vorlesung_matplotlib

November 23, 2022

1 matplotlib

Will man mit Python Abbildung erzeugen, greift man meistens auf das Modul `matplotlib` zurück. Dieses Modul bietet eine Vielzahl an Möglichkeiten um Daten zu visualisieren. Der Syntax orientiert sich dabei stark an dem kommerziellen Programm *matlab*.

Wie immer werden zuerst die Module importiert. *matplotlib* benötigt *numpy* das daher zuerst eingebunden wird. *matplotlib* besteht aus verschiedenen Submodulen, die einzeln eingebunden werden. Hier verwenden wir das Submodul *pyplot*, für das sich die Abkürzung *plt* etabliert hat.

[1]:
```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
```

Die Zeile `%matplotlib inline` ist etwas Jupyter-Magie. Es gibt ein paar Befehle, die man an Jupyter schicken kann. Diese beginnen immer mit `%`. Hier wird Jupyter gesagt wo/wie die Abbildungen auftauchen sollen. (Im Prinzip findet es Jupyter sonst selbst heraus, leider aber erst **nach** der ersten Abbildung.)

1.1 Erster Plot

matplotlib stellt numpy-arrays grafisch dar. Daher erzeugen wir zuerst ein Array

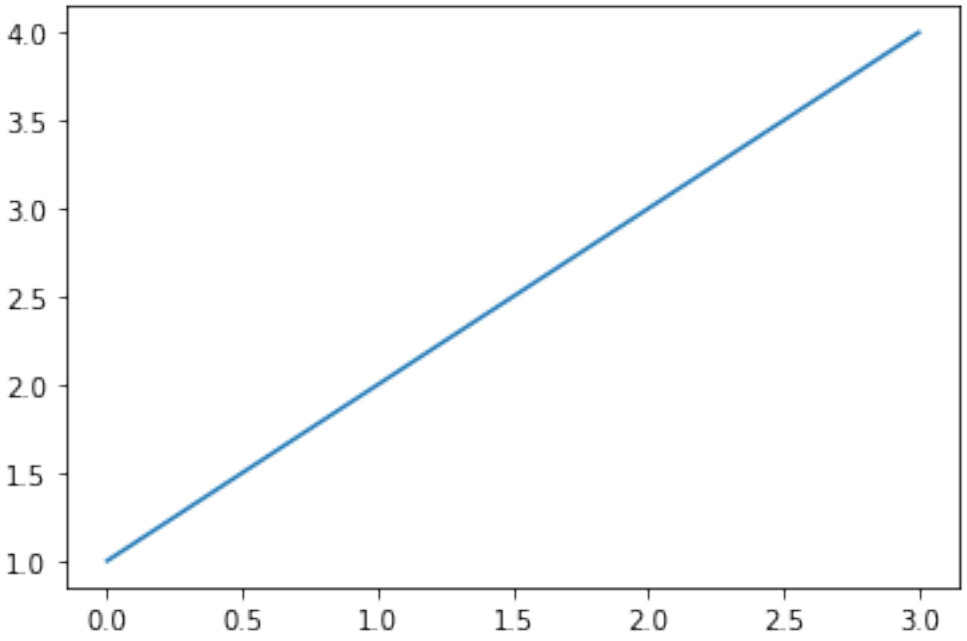
[2]:
```
a = np.arange(1, 5)
print(a)
```

```
[1 2 3 4]
```

und plotten es dann:

[3]:
```
plt.plot(a)
```

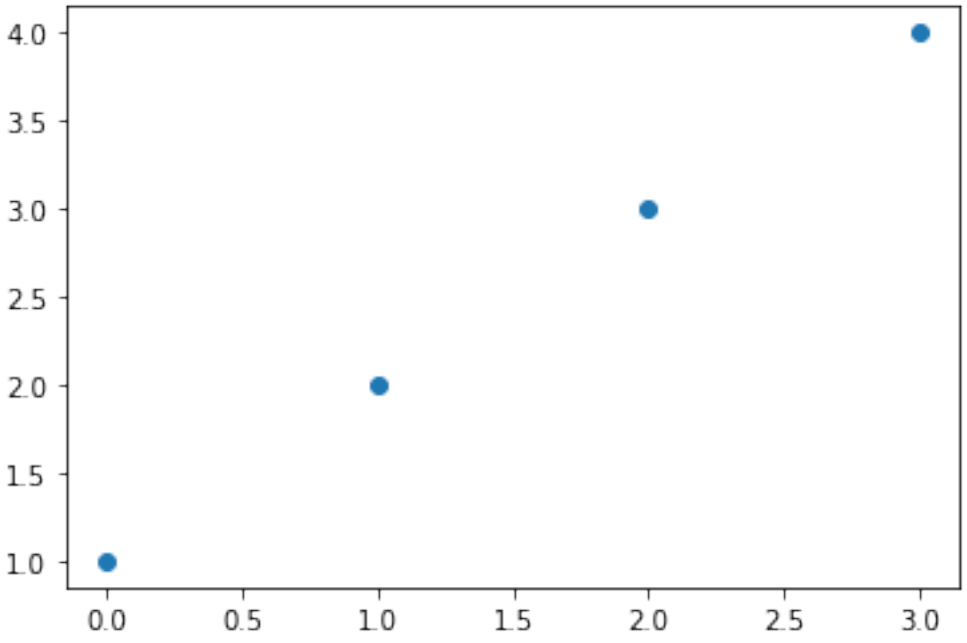
[3]:
```
[<matplotlib.lines.Line2D at 0x7f2541c9f9a0>]
```



obwohl wir vier Werte übergeben haben, erhalten wir eine Linie. Dies liegt an einem für uns etwas ungünstigen Standardverhalten: plt.plot() verbindet normalerweise alle Punkte mit einer Linie. Inbesondere Messdaten sollte aber als einzelne Punkte dargestellt werden. Dafür muss man `plt.plot()` explizit sagen das und mit welchem Symbol die Werte dargestellt werden.

