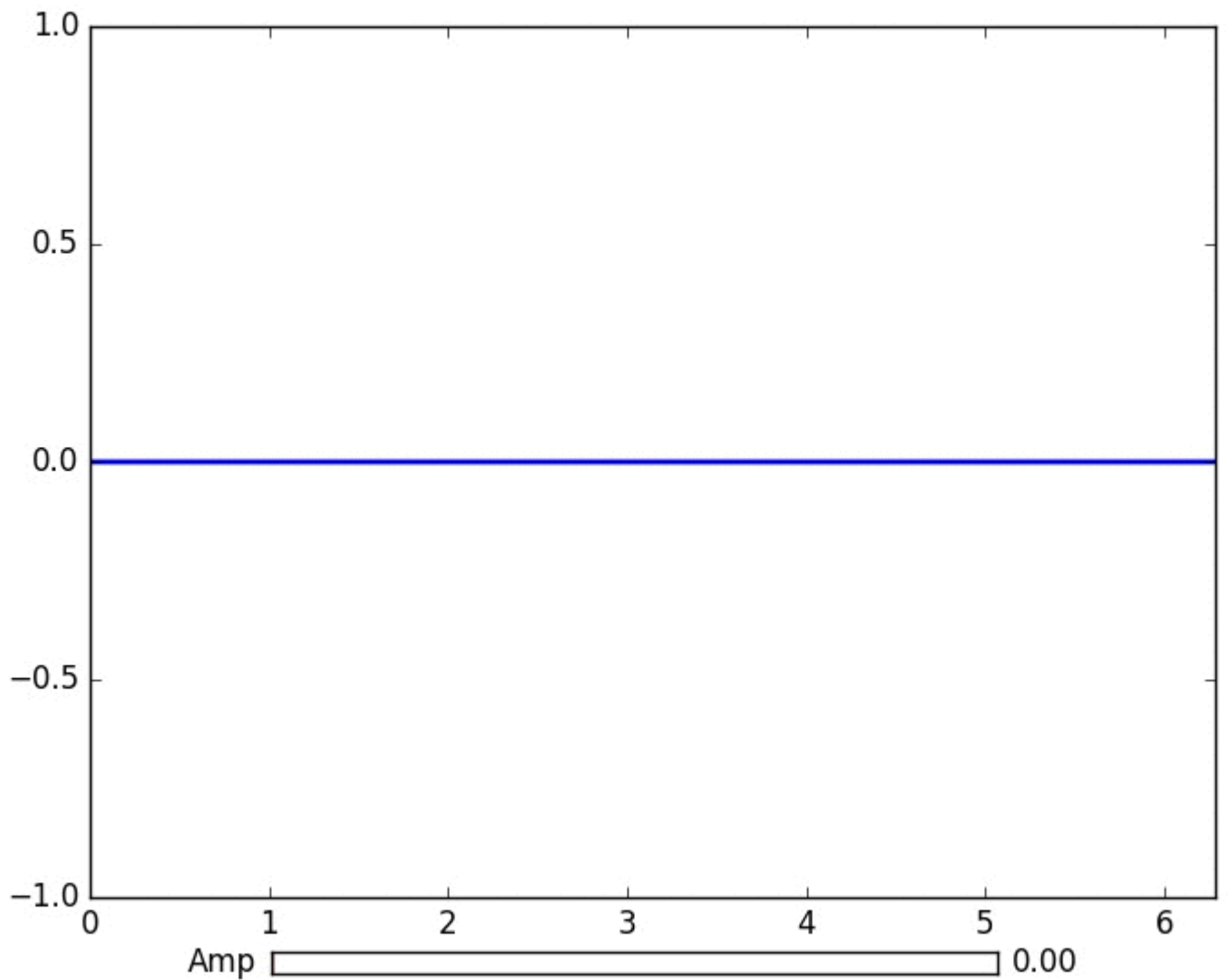
axamp = plt.axes([0.25, .03, 0.50, 0.02])
# Slider
samp = Slider(axamp, 'Amp', 0, 1, valinit=initial_amp)

def update(val):
    # amp is the current value of the slider
    amp = samp.val
    # update curve
    l.set_ydata(amp*np.sin(t))
    # redraw canvas while idle
    fig.canvas.draw_idle()

# call update function on slider value change
samp.on_changed(update)

plt.show()

```



Andere verfügbare Widgets:

- [AxesWidget](#)
- [Taste](#)
- [CheckButtons](#)
- [Mauszeiger](#)
- [EllipseSelector](#)
- [Lasso](#)
- [LassoSelector](#)
- [LockDraw](#)
- [MultiCursor](#)
- [Radio Knöpfe](#)
- [RechteckSelektor](#)
- [SpanSelector](#)
- [SubplotTool](#)
- [ToolHandles](#)

Live-Daten von Pipe mit Matplotlib darstellen

Dies kann nützlich sein, wenn Sie eingehende Daten in Echtzeit visualisieren möchten. Diese Daten könnten zum Beispiel von einem Mikrocontroller stammen, der ein analoges Signal kontinuierlich abtastet.

In diesem Beispiel erhalten wir unsere Daten von einer Named Pipe (auch als FIFO bezeichnet). In diesem Beispiel sollten die Daten in der Pipe Zahlen enthalten, die durch Zeilenumbrüche getrennt sind. Sie können dies jedoch an Ihre Wünsche anpassen.

Beispieldaten:

```
100
123.5
1589
```

Weitere Informationen zu Named Pipes

Wir werden auch den Datentyp deque aus den Standardbibliothekensammlungen verwenden. Ein Deque-Objekt funktioniert ziemlich ähnlich wie eine Liste. Mit einem Deque-Objekt ist es jedoch recht einfach, etwas an das Objekt anzuhängen, während das Deque-Objekt auf einer festen Länge bleibt. Dies ermöglicht es uns, die x-Achse auf einer festen Länge zu halten, anstatt den Graphen immer zu vergrößern und zu zerquetschen. [Weitere Informationen zu Deque-Objekten](#)

Die Wahl des richtigen Backends ist entscheidend für die Leistung. Prüfen Sie, welche Backends auf Ihrem Betriebssystem funktionieren, und wählen Sie ein schnelles aus. Für mich funktionierten nur qt4agg und das Standard-Backend, aber der Standard war zu langsam. [Weitere Informationen zu den Backends in Matplotlib](#)

Dieses Beispiel basiert auf [dem Matplotlib-Beispiel für das Aufzeichnen von Zufallsdaten](#) .

Keines der 'Zeichen in diesem Code soll entfernt werden.

```
import matplotlib
import collections
#selecting the right backend, change qt4agg to your desired backend
matplotlib.use('qt4agg')
import matplotlib.pyplot as plt
import matplotlib.animation as animation

#command to open the pipe
datapipe = open('path to your pipe','r')

#amount of data to be displayed at once, this is the size of the x axis
#increasing this amount also makes plotting slightly slower
data_amount = 1000

#set the size of the deque object
datalist = collections.deque([0]*data_amount,data_amount)

#configure the graph itself
fig, ax = plt.subplots()
line, = ax.plot([0,]*data_amount)

#size of the y axis is set here
ax.set_ylim(0,256)
```

```

def update(data):
    line.set_ydata(data)
    return line,

def data_gen():
    while True:
        """
        We read two data points in at once, to improve speed
        You can read more at once to increase speed
        Or you can read just one at a time for improved animation smoothness
        data from the pipe comes in as a string,
        and is seperated with a newline character,
        which is why we use respectively eval and rstrip.
        """
        datalist.append(eval((datapipe.readline()).rstrip('\n')))
        datalist.append(eval((datapipe.readline()).rstrip('\n')))
        yield datalist

ani = animation.FuncAnimation(fig, update, data_gen, interval=0, blit=True)
plt.show()

```

Wenn Ihr Diagramm nach einiger Zeit verzögert wird, fügen Sie weitere Daten von `datalist.append` hinzu, sodass in jedem Frame mehr Zeilen gelesen werden. Oder wählen Sie ein schnelleres Backend, wenn Sie können.

Dies funktionierte mit 150 Hz-Daten aus einer Pipe auf meinem 1,7-GHz-i3 4005u.

Animationen und interaktives Plotten online lesen:

<https://riptutorial.com/de/matplotlib/topic/6983/animationen-und-interaktives-plotten>

Kapitel 3: Bildbearbeitung

Examples

Bilder öffnen

Matplotlib enthält das `image` zur Bildmanipulation

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
```

Bilder werden mit der `imread` Funktion aus der Datei gelesen (nur `.png`):

```
img = mpimg.imread('my_image.png')
```

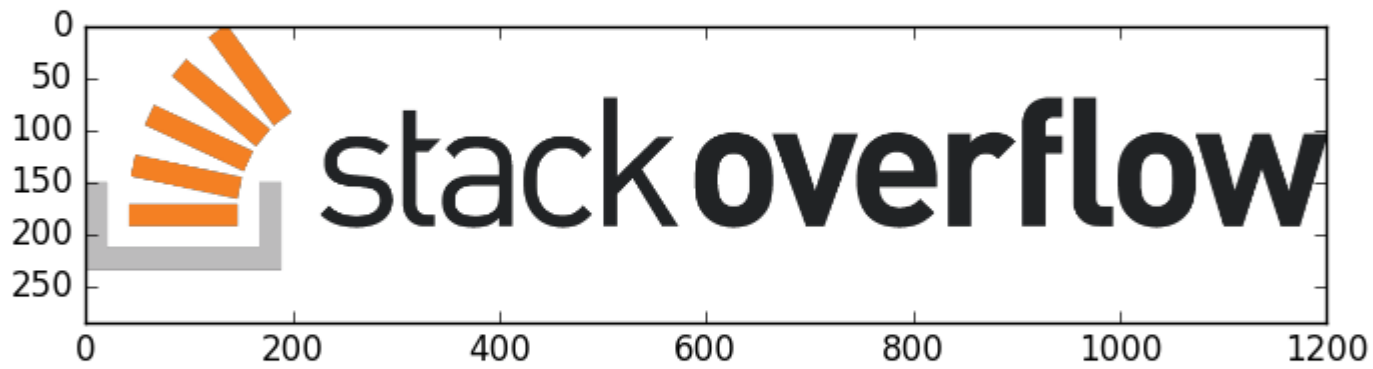
und sie werden von der `imshow` Funktion gerendert:

```
plt.imshow(img)
```

Lassen Sie uns das *Grundstück* [Stack - Überlauf - Logo](#) :

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
img = mpimg.imread('so-logo.png')
plt.imshow(img)
plt.show()
```

Die resultierende Darstellung ist



Bildbearbeitung online lesen: <https://riptutorial.com/de/matplotlib/topic/4575/bildbearbeitung>

Kapitel 4: Boxplots

Examples

Grundlegende Boxplots

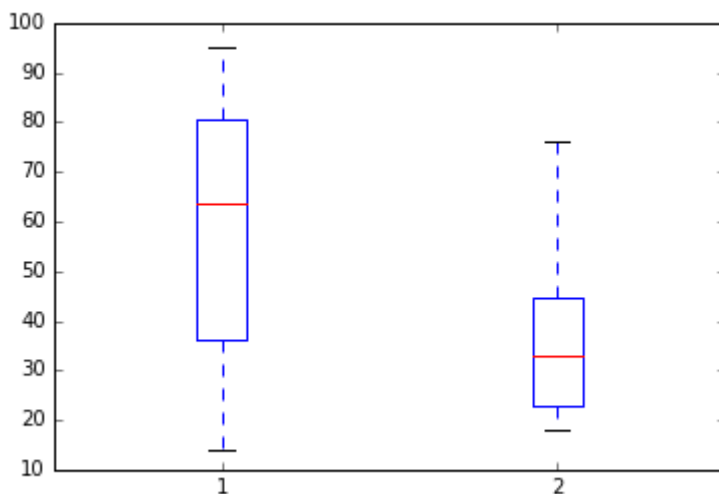
Boxplots sind beschreibende Diagramme, mit deren Hilfe die Verteilung verschiedener Datenreihen verglichen werden kann. Sie sind *deskriptiv*, weil sie Messwerte (z. B. den *Median*) *anzeigen*, die keine zugrunde liegende Wahrscheinlichkeitsverteilung annehmen.

Das einfachste Beispiel eines Boxplots in matplotlib kann durch einfaches Übergeben der Daten als Listenliste erreicht werden:

```
import matplotlib as plt

dataline1 = [43,76,34,63,56,82,87,55,64,87,95,23,14,65,67,25,23,85]
dataline2 = [34,45,34,23,43,76,26,18,24,74,23,56,23,23,34,56,32,23]
data = [ dataline1, dataline2 ]

plt.boxplot( data )
```

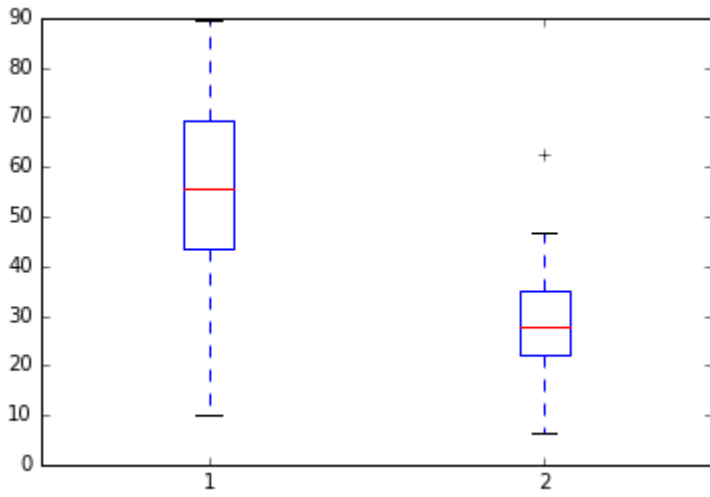


Es ist jedoch üblich, `numpy` Arrays als Parameter für die `numpy` zu verwenden, da diese häufig das Ergebnis vorheriger Berechnungen sind. Dies kann wie folgt durchgeführt werden:

```
import numpy as np
import matplotlib as plt

np.random.seed(123)
dataline1 = np.random.normal( loc=50, scale=20, size=18 )
dataline2 = np.random.normal( loc=30, scale=10, size=18 )
data = np.stack( [ dataline1, dataline2 ], axis=1 )

plt.boxplot( data )
```



Boxplots online lesen: <https://riptutorial.com/de/matplotlib/topic/6086/boxplots>

Kapitel 5: Boxplots

Examples

Boxplot-Funktion

Matplotlib hat eine eigene Implementierung von **Boxplot**. Der relevante Aspekt dieser Funktion ist, dass das Boxplot standardmäßig den Median (Perzentil 50%) mit einer roten Linie anzeigt. Das Kästchen steht für Q1 und Q3 (Perzentile 25 und 75), und die Whisker geben eine Vorstellung von dem Bereich der Daten (möglicherweise bei $Q1 - 1,5 IQR$; $Q3 + 1,5 IQR$; IQR ist der Interquartilbereich, aber dies hat keine Bestätigung). Beachten Sie auch, dass Proben außerhalb dieses Bereichs als Marker angezeigt werden (diese werden Flieger genannt).

HINWEIS: Nicht alle Implementierungen von *Boxplot* folgen den gleichen Regeln. Möglicherweise verwendet das allgemeinste Boxplot-Diagramm die Whisker, um das Minimum und das Maximum darzustellen (wodurch Flieger nicht existieren). Beachten Sie auch, dass dieses Diagramm manchmal als *Box-Whisker-Diagramm* und *Box-Whisker-Diagramm bezeichnet wird*.

Das folgende Rezept zeigt einige der Möglichkeiten, die Sie mit der aktuellen Matplotlib-Implementierung von Boxplot machen können:

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

X1 = np.random.normal(0, 1, 500)
X2 = np.random.normal(0.3, 1, 500)

# The most simple boxplot
plt.boxplot(X1)
plt.show()

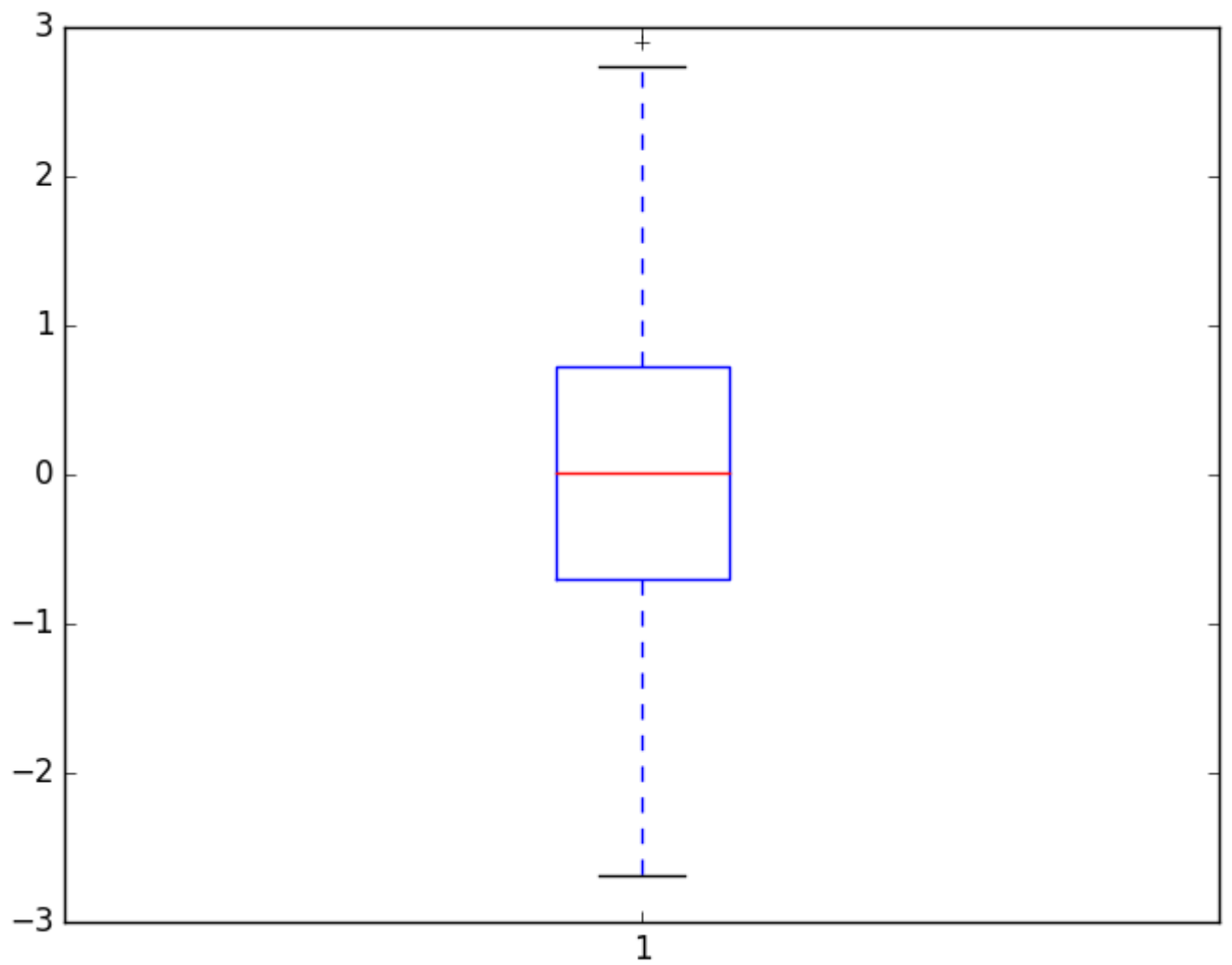
# Changing some of its features
plt.boxplot(X1, notch=True, sym="o") # Use sym="" to shown no fliers; also showfliers=False
plt.show()

# Showing multiple boxplots on the same window
plt.boxplot((X1, X2), notch=True, sym="o", labels=["Set 1", "Set 2"])
plt.show()

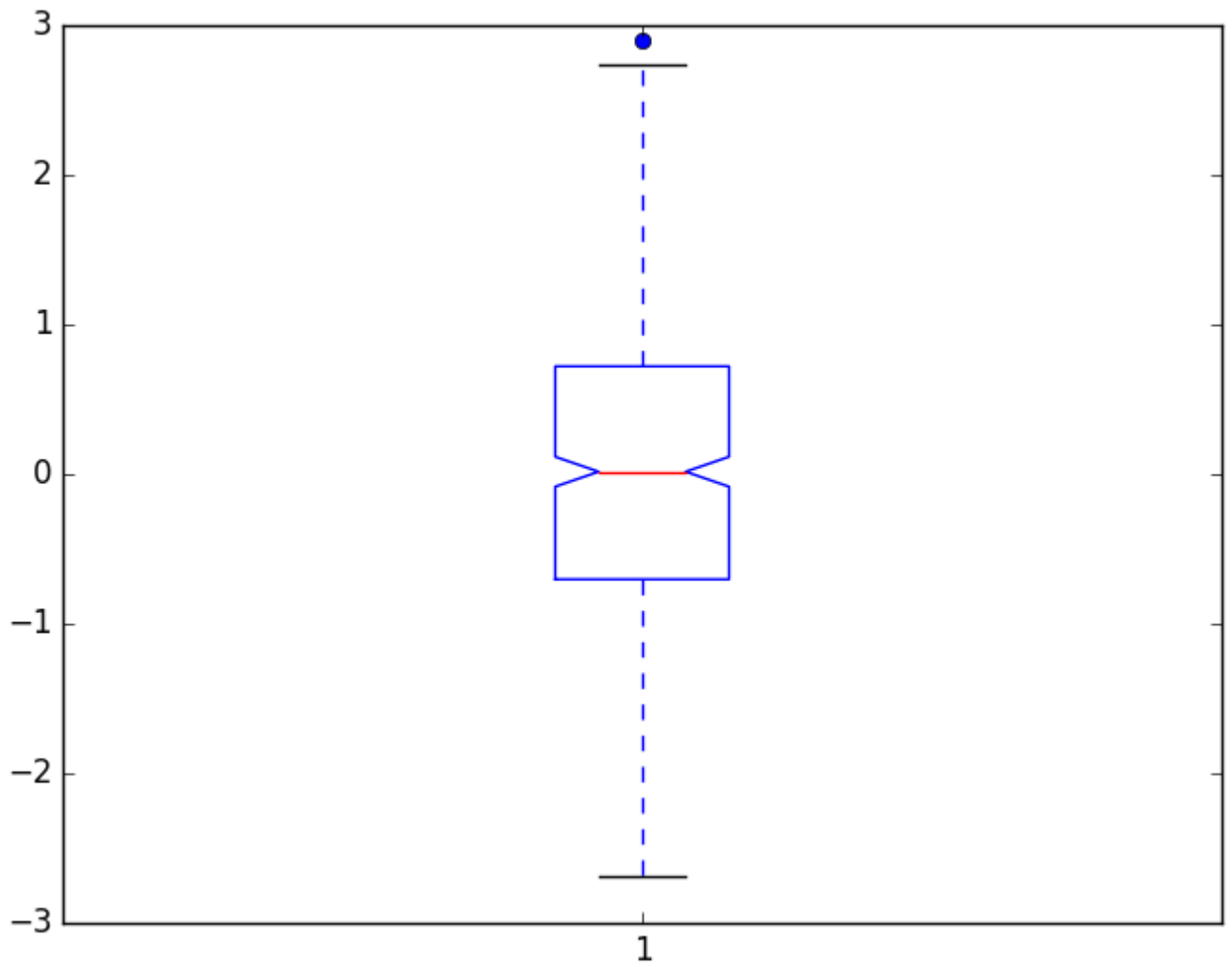
# Hidding features of the boxplot
plt.boxplot(X2, notch=False, showfliers=False, showbox=False, showcaps=False, positions=[4],
labels=["Set 2"])
plt.show()

# Advanced customization of the boxplot
line_props = dict(color="r", alpha=0.3)
bbox_props = dict(color="g", alpha=0.9, linestyle="dashdot")
flier_props = dict(marker="o", markersize=17)
plt.boxplot(X1, notch=True, whiskerprops=line_props, boxprops=bbox_props,
flierprops=flier_props)
plt.show()
```

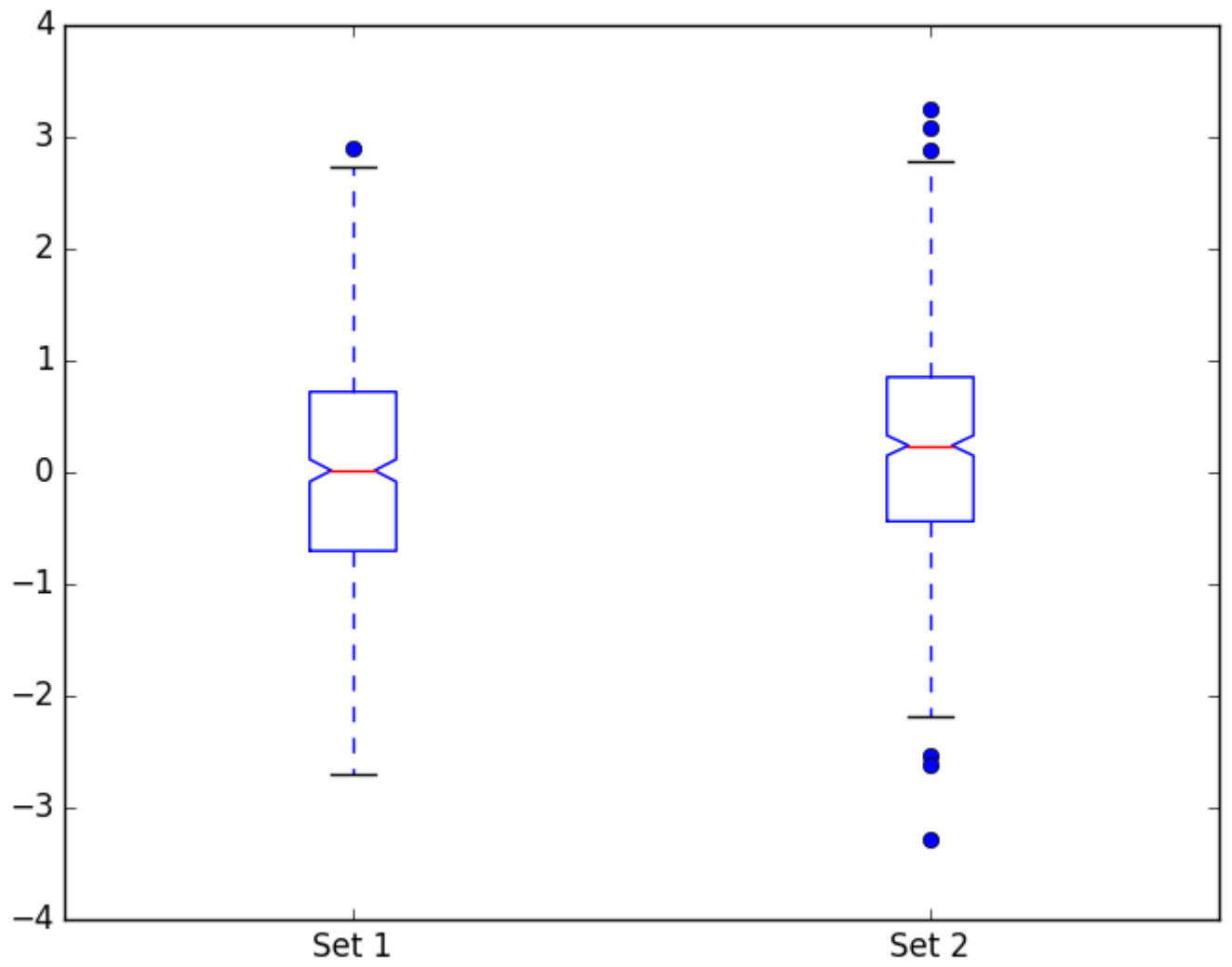
Dies führt zu folgenden Darstellungen:



