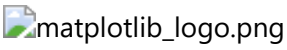
Visualization with matplotlib I



Content:

Introduction

One of the most important modules in Python for 2-dimensional visualization is Matplotlib. It is a very comprehensive package with many display options for 2-dimensional data, whether xy plots, bar charts, pie charts, box plots, contours, vectors, scatters, and so on.

Matplotlib home page https://matplotlib.org/

Matplotlib provides many example scripts with a short description, see https://matplotlib.org/stable/gallery. For the beginner it is sometimes difficult to get started, but through the examples gallery with the many examples you can quickly find something that suits your needs.

Matplotlib's collection of style functions **pyplot** work very similar to MATLAB. For a detailed description see https://matplotlib.org/stable/tutorials.

Preliminary work

Jupyter notebook offers the possibility to display the plots within the notebook. Therefore we have to turn on the matplotlib mode.

```
    %matplotlib inline
```

```
%matplotlib inline
```

Import plot library

Import the function collection `pyplot` from matplotlib and use the shortcut `plt` for it.

```
import matplotlib.pyplot as plt
```

```
import matplotlib.pyplot as plt
```

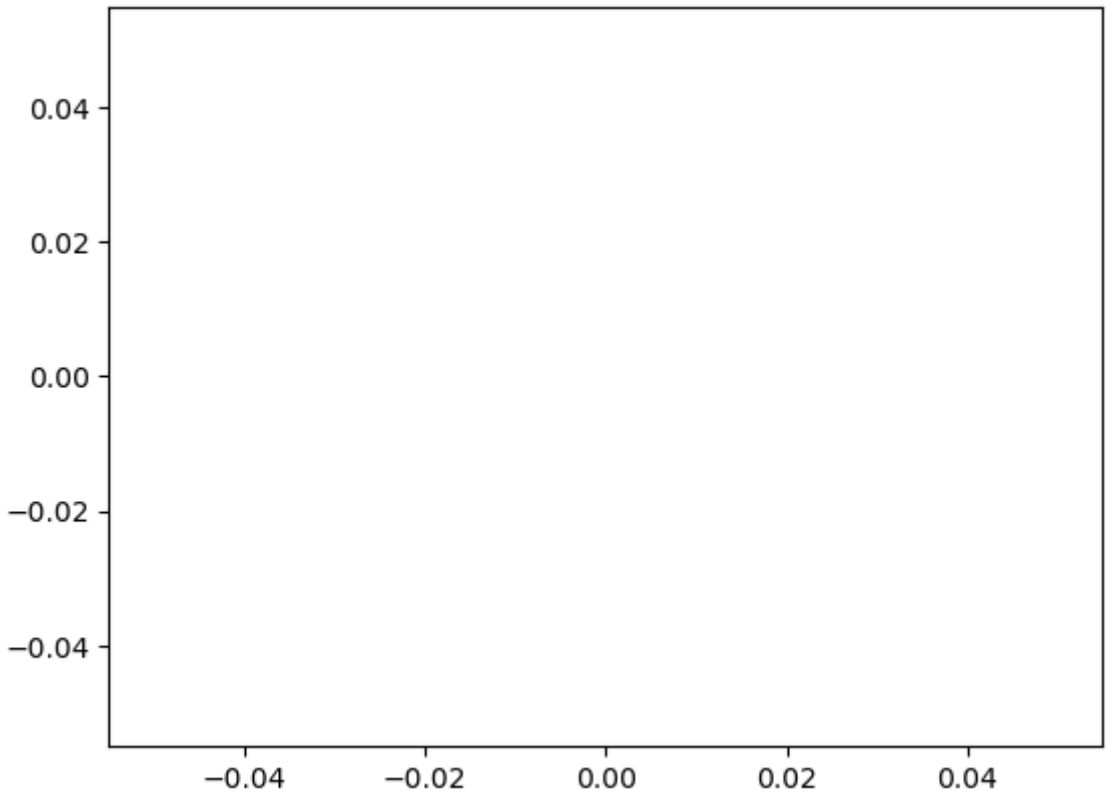
A simple plot

With the `plot` function we can create an empty plot.

```
plt.plot()
```

```
plt.plot()
```

```
[]
```