[4]:
```
plt.plot(a, 'o')
```

[4]:
```
[<matplotlib.lines.Line2D at 0x7f253fb3d790>]
```

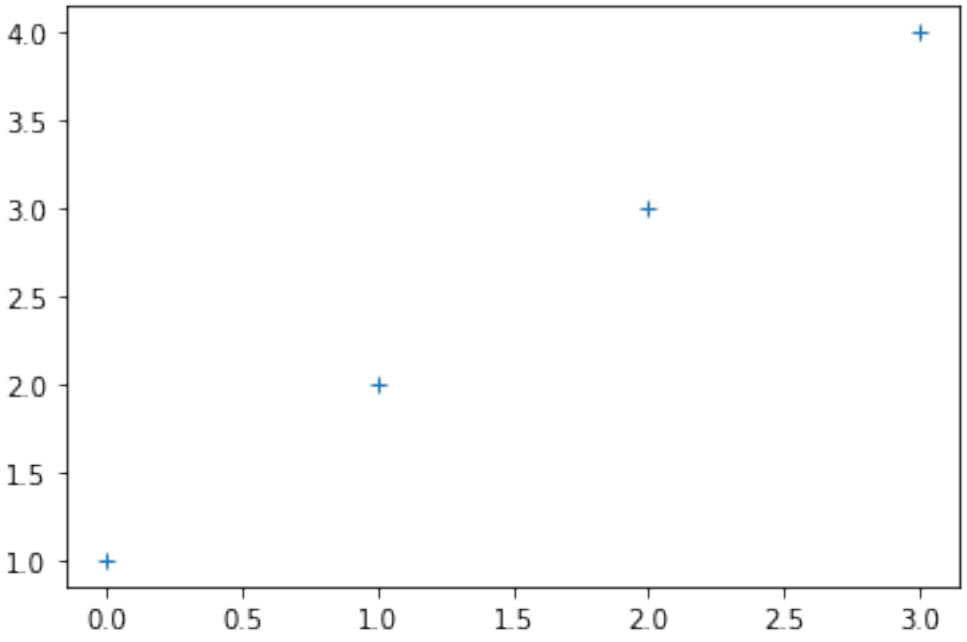


Jetzt erkent man auch, dass vier Werte geplottet wurden. Die Werte wurden in y-Richtung aufgetragen, als x-Werte wurden die Indizes verwendet.

Der Buchstabe ist eine Abkürzung für das Symbol, ein Pluszeichen bekomme ich z.B. so

[5]:
```
plt.plot(a, '+')
```

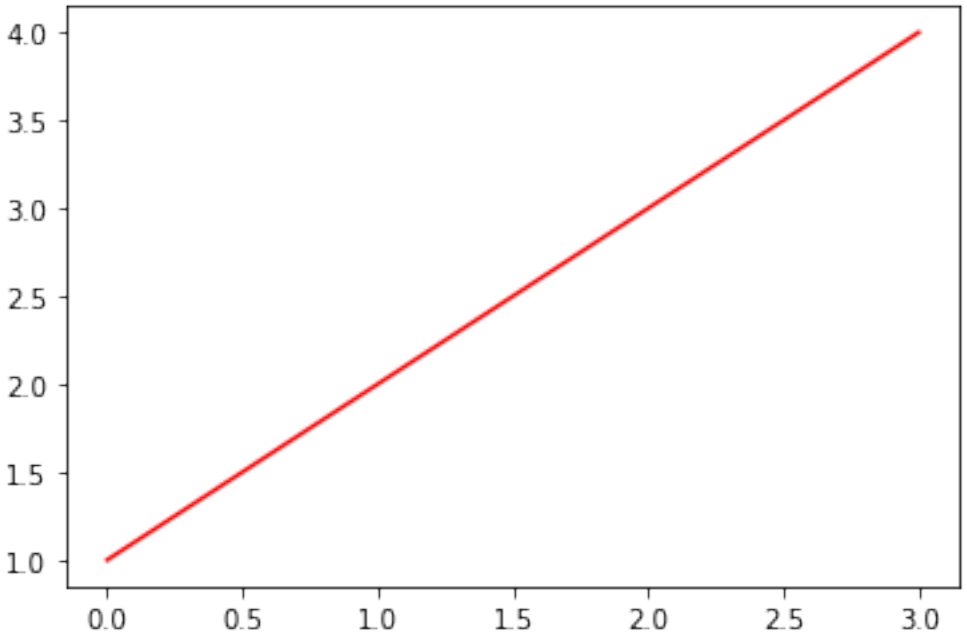
[5]:
```
[<matplotlib.lines.Line2D at 0x7f253fab6be0>]
```