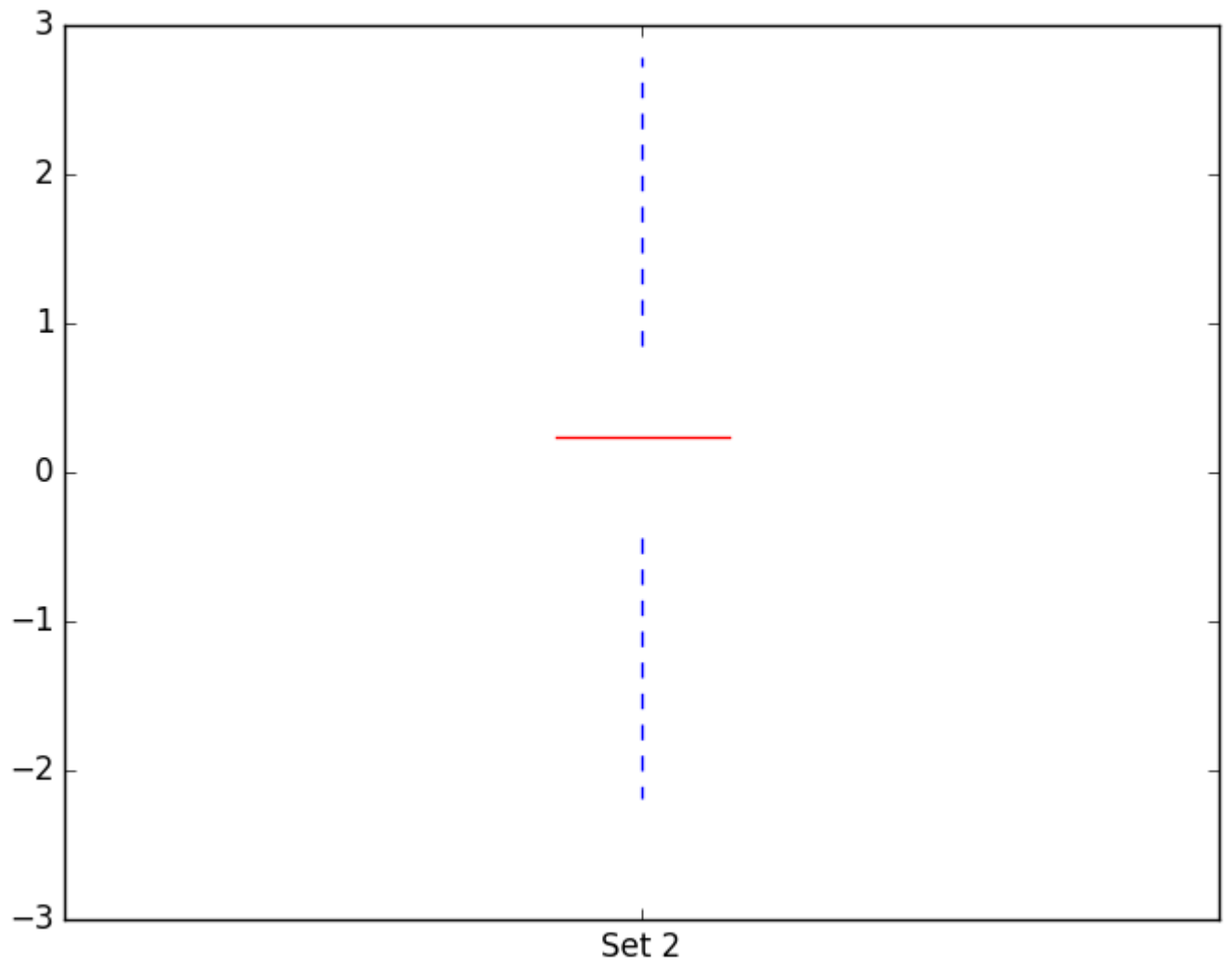
1. *Standard-Matplotlib-Boxplot*



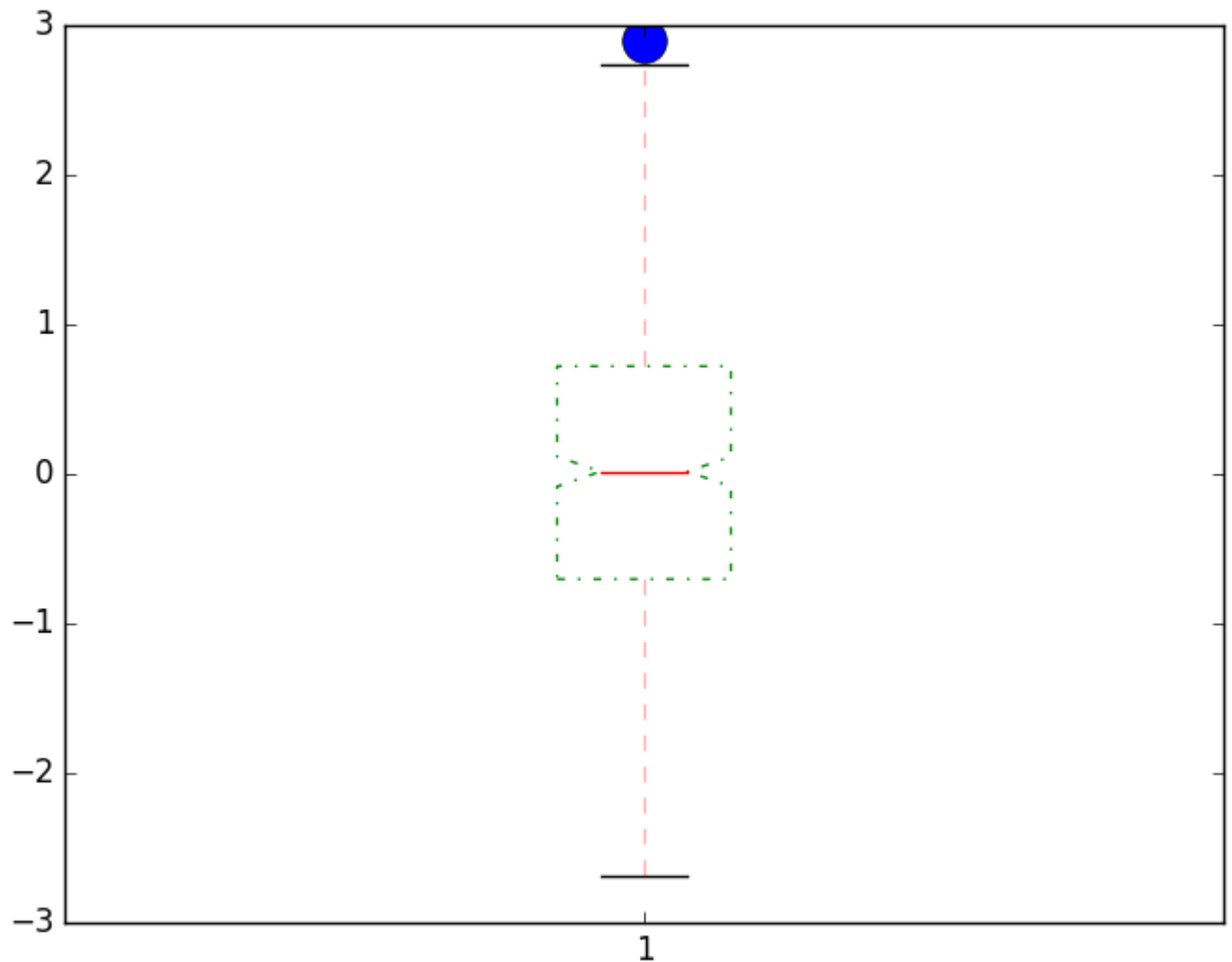
2. Einige Funktionen des Boxplots mit Funktionsargumenten ändern



3. Mehrere Boxplots im selben Plotfenster



4. Einige Funktionen des Boxplots verstecken



5. Erweiterte Anpassung eines Boxplots mit Requisiten

Wenn Sie beabsichtigen, eine erweiterte Anpassung Ihrer boxplot zu tun, sollten Sie wissen, dass die [Requisiten](#) Wörterbücher, die Sie (zum Beispiel) bauen:

```
line_props = dict(color="r", alpha=0.3)
bbox_props = dict(color="g", alpha=0.9, linestyle="dashdot")
flier_props = dict(marker="o", markersize=17)
plt.boxplot(X1, notch=True, whiskerprops=line_props, boxprops=bbox_props,
            flierprops=flier_props)
plt.show()
```

... beziehen sich meistens (wenn nicht alle) auf [Line2D](#)- Objekte. Das bedeutet, dass nur die in dieser Klasse verfügbaren Argumente geändert werden können. Sie werden feststellen, dass Schlüsselwörter wie `whiskerprops`, `boxprops`, `flierprops` und `capprops`. Dies sind die Elemente, die Sie benötigen, um ein Props-Wörterbuch bereitzustellen, um es weiter anzupassen.

HINWEIS: Eine weitere Anpassung des Boxplots mit dieser Implementierung kann sich als schwierig erweisen. In einigen Fällen kann die Verwendung anderer Matplotlib-

Elemente wie [Patches](#) zum Erstellen eines eigenen Boxplots von Vorteil sein (z. B. erhebliche Änderungen am Boxelement).

Boxplots online lesen: <https://riptutorial.com/de/matplotlib/topic/6368/boxplots>

Kapitel 6: Colormaps

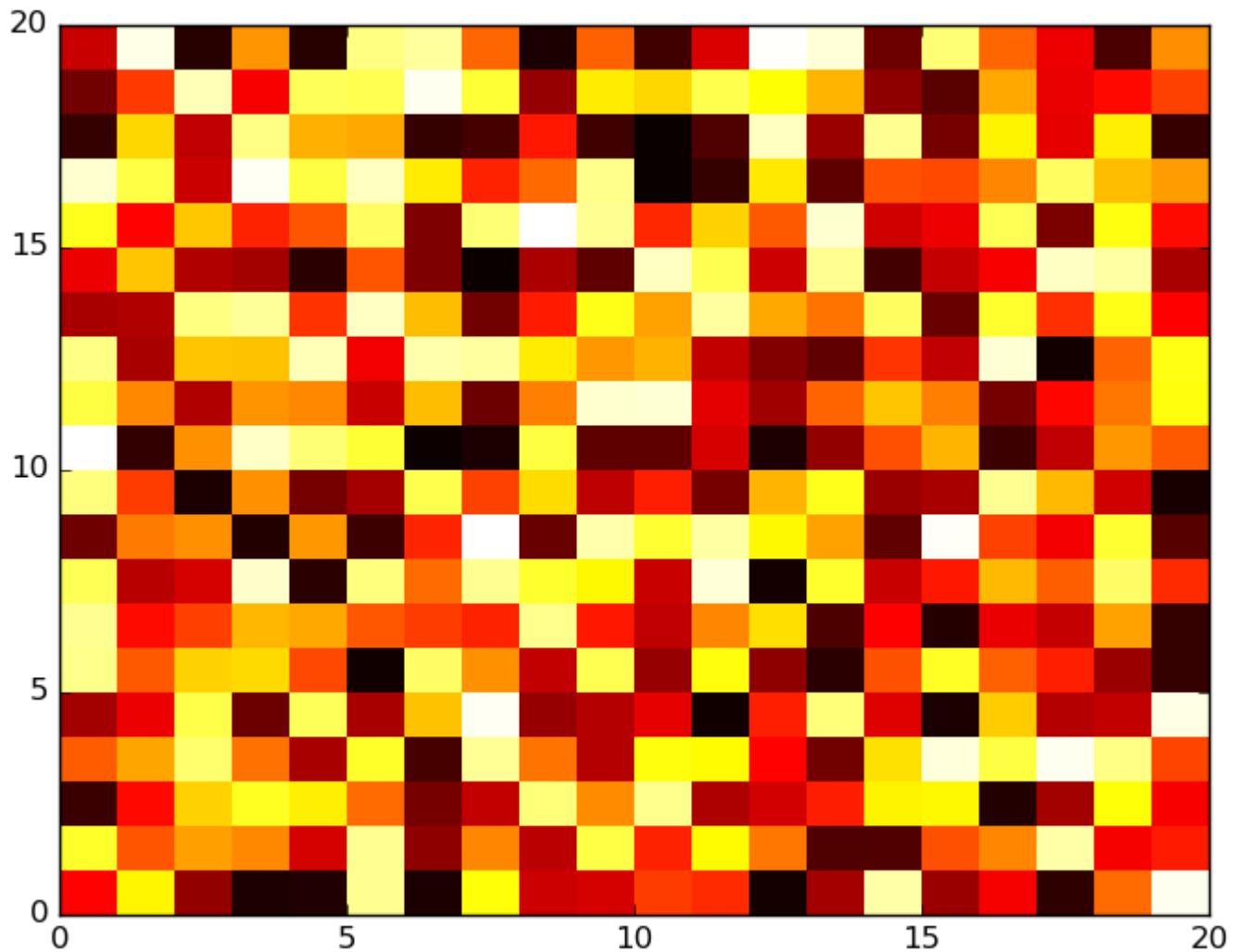
Examples

Grundlegende Verwendung

Die Verwendung von eingebauten Colormaps besteht einfach darin, den Namen der erforderlichen Colormap (wie in [der Colormaps-Referenz angegeben](#)) an die `pcolormesh` Funktion (wie `pcolormesh` oder `contourf`) zu übergeben, die dies erwartet, normalerweise in Form eines `cmap` Schlüsselwortarguments:

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

plt.figure()
plt.pcolormesh(np.random.rand(20,20), cmap='hot')
plt.show()
```



Colormaps sind besonders nützlich, um dreidimensionale Daten in zweidimensionalen Darstellungen zu visualisieren, eine gute Farbkarte kann jedoch auch eine korrekte dreidimensionale Darstellung viel klarer machen:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator

# generate example data
import numpy as np
x,y = np.meshgrid(np.linspace(-1,1,15), np.linspace(-1,1,15))
z = np.cos(x*np.pi)*np.sin(y*np.pi)

# actual plotting example
fig = plt.figure()
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(x,y,z,rstride=1,cstride=1,cmap='viridis')
ax2 = fig.add_subplot(122)
cf = ax2.contourf(x,y,z, 51, vmin=-1, vmax=1, cmap='viridis')
cbar = fig.colorbar(cf)
cbar.locator = LinearLocator(numticks=11)
cbar.update_ticks()
```

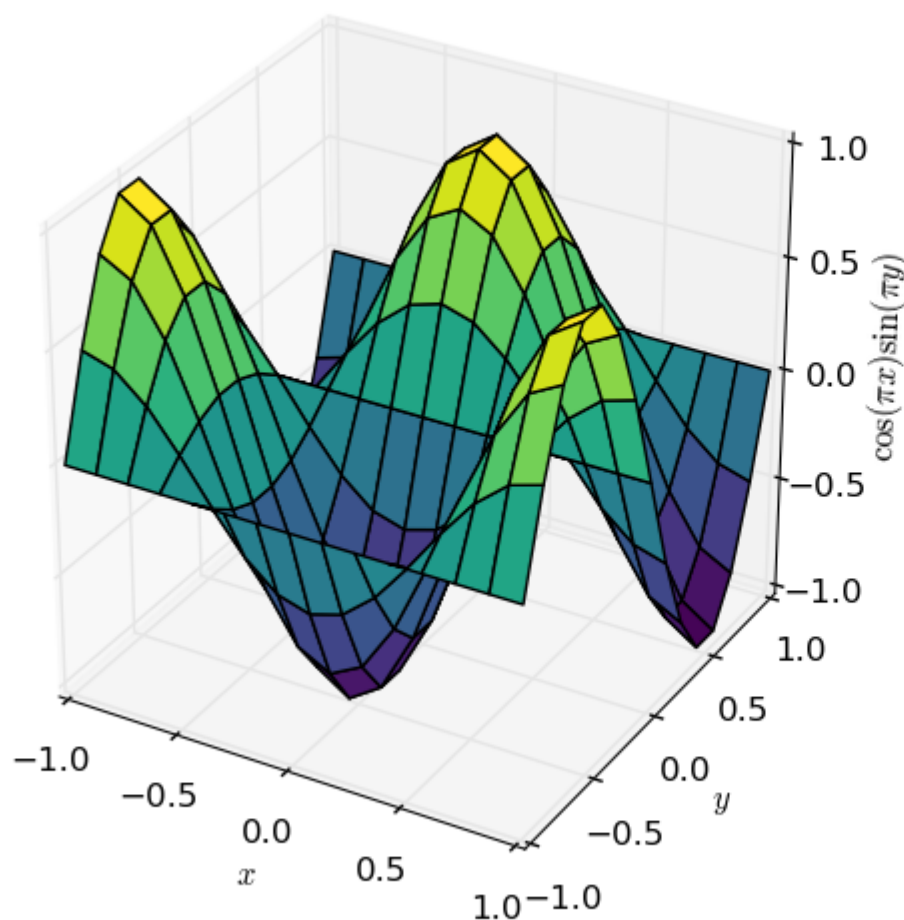
```

for ax in {ax1, ax2}:
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'$y$')
    ax.set_xlim([-1,1])
    ax.set_ylim([-1,1])
    ax.set_aspect('equal')

ax1.set_zlim([-1,1])
ax1.set_zlabel(r'$\cos(\pi x) \sin(\pi y)$')

plt.show()

```



Benutzerdefinierte Colormaps verwenden

Neben den integrierten Colormaps, die in [der Colormaps-Referenz definiert sind](#) (und deren umgekehrten Maps, an deren Namen `'_r'` angehängt ist), können auch benutzerdefinierte Colormaps definiert werden. Der Schlüssel ist das `matplotlib.cm` Modul.

Das folgende Beispiel definiert eine sehr einfache Farbzuoordnung mit `cm.register_cmap`, die eine einzelne Farbe enthält, wobei die Opazität (Alpha-Wert) der Farbe im Datenbereich zwischen

vollständig undurchsichtig und vollständig transparent interpoliert wird. Beachten Sie, dass die wichtigsten Linien aus Sicht der Colormap der Import von `cm`, der Aufruf von `register_cmap` und die Übergabe der Colormap an `plot_surface`.

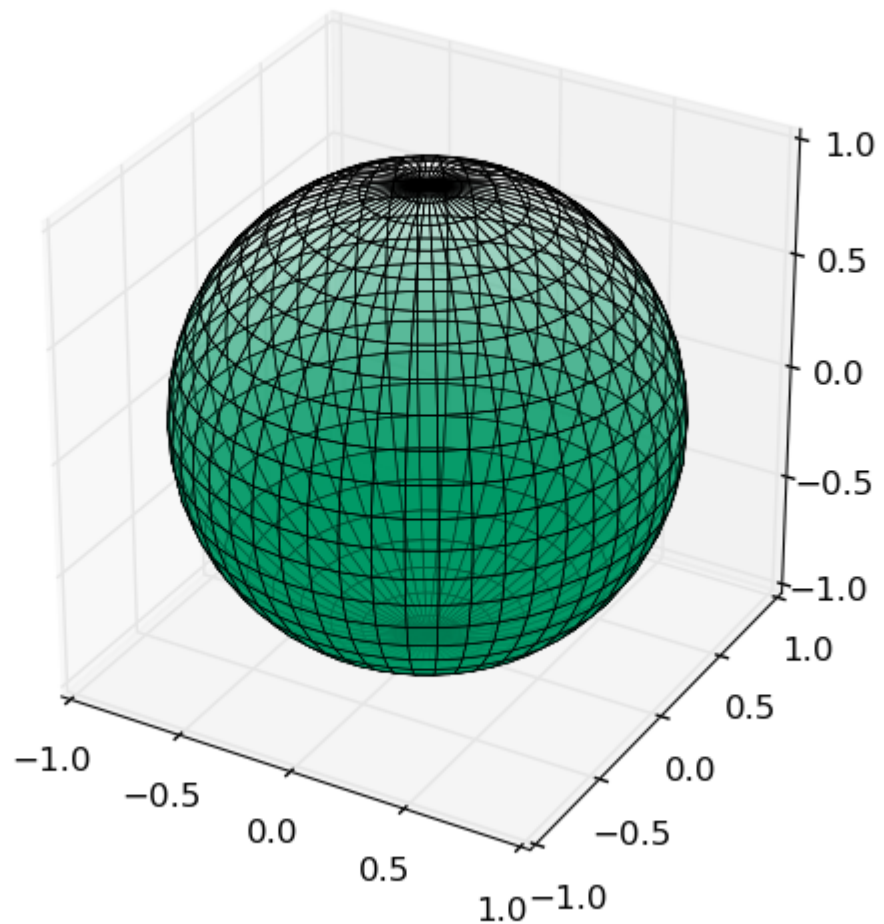
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm

# generate data for sphere
from numpy import pi, meshgrid, linspace, sin, cos
th, ph = meshgrid(linspace(0, pi, 25), linspace(0, 2*pi, 51))
x, y, z = sin(th)*cos(ph), sin(th)*sin(ph), cos(th)

# define custom colormap with fixed colour and alpha gradient
# use simple linear interpolation in the entire scale
cm.register_cmap(name='alpha_gradient',
                 data={'red': [(0., 0, 0),
                              (1., 0, 0)],
                       'green': [(0., 0.6, 0.6),
                                  (1., 0.6, 0.6)],
                       'blue': [(0., 0.4, 0.4),
                                 (1., 0.4, 0.4)],
                       'alpha': [(0., 1, 1),
                                  (1., 0, 0)]})

# plot sphere with custom colormap; constrain mapping to between |z|=0.7 for enhanced effect
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, cmap='alpha_gradient', vmin=-0.7, vmax=0.7, rstride=1, cstride=1, linewidth=0.5, edgecolor='b')
ax.set_xlim([-1, 1])
ax.set_ylim([-1, 1])
ax.set_zlim([-1, 1])
ax.set_aspect('equal')

plt.show()
```



In komplizierteren Szenarien kann eine Liste von R / G / B (/ A) -Werten definiert werden, in die Matplotlib linear interpoliert, um die in den entsprechenden Plots verwendeten Farben zu bestimmen.

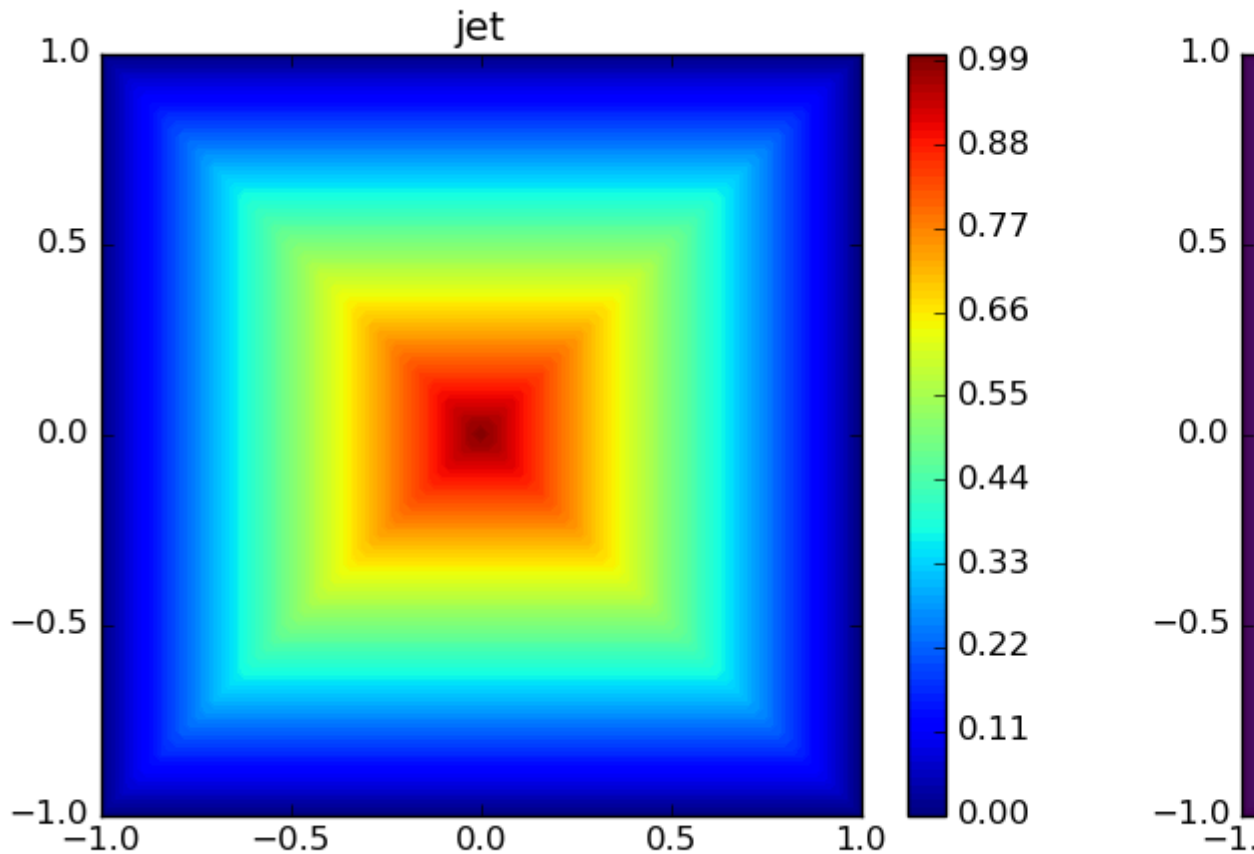
Wahrnehmungsmäßig einheitliche Farbkarten

Die ursprüngliche Standardfarbkarte von MATLAB (ersetzt in Version R2014b) mit dem Namen `jet` ist aufgrund ihres hohen Kontrasts und ihrer Bekanntheit allgegenwärtig (und war aus Kompatibilitätsgründen die Standardeinstellung von Matplotlib). Trotz seiner Beliebtheit [weisen herkömmliche Colormaps häufig Mängel auf](#), wenn es darum geht, Daten genau darzustellen. Die wahrgenommene Änderung in diesen Colormaps entspricht nicht den Datenänderungen. und eine Konvertierung der Farbkarte in Graustufen (z. B. durch Drucken einer Abbildung mit einem Schwarzweißdrucker) kann zu Informationsverlust führen.

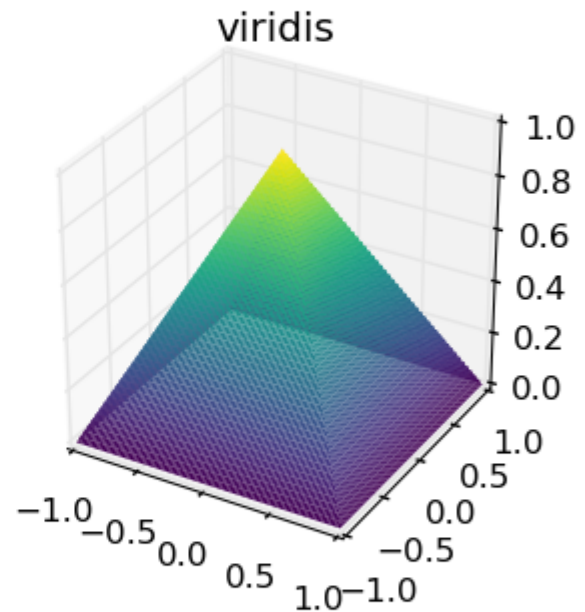
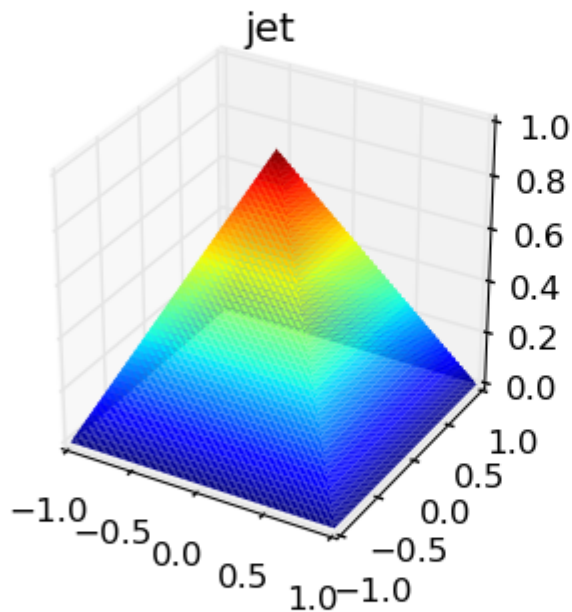
Perzeptuell einheitliche Colormaps wurden eingeführt, um die Datenvisualisierung so genau und zugänglich wie möglich zu machen. Matplotlib [hat](#) in Version 1.5 [vier neue, perzeptuell einheitliche Colormaps eingeführt](#), von denen eine (namens `viridis`) der Standard von Version 2.0 ist. Diese vier Farbkarten (`viridis`, `inferno`, `plasma` und `magma`) sind alle aus Sicht der Wahrnehmung

optimal. Diese sollten standardmäßig für die Datenvisualisierung verwendet werden, es sei denn, es gibt sehr gute Gründe, dies nicht zu tun. Diese Colormaps führen so wenig Verzerrung wie möglich aus (indem keine Features erstellt werden, für die es anfangs keine gibt), und sie eignen sich für ein Publikum mit eingeschränkter Farbwahrnehmung.

Betrachten Sie als Beispiel für die visuelle Verzerrung von Daten die folgenden zwei Draufsichtdiagramme von pyramidenartigen Objekten:



Welche der beiden ist eine richtige Pyramide? Die Antwort ist natürlich, dass beide das sind, aber dies ist bei der Darstellung mit der `jet` Colormap alles andere als offensichtlich:



Dieses Merkmal ist der Kern der Wahrnehmungsgleichförmigkeit.

Benutzerdefinierte diskrete Colormap

Wenn Sie vordefinierte Bereiche festgelegt haben und bestimmte Farben für diese Bereiche verwenden möchten, können Sie eine benutzerdefinierte Farbzuoordnung festlegen. Zum Beispiel:

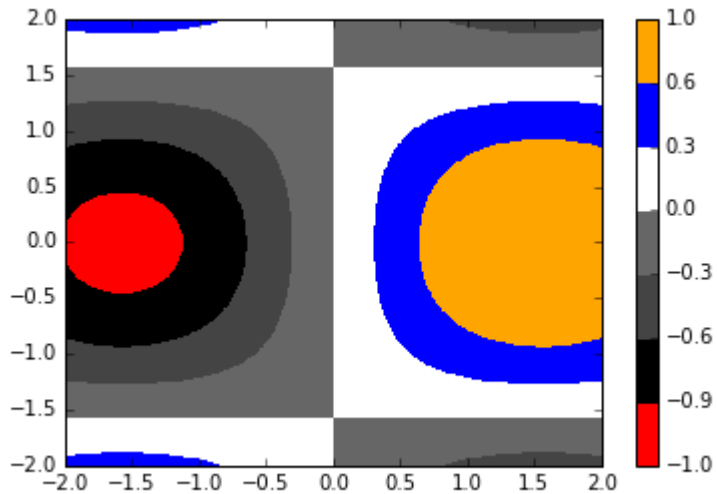
```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.colors

x = np.linspace(-2,2,500)
y = np.linspace(-2,2,500)
XX, YY = np.meshgrid(x, y)
Z = np.sin(XX) * np.cos(YY)

cmap = colors.ListedColormap(['red', '#000000', '#444444', '#666666', '#ffffff', 'blue',
'orange'])
boundaries = [-1, -0.9, -0.6, -0.3, 0, 0.3, 0.6, 1]
norm = colors.BoundaryNorm(boundaries, cmap.N, clip=True)
```