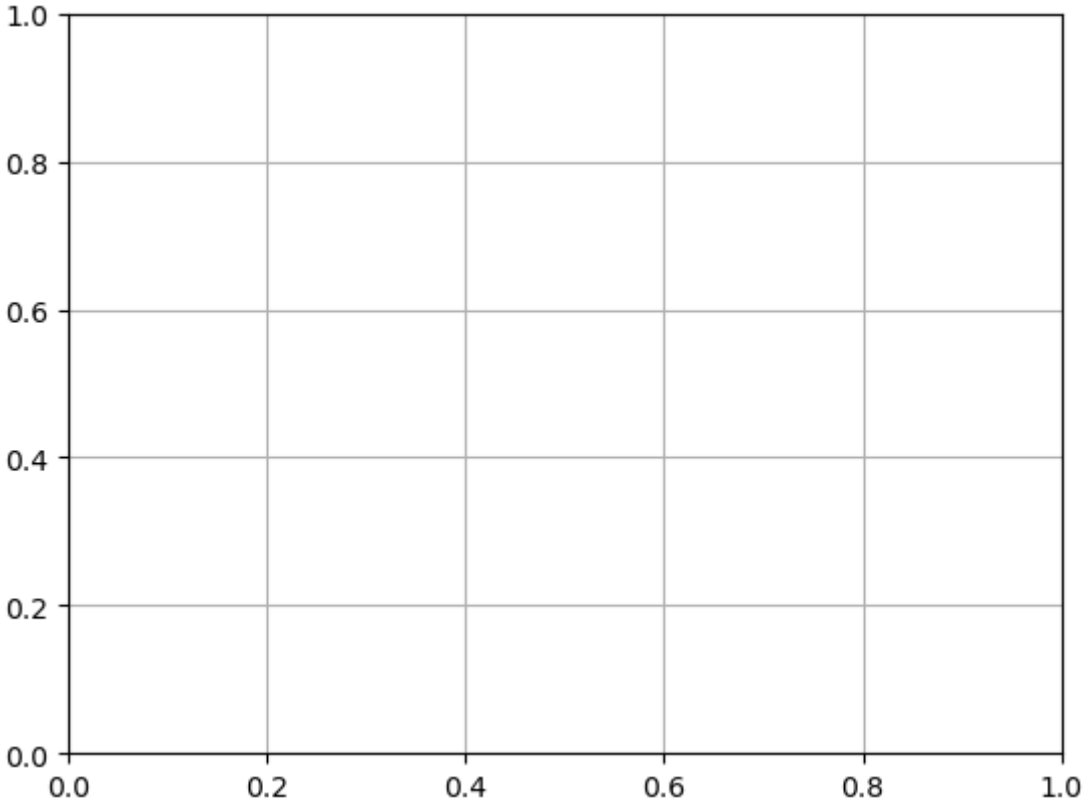


Add grid lines

If you want to add some grid lines to the plot use the `grid` function.

```
plt.grid()
```

```
plt.grid()
```



Basic concepts

Matplotlib allows us to create a basic plot in an comfortable way. Only two lists or arrays are needed.

Create some sample data:

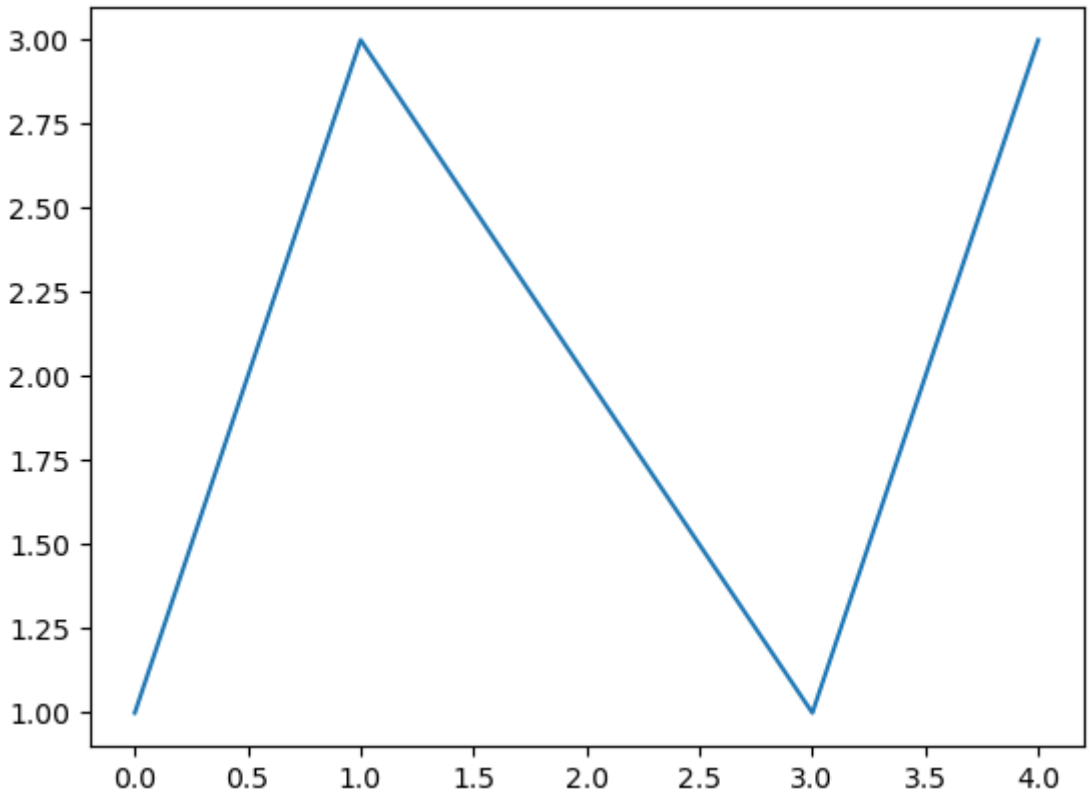
```
y = [1,3,2,1,3]
```

```
y = [1,3,2,1,3]
```

Plot the sample data:

```
plt.plot(y)
```

```
[<matplotlib.lines.Line2D at 0x7fffc85213a0>]
```