Farben kann man über die Option *color* einstellen

[6]:
```
plt.plot(a, color="red")
```

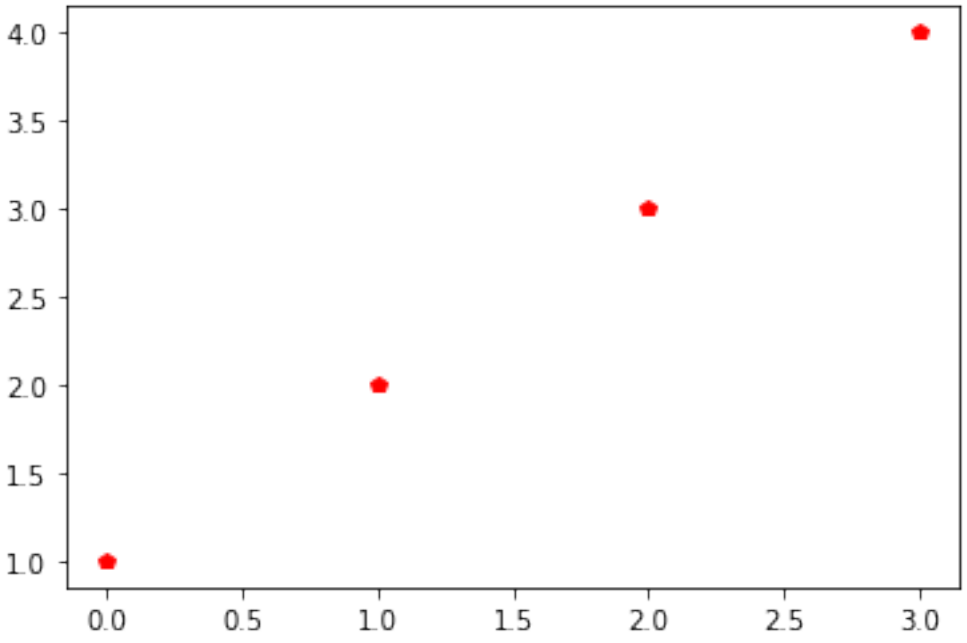
[6]:
```
[<matplotlib.lines.Line2D at 0x7f253fa3c7f0>]
```



oder auch als Abkürzung übergeben

[7]:
```
plt.plot(a, 'pr')
```

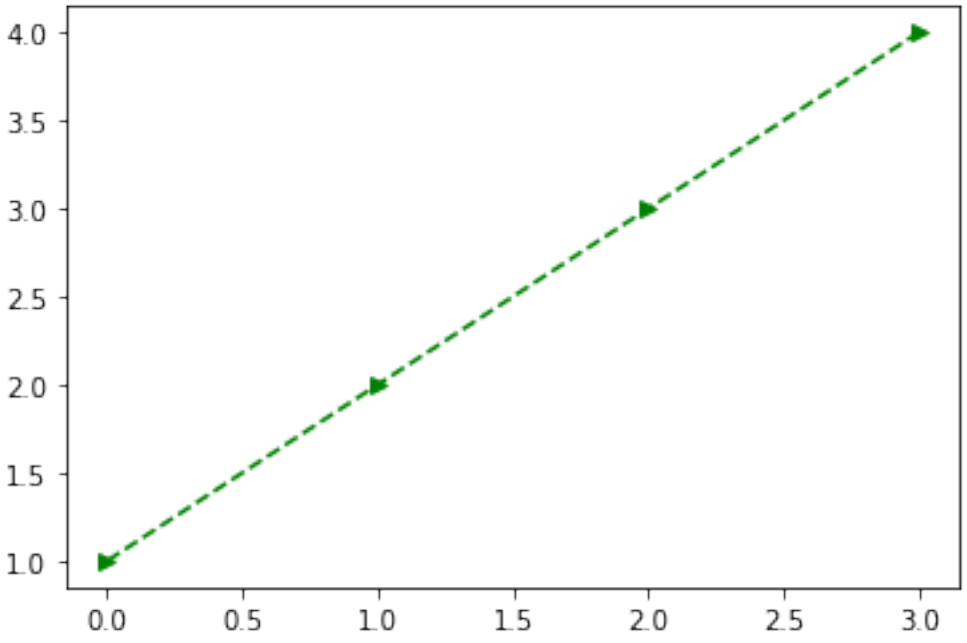
[7]:
```
[<matplotlib.lines.Line2D at 0x7f253f9c0760>]
```



sogar der Linienstil funktioniert über Abkürzungen

[8]:
```
plt.plot(a, '>g--')
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7f253f938b50>]
```



Sollen Funktionen dargestellt werden die Funktionen von Numpy praktisch. Erstelle ich mit `linspace()` einen Array von X-Koordinaten