```
plt.pcolormesh(x,y,Z, cmap=cmap, norm=norm)
plt.colorbar()
plt.show()
```

Produziert



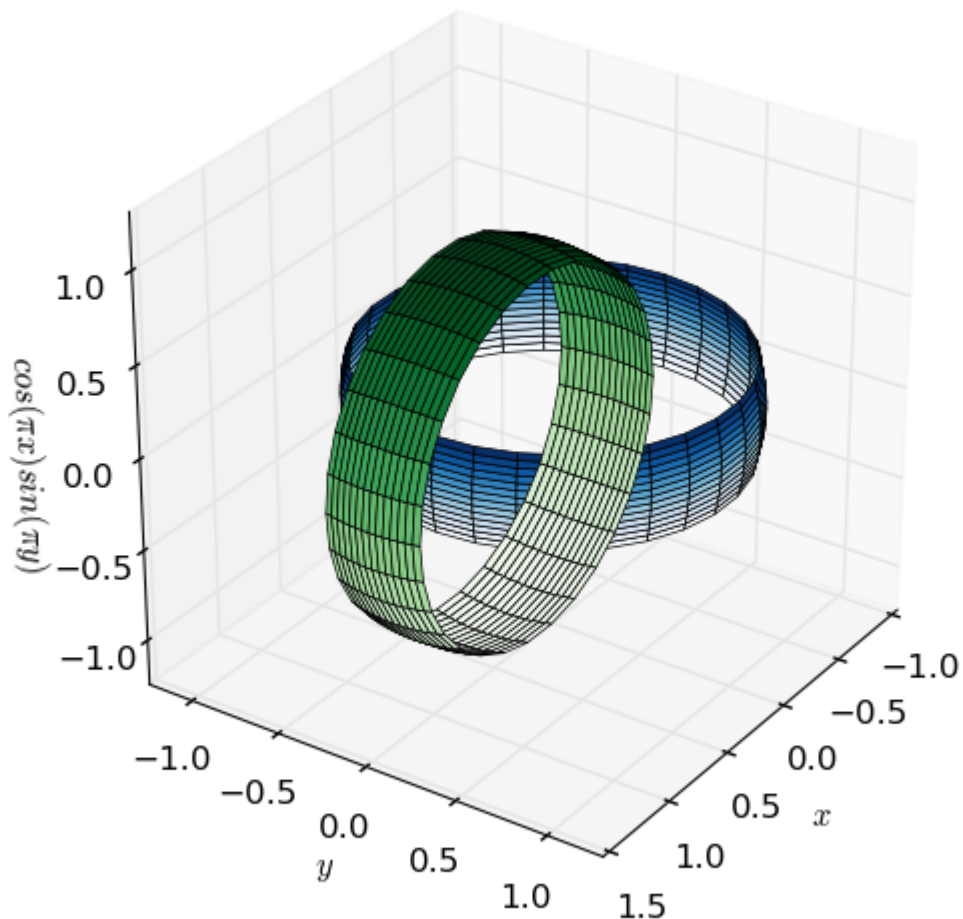
Die Farbe i wird für Werte zwischen der Grenze i und $i + 1$ verwendet. Farben können durch Namen ('red' , 'green'), HTML-Codes ('#ffaa44' , '#441188') oder RGB-Tupel ((0.2, 0.9, 0.45)) angegeben werden.

Colormaps online lesen: <https://riptutorial.com/de/matplotlib/topic/3385/colormaps>

Kapitel 7: Dreidimensionale Diagramme

Bemerkungen

Das dreidimensionale Plotten in Matplotlib war in der Vergangenheit etwas schwierig, da die Rendering-Engine inhärent 2d ist. Die Tatsache, dass 3D-Setups durch das Plotten eines 2d-Blocks nach dem anderen gerendert werden, bedeutet, dass häufig **Rendering-Probleme** mit der scheinbaren Tiefe von Objekten verbunden sind. Der Kern des Problems besteht darin, dass zwei nicht verbundene Objekte entweder vollständig hintereinander oder vollständig voreinander sein können, was zu Artefakten führt, wie in der folgenden Abbildung zweier ineinandergreifender Ringe dargestellt (Klicken Sie für ein animiertes GIF):



Dies kann jedoch behoben werden. Dieses Artefakt ist nur vorhanden, wenn mehrere Flächen in derselben Zeichnung geplottet werden - da jede als flache 2D-Form dargestellt wird, wobei ein einzelner Parameter den Sichtabstand bestimmt. Sie werden feststellen, dass eine einzelne komplizierte Oberfläche nicht das gleiche Problem aufweist.

Abhilfe schaffen Sie, indem Sie die Plotobjekte mithilfe von transparenten Brücken zusammenfügen:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import erf

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.arange(0, 6, 0.25)
Y = np.arange(0, 6, 0.25)
X, Y = np.meshgrid(X, Y)

Z1 = np.empty_like(X)
Z2 = np.empty_like(X)
C1 = np.empty_like(X, dtype=object)
C2 = np.empty_like(X, dtype=object)

for i in range(len(X)):
    for j in range(len(X[0])):
        z1 = 0.5*(erf((X[i,j]+Y[i,j]-4.5)*0.5)+1)
        z2 = 0.5*(erf((-X[i,j]-Y[i,j]+4.5)*0.5)+1)
        Z1[i,j] = z1
        Z2[i,j] = z2

        # If you want to grab a colour from a matplotlib cmap function,
        # you need to give it a number between 0 and 1. z1 and z2 are
        # already in this range, so it just works as is.
        C1[i,j] = plt.get_cmap("Oranges")(z1)
        C2[i,j] = plt.get_cmap("Blues")(z2)

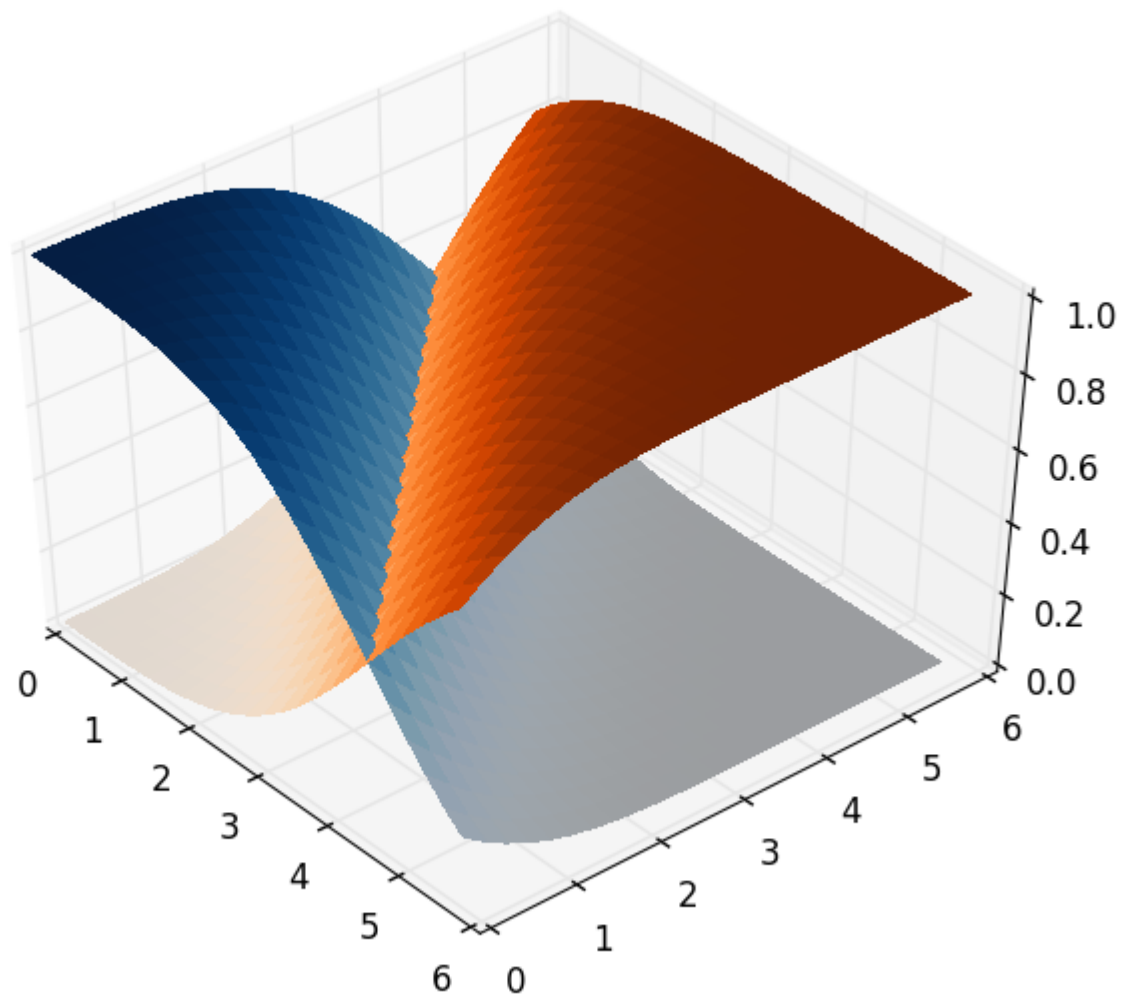
# Create a transparent bridge region
X_bridge = np.vstack([X[-1,:],X[-1,:]])
Y_bridge = np.vstack([Y[-1,:],Y[-1,:]])
Z_bridge = np.vstack([Z1[-1,:],Z2[-1,:]])
color_bridge = np.empty_like(Z_bridge, dtype=object)

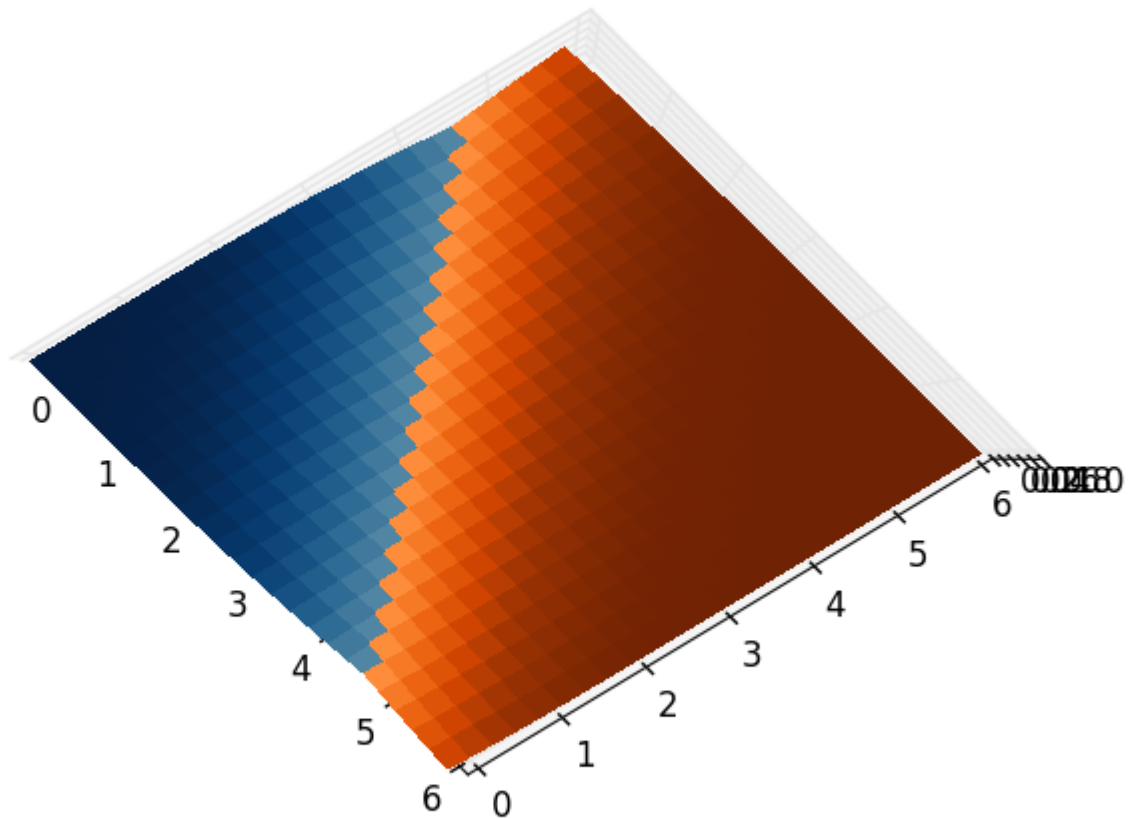
color_bridge.fill((1,1,1,0)) # RGBA colour, onlt the last component matters - it represents
the alpha / opacity.

# Join the two surfaces flipping one of them (using also the bridge)
X_full = np.vstack([X, X_bridge, np.flipud(X)])
Y_full = np.vstack([Y, Y_bridge, np.flipud(Y)])
Z_full = np.vstack([Z1, Z_bridge, np.flipud(Z2)])
color_full = np.vstack([C1, color_bridge, np.flipud(C2)])

surf_full = ax.plot_surface(X_full, Y_full, Z_full, rstride=1, cstride=1,
                            facecolors=color_full, linewidth=0,
                            antialiased=False)

plt.show()
```





Examples

Dreidimensionale Achsen erstellen

Matplotlib-Achsen sind standardmäßig zweidimensional. Um dreidimensionale Diagramme zu erstellen, müssen Sie die `Axes3D` Klasse aus dem [mplot3d-Toolkit importieren](#), um eine neue Art der Projektion für Achsen zu ermöglichen, nämlich '3d' :

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

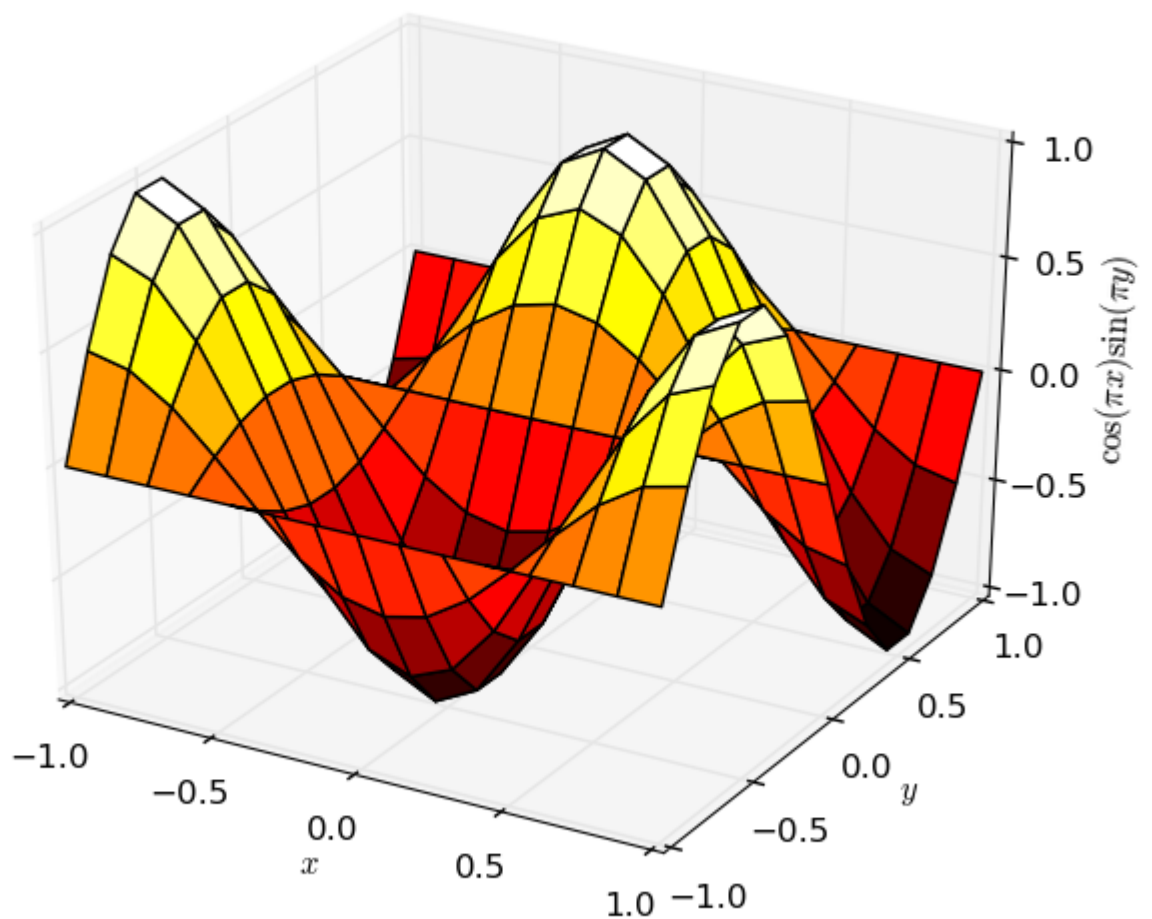
Neben den einfachen Verallgemeinerungen von zweidimensionalen Plots (wie [Liniendiagramme](#), [Streudiagramme](#), [Balkendiagramme](#), [Liniendiagramme](#)), mehr [Oberflächen Plotten Methoden](#) zur Verfügung, beispielsweise `ax.plot_surface` :

```

# generate example data
import numpy as np
x,y = np.meshgrid(np.linspace(-1,1,15),np.linspace(-1,1,15))
z = np.cos(x*np.pi)*np.sin(y*np.pi)

# actual plotting example
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# rstride and cstride are row and column stride (step size)
ax.plot_surface(x,y,z,rstride=1,cstride=1,cmap='hot')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.set_zlabel(r'$\cos(\pi x) \sin(\pi y)$')
plt.show()

```



Dreidimensionale Diagramme online lesen:

<https://riptutorial.com/de/matplotlib/topic/1880/dreidimensionale-diagramme>

Kapitel 8: Figuren und Äxte Objekte

Examples

Eine Figur erstellen

Die Abbildung enthält alle Plotelemente. Die Hauptmethode zum Erstellen einer Figur in `matplotlib` ist die Verwendung von `pyplot`.

```
import matplotlib.pyplot as plt
fig = plt.figure()
```

Sie können optional eine Nummer angeben, mit der Sie auf eine zuvor erstellte Figur zugreifen können. Wenn keine Nummer angegeben wird, wird die ID der zuletzt erstellten Figur erhöht und stattdessen verwendet. Zahlen werden ab 1, nicht 0 indiziert.

```
import matplotlib.pyplot as plt
fig = plt.figure()
fig == plt.figure(1) # True
```

Anstelle einer Zahl können Zahlen auch durch eine Zeichenfolge identifiziert werden. Wenn Sie ein interaktives Backend verwenden, wird auch der Fenstertitel festgelegt.

```
import matplotlib.pyplot as plt
fig = plt.figure('image')
```

Zur Wahl der Figur verwenden

```
plt.figure(fig.number) # or
plt.figure(1)
```

Achsen erstellen

Es gibt zwei Möglichkeiten, Achsen in `matplotlib` zu erstellen: mit `pyplot` oder mit der objektorientierten API.

Pyplot verwenden:

```
import matplotlib.pyplot as plt
ax = plt.subplot(3, 2, 1) # 3 rows, 2 columns, the first subplot
```

Verwenden der objektorientierten API:

```
import matplotlib.pyplot as plt
fig = plt.figure()
```

```
ax = fig.add_subplot(3, 2, 1)
```

Mit der Komfortfunktion `plt.subplots()` können Sie eine Abbildung und eine Sammlung von Unterplots in einem Befehl erstellen:

```
import matplotlib.pyplot as plt  
  
fig, (ax1, ax2) = plt.subplots(ncols=2, nrows=1) # 1 row, 2 columns
```

Figuren und Äxte Objekte online lesen: <https://riptutorial.com/de/matplotlib/topic/2307/figuren-und-axte-objekte>

Kapitel 9: Figurfenster schließen

Syntax

- `plt.close ()` # schließt die aktuell aktive Figur
- `plt.close (fig)` # schließt die Figur mit dem Griff 'fig'
- `plt.close (num)` # schließt die Figurennummer 'num'
- `plt.close (name)` # schließt die Figur mit der Bezeichnung 'name'
- `plt.close ('all')` # schließt alle Zahlen

Examples

Schließung der aktuell aktiven Figur mittels Pyplot

Die Pyplot-Schnittstelle zu `matplotlib` möglicherweise die einfachste Möglichkeit, eine Figur zu schließen.

```
import matplotlib.pyplot as plt
plt.plot([0, 1], [0, 1])
plt.close()
```

Eine bestimmte Figur mit `plt.close ()` schließen

Eine bestimmte Figur kann durch Halten des Griffs geschlossen werden

```
import matplotlib.pyplot as plt

fig1 = plt.figure() # create first figure
plt.plot([0, 1], [0, 1])

fig2 = plt.figure() # create second figure
plt.plot([0, 1], [0, 1])

plt.close(fig1) # close first figure although second one is active
```

Figurfenster schließen online lesen: <https://riptutorial.com/de/matplotlib/topic/6628/figurfenster-schlie-en>

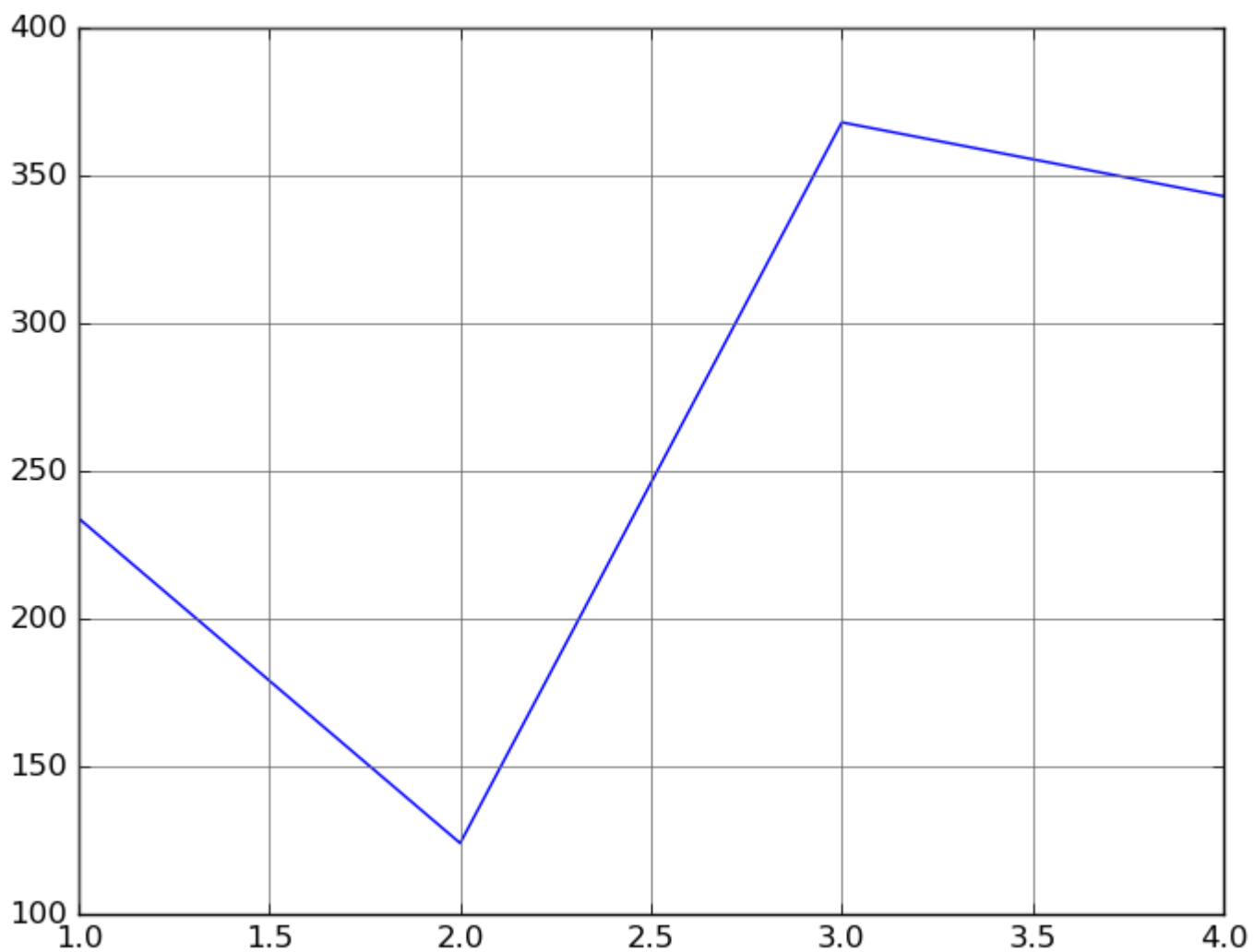
Kapitel 10: Gitterlinien und Häkchen

Examples

Plot mit Gitternetzlinien

Plot mit Gitternetzlinien

Example Of Plot With Grid Lines



```
import matplotlib.pyplot as plt

# The Data
x = [1, 2, 3, 4]
y = [234, 124, 368, 343]

# Create the figure and axes objects
fig, ax = plt.subplots(1, figsize=(8, 6))
```

```
fig.suptitle('Example Of Plot With Grid Lines')

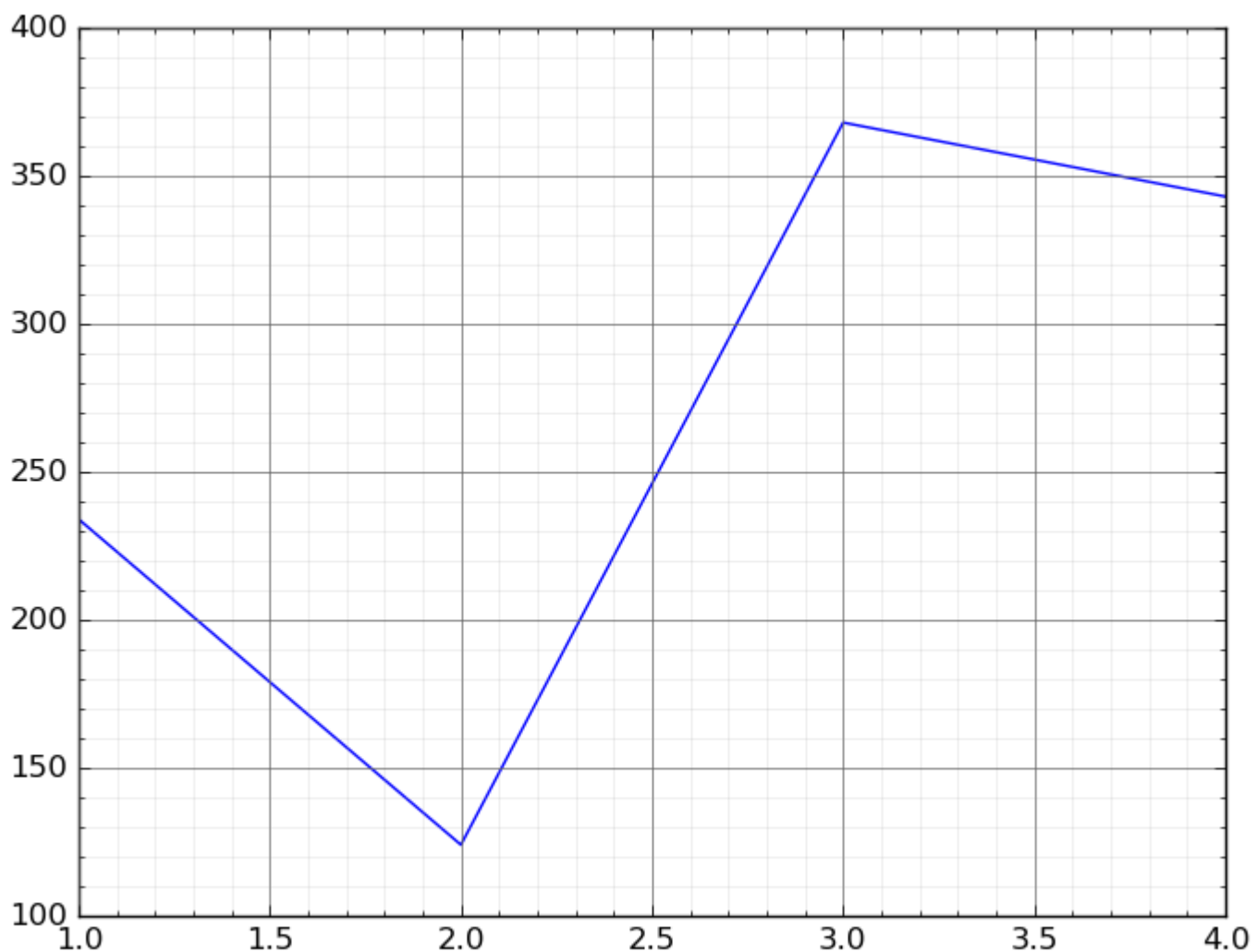
# Plot the data
ax.plot(x,y)

# Show the grid lines as dark grey lines
plt.grid(b=True, which='major', color='#666666', linestyle='-')

plt.show()
```