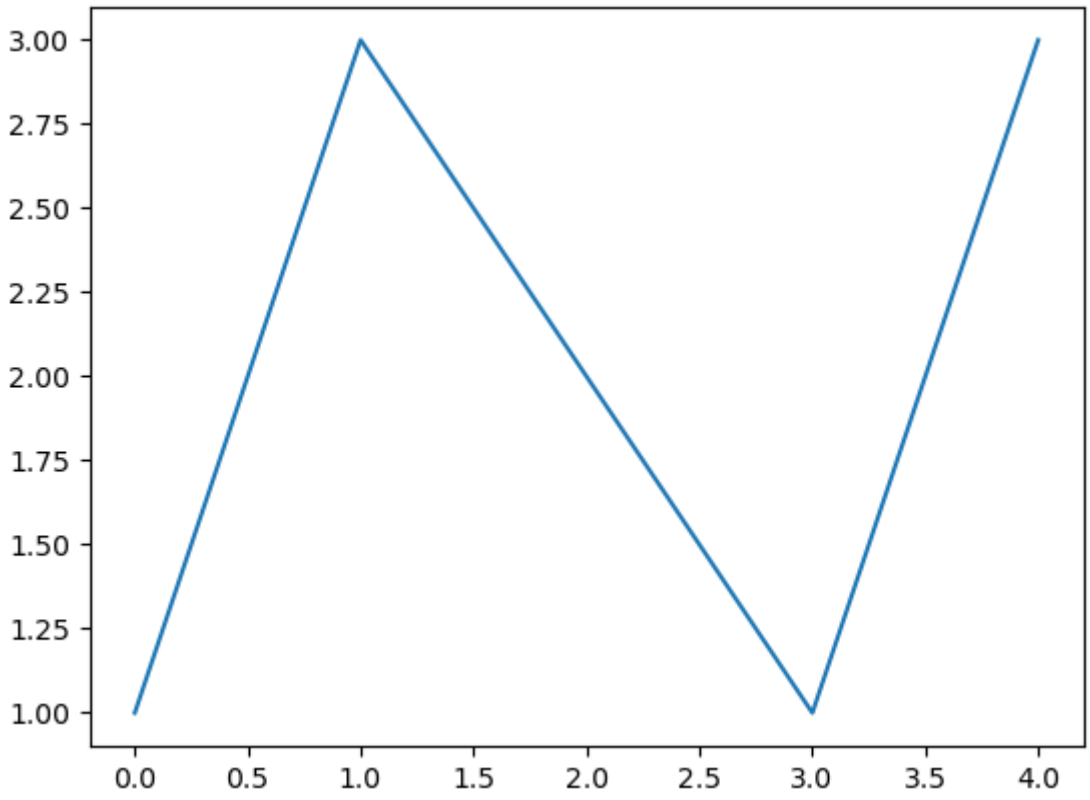


Note: If you want to get rid of the Matplotlib message like the

```
[<matplotlib.lines.Line2D at 0x10f402500>]
```

from above you can add a semicolon to the line or write the expression to a variable.

```
plt.plot(y);
```



For the next example we assign a list of integers to the y-values:

```
x = [1,3,6,7,9]
```

Now we use the two lists to generate a line plot with Matplotslib's `plot` function. Running the following command line will display the line plot inside this notebook (if you have set `%matplotlib inline`).

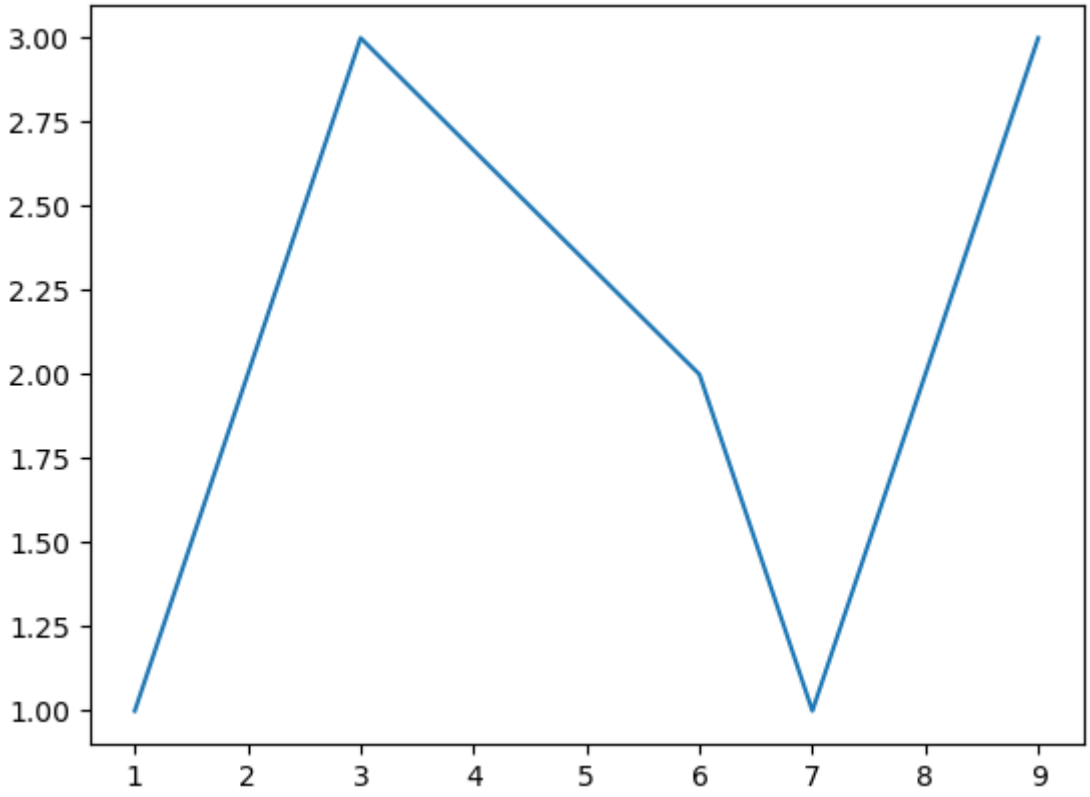
```
plt.plot(x, y)
```

```
print(x)
print(y)
```

```
[1, 3, 6, 7, 9]
[1, 3, 2, 1, 3]
```

```
x = [1, 3, 6, 7, 9]
y = [1, 3, 2, 1, 3]
```

```
plt.plot(x, y);
```



This plot is more than minimalistic and we will show you how to change the **line style** with the parameter `linestyle`.