[9]:
```
x = np.linspace(0, np.pi, 100)
```

und berechne damit eine Funktion

[10]:
```
y = np.sin(x)
y
```

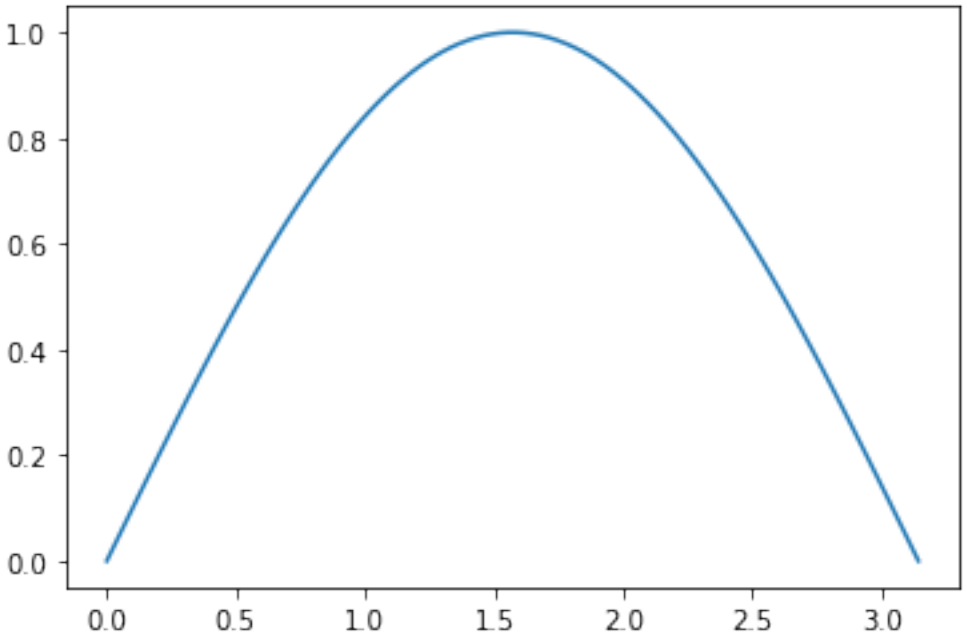
[10]:
```
array([0.00000000e+00, 3.17279335e-02, 6.34239197e-02, 9.50560433e-02,
       1.26592454e-01, 1.58001396e-01, 1.89251244e-01, 2.20310533e-01,
       2.51147987e-01, 2.81732557e-01, 3.12033446e-01, 3.42020143e-01,
       3.71662456e-01, 4.00930535e-01, 4.29794912e-01, 4.58226522e-01,
       4.86196736e-01, 5.13677392e-01, 5.40640817e-01, 5.67059864e-01,
       5.92907929e-01, 6.18158986e-01, 6.42787610e-01, 6.66769001e-01,
       6.90079011e-01, 7.12694171e-01, 7.34591709e-01, 7.55749574e-01,
       7.76146464e-01, 7.95761841e-01, 8.14575952e-01, 8.32569855e-01,
       8.49725430e-01, 8.66025404e-01, 8.81453363e-01, 8.95993774e-01,
       9.09631995e-01, 9.22354294e-01, 9.34147860e-01, 9.45000819e-01,
       9.54902241e-01, 9.63842159e-01, 9.71811568e-01, 9.78802446e-01,
       9.84807753e-01, 9.89821442e-01, 9.93838464e-01, 9.96854776e-01,
       9.98867339e-01, 9.99874128e-01, 9.99874128e-01, 9.98867339e-01,
       9.96854776e-01, 9.93838464e-01, 9.89821442e-01, 9.84807753e-01,
       9.78802446e-01, 9.71811568e-01, 9.63842159e-01, 9.54902241e-01,
       9.45000819e-01, 9.34147860e-01, 9.22354294e-01, 9.09631995e-01,
       8.95993774e-01, 8.81453363e-01, 8.66025404e-01, 8.49725430e-01,
       8.32569855e-01, 8.14575952e-01, 7.95761841e-01, 7.76146464e-01,
       7.55749574e-01, 7.34591709e-01, 7.12694171e-01, 6.90079011e-01,
       6.66769001e-01, 6.42787610e-01, 6.18158986e-01, 5.92907929e-01,
       5.67059864e-01, 5.40640817e-01, 5.13677392e-01, 4.86196736e-01,
       4.58226522e-01, 4.29794912e-01, 4.00930535e-01, 3.71662456e-01,
       3.42020143e-01, 3.12033446e-01, 2.81732557e-01, 2.51147987e-01,
       2.20310533e-01, 1.89251244e-01, 1.58001396e-01, 1.26592454e-01,
       9.50560433e-02, 6.34239197e-02, 3.17279335e-02, 1.22464680e-16])
```

erhalte ich einen Array von y-Koordinaten.

Dies kann ich nun nutzen um den Funktionsverlauf darzustellen. Übergebe ich `plot()` zwei Arrays, wird der erste als eine Liste von X-Koordinaten, der zweite als eine Liste von Y-Koordinaten interpretiert und ich sehe den Funktionsverlauf.

[11]:
```
plt.plot(x, y)
```

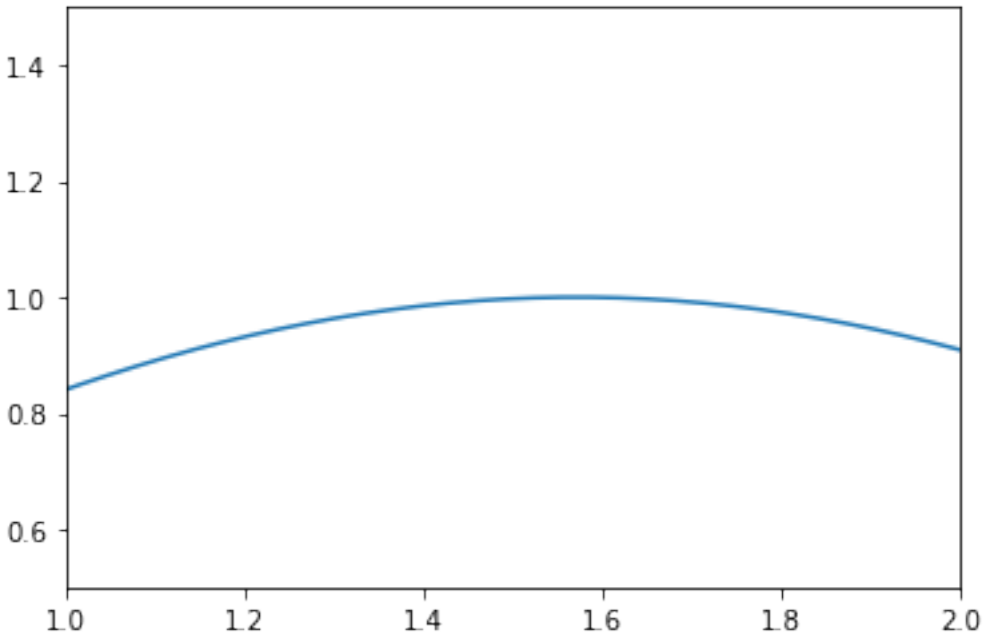
[11]:
```
[<matplotlib.lines.Line2D at 0x7f253f8beb20>]
```