Plot mit großen und kleinen Rasterlinien

Example Of Plot With Major and Minor Grid Lines



```
import matplotlib.pyplot as plt

# The Data
x = [1, 2, 3, 4]
y = [234, 124, 368, 343]

# Create the figure and axes objects
```

```
fig, ax = plt.subplots(1, figsize=(8, 6))
fig.suptitle('Example Of Plot With Major and Minor Grid Lines')

# Plot the data
ax.plot(x,y)

# Show the major grid lines with dark grey lines
plt.grid(b=True, which='major', color='#666666', linestyle='-')

# Show the minor grid lines with very faint and almost transparent grey lines
plt.minorticks_on()
plt.grid(b=True, which='minor', color='#999999', linestyle='-', alpha=0.2)

plt.show()
```

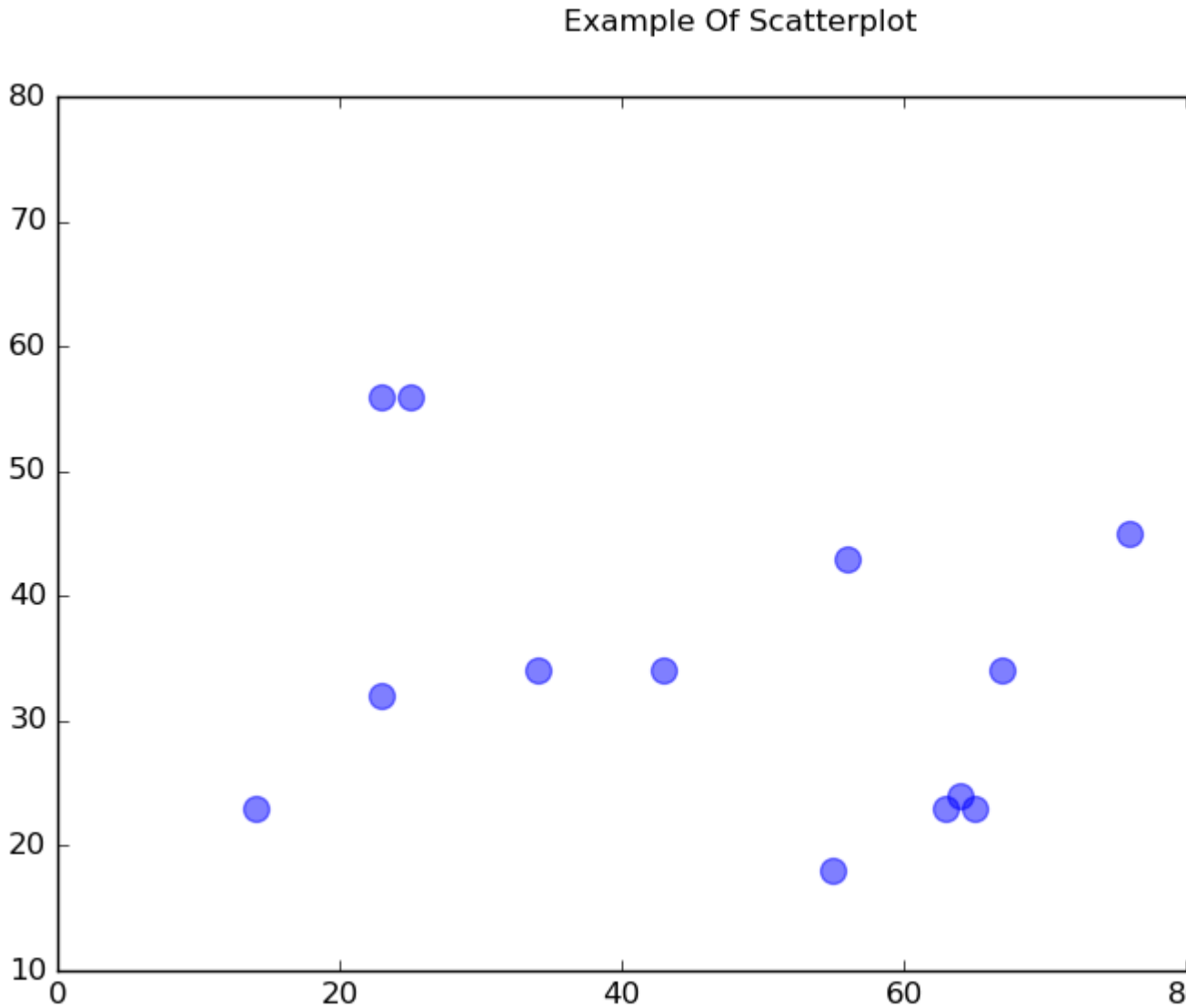
Gitterlinien und Hackchen online lesen: <https://riptutorial.com/de/matplotlib/topic/4029/gitterlinien-und-hakchen>

Kapitel 11: Grundlegende Diagramme

Examples

Streudiagramme

Ein einfaches Streudiagramm



```
import matplotlib.pyplot as plt

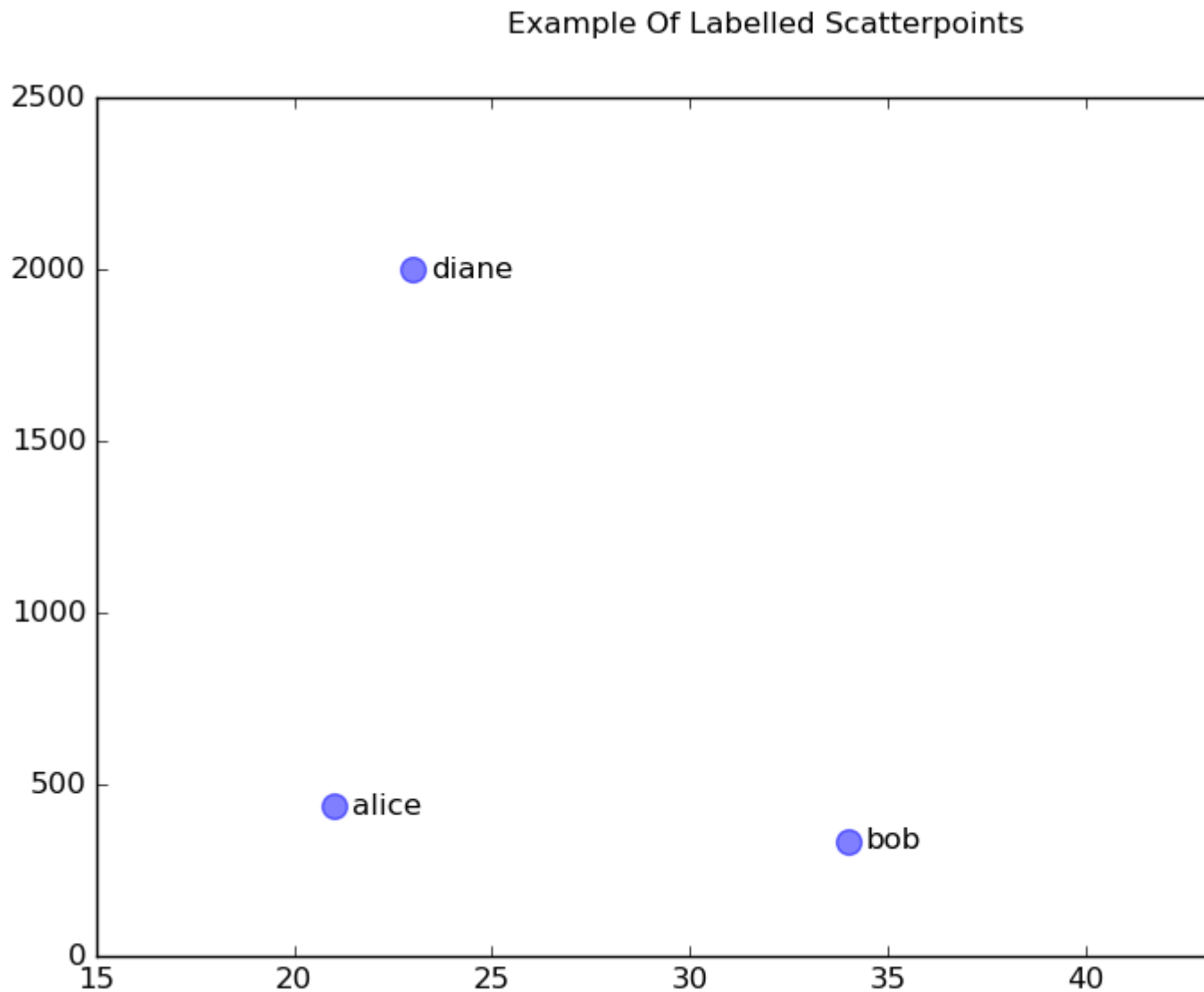
# Data
x = [43, 76, 34, 63, 56, 82, 87, 55, 64, 87, 95, 23, 14, 65, 67, 25, 23, 85]
y = [34, 45, 34, 23, 43, 76, 26, 18, 24, 74, 23, 56, 23, 23, 34, 56, 32, 23]

fig, ax = plt.subplots(1, figsize=(10, 6))
fig.suptitle('Example Of Scatterplot')
```

```
# Create the Scatter Plot
ax.scatter(x, y,
          color="blue", # Color of the dots
          s=100,        # Size of the dots
          alpha=0.5,    # Alpha/transparency of the dots (1 is opaque, 0 is transparent)
          linewidths=1) # Size of edge around the dots

# Show the plot
plt.show()
```

Ein Streudiagramm mit beschrifteten Punkten



```
import matplotlib.pyplot as plt

# Data
x = [21, 34, 44, 23]
y = [435, 334, 656, 1999]
labels = ["alice", "bob", "charlie", "diane"]

# Create the figure and axes objects
```

```

fig, ax = plt.subplots(1, figsize=(10, 6))
fig.suptitle('Example Of Labelled Scatterpoints')

# Plot the scatter points
ax.scatter(x, y,
           color="blue", # Color of the dots
           s=100,        # Size of the dots
           alpha=0.5,    # Alpha of the dots
           linewidths=1) # Size of edge around the dots

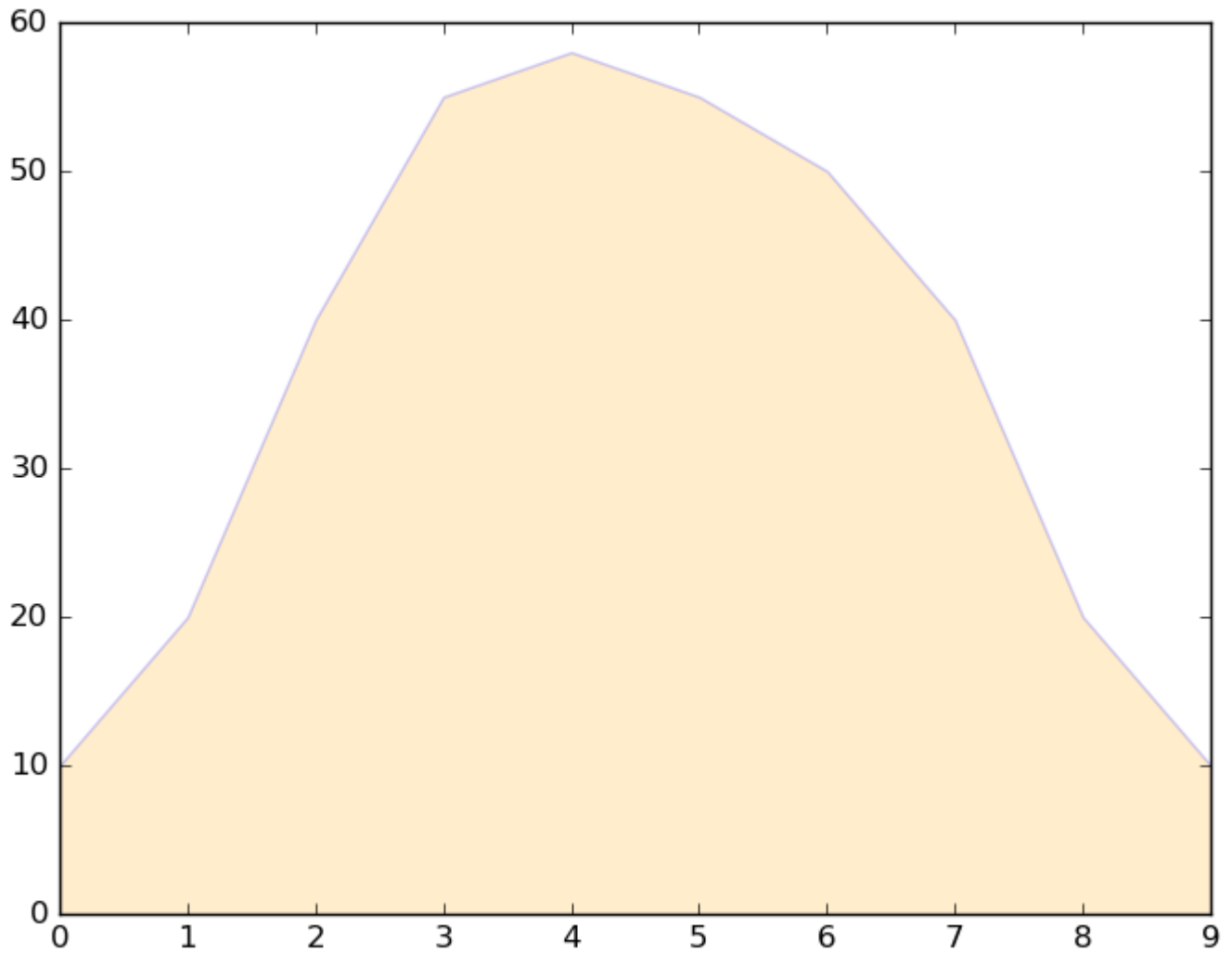
# Add the participant names as text labels for each point
for x_pos, y_pos, label in zip(x, y, labels):
    ax.annotate(label, # The label for this point
               xy=(x_pos, y_pos), # Position of the corresponding point
               xytext=(7, 0), # Offset text by 7 points to the right
               textcoords='offset points', # tell it to use offset points
               ha='left', # Horizontally aligned to the left
               va='center') # Vertical alignment is centered

# Show the plot
plt.show()

```

Schattierte Pläne

Schattenbereich unter einer Linie



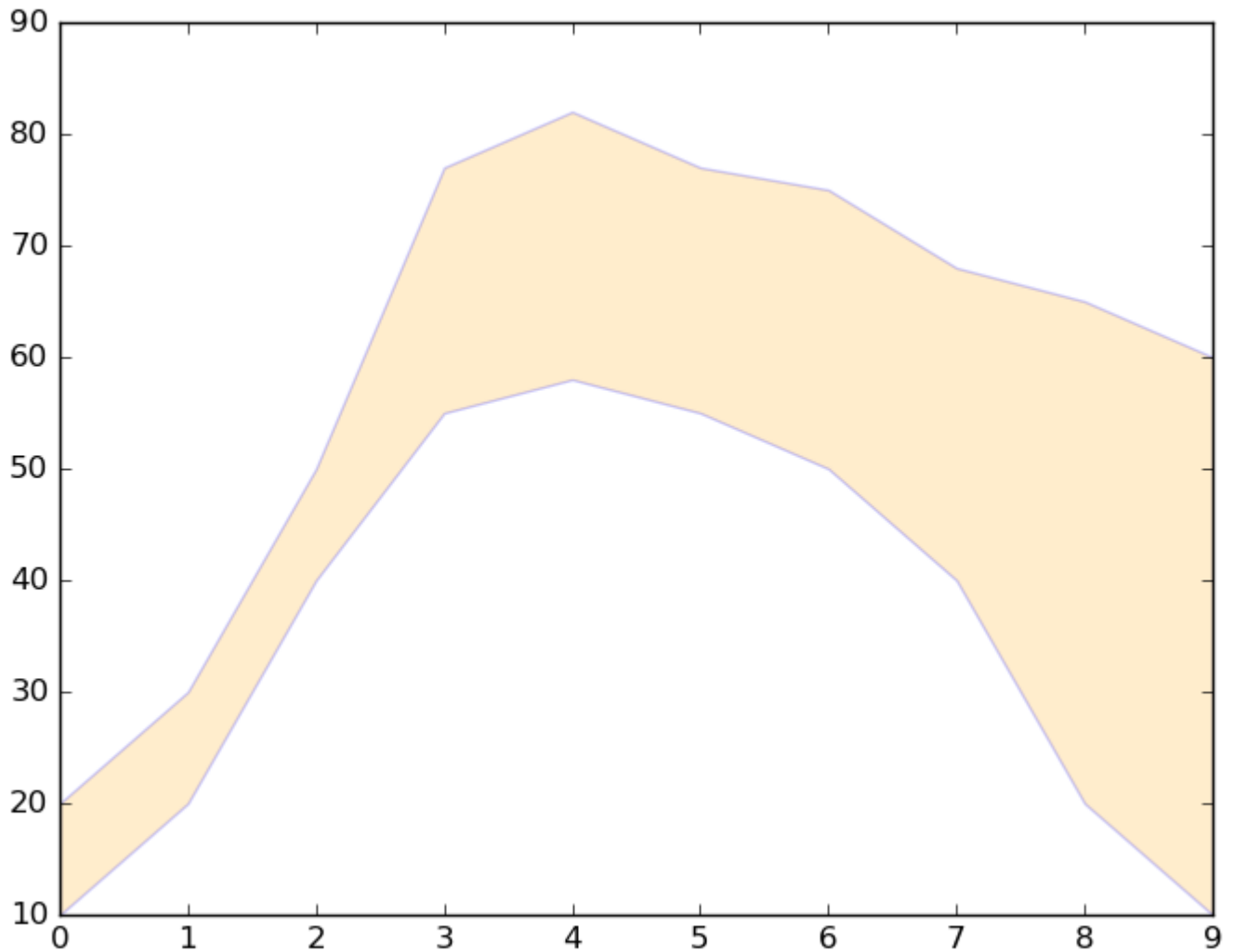
```
import matplotlib.pyplot as plt

# Data
x = [0,1,2,3,4,5,6,7,8,9]
y1 = [10,20,40,55,58,55,50,40,20,10]

# Shade the area between y1 and line y=0
plt.fill_between(x, y1, 0,
                facecolor="orange", # The fill color
                color='blue',      # The outline color
                alpha=0.2)         # Transparency of the fill

# Show the plot
plt.show()
```

Schattierte Region zwischen zwei Zeilen



```
import matplotlib.pyplot as plt

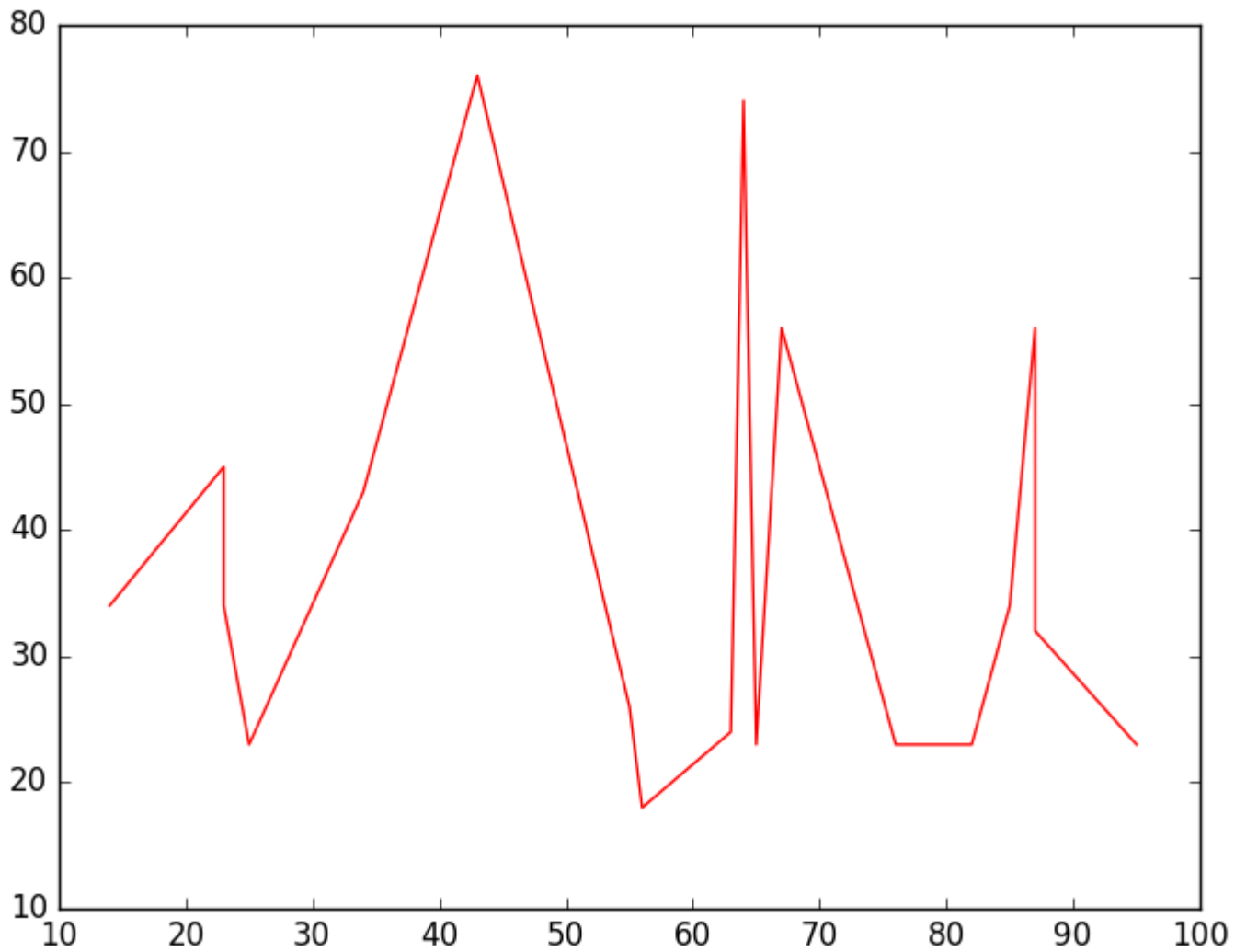
# Data
x = [0,1,2,3,4,5,6,7,8,9]
y1 = [10,20,40,55,58,55,50,40,20,10]
y2 = [20,30,50,77,82,77,75,68,65,60]

# Shade the area between y1 and y2
plt.fill_between(x, y1, y2,
                facecolor="orange", # The fill color
                color='blue',      # The outline color
                alpha=0.2)         # Transparency of the fill

# Show the plot
plt.show()
```

Liniendiagramme

Einfaches Liniendiagramm



```
import matplotlib.pyplot as plt

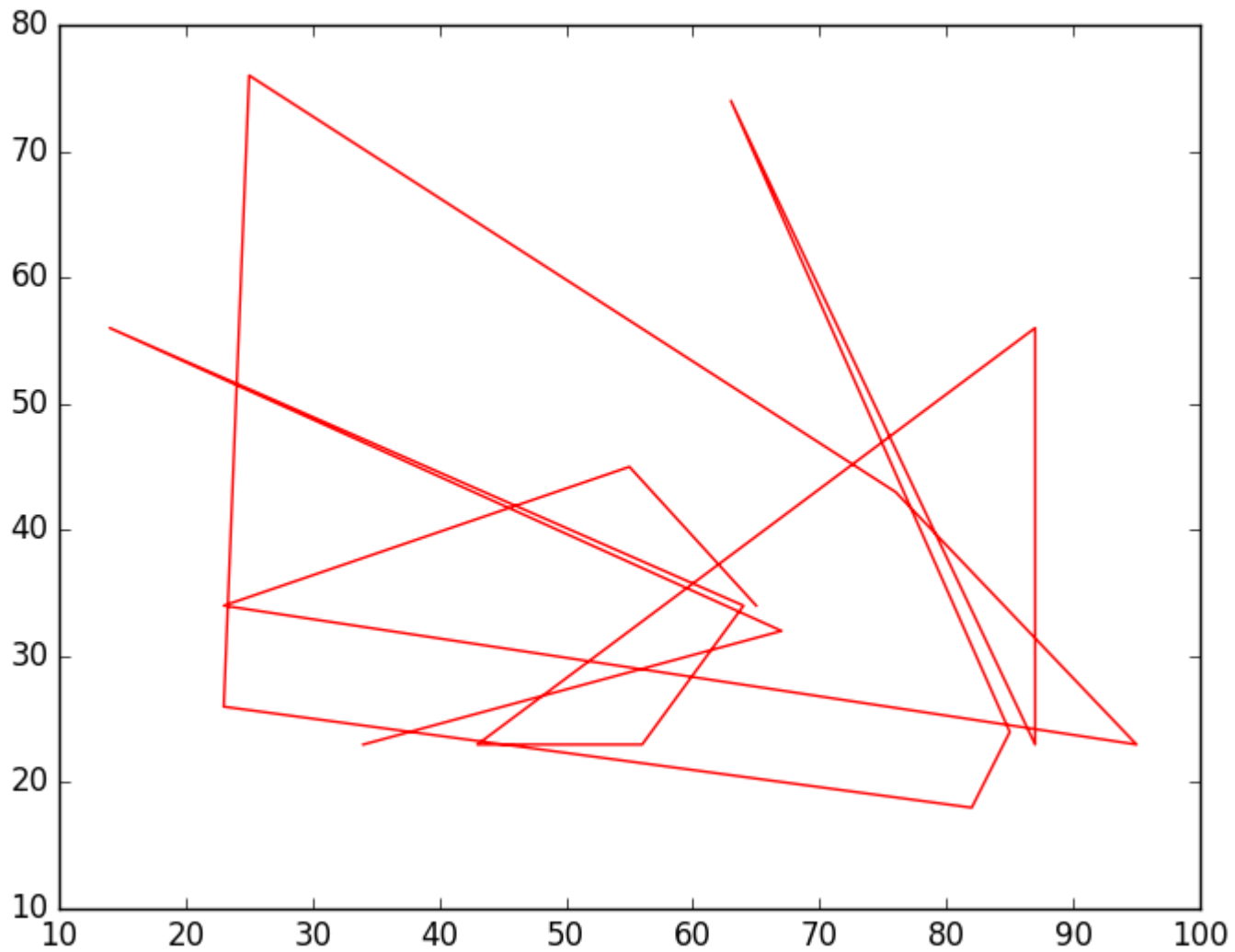
# Data
x = [14,23,23,25,34,43,55,56,63,64,65,67,76,82,85,87,87,95]
y = [34,45,34,23,43,76,26,18,24,74,23,56,23,23,34,56,32,23]

# Create the plot
plt.plot(x, y, 'r-')
# r- is a style code meaning red solid line

# Show the plot
plt.show()
```

Beachten Sie, dass y im Allgemeinen keine Funktion von x und dass die Werte in x nicht sortiert werden müssen. So sieht ein Liniendiagramm mit unsortierten x -Werten aus:

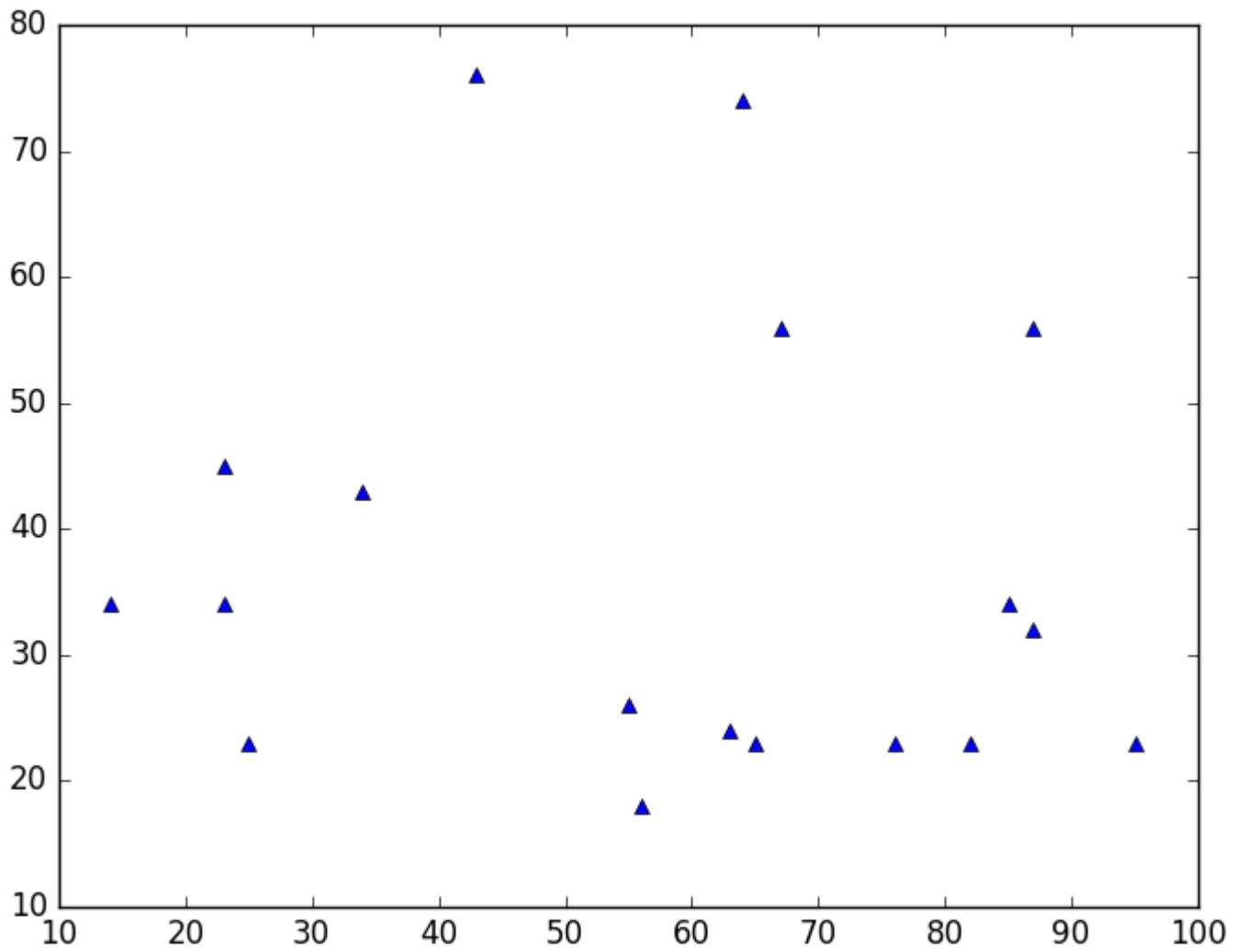
```
# shuffle the elements in x
np.random.shuffle(x)
plt.plot(x, y, 'r-')
plt.show()
```



Datenplot

Dies ähnelt einem [Streudiagramm](#), verwendet jedoch stattdessen die Funktion `plt.plot()`. Der einzige Unterschied im Code ist das Stilargument.