Will ich nur einen Ausschnitt meiner Kurve auch darstellen, so helfen `xlim()` und `ylim()`

[12]:
```
plt.xlim(1, 2)
plt.ylim(0.5, 1.5)
plt.plot(x, y)
```

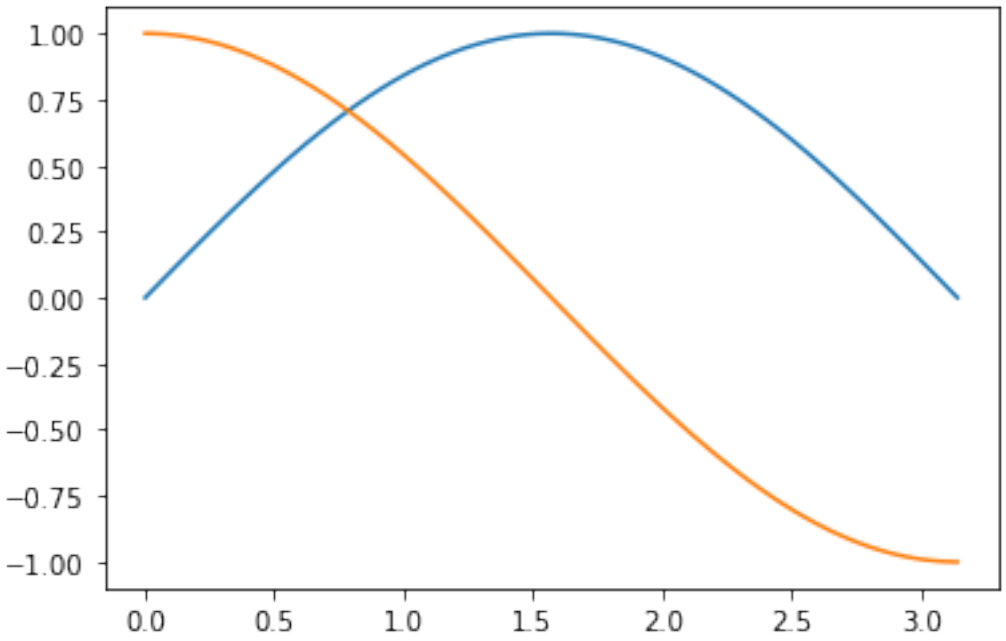
[12]:
```
[<matplotlib.lines.Line2D at 0x7f253f83f460>]
```



Wird `plot()` in einer Jupyterzelle mehrfach aufgerufen, werden die entsprechenden Kurven in einer Abbildung gezeichnet. Matplotlib wechselt dabei von sich aus die Farbe der Kurven.

[13]:
```
plt.plot(x, y)
plt.plot(x, np.cos(x))
```

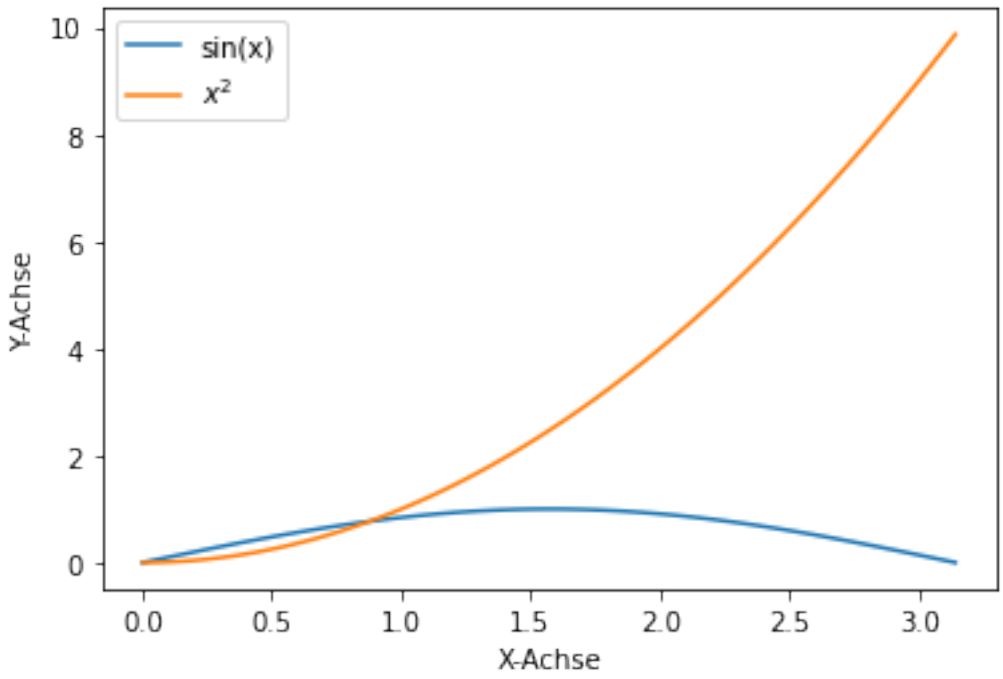
[13]:
```
[<matplotlib.lines.Line2D at 0x7f253f82f1c0>]
```



Die Abbildung muss jetzt noch mit Achsenbeschriftungen und einer Legende versorgt werden:

[14]:
```
plt.plot(x, y)
plt.plot(x, x**2)
plt.xlabel('X-Achse')
plt.ylabel('Y-Achse')
plt.legend(['sin(x)', '$x^2$'])
```

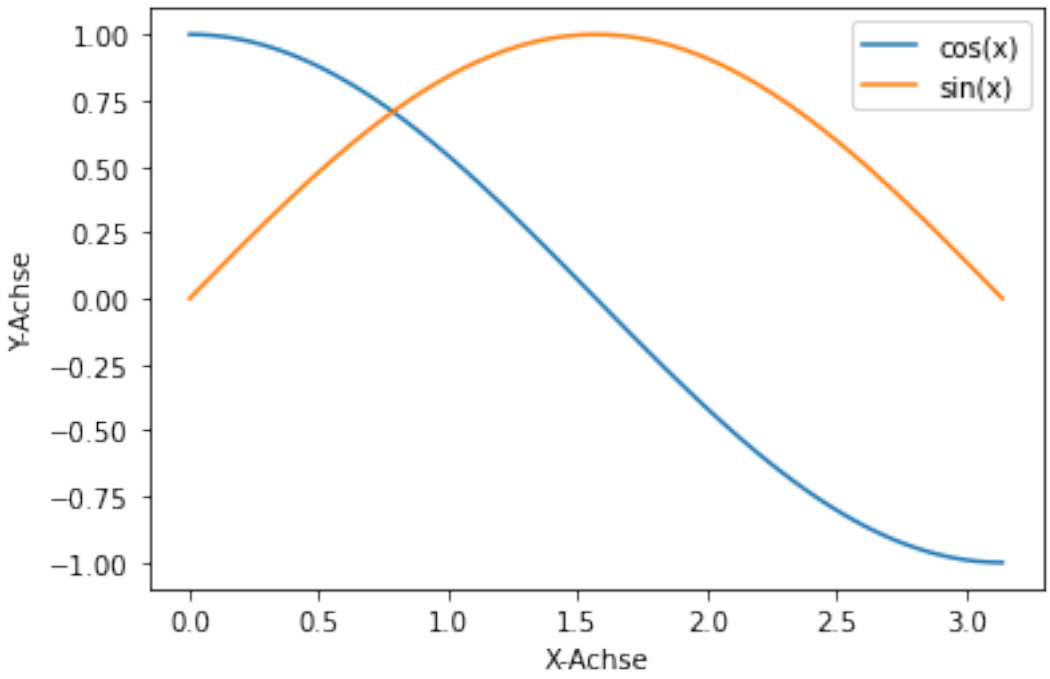
[14]:
```
<matplotlib.legend.Legend at 0x7f253fad5850>
```



Bei Legenden ist es meistens günstiger die entsprechenden Beschriftungen als *label* gleich bei `plot()` anzugeben. Dann gibt man bei `legend()` keine Beschriftungen an. Ich kann aber trotzdem noch z.B. die Position der Legende angeben.

Am Ende möchte man die Abbildung vielleicht als Datei. `savefig(FILENAME)` erzeugt diese Datei, wobei die Dateiendung das Datenformat bestimmt.

[15]:
```
plt.plot(x, np.cos(x), label="cos(x)")
plt.plot(x, np.sin(x), label="sin(x)")
plt.xlabel('X-Achse')
plt.ylabel('Y-Achse')
plt.legend(loc='upper right')
plt.savefig('plot1.pdf')
```