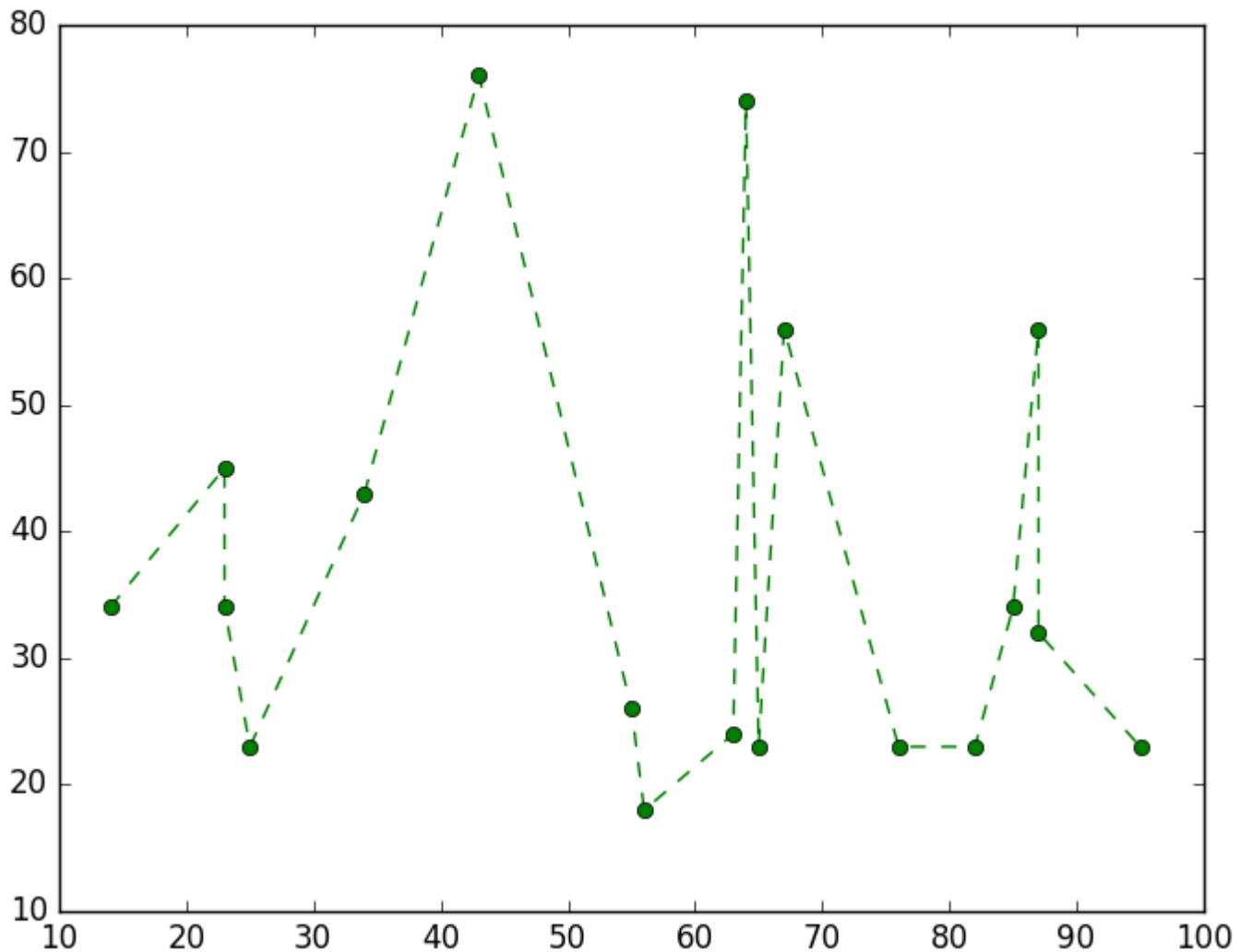
```
plt.plot(x, y, 'b^')  
# Create blue up-facing triangles
```



Daten und Zeile

Das style-Argument kann Symbole sowohl für Marker als auch für den Linienstil annehmen:

```
plt.plot(x, y, 'go--')  
# green circles and dashed line
```

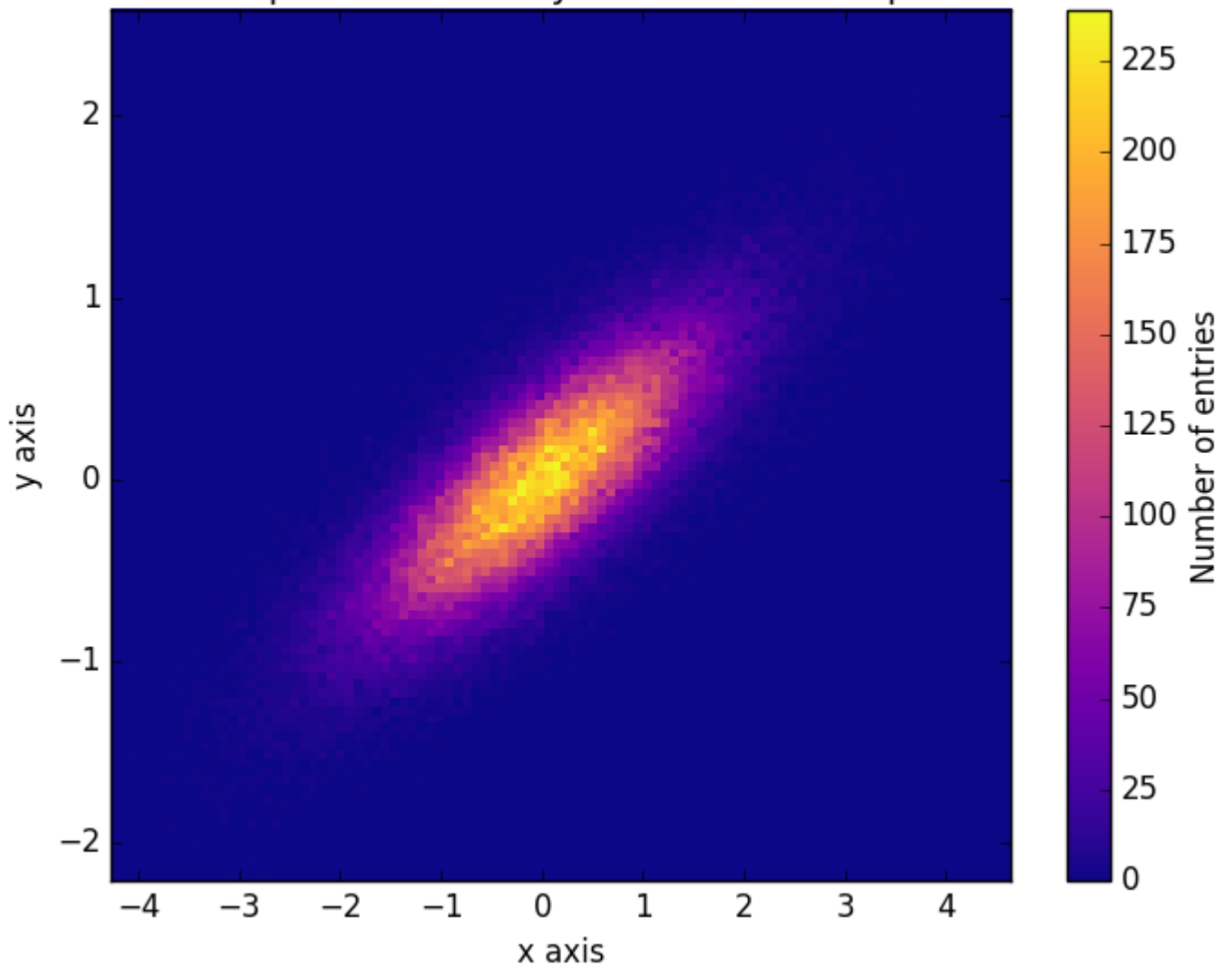



Heatmap

Heatmaps sind nützlich, um Skalarfunktionen zweier Variablen zu visualisieren. Sie liefern ein „flaches“ Bild von zweidimensionalen Histogrammen (die zum Beispiel die Dichte eines bestimmten Bereichs darstellen).

Der folgende Quellcode zeigt Heatmaps, bei denen bivariate normalverteilte Zahlen, die in beiden Richtungen auf 0 zentriert sind (Mittelwerte $[0.0, 0.0]$), und a mit einer gegebenen Kovarianzmatrix verwendet werden. Die Daten werden mit der numpy-Funktion [numpy.random.multivariate_normal](#) generiert. es wird dann der `hist2d` Funktion von [matplotlib.pyplot](#) zugeführt.

Heatmap of 2D normally distributed data points



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

# Define numbers of generated data points and bins per axis.
N_numbers = 100000
N_bins = 100

# set random seed
np.random.seed(0)

# Generate 2D normally distributed numbers.
x, y = np.random.multivariate_normal(
    mean=[0.0, 0.0],      # mean
    cov=[[1.0, 0.4],
         [0.4, 0.25]],   # covariance matrix
    size=N_numbers
).T                      # transpose to get columns

# Construct 2D histogram from data using the 'plasma' colormap
plt.hist2d(x, y, bins=N_bins, normed=False, cmap='plasma')
```

```

# Plot a colorbar with label.
cb = plt.colorbar()
cb.set_label('Number of entries')

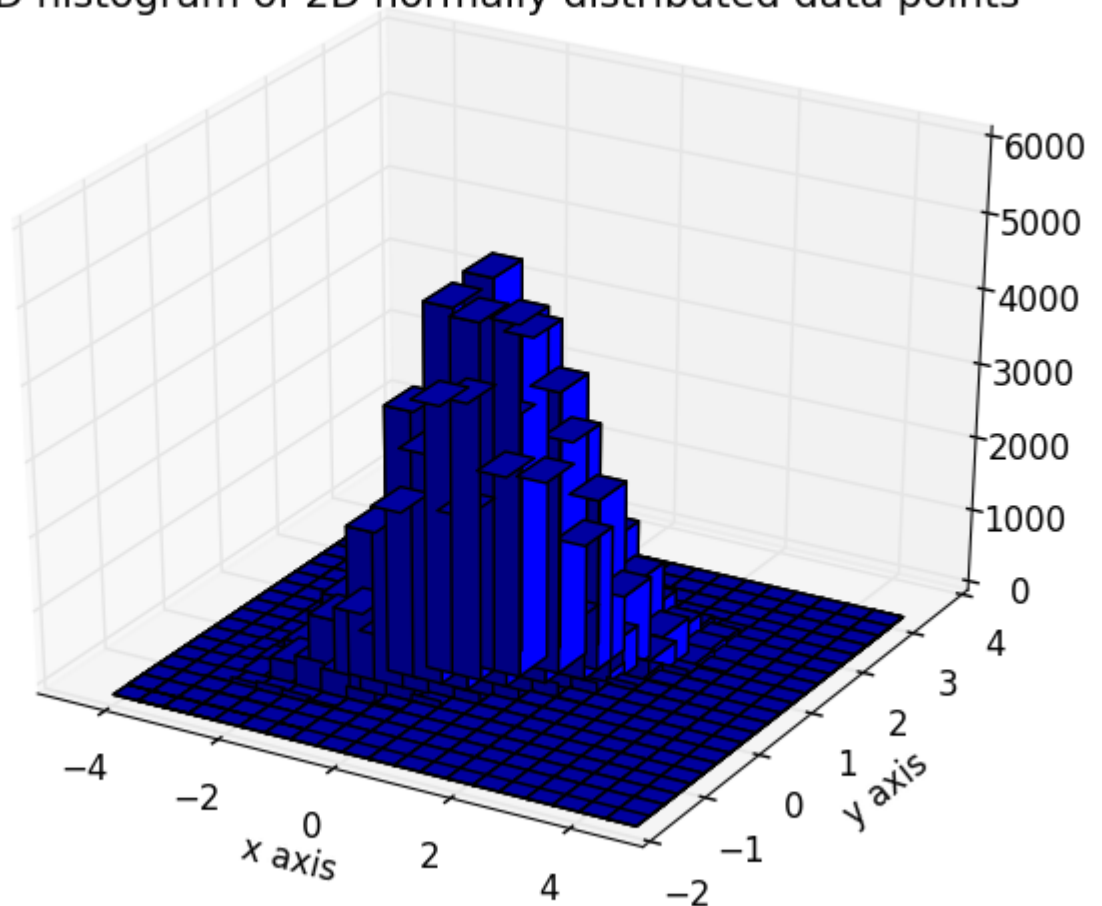
# Add title and labels to plot.
plt.title('Heatmap of 2D normally distributed data points')
plt.xlabel('x axis')
plt.ylabel('y axis')

# Show the plot.
plt.show()

```

Hier sind die gleichen Daten als 3D-Histogramm dargestellt (hier werden nur 20 Bins aus Effizienzgründen verwendet). Der Code basiert auf [dieser Matplotlib-Demo](#) .

3D histogram of 2D normally distributed data points



```

from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

# Define numbers of generated data points and bins per axis.

```

```

N_numbers = 100000
N_bins = 20

# set random seed
np.random.seed(0)

# Generate 2D normally distributed numbers.
x, y = np.random.multivariate_normal(
    mean=[0.0, 0.0],      # mean
    cov=[[1.0, 0.4],
         [0.4, 0.25]],   # covariance matrix
    size=N_numbers
).T                      # transpose to get columns

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
hist, xedges, yedges = np.histogram2d(x, y, bins=N_bins)

# Add title and labels to plot.
plt.title('3D histogram of 2D normally distributed data points')
plt.xlabel('x axis')
plt.ylabel('y axis')

# Construct arrays for the anchor positions of the bars.
# Note: np.meshgrid gives arrays in (ny, nx) so we use 'F' to flatten xpos,
# ypos in column-major order. For numpy >= 1.7, we could instead call meshgrid
# with indexing='ij'.
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25)
xpos = xpos.flatten('F')
ypos = ypos.flatten('F')
zpos = np.zeros_like(xpos)

# Construct arrays with the dimensions for the 16 bars.
dx = 0.5 * np.ones_like(zpos)
dy = dx.copy()
dz = hist.flatten()

ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color='b', zsort='average')

# Show the plot.
plt.show()

```

Grundlegende Diagramme online lesen:

<https://riptutorial.com/de/matplotlib/topic/3266/grundlegende-diagramme>

Kapitel 12: Histogramm

Examples

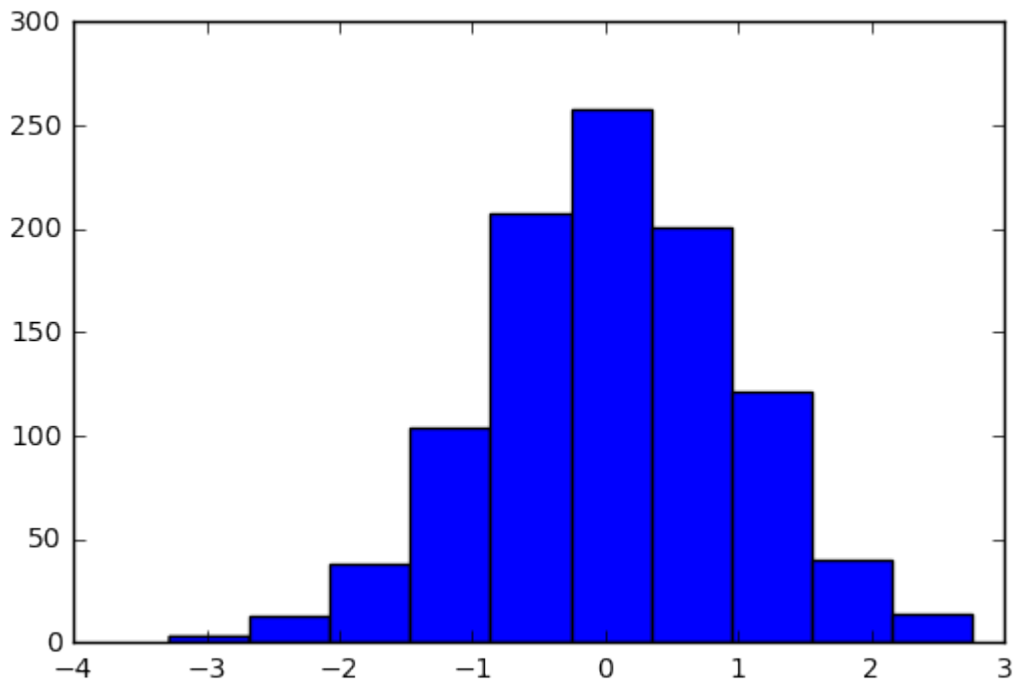
Einfaches Histogramm

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

# generate 1000 data points with normal distribution
data = np.random.randn(1000)

plt.hist(data)

plt.show()
```



Histogramm online lesen: <https://riptutorial.com/de/matplotlib/topic/7329/histogramm>

Kapitel 13: Integration mit TeX / LaTeX

Bemerkungen

- Die LaTeX-Unterstützung von Matplotlib erfordert eine funktionierende LaTeX-Installation, dvipng (möglicherweise in Ihrer LaTeX-Installation enthalten) und Ghostscript (GPL Ghostscript 8.60 oder höher wird empfohlen).
- Die pgf-Unterstützung von Matplotlib erfordert eine aktuelle LaTeX-Installation, die die TikZ / PGF-Pakete (wie TeXLive) enthält, vorzugsweise mit installiertem XeLaTeX oder LuaLaTeX.

Examples

Einfügen von TeX-Formeln in Diagramme

TeX-Formeln können mit der Funktion `rc` die `rc` eingefügt werden

```
import matplotlib.pyplot as plt
plt.rc(usetex = True)
```

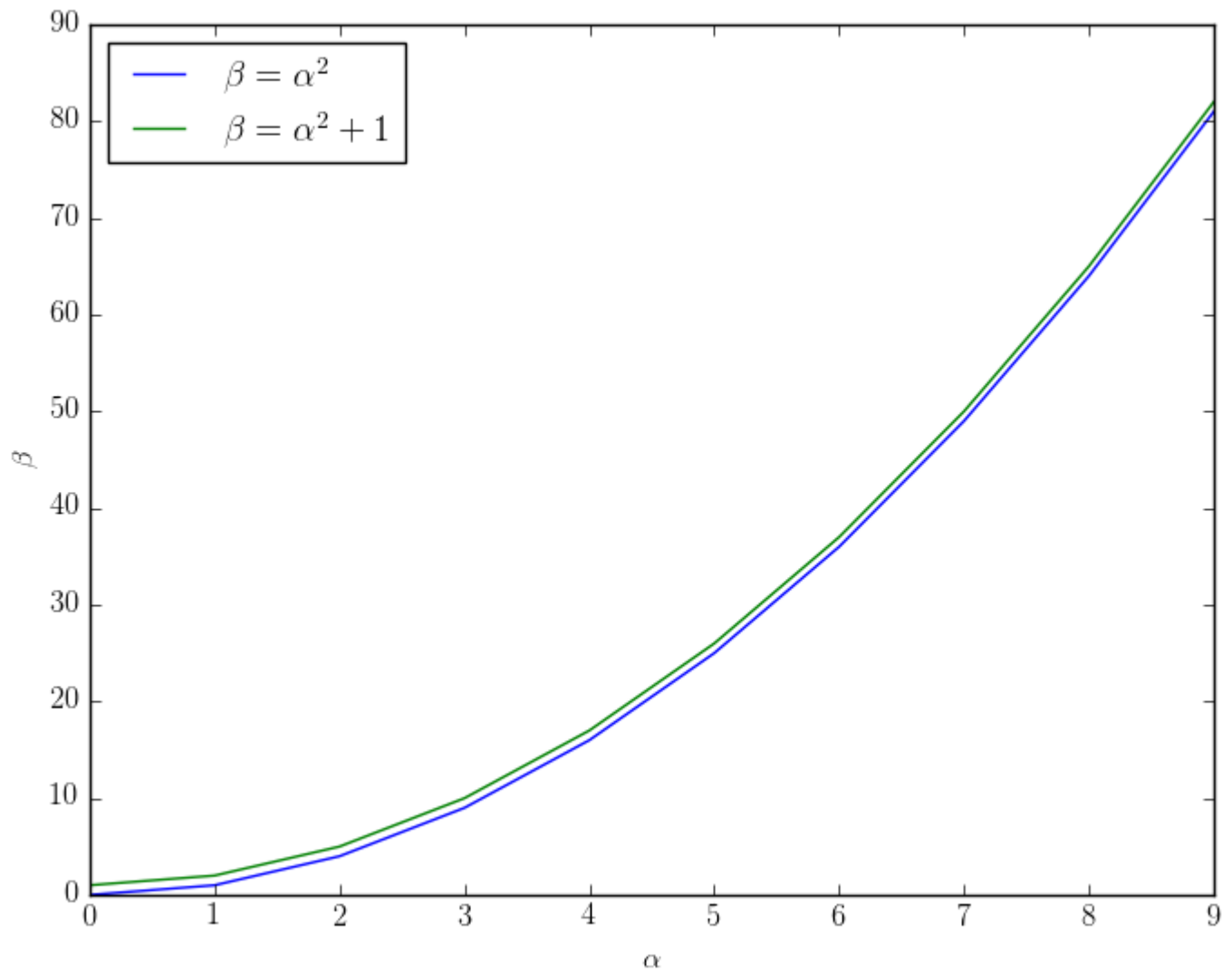
oder auf die `rcParams` zugreifen:

```
import matplotlib.pyplot as plt
params = {'tex.usetex': True}
plt.rcParams.update(params)
```

TeX verwendet den Backslash `\` für Befehle und Symbole, die mit [Sonderzeichen](#) in Python-Zeichenfolgen in Konflikt stehen können. Um literale Backslashes in einer Python-Zeichenfolge verwenden zu können, müssen sie entweder mit Escapezeichen versehen oder in eine unformatierte Zeichenfolge eingefügt werden:

```
plt.xlabel('\alpha')
plt.xlabel(r'\alpha')
```

Die folgende Handlung



kann durch den Code erzeugt werden

```
import matplotlib.pyplot as plt
plt.rc(usetex = True)
x = range(0,10)
y = [t**2 for t in x]
z = [t**2+1 for t in x]
plt.plot(x, y, label = r'\beta=\alpha^2$')
plt.plot(x, z, label = r'\beta=\alpha^2+1$')
plt.xlabel(r'\alpha$')
plt.ylabel(r'\beta$')
plt.legend(loc=0)
plt.show()
```

Angezeigte Gleichungen (z. B. $\beta = \alpha^2$ oder $\beta = \alpha^2 + 1$) werden nicht unterstützt. Trotzdem ist der angezeigte mathematische Stil mit `\displaystyle`.

Um Latex-Pakete zu laden, verwenden Sie das Argument `tex.latex.preamble`:

```
params = {'text.latex.preamble' : [r'\usepackage{siunitx}', r'\usepackage{amsmath}']}
plt.rcParams.update(params)
```

Beachten Sie jedoch die Warnung in der [Beispieldatei matplotlibrc](#) :

```
#text.latex.preamble : # IMPROPER USE OF THIS FEATURE WILL LEAD TO LATEX FAILURES
                        # AND IS THEREFORE UNSUPPORTED. PLEASE DO NOT ASK FOR HELP
                        # IF THIS FEATURE DOES NOT DO WHAT YOU EXPECT IT TO.
                        # preamble is a comma separated list of LaTeX statements
                        # that are included in the LaTeX document preamble.
                        # An example:
                        # text.latex.preamble : \usepackage{bm},\usepackage{euler}
                        # The following packages are always loaded with usetex, so
                        # beware of package collisions: color, geometry, graphicx,
                        # typelcm, textcomp. Adobe Postscript (PSSNFS) font packages
                        # may also be loaded, depending on your font settings
```

Speichern und Exportieren von Plots, die TeX verwenden

Um Plots mit matplotlib in TeX - Dokumente erstellt sind, sollten sie gespeichert als sein `pdf` oder `eps` - Dateien. Auf diese Weise wird jeglicher Text in der Zeichnung (einschließlich TeX-Formeln) als Text im endgültigen Dokument dargestellt.

```
import matplotlib.pyplot as plt
plt.rc(usetex=True)
x = range(0, 10)
y = [t**2 for t in x]
z = [t**2+1 for t in x]
plt.plot(x, y, label=r'\beta=\alpha^2$')
plt.plot(x, z, label=r'\beta=\alpha^2+1$')
plt.xlabel(r'\alpha$')
plt.ylabel(r'\beta$')
plt.legend(loc=0)
plt.savefig('my_pdf_plot.pdf') # Saving plot to pdf file
plt.savefig('my_eps_plot.eps') # Saving plot to eps file
```

Diagramme in matplotlib können mit dem Makro `pgf` Makro in TeX-Code exportiert werden, um Grafiken anzuzeigen.

```
import matplotlib.pyplot as plt
plt.rc(usetex=True)
x = range(0, 10)
y = [t**2 for t in x]
z = [t**2+1 for t in x]
plt.plot(x, y, label=r'\beta=\alpha^2$')
plt.plot(x, z, label=r'\beta=\alpha^2+1$')
plt.xlabel(r'\alpha$')
plt.ylabel(r'\beta$')
plt.legend(loc=0)
plt.savefig('my_pgf_plot.pgf')
```

Verwenden Sie den Befehl `rc` , um die verwendete TeX-Engine zu ändern

```
plt.rc('pgf', texsystem='pdflatex') # or luatex, xelatex...
```

Um die `.pgf` Figur aufzunehmen, schreiben Sie in Ihr LaTeX-Dokument


```
\usepackage{pgf}
\input{my_pgf_plot.pgf}
```

Integration mit TeX / LaTeX online lesen:

<https://riptutorial.com/de/matplotlib/topic/2962/integration-mit-tex---latex>

Kapitel 14: Konturkarten

Examples

Einfache gefüllte Konturdarstellung

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

# generate 101 x and y values between -10 and 10
x = np.linspace(-10, 10, 101)
y = np.linspace(-10, 10, 101)

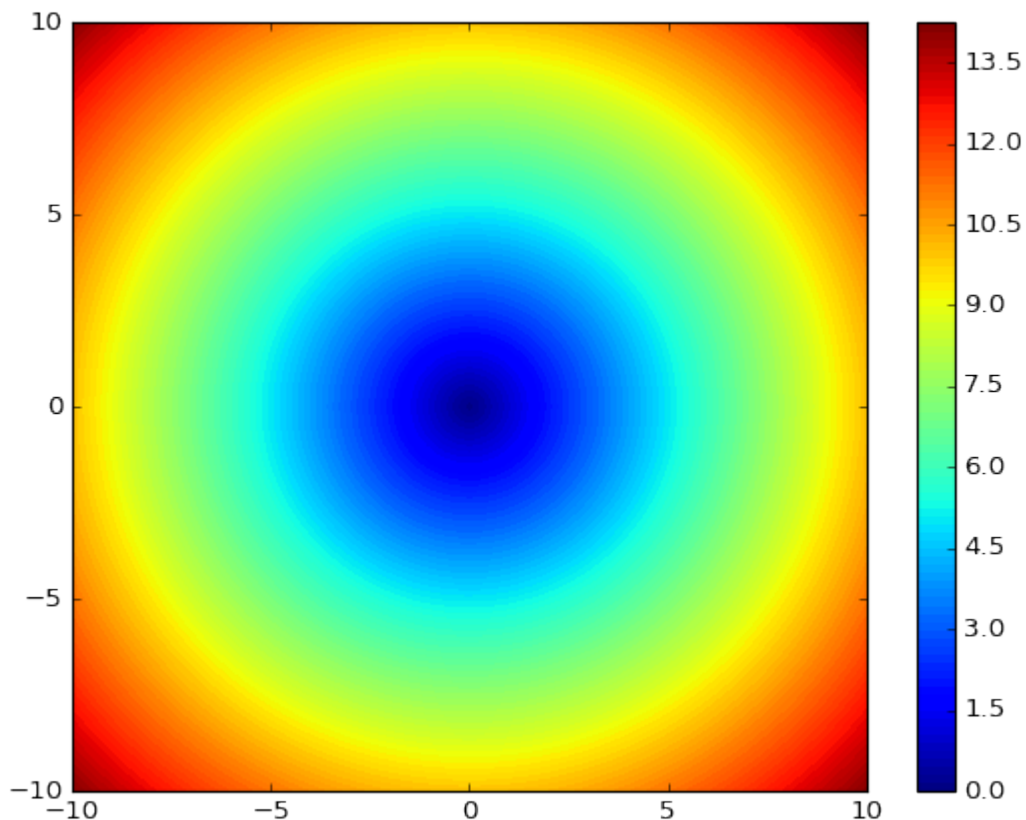
# make X and Y matrices representing x and y values of 2d plane
X, Y = np.meshgrid(x, y)