[16]:
```
help(plt.plot)
```

```
Help on function plot in module matplotlib.pyplot:

plot(*args, scalex=True, scaley=True, data=None, **kwargs)
    Plot y versus x as lines and/or markers.

    Call signatures::

        plot([x], y, [fmt], *, data=None, **kwargs)
        plot([x], y, [fmt], [x2], y2, [fmt2], …, **kwargs)

    The coordinates of the points or line nodes are given by *x*, *y*.

    The optional parameter *fmt* is a convenient way for defining basic
    formatting like color, marker and linestyle. It's a shortcut string
    notation described in the *Notes* section below.

    >>> plot(x, y)        # plot x and y using default line style and color
    >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
    >>> plot(y)           # plot y using x as index array 0..N-1
    >>> plot(y, 'r+')     # ditto, but with red plusses

    You can use `.Line2D` properties as keyword arguments for more
    control on the appearance. Line properties and *fmt* can be mixed.
```

```
The following two calls yield identical results:

>>> plot(x, y, 'go--', linewidth=2, markersize=12)
>>> plot(x, y, color='green', marker='o', linestyle='dashed',
…      linewidth=2, markersize=12)

When conflicting with *fmt*, keyword arguments take precedence.


**Plotting labelled data**

There's a convenient way for plotting objects with labelled data (i.e.
data that can be accessed by index ``obj['y']``). Instead of giving
the data in *x* and *y*, you can provide the object in the *data*
parameter and just give the labels for *x* and *y*::

>>> plot('xlabel', 'ylabel', data=obj)

All indexable objects are supported. This could e.g. be a `dict`, a
`pandas.DataFrame` or a structured numpy array.


**Plotting multiple sets of data**

There are various ways to plot multiple sets of data.

- The most straight forward way is just to call `plot` multiple times.
  Example:

  >>> plot(x1, y1, 'bo')
  >>> plot(x2, y2, 'go')

- If *x* and/or *y* are 2D arrays a separate data set will be drawn
  for every column. If both *x* and *y* are 2D, they must have the
  same shape. If only one of them is 2D with shape (N, m) the other
  must have length N and will be used for every data set m.

  Example:

  >>> x = [1, 2, 3]
  >>> y = np.array([[1, 2], [3, 4], [5, 6]])
  >>> plot(x, y)

  is equivalent to:

  >>> for col in range(y.shape[1]):
  …     plot(x, y[:, col])
```

```
- The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
  groups::

  >>> plot(x1, y1, 'g^', x2, y2, 'g-')

  In this case, any additional keyword argument applies to all
  datasets. Also this syntax cannot be combined with the *data*
  parameter.

By default, each line is assigned a different style specified by a
'style cycle'. The *fmt* and line property parameters are only
necessary if you want explicit deviations from these defaults.
Alternatively, you can also change the style cycle using
:rc:`axes.prop_cycle`.


Parameters
----------
x, y : array-like or scalar
    The horizontal / vertical coordinates of the data points.
    *x* values are optional and default to ``range(len(y))``.

    Commonly, these parameters are 1D arrays.

    They can also be scalars, or two-dimensional (in that case, the
    columns represent separate data sets).

    These arguments cannot be passed as keywords.

fmt : str, optional
    A format string, e.g. 'ro' for red circles. See the *Notes*
    section for a full description of the format strings.

    Format strings are just an abbreviation for quickly setting
    basic line properties. All of these and more can also be
    controlled by keyword arguments.

    This argument cannot be passed as keyword.

data : indexable object, optional
    An object with labelled data. If given, provide the label names to
    plot in *x* and *y*.

    .. note::
        Technically there's a slight ambiguity in calls where the
        second label is a valid *fmt*. ``plot('n', 'o', data=obj)``
        could be ``plt(x, y)`` or ``plt(y, fmt)``. In such cases,
        the former interpretation is chosen, but a warning is issued.
```

```
            You may suppress the warning by adding an empty format string
            ``plot('n', 'o', '', data=obj)``.

    Returns
    -------
    list of `.Line2D`
        A list of lines representing the plotted data.

    Other Parameters
    ----------------
    scalex, scaley : bool, default: True
        These parameters determine if the view limits are adapted to the
        data limits. The values are passed on to
        `~.axes.Axes.autoscale_view`.

    **kwargs : `.Line2D` properties, optional
        *kwargs* are used to specify properties like a line label (for
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1, 2, 3], [1, 2, 3], 'go-', label='line 1', linewidth=2)
        >>> plot([1, 2, 3], [1, 4, 9], 'rs', label='line 2')

        If you specify multiple lines with one plot call, the kwargs apply
        to all those lines. In case the label object is iterable, each
        element is used as labels for each set of data.

        Here is a list of available `.Line2D` properties:

        Properties:
        agg_filter: a filter function, which takes a (m, n, 3) float array and a
dpi value, and returns a (m, n, 3) array and two offsets from the bottom left
corner of the image
        alpha: scalar or None
        animated: bool
        antialiased or aa: bool
        clip_box: `.Bbox`
        clip_on: bool
        clip_path: Patch or (Path, Transform) or None
        color or c: color
        dash_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
        dash_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
        dashes: sequence of floats (on/off ink in points) or (None, None)
        data: (2, N) array or two 1D arrays
        drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-
post'}, default: 'default'
        figure: `.Figure`
        fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
```

```
        gapcolor: color or None
        gid: str
        in_layout: bool
        label: object
        linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), …}
        linewidth or lw: float
        marker: marker style string, `~.path.Path` or `~.markers.MarkerStyle`
        markeredgecolor or mec: color
        markeredgewidth or mew: float
        markerfacecolor or mfc: color
        markerfacecoloralt or mfcalt: color
        markersize or ms: float
        markevery: None or int or (int, int) or slice or list[int] or float or
(float, float) or list[bool]
        mouseover: bool
        path_effects: `.AbstractPathEffect`
        picker: float or callable[[Artist, Event], tuple[bool, dict]]
        pickradius: unknown
        rasterized: bool
        sketch_params: (scale: float, length: float, randomness: float)
        snap: bool or None
        solid_capstyle: `.CapStyle` or {'butt', 'projecting', 'round'}
        solid_joinstyle: `.JoinStyle` or {'miter', 'round', 'bevel'}
        transform: unknown
        url: str
        visible: bool
        xdata: 1D array
        ydata: 1D array
        zorder: float

    See Also
    --------
    scatter : XY scatter plot with markers of varying size and/or color (
        sometimes also called bubble chart).

    Notes
    -----
    **Format Strings**

    A format string consists of a part for color, marker and line::

        fmt = '[marker][line][color]'

    Each of them is optional. If not provided, the value from the style
    cycle is used. Exception: If ``line`` is given, but no ``marker``,
    the data will be a line without markers.

    Other combinations such as ``[color][marker][line]`` are also
```

supported, but note that their parsing may be ambiguous.