# compute z value of a point as a function of x and y (z = 12 distance form 0,0)
Z = np.sqrt(X ** 2 + Y ** 2)

# plot filled contour map with 100 levels
cs = plt.contourf(X, Y, Z, 100)

# add default colorbar for the map
plt.colorbar(cs)
```

Ergebnis:



Einfache Konturdarstellung

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

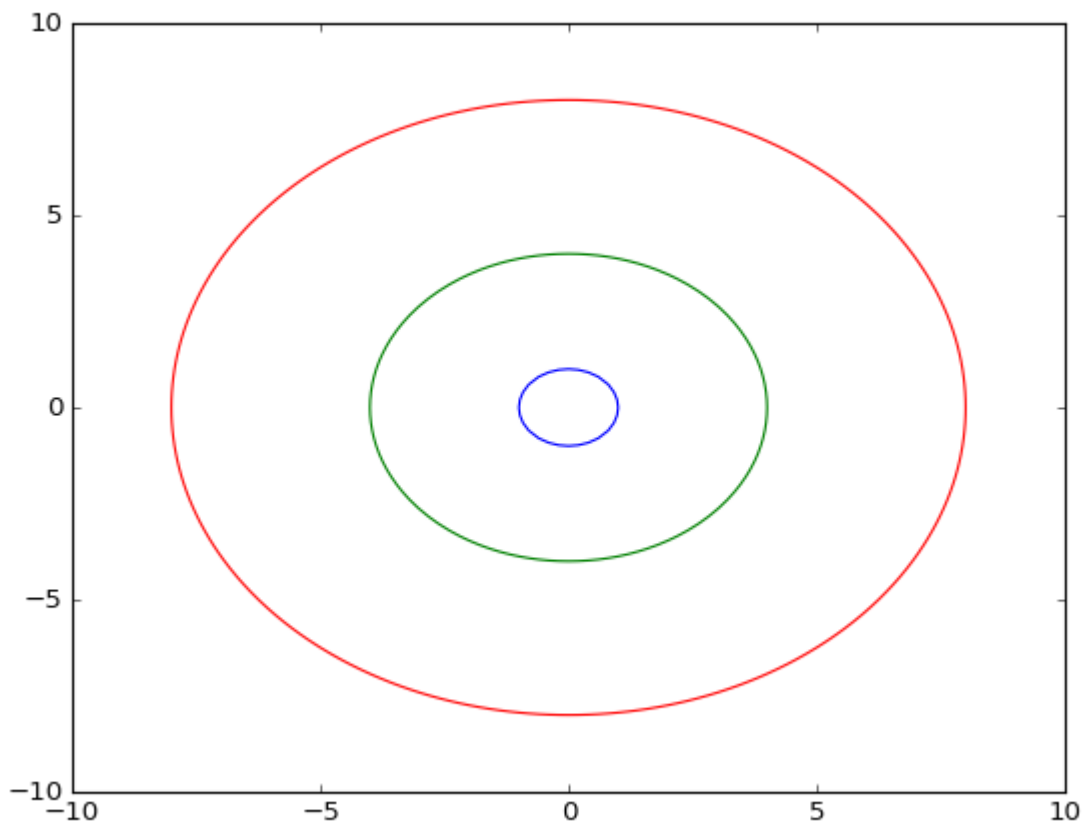
# generate 101 x and y values between -10 and 10
x = np.linspace(-10, 10, 101)
y = np.linspace(-10, 10, 101)

# make X and Y matrices representing x and y values of 2d plane
X, Y = np.meshgrid(x, y)

# compute z value of a point as a function of x and y (z = 12 distance form 0,0)
Z = np.sqrt(X ** 2 + Y ** 2)

# plot contour map with 3 levels
# colors: up to 1 - blue, from 1 to 4 - green, from 4 to 8 - red
plt.contour(X, Y, Z, [1, 4, 8], colors=['b', 'g', 'r'])
```

Ergebnis:



Konturkarten online lesen: <https://riptutorial.com/de/matplotlib/topic/8644/konturkarten>

Kapitel 15: Koordinatensysteme

Bemerkungen

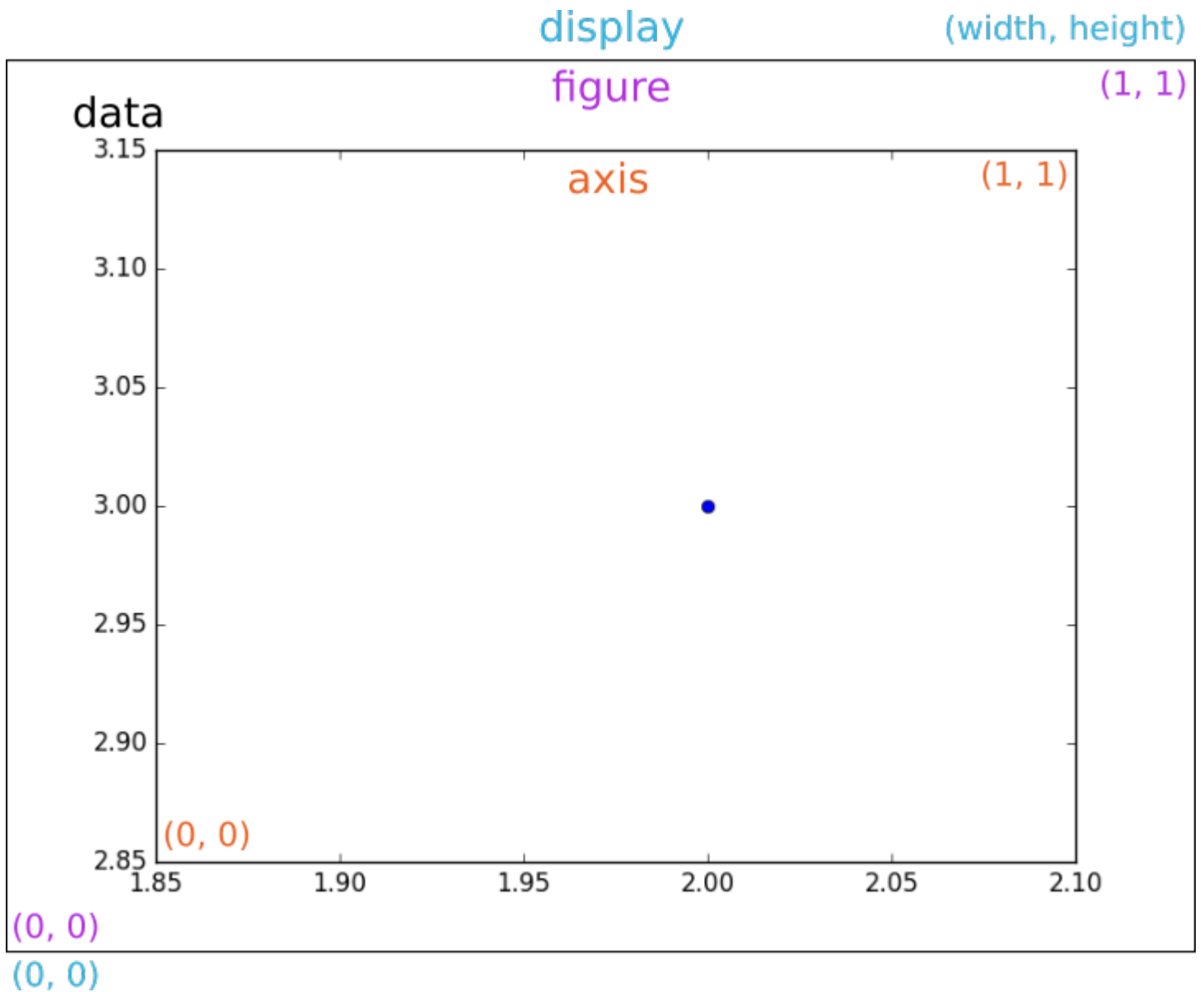
Matplotlib verfügt über vier verschiedene Koordinatensysteme, mit denen das Positionieren verschiedener Objekte, z. B. Text, vereinfacht werden kann. Jedes System hat ein entsprechendes Transformationsobjekt, das Koordinaten von diesem System in das sogenannte Anzeigekoordinatensystem transformiert.

Das Datenkoordinatensystem ist das System, das durch die Daten in den jeweiligen Achsen definiert wird. Dies ist nützlich, wenn Sie versuchen, ein Objekt relativ zu den aufgezeichneten Daten zu positionieren. Der Bereich wird durch die `xlim` und `ylim` Eigenschaften der `Axes`. Das entsprechende Transformationsobjekt ist `ax.transData`.

Achsen-Koordinatensystem ist das an sein `Axes` Objekt gebundene System. Punkte (0, 0) und (1, 1) definieren die unteren linken und rechten oberen Ecken der Achsen. Daher ist es hilfreich, wenn Sie sich relativ zu den Achsen wie in der Mitte des Diagramms positionieren. Das entsprechende Transformationsobjekt ist `ax.transAxes`.

Das Koordinatensystem der Figur ist analog zum Koordinatensystem der Achsen, außer dass es an die `Figure` gebunden ist. Die Punkte (0, 0) und (1, 1) stehen für die linke untere und die rechte obere Ecke der Figur. Dies ist nützlich, wenn Sie versuchen, etwas relativ zum gesamten Bild zu positionieren. Das entsprechende Transformationsobjekt ist `fig.transFigure`.

Das Anzeigekoordinatensystem ist das in Pixel angegebene Bildsystem. Punkte (0, 0) und (Breite, Höhe) sind die unteren linken und oberen rechten Pixel des Bildes oder der Anzeige. Es kann absolut zum Positionieren verwendet werden. Da Transformationsobjekte Koordinaten in dieses Koordinatensystem transformieren, ist dem Anzeigesystem kein Transformationsobjekt zugeordnet. `None` oder `matplotlib.transforms.IdentityTransform()` kann jedoch bei Bedarf verwendet werden.



Weitere Details finden Sie [hier](#) .

Examples

Koordinatensysteme und Text

Die Koordinatensysteme von Matplotlib sind sehr praktisch, wenn Sie versuchen, die von Ihnen erstellten Plots mit Anmerkungen zu versehen. Manchmal möchten Sie Text relativ zu Ihren Daten positionieren, z. B. wenn Sie versuchen, einen bestimmten Punkt zu kennzeichnen. Zu anderen Zeiten möchten Sie vielleicht einen Text oberhalb der Figur hinzufügen. Dies kann leicht erreicht werden, indem ein geeignetes Koordinatensystem ausgewählt wird, indem ein Transformationsobjekt an den `transform` im Aufruf von `text()` .

```
import matplotlib.pyplot as plt

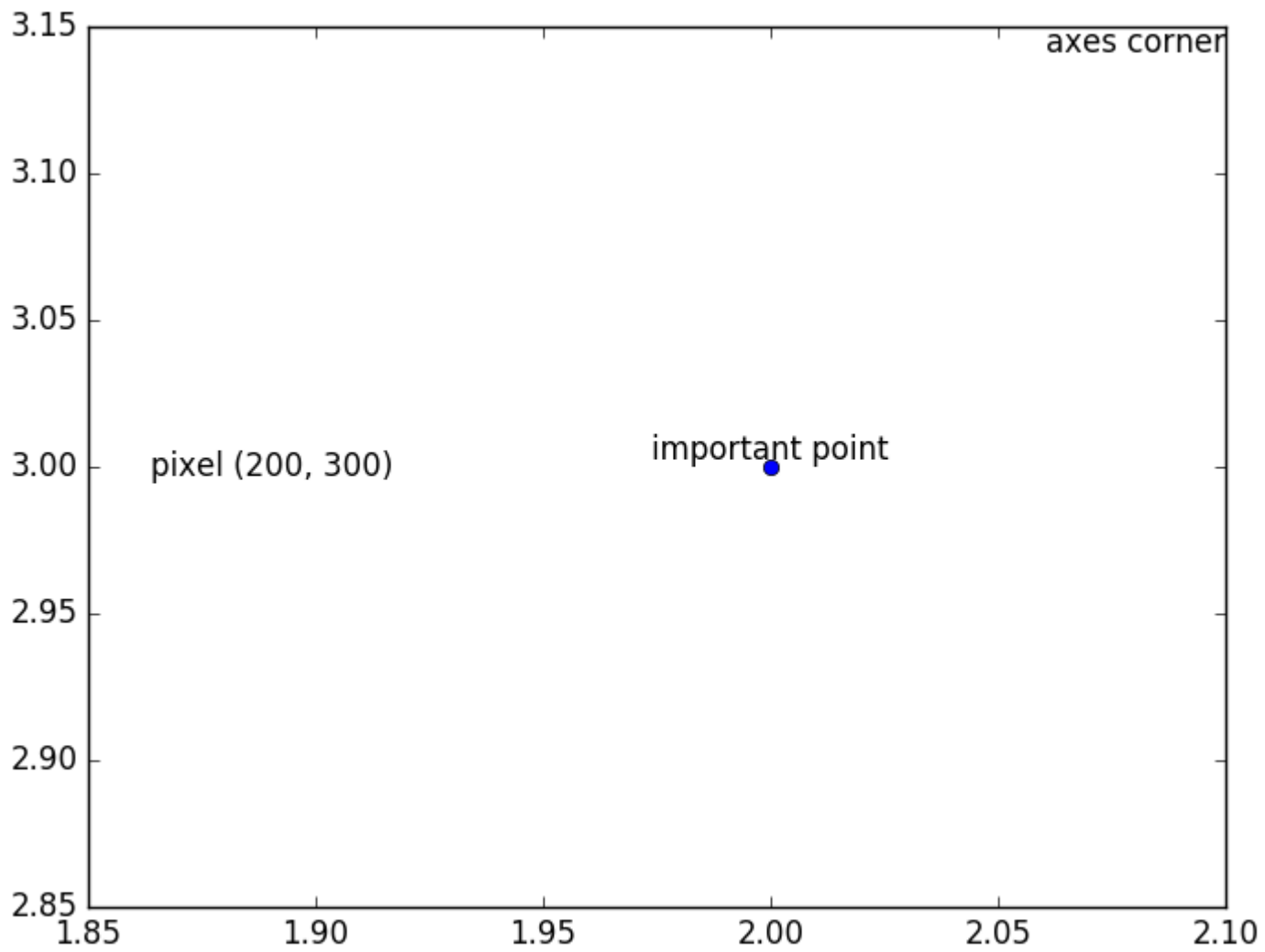
fig, ax = plt.subplots()

ax.plot([2.], [3.], 'bo')
```

```
plt.text( # position text relative to data
    2., 3., 'important point', # x, y, text,
    ha='center', va='bottom', # text alignment,
    transform=ax.transData # coordinate system transformation
)
plt.text( # position text relative to Axes
    1.0, 1.0, 'axes corner',
    ha='right', va='top',
    transform=ax.transAxes
)
plt.text( # position text relative to Figure
    0.0, 1.0, 'figure corner',
    ha='left', va='top',
    transform=fig.transFigure
)
plt.text( # position text absolutely at specific pixel on image
    200, 300, 'pixel (200, 300)',
    ha='center', va='center',
    transform=None
)

plt.show()
```

figure corner



Koordinatensysteme online lesen:

<https://riptutorial.com/de/matplotlib/topic/4566/koordinatensysteme>

Kapitel 16: Legenden

Examples

Einfache Legende

Angenommen, Sie haben mehrere Linien in derselben Grafik, jede mit einer anderen Farbe, und Sie möchten eine Legende erstellen, um zu erkennen, was jede Linie darstellt. Sie können dies tun, indem Sie bei Aufruf von `plot()` eine Beschriftung an jede der Zeilen übergeben, z. B. wird die folgende Zeile mit *"My Line 1"* bezeichnet .

```
ax.plot(x, y1, color="red", label="My Line 1")
```

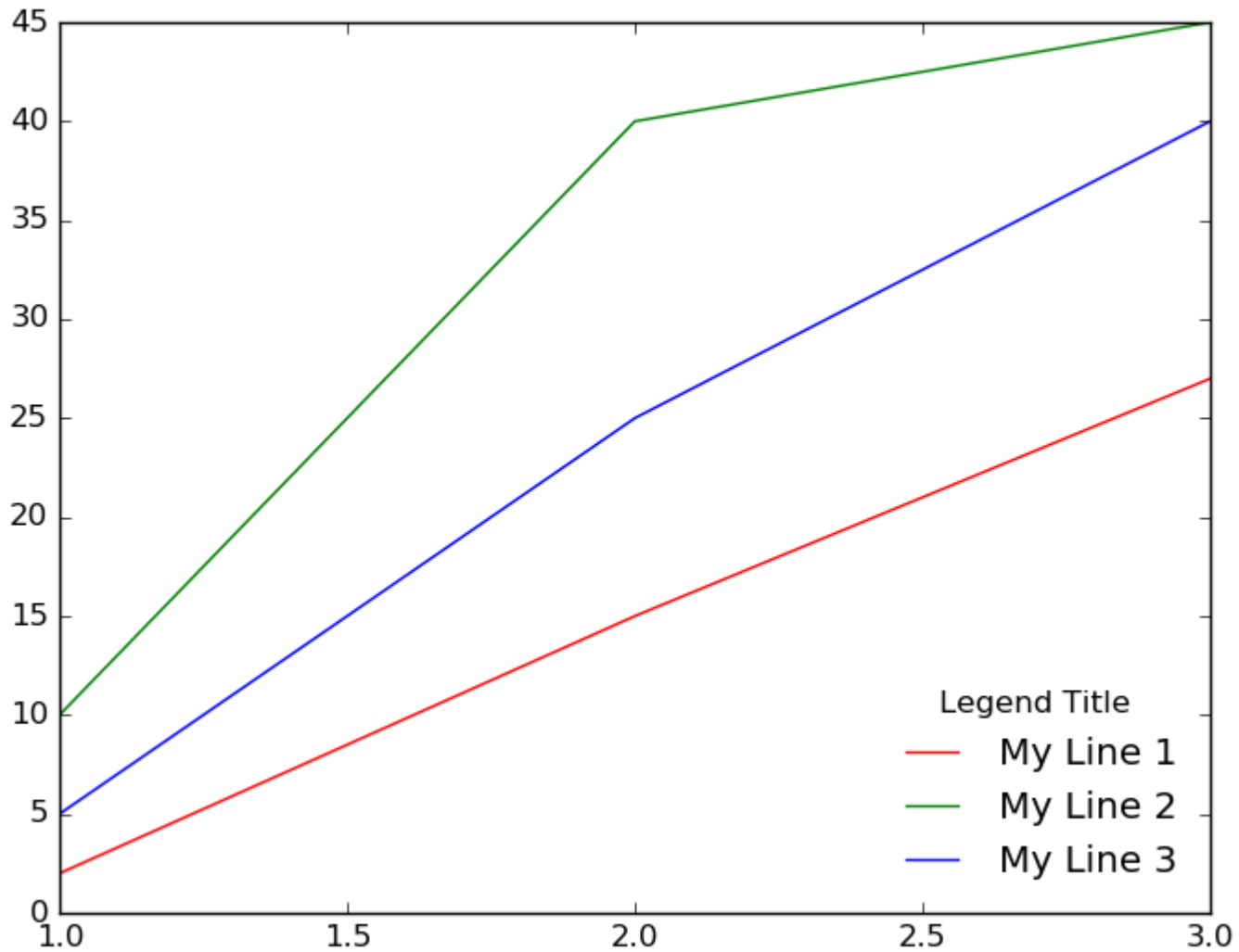
Dies gibt den Text an, der in der Legende für diese Zeile angezeigt wird. Um die eigentliche Legende sichtbar zu machen, können wir `ax.legend()`

Standardmäßig wird eine Legende in einem Kästchen in der oberen rechten Ecke des Diagramms erstellt. Sie können Argumente an `legend()` , um sie anzupassen. Zum Beispiel können wir es in der unteren rechten Ecke positionieren, ohne ein Rahmenrahmen, und einen Titel für die Legende erstellen, indem Sie Folgendes aufrufen:

```
ax.legend(loc="lower right", title="Legend Title", frameon=False)
```

Unten ist ein Beispiel:

Simple Legend Example



```
import matplotlib.pyplot as plt

# The data
x = [1, 2, 3]
y1 = [2, 15, 27]
y2 = [10, 40, 45]
y3 = [5, 25, 40]

# Initialize the figure and axes
fig, ax = plt.subplots(1, figsize=(8, 6))

# Set the title for the figure
fig.suptitle('Simple Legend Example ', fontsize=15)

# Draw all the lines in the same plot, assigning a label for each one to be
# shown in the legend
ax.plot(x, y1, color="red", label="My Line 1")
ax.plot(x, y2, color="green", label="My Line 2")
ax.plot(x, y3, color="blue", label="My Line 3")

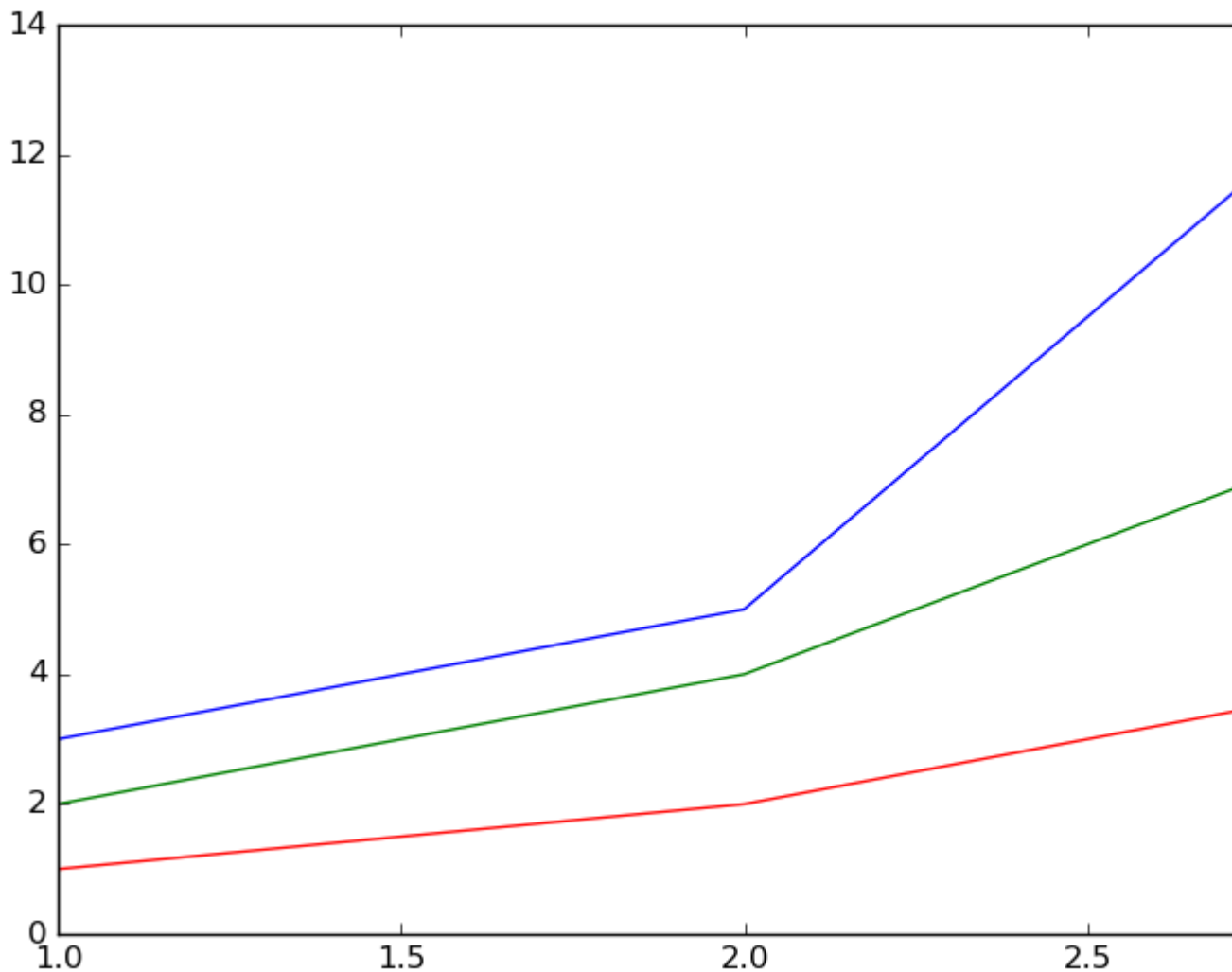
# Add a legend with title, position it on the lower right (loc) with no box framing (frameon)
ax.legend(loc="lower right", title="Legend Title", frameon=False)
```

```
# Show the plot
plt.show()
```

Legende außerhalb des Grundstücks platziert

Manchmal ist es notwendig oder wünschenswert, die Legende außerhalb der Handlung zu platzieren. Der folgende Code zeigt, wie das geht.

Example of a Legend Being Placed Outside of Plot



```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 1, figsize=(10,6)) # make the figure with the size 10 x 6 inches
fig.suptitle('Example of a Legend Being Placed Outside of Plot')

# The data
x = [1, 2, 3]
y1 = [1, 2, 4]
y2 = [2, 4, 8]
y3 = [3, 5, 14]

# Labels to use for each line
line_labels = ["Item A", "Item B", "Item C"]
```

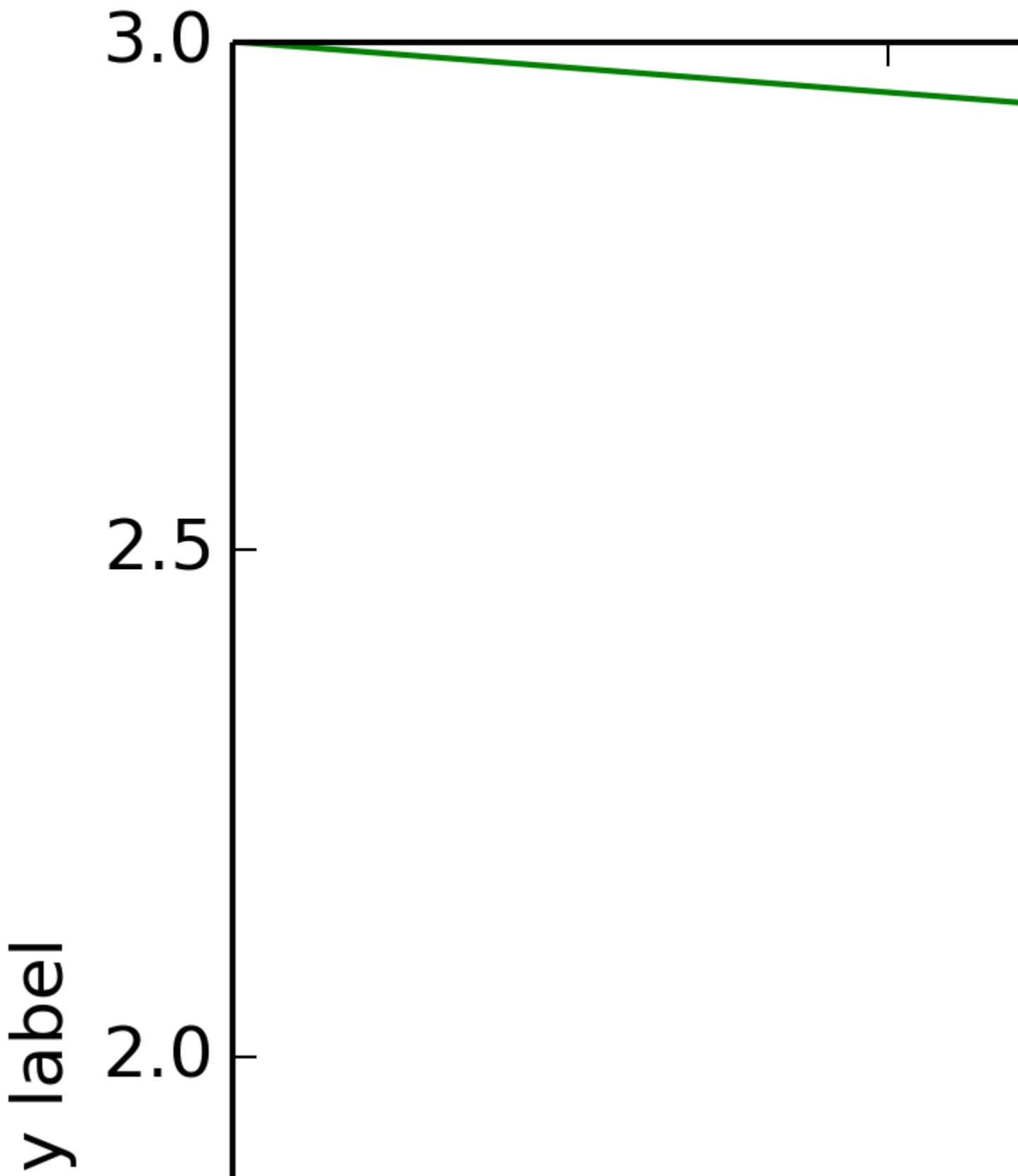
```
# Create the lines, assigning different colors for each one.
# Also store the created line objects
l1 = ax.plot(x, y1, color="red")[0]
l2 = ax.plot(x, y2, color="green")[0]
l3 = ax.plot(x, y3, color="blue")[0]

fig.legend([l1, l2, l3],          # List of the line objects
           labels= line_labels,  # The labels for each line
           loc="center right",   # Position of the legend
           borderaxespad=0.1,   # Add little spacing around the legend box
           title="Legend Title") # Title for the legend

# Adjust the scaling factor to fit your legend text completely outside the plot
# (smaller value results in more space being made for the legend)
plt.subplots_adjust(right=0.85)

plt.show()
```

Eine andere Möglichkeit, die Legende außerhalb der `bbox_to_anchor` `bbox_extra_artists` ist die Verwendung von `bbox_to_anchor + bbox_extra_artists + bbox_inches='tight'` , wie im folgenden Beispiel gezeigt:



erstellen, anstatt eine Legende auf den *Achsen* Ebene zu schaffen (die eine eigene Legende für jede subplot schaffen). Dies wird durch Aufrufen von `fig.legend()` wie im Code für den folgenden Code zu sehen ist.

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(10,4))
fig.suptitle('Example of a Single Legend Shared Across Multiple Subplots')

# The data
x = [1, 2, 3]
y1 = [1, 2, 3]
y2 = [3, 1, 3]
y3 = [1, 3, 1]
y4 = [2, 2, 3]

# Labels to use in the legend for each line
line_labels = ["Line A", "Line B", "Line C", "Line D"]

# Create the sub-plots, assigning a different color for each line.
# Also store the line objects created
l1 = ax1.plot(x, y1, color="red")[0]
l2 = ax2.plot(x, y2, color="green")[0]
l3 = ax3.plot(x, y3, color="blue")[0]
l4 = ax3.plot(x, y4, color="orange")[0] # A second line in the third subplot

# Create the legend
fig.legend([l1, l2, l3, l4],      # The line objects
          labels=line_labels,    # The labels for each line
          loc="center right",    # Position of legend
          borderaxespad=0.1,    # Small spacing around legend box
          title="Legend Title"  # Title for the legend
          )

# Adjust the scaling factor to fit your legend text completely outside the plot
# (smaller value results in more space being made for the legend)
plt.subplots_adjust(right=0.85)

plt.show()
```

Zu dem obigen Beispiel ist Folgendes zu beachten:

```
l1 = ax1.plot(x, y1, color="red")[0]
```

Wenn `plot()` aufgerufen wird, wird eine Liste von **line2D**- Objekten zurückgegeben. In diesem Fall wird nur eine Liste mit einem einzelnen *line2D*- Objekt zurückgegeben, die mit der Indexierung `[0]` extrahiert und in `l1` gespeichert wird.

Eine Liste aller *line2D*- Objekte, an denen wir interessiert sind, einschließlich der Legende, muss als erstes Argument an `fig.legend()`. Das zweite Argument zu `fig.legend()` ist ebenfalls notwendig. Es soll eine Liste von Strings sein, die als Beschriftungen für jede Zeile in der Legende verwendet werden.

Die anderen Argumente, die an `fig.legend()` sind rein optional und helfen nur bei der Feinabstimmung der Ästhetik der Legende.

Mehrere Legenden auf derselben Achse

Wenn Sie `plt.legend()` oder `ax.legend()` mehrmals `ax.legend()` , wird die erste Legende entfernt und eine neue Legende gezeichnet. Nach den [offiziellen Unterlagen](#) :

Dies wurde gemacht, damit der Befehl `legend()` wiederholt aufgerufen werden kann, um die Legende auf die neuesten Ziehpunkte der Achsen zu aktualisieren

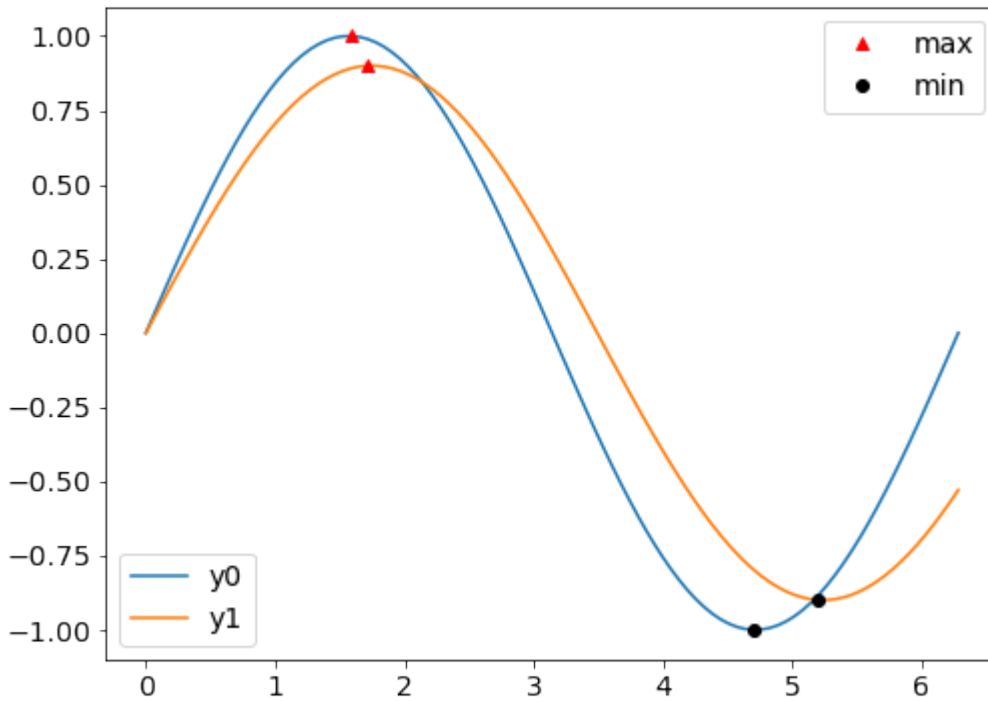
Aber keine Angst: Es ist immer noch recht einfach, einer Achse eine zweite Legende (oder eine dritte oder vierte) hinzuzufügen. In diesem Beispiel zeichnen wir zwei Linien und dann Marker auf ihren jeweiligen Maxima und Minima. Eine Legende bezieht sich auf die Linien und die andere auf die Markierungen.

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

# Generate data for plotting:
x = np.linspace(0,2*np.pi,100)
y0 = np.sin(x)
y1 = .9*np.sin(.9*x)
# Find their maxima and minima and store
maxes = np.empty((2,2))
mins = np.empty((2,2))
for k,y in enumerate([y0,y1]):
    maxloc = y.argmax()
    maxes[k] = x[maxloc], y[maxloc]
    minloc = y.argmin()
    mins[k] = x[minloc], y[minloc]

# Instantiate figure and plot
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,y0, label='y0')
ax.plot(x,y1, label='y1')
# Plot maxima and minima, and keep references to the lines
maxline, = ax.plot(maxes[:,0], maxes[:,1], 'r^')
minline, = ax.plot(mins[:,0], mins[:,1], 'ko')

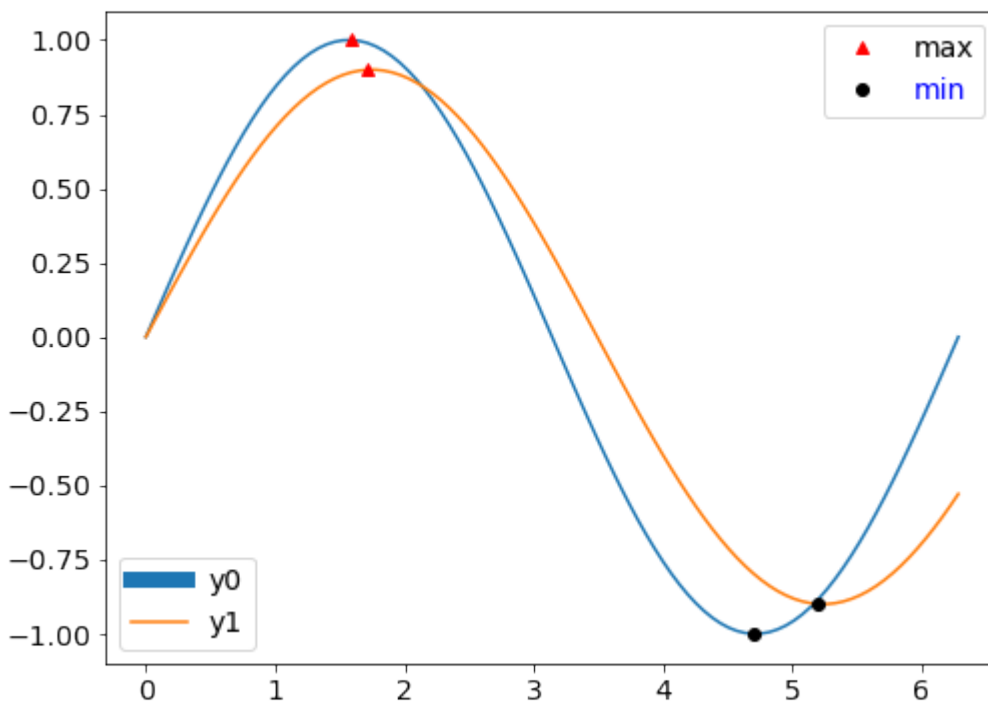
# Add first legend: only labeled data is included
leg1 = ax.legend(loc='lower left')
# Add second legend for the maxes and mins.
# leg1 will be removed from figure
leg2 = ax.legend([maxline,minline],['max','min'], loc='upper right')
# Manually add the first legend back
ax.add_artist(leg1)
```



Der Schlüssel ist, um sicherzustellen, dass Sie Verweise auf die Legendenobjekte haben. Die erste `leg1` Sie instanzieren (`leg1`), wird aus der Figur entfernt, wenn Sie die zweite hinzufügen. Das `leg1` Objekt ist jedoch noch vorhanden und kann mit `ax.add_artist` wieder `ax.add_artist`.

Das wirklich Tolle ist, dass man *beide* Legenden noch manipulieren kann. Fügen Sie beispielsweise am Ende des obigen Codes Folgendes hinzu:

```
leg1.get_lines()[0].set_lw(8)
leg2.get_texts()[1].set_color('b')
```



Schließlich ist es erwähnenswert, dass im Beispiel nur die Linien beim `ax.legend()` mit

Beschriftungen versehen wurden, was bedeutet, dass `ax.legend()` nur diese Zeilen zum `leg1`. Die Legende für die Marker (`leg2`) erforderte daher die Zeilen und Bezeichnungen als Argumente, wenn sie instanziiert wurden. Alternativ könnten wir die Markierungen auch beschriftet haben, wenn sie gezeichnet wurden. Aber dann hätten *beide* Aufrufe von `ax.legend` zusätzliche Argumente benötigt, so dass jede Legende nur die Elemente enthielt, die wir wollten.

Legenden online lesen: <https://riptutorial.com/de/matplotlib/topic/2840/legenden>

Kapitel 17: LogLog-Darstellung

Einführung

Die LogLog-Darstellung ist eine Möglichkeit, eine Exponentialfunktion linear darzustellen.

Examples

LogLog-Darstellung

Sei $y(x) = A \cdot x^a$, zum Beispiel $A = 30$ und $a = 3,5$. Wenn Sie den natürlichen Logarithmus (\ln) beider Seiten verwenden, erhalten Sie (unter Verwendung der allgemeinen Regeln für Logarithmen): $\ln(y) = \ln(A \cdot x^a) = \ln(A) + a \cdot \ln(x)$. Ein Diagramm mit logarithmischen Achsen für x und y ist also eine lineare Kurve. Die Steigung dieser Kurve ist der Exponent a von $y(x)$, während der y -Achsenabschnitt $y(0)$ der natürliche Logarithmus von A ist, $\ln(A) = \ln(30) = 3.401$.

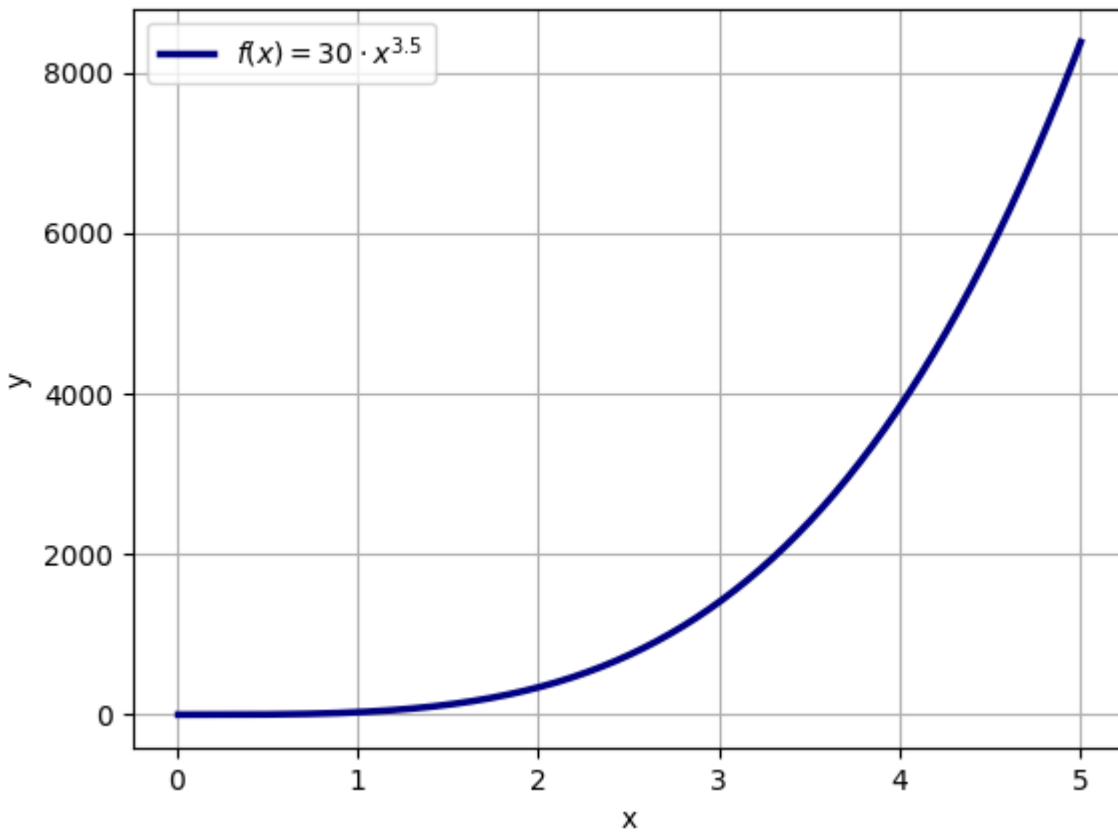
Das folgende Beispiel veranschaulicht die Beziehung zwischen einer Exponentialfunktion und der linearen Protokolldarstellung (die Funktion ist $y = A \cdot x^a$ mit $A = 30$ und $a = 3,5$):

```
import numpy as np
import matplotlib.pyplot as plt
A = 30
a = 3.5
x = np.linspace(0.01, 5, 10000)
y = A * x**a

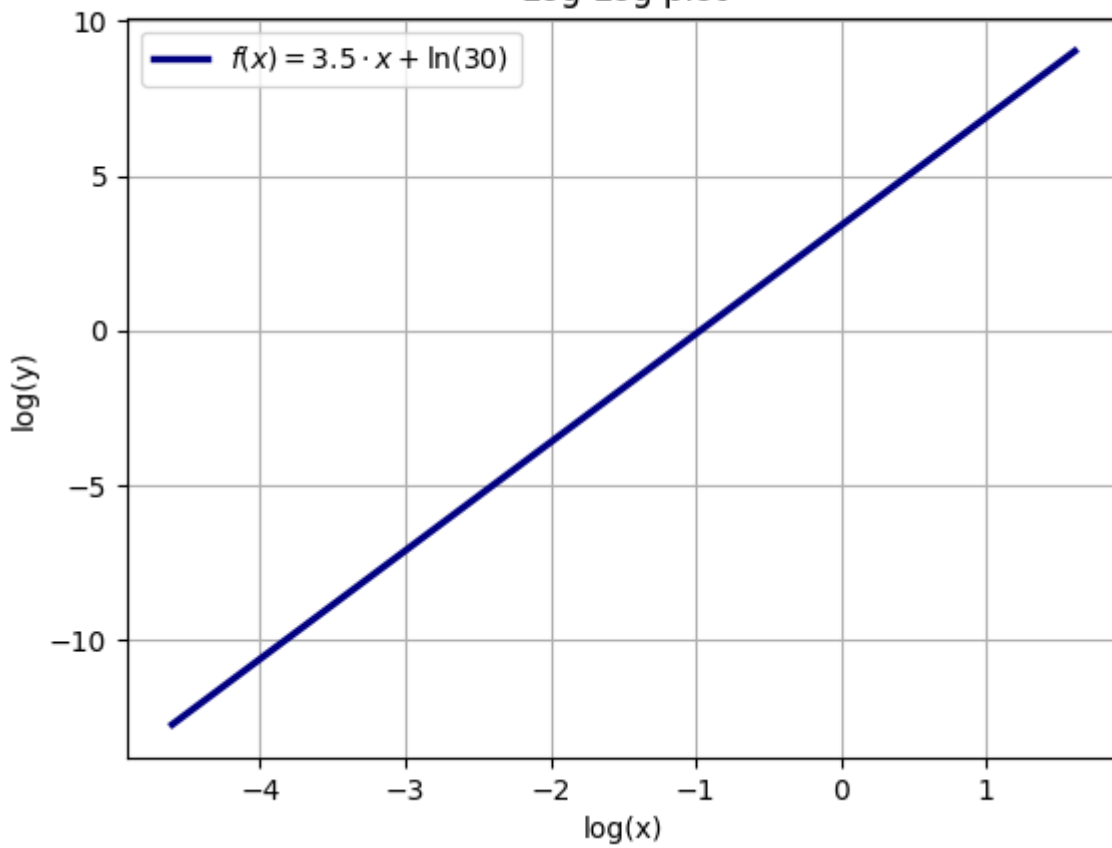
ax = plt.gca()
plt.plot(x, y, linewidth=2.5, color='navy', label=r'$f(x) = 30 \cdot x^{3.5}$')
plt.legend(loc='upper left')
plt.xlabel(r'x')
plt.ylabel(r'y')
ax.grid(True)
plt.title(r'Normal plot')
plt.show()
plt.clf()

xlog = np.log(x)
ylog = np.log(y)
ax = plt.gca()
plt.plot(xlog, ylog, linewidth=2.5, color='navy', label=r'$f(x) = 3.5 \cdot x + \ln(30)$')
plt.legend(loc='best')
plt.xlabel(r'log(x)')
plt.ylabel(r'log(y)')
ax.grid(True)
plt.title(r'Log-Log plot')
plt.show()
plt.clf()
```

Normal plot



Log-Log plot



LogLog-Darstellung online lesen: <https://riptutorial.com/de/matplotlib/topic/10145/loglog-darstellung>

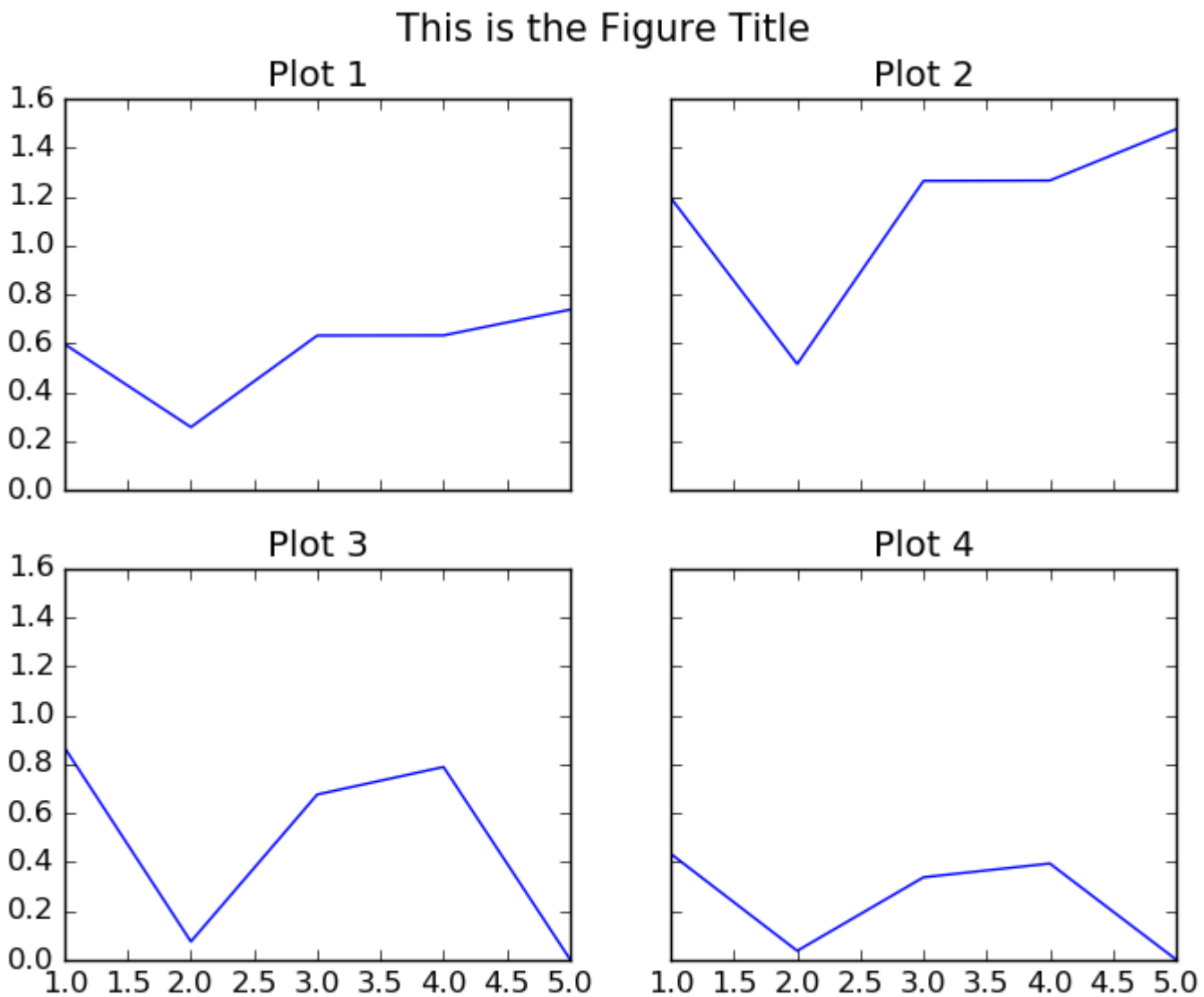
Kapitel 18: Mehrere Plots

Syntax

- Listenpunkt

Examples

Raster von Subplots mit Subplot



```
"""
=====
CREATE A 2 BY 2 GRID OF SUB-PLOTS WITHIN THE SAME FIGURE.
=====
"""
import matplotlib.pyplot as plt
```

```

# The data
x = [1,2,3,4,5]
y1 = [0.59705847, 0.25786401, 0.63213726, 0.63287317, 0.73791151]
y2 = [1.19411694, 0.51572803, 1.26427451, 1.26574635, 1.47582302]
y3 = [0.86793828, 0.07563408, 0.67670068, 0.78932712, 0.0043694]
# 5 more random values
y4 = [0.43396914, 0.03781704, 0.33835034, 0.39466356, 0.0021847]

# Initialise the figure and a subplot axes. Each subplot sharing (showing) the
# same range of values for the x and y axis in the plots.
fig, axes = plt.subplots(2, 2, figsize=(8, 6), sharex=True, sharey=True)

# Set the title for the figure
fig.suptitle('This is the Figure Title', fontsize=15)

# Top Left Subplot
axes[0,0].plot(x, y1)
axes[0,0].set_title("Plot 1")

# Top Right Subplot
axes[0,1].plot(x, y2)
axes[0,1].set_title("Plot 2")

# Bottom Left Subplot
axes[1,0].plot(x, y3)
axes[1,0].set_title("Plot 3")

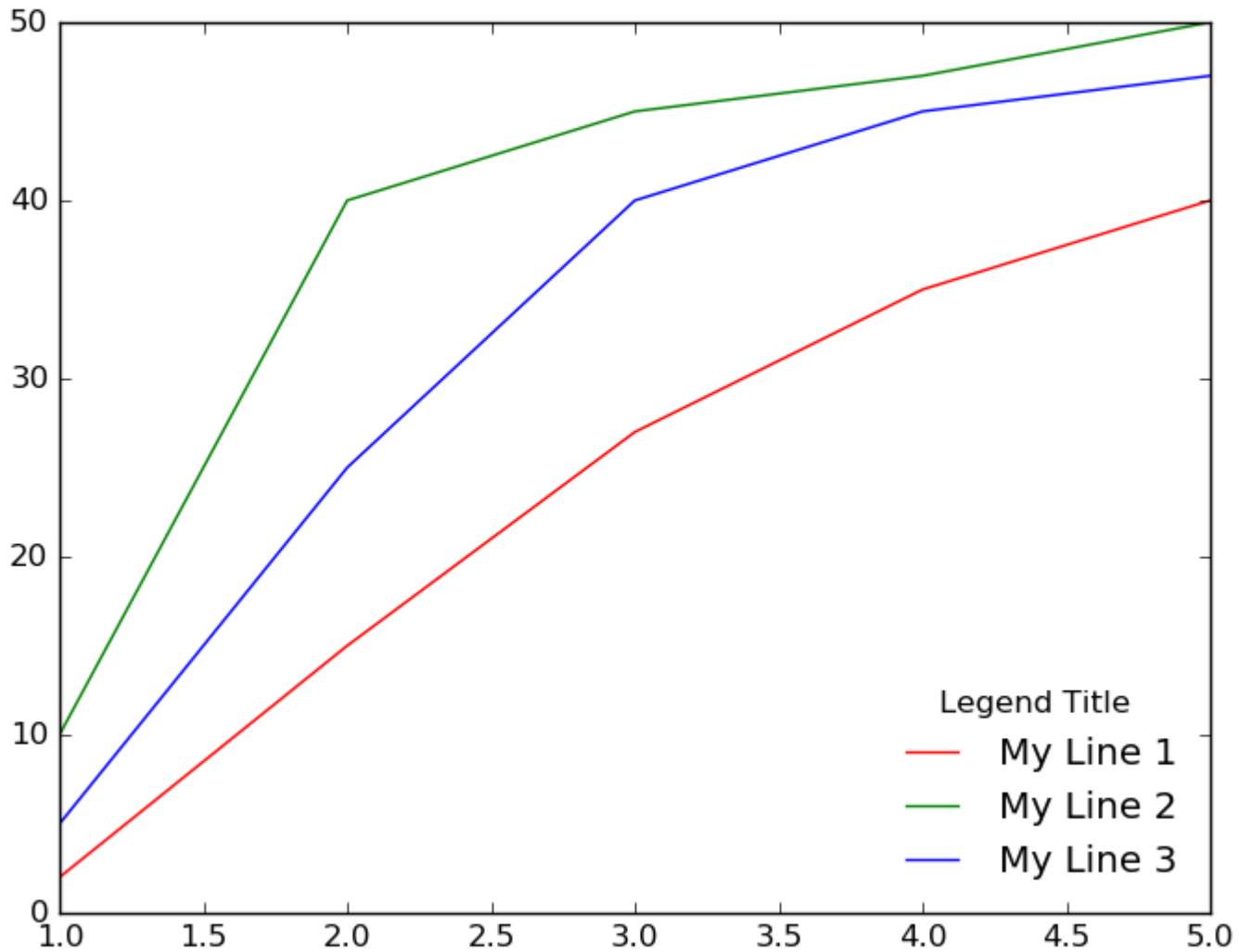
# Bottom Right Subplot
axes[1,1].plot(x, y4)
axes[1,1].set_title("Plot 4")

plt.show()

```

Mehrere Linien / Kurven in derselben Zeichnung

Multiple Lines in Same Plot



```
"""
=====
                        DRAW MULTIPLE LINES IN THE SAME PLOT
=====
"""
import matplotlib.pyplot as plt

# The data
x = [1, 2, 3, 4, 5]
y1 = [2, 15, 27, 35, 40]
y2 = [10, 40, 45, 47, 50]
y3 = [5, 25, 40, 45, 47]

# Initialise the figure and axes.
fig, ax = plt.subplots(1, figsize=(8, 6))

# Set the title for the figure
fig.suptitle('Multiple Lines in Same Plot', fontsize=15)

# Draw all the lines in the same plot, assigning a label for each one to be
# shown in the legend.
ax.plot(x, y1, color="red", label="My Line 1")
ax.plot(x, y2, color="green", label="My Line 2")
```

```

ax.plot(x, y3, color="blue", label="My Line 3")

# Add a legend, and position it on the lower right (with no box)
plt.legend(loc="lower right", title="Legend Title", frameon=False)

plt.show()

```

Mehrere Plots mit Gitterspez

Das Paket `gridspec` ermöglicht eine bessere Kontrolle über die Platzierung von Unterplots. Es erleichtert die Kontrolle der Ränder der Plots und des Abstands zwischen den einzelnen Subplots. Darüber hinaus können Achsen unterschiedlicher Größe in derselben Figur definiert werden, indem Achsen definiert werden, die mehrere Gitterpositionen einnehmen.