Markers

character	description
``'.'``	point marker
``','``	pixel marker
``'o'``	circle marker
``'v'``	triangle_down marker
``'^'``	triangle_up marker
``'<'``	triangle_left marker
``'>'``	triangle_right marker
``'1'``	tri_down marker
``'2'``	tri_up marker
``'3'``	tri_left marker
``'4'``	tri_right marker
``'8'``	octagon marker
``'s'``	square marker
``'p'``	pentagon marker
``'P'``	plus (filled) marker
``'*'``	star marker
``'h'``	hexagon1 marker
``'H'``	hexagon2 marker
``'+'``	plus marker
``'x'``	x marker
``'X'``	x (filled) marker
``'D'``	diamond marker
``'d'``	thin_diamond marker
``'|'``	vline marker
``'_'``	hline marker

Line Styles

character	description
``'-'``	solid line style
``'--'``	dashed line style
``'-.'``	dash-dot line style
``':'``	dotted line style

Example format strings::

```
    'b'    # blue markers with default shape
```

```
    'or'   # red circles
    '-g'   # green solid line
    '--'   # dashed line with default color
    '^k:'  # black triangle_up markers connected by a dotted line

**Colors**

The supported color abbreviations are the single letter codes

=============    ==============================
character        color
=============    ==============================
``'b'``          blue
``'g'``          green
``'r'``          red
``'c'``          cyan
``'m'``          magenta
``'y'``          yellow
``'k'``          black
``'w'``          white
=============    ==============================

and the ``'CN'`` colors that index into the default property cycle.

If the color is the only part of the format string, you can
additionally use any  `matplotlib.colors` spec, e.g. full names
(``'green'``) or hex strings (``'#008000'``).
```

1.2 Subplots, Axes, Objektoriente Programmierung (OOP) bei Abbildungen

Bisher haben wir alle Einstellung über Funktionen von `pyplot` vorgenommen. Diese Funktionen wirken implizit auf die aktuelle Abbildung (figure).

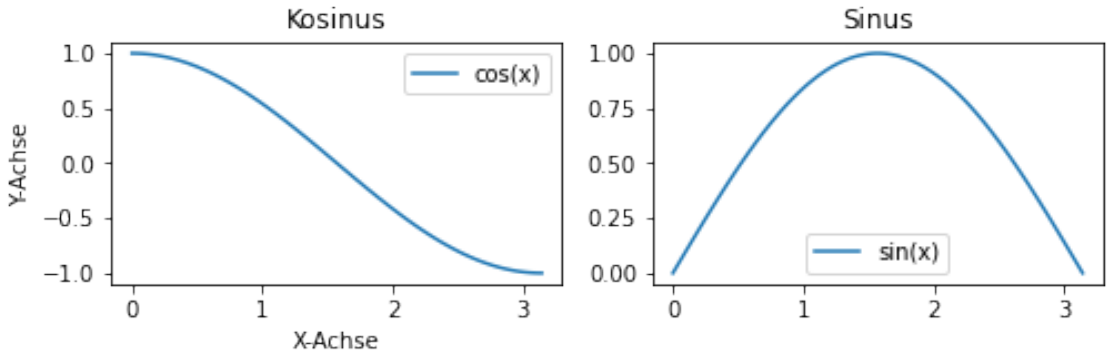
Es gibt eine alternative Methode die explizit auf eine (Teil-)Abbildung angewendet wird. Hierbei kommen wir in den Bereich der Objektorientieren Programmierung die über diese Veranstaltung hinaus geht. Da diese Methode allerdings in Theorie-A zur Anwendung kam kurz zur Vollständigkeit:

Mit der Funktion `plt.subplots()` kann man Abbildungen mit einer oder mehreren Teilabbildungen erstellen. Die Funktion gibt nun zwei Dinge zurück: 1. einen Verweis auf die Abbildung (*figure*) 2. eine Liste mit Verweisen auf die Teilabbildungen (*axes*)

(Bei Matplotlib heissen die Teilabbildungen *"axes"*, eine etwas ungünstige Wahl, da es auch "Axis" gibt).

Jede Teilabbildung hat nun ihre eigenen Funktionen, sogennante "Methoden". Diese ruft man auf, indem man den Funktionsnamen an die Variable mit der Teilabbildung anhängt. Das ganze sieht dann so aus:

[17]:
```
fig, ax = plt.subplots(1, 2, figsize=[8, 2])
ax[0].plot(x, np.cos(x), label='cos(x)')
ax[1].plot(x, np.sin(x), label='sin(x)')
ax[0].set_xlabel('X-Achse')  # Füge Beschriftung an der  X-Achse hinzu
ax[0].set_ylabel('Y-Achse')  # Füge Beschrifung an der Y-Achse hinzu.
ax[0].set_title("Kosinus")   # Titel der ersten Teilabbildung
ax[1].set_title("Sinus")     # Titel der zweiten Teilabbildung
ax[0].legend();              # Hinzufügen einer Legende
ax[1].legend();
fig.savefig('plot2.pdf')
```



Das ist praktisch wenn man wirklich eine Abbildung mit Unterabbildungen erzeugen will, wird aber auch oft für eine einzelne Abbildung genutzt. (*ax* ist dann allerdings keine Liste).