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

# Make some data
t = np.arange(0, 2, 0.01)
y1 = np.sin(2*np.pi * t)
y2 = np.cos(2*np.pi * t)
y3 = np.exp(t)
y4 = np.exp(-t)

# Initialize the grid with 3 rows and 3 columns
ncols = 3
nrows = 3
grid = GridSpec(nrows, ncols,
                left=0.1, bottom=0.15, right=0.94, top=0.94, wspace=0.3, hspace=0.3)

fig = plt.figure(0)
fig.clf()

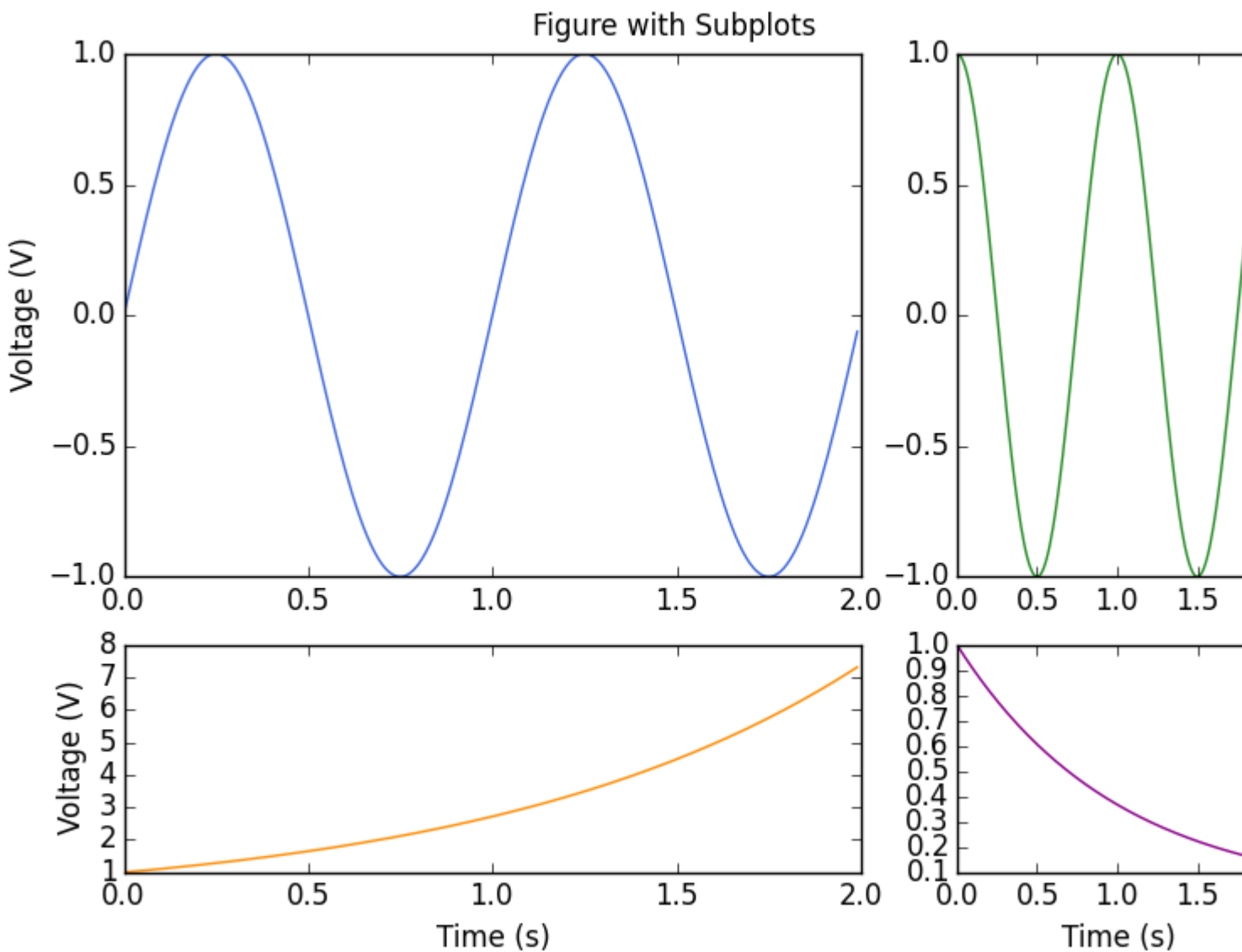
# Add axes which can span multiple grid boxes
ax1 = fig.add_subplot(grid[0:2, 0:2])
ax2 = fig.add_subplot(grid[0:2, 2])
ax3 = fig.add_subplot(grid[2, 0:2])
ax4 = fig.add_subplot(grid[2, 2])

ax1.plot(t, y1, color='royalblue')
ax2.plot(t, y2, color='forestgreen')
ax3.plot(t, y3, color='darkorange')
ax4.plot(t, y4, color='darkmagenta')

# Add labels and titles
fig.suptitle('Figure with Subplots')
ax1.set_ylabel('Voltage (V)')
ax3.set_ylabel('Voltage (V)')
ax3.set_xlabel('Time (s)')
ax4.set_xlabel('Time (s)')

```

Dieser Code erzeugt die unten gezeigte Darstellung.



Eine Darstellung von 2 Funktionen auf der gemeinsamen x-Achse.

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

# create some data
x = np.arange(-2, 20, 0.5)          # values of x
y1 = map(lambda x: -4.0/3.0*x + 16, x)  # values of y1(x)
y2 = map(lambda x: 0.2*x**2 - 5*x + 32, x) # svalues of y2(x)

fig = plt.figure()
ax1 = fig.add_subplot(111)

# create line plot of y1(x)
line1, = ax1.plot(x, y1, 'g', label="Function y1")
ax1.set_xlabel('x')
ax1.set_ylabel('y1', color='g')

# create shared axis for y2(x)
ax2 = ax1.twinx()
```



```

# create line plot of y2(x)
line2, = ax2.plot(x, y2, 'r', label="Function y2")
ax2.set_ylabel('y2', color='r')

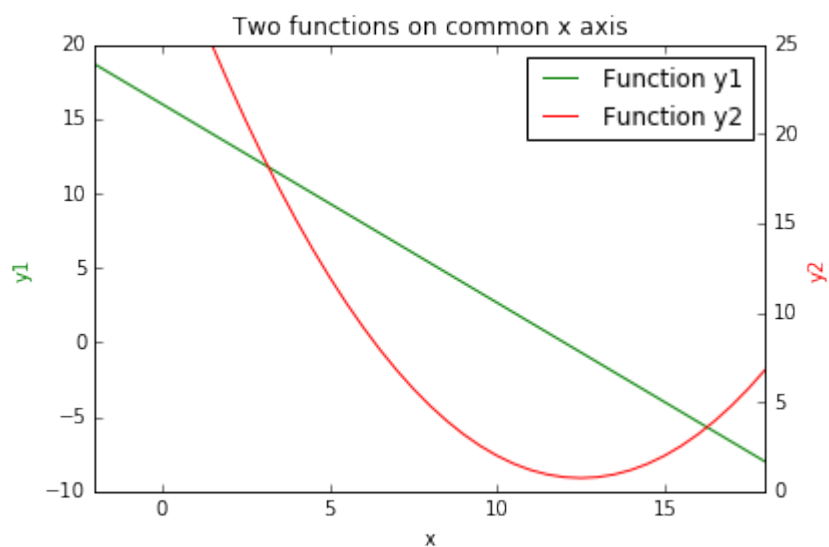
# set title, plot limits, etc
plt.title('Two functions on common x axis')
plt.xlim(-2, 18)
plt.ylim(0, 25)

# add a legend, and position it on the upper right
plt.legend((line1, line2), ('Function y1', 'Function y2'))

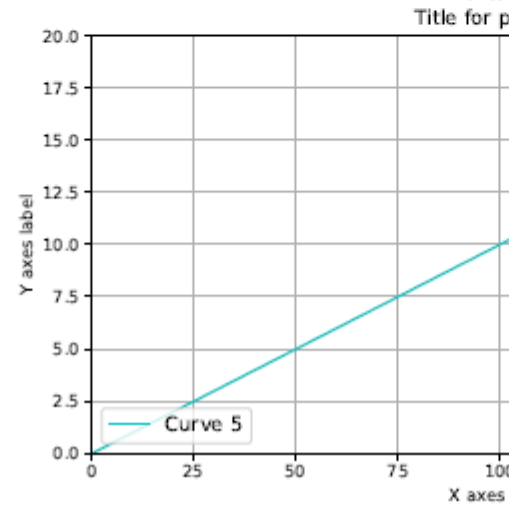
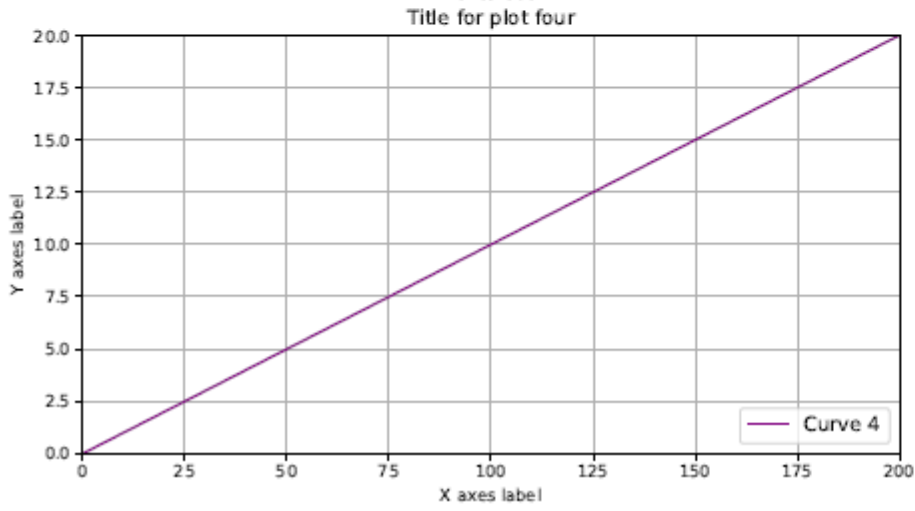
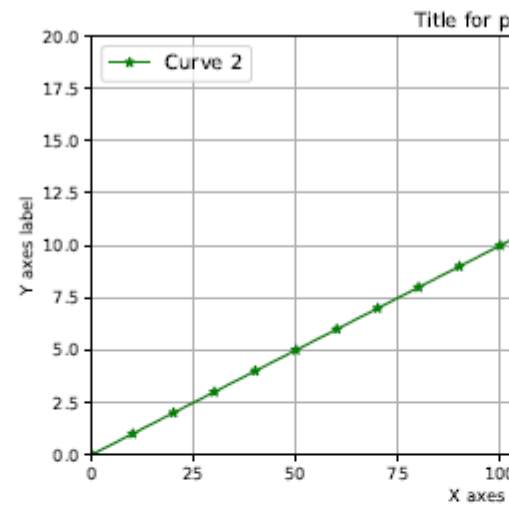
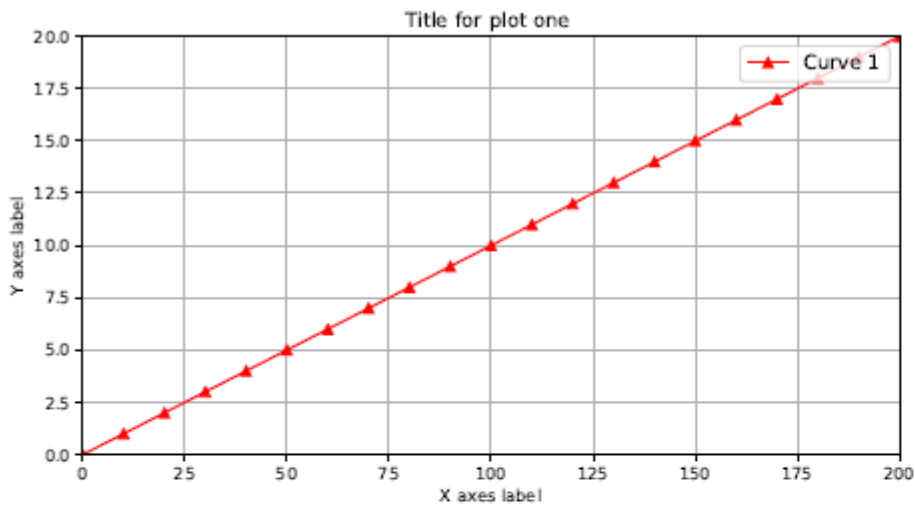
plt.show()

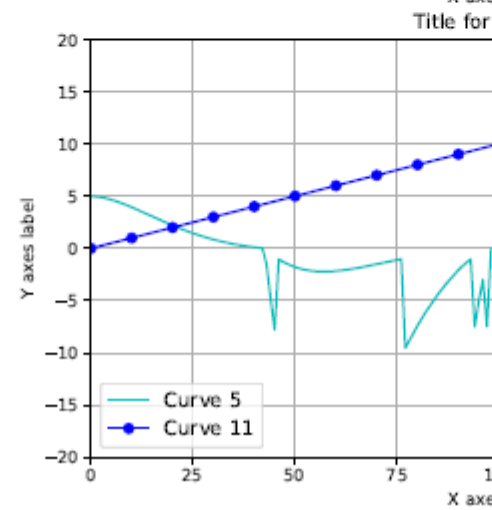
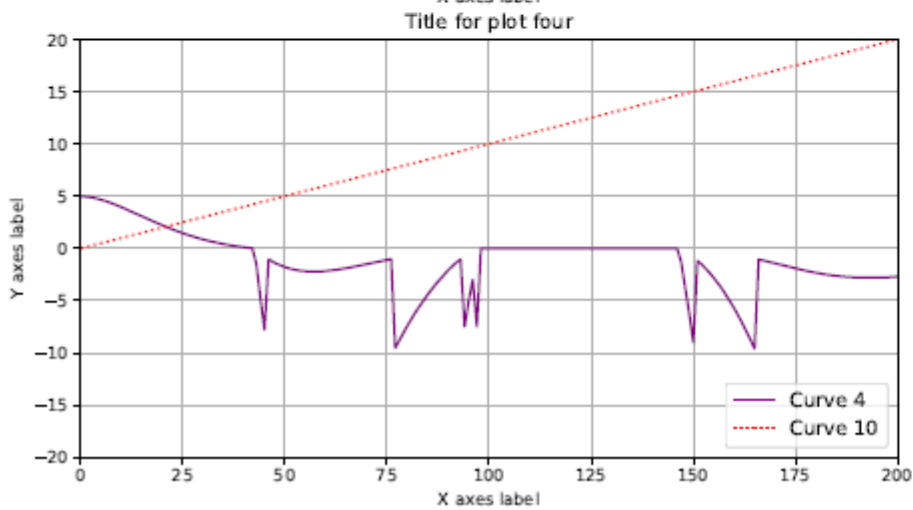
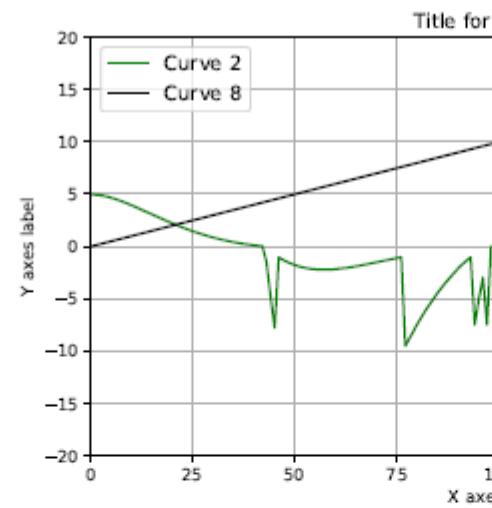
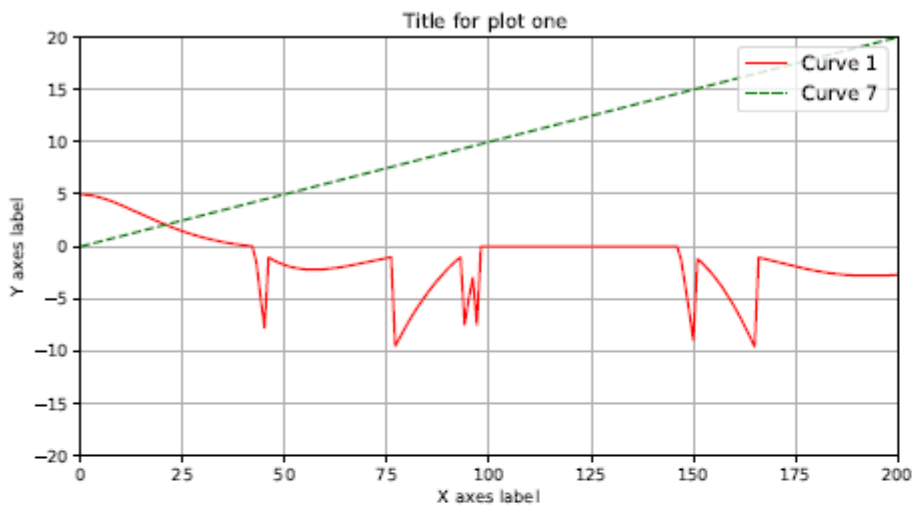
```

Dieser Code erzeugt die unten gezeigte Darstellung.



Mehrere Plots und Multiple Plot-Funktionen





```
CAE.csv
1 TIME,Acceleration
2 0,4.992235
3 0.09952711,4.956489
4 0.1999273,4.915645
5 0.2994544,4.850395
6 0.3998545,4.763977
7 0.4993816,4.65888
8 0.5997818,4.537595
9 0.6993089,4.402862
10 0.799709,4.256423
11 0.8992361,4.100522
12 0.9996362,3.937148
13 1.099163,3.768047
14 1.199564,3.579082
```

```
import matplotlib
matplotlib.use("TKAgg")

# module to save pdf files
from matplotlib.backends.backend_pdf import PdfPages

import matplotlib.pyplot as plt # module to plot

import pandas as pd # module to read csv file
```

```

# module to allow user to select csv file
from tkinter.filedialog import askopenfilename

# module to allow user to select save directory
from tkinter.filedialog import askdirectory

#=====
# User chosen Data for plots
#=====

# User choose csv file then read csv file
filename = askopenfilename() # user selected file
data = pd.read_csv(filename, delimiter=',')

# check to see if data is reading correctly
#print(data)

#=====
# Plots on two different Figures and sets the size of the figures
#=====

# figure size = (width,height)
f1 = plt.figure(figsize=(30,10))
f2 = plt.figure(figsize=(30,10))

#-----
# Figure 1 with 6 plots
#-----

# plot one
# Plot column labeled TIME from csv file and color it red
# subplot(2 Rows, 3 Columns, First subplot,)
ax1 = f1.add_subplot(2,3,1)
ax1.plot(data[["TIME"]], label = 'Curve 1', color = "r", marker = '^', markevery = 10)
# added line marker triangle

# plot two
# plot column labeled TIME from csv file and color it green
# subplot(2 Rows, 3 Columns, Second subplot)
ax2 = f1.add_subplot(2,3,2)
ax2.plot(data[["TIME"]], label = 'Curve 2', color = "g", marker = '*', markevery = 10)
# added line marker star

# plot three
# plot column labeled TIME from csv file and color it blue
# subplot(2 Rows, 3 Columns, Third subplot)
ax3 = f1.add_subplot(2,3,3)
ax3.plot(data[["TIME"]], label = 'Curve 3', color = "b", marker = 'D', markevery = 10)
# added line marker diamond

# plot four
# plot column labeled TIME from csv file and color it purple
# subplot(2 Rows, 3 Columns, Fourth subplot)
ax4 = f1.add_subplot(2,3,4)
ax4.plot(data[["TIME"]], label = 'Curve 4', color = "#800080")

```

```

# plot five
# plot column labeled TIME from csv file and color it cyan
# subplot(2 Rows, 3 Columns, Fifth subplot)
ax5 = f1.add_subplot(2,3,5)
ax5.plot(data[["TIME"]], label = 'Curve 5', color = "c")

# plot six
# plot column labeled TIME from csv file and color it black
# subplot(2 Rows, 3 Columns, Sixth subplot)
ax6 = f1.add_subplot(2,3,6)
ax6.plot(data[["TIME"]], label = 'Curve 6', color = "k")

#-----
# Figure 2 with 6 plots
#-----

# plot one
# Curve 1: plot column labeled Acceleration from csv file and color it red
# Curve 2: plot column labeled      TIME      from csv file and color it green
# subplot(2 Rows, 3 Columns, First subplot)
ax10 = f2.add_subplot(2,3,1)
ax10.plot(data[["Acceleration"]], label = 'Curve 1', color = "r")
ax10.plot(data[["TIME"]], label = 'Curve 7', color="g", linestyle = '--')
# dashed line

# plot two
# Curve 1: plot column labeled Acceleration from csv file and color it green
# Curve 2: plot column labeled      TIME      from csv file and color it black
# subplot(2 Rows, 3 Columns, Second subplot)
ax20 = f2.add_subplot(2,3,2)
ax20.plot(data[["Acceleration"]], label = 'Curve 2', color = "g")
ax20.plot(data[["TIME"]], label = 'Curve 8', color = "k", linestyle = '-')
# solid line (default)

# plot three
# Curve 1: plot column labeled Acceleration from csv file and color it blue
# Curve 2: plot column labeled      TIME      from csv file and color it purple
# subplot(2 Rows, 3 Columns, Third subplot)
ax30 = f2.add_subplot(2,3,3)
ax30.plot(data[["Acceleration"]], label = 'Curve 3', color = "b")
ax30.plot(data[["TIME"]], label = 'Curve 9', color = "#800080", linestyle = '-.')
# dash_dot line

# plot four
# Curve 1: plot column labeled Acceleration from csv file and color it purple
# Curve 2: plot column labeled      TIME      from csv file and color it red
# subplot(2 Rows, 3 Columns, Fourth subplot)
ax40 = f2.add_subplot(2,3,4)
ax40.plot(data[["Acceleration"]], label = 'Curve 4', color = "#800080")
ax40.plot(data[["TIME"]], label = 'Curve 10', color = "r", linestyle = ':')
# dotted line

# plot five
# Curve 1: plot column labeled Acceleration from csv file and color it cyan
# Curve 2: plot column labeled      TIME      from csv file and color it blue
# subplot(2 Rows, 3 Columns, Fifth subplot)

```

```

ax50 = f2.add_subplot(2,3,5)
ax50.plot(data[["Acceleration"]], label = 'Curve 5', color = "c")
ax50.plot(data[["TIME"]], label = 'Curve 11', color = "b", marker = 'o', markevery = 10)
# added line marker circle

# plot six
# Curve 1: plot column labeled Acceleration from csv file and color it black
# Curve 2: plot column labeled      TIME      from csv file and color it cyan
# subplot(2 Rows, 3 Columns, Sixth subplot)
ax60 = f2.add_subplot(2,3,6)
ax60.plot(data[["Acceleration"]], label = 'Curve 6', color = "k")
ax60.plot(data[["TIME"]], label = 'Curve 12', color = "c", marker = 's', markevery = 10)
# added line marker square

=====
# Figure Plot options
=====

#-----
# Figure 1 options
#-----

#switch to figure one for editing
plt.figure(1)

# Plot one options
ax1.legend(loc='upper right', fontsize='large')
ax1.set_title('Title for plot one ')
ax1.set_xlabel('X axes label')
ax1.set_ylabel('Y axes label')
ax1.grid(True)
ax1.set_xlim([0,200])
ax1.set_ylim([0,20])

# Plot two options
ax2.legend(loc='upper left', fontsize='large')
ax2.set_title('Title for plot two ')
ax2.set_xlabel('X axes label')
ax2.set_ylabel('Y axes label')
ax2.grid(True)
ax2.set_xlim([0,200])
ax2.set_ylim([0,20])

# Plot three options
ax3.legend(loc='upper center', fontsize='large')
ax3.set_title('Title for plot three ')
ax3.set_xlabel('X axes label')
ax3.set_ylabel('Y axes label')
ax3.grid(True)
ax3.set_xlim([0,200])
ax3.set_ylim([0,20])

# Plot four options
ax4.legend(loc='lower right', fontsize='large')
ax4.set_title('Title for plot four')
ax4.set_xlabel('X axes label')
ax4.set_ylabel('Y axes label')
ax4.grid(True)
ax4.set_xlim([0,200])

```

```

ax4.set_ylim([0,20])

# Plot five options
ax5.legend(loc='lower left', fontsize='large')
ax5.set_title('Title for plot five ')
ax5.set_xlabel('X axes label')
ax5.set_ylabel('Y axes label')
ax5.grid(True)
ax5.set_xlim([0,200])
ax5.set_ylim([0,20])

# Plot six options
ax6.legend(loc='lower center', fontsize='large')
ax6.set_title('Title for plot six')
ax6.set_xlabel('X axes label')
ax6.set_ylabel('Y axes label')
ax6.grid(True)
ax6.set_xlim([0,200])
ax6.set_ylim([0,20])

#-----
# Figure 2 options
#-----

#switch to figure two for editing
plt.figure(2)

# Plot one options
ax10.legend(loc='upper right', fontsize='large')
ax10.set_title('Title for plot one ')
ax10.set_xlabel('X axes label')
ax10.set_ylabel('Y axes label')
ax10.grid(True)
ax10.set_xlim([0,200])
ax10.set_ylim([-20,20])

# Plot two options
ax20.legend(loc='upper left', fontsize='large')
ax20.set_title('Title for plot two ')
ax20.set_xlabel('X axes label')
ax20.set_ylabel('Y axes label')
ax20.grid(True)
ax20.set_xlim([0,200])
ax20.set_ylim([-20,20])

# Plot three options
ax30.legend(loc='upper center', fontsize='large')
ax30.set_title('Title for plot three ')
ax30.set_xlabel('X axes label')
ax30.set_ylabel('Y axes label')
ax30.grid(True)
ax30.set_xlim([0,200])
ax30.set_ylim([-20,20])

# Plot four options
ax40.legend(loc='lower right', fontsize='large')
ax40.set_title('Title for plot four')
ax40.set_xlabel('X axes label')
ax40.set_ylabel('Y axes label')
ax40.grid(True)
ax40.set_xlim([0,200])

```

```

ax40.set_ylim([-20,20])

# Plot five options
ax50.legend(loc='lower left', fontsize='large')
ax50.set_title('Title for plot five ')
ax50.set_xlabel('X axes label')
ax50.set_ylabel('Y axes label')
ax50.grid(True)
ax50.set_xlim([0,200])
ax50.set_ylim([-20,20])

# Plot six options
ax60.legend(loc='lower center', fontsize='large')
ax60.set_title('Title for plot six')
ax60.set_xlabel('X axes label')
ax60.set_ylabel('Y axes label')
ax60.grid(True)
ax60.set_xlim([0,200])
ax60.set_ylim([-20,20])

#=====
# User chosen file location Save PDF
#=====

savefilename = askdirectory()# user selected file path
pdf = PdfPages(f'{savefilename}/longplot.pdf')
# using formatted string literals ("f-strings")to place the variable into the string

# save both figures into one pdf file
pdf.savefig(1)
pdf.savefig(2)

pdf.close()

#=====
# Show plot
#=====

# manually set the subplot spacing when there are multiple plots
#plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace =None, hspace=None )

# Automaticlly adds space between plots
plt.tight_layout()

plt.show()

```

Mehrere Plots online lesen: <https://riptutorial.com/de/matplotlib/topic/3279/mehrere-plots>

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit matplotlib	Amitay Stern , ChaoticTwist , Chr , Chris Mueller , Community , dermen , evtoh , farenorth , Josh , jrjc , pmos , Serenity , tacaswell
2	Animationen und interaktives Plotten	FiN , smurfendrek123 , user2314737
3	Bildbearbeitung	Bosoneando
4	Boxplots	Luis
5	Colormaps	Andras Deak , Xevaquor
6	Dreidimensionale Diagramme	Andras Deak , Serenity , will
7	Figuren und Äxte Objekte	David Zwicker , Josh , Serenity , tom
8	Figurfenster schließen	Brian , David Zwicker
9	Gitterlinien und Häkchen	ronrest
10	Grundlegende Diagramme	Franck Deroncourt , Josh , ml4294 , ronrest , Scimonster , Serenity , user2314737
11	Histogramm	Yegor Kishilov
12	Integration mit TeX / LaTeX	Andras Deak , Bosoneando , Chris Mueller , Næreen , Serenity
13	Konturkarten	Eugene Loy , Serenity
14	Koordinatensysteme	jure
15	Legenden	Andras Deak , Franck Deroncourt , ronrest , saintsfan342000 , Serenity
16	LogLog-Darstellung	ml4294
17	Mehrere Plots	Chris Mueller , Robert Branam , ronrest , swatchai