You can try some of the following line styles. The code string as well as the short code string can be used.

```
Line style      code       short code
-------------------------------------
Solid line     'solid'     '-'
Dashed line    'dashed'    '--'
Dotted line    'dotted'    ':'
Dash dot line  'dashdot'   '-.'
```

> Note: There are many more line styles available. See:
> https://matplotlib.org/3.1.0/gallery/lines_bars_and_markers/linestyles.html

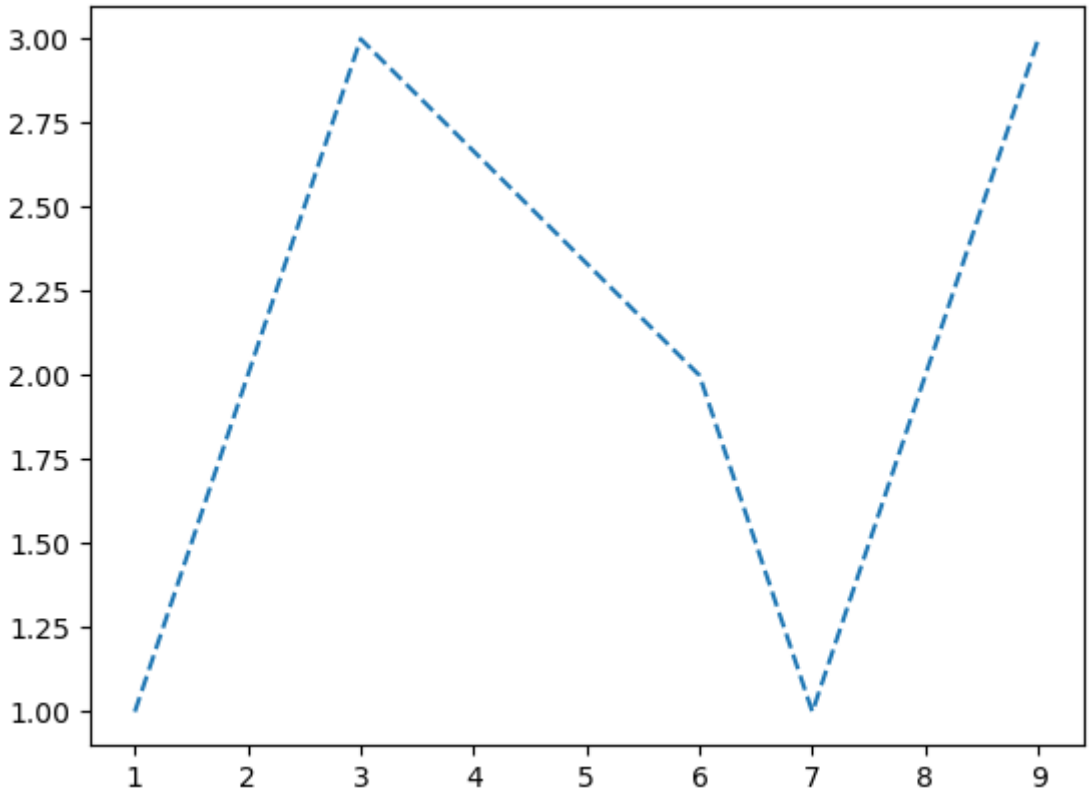
```
plt.plot(x, y, linestyle=':')

or
```

```
plt.plot(x, y, ':')
```

```
plt.plot(x, y, '--')
```

```
[<matplotlib.lines.Line2D at 0x7fffc817a8b0>]
```



Let's switch from line to a **marker type**. You can do it with a short code or use the `marker` parameter.

```
Circle          'o'
Triangle up     '^'
Triangle down   'v'
Square          's'
Star            '*'
Plus            '+'
X               'x'
Diamond         'D'
Diamond thin    'd'
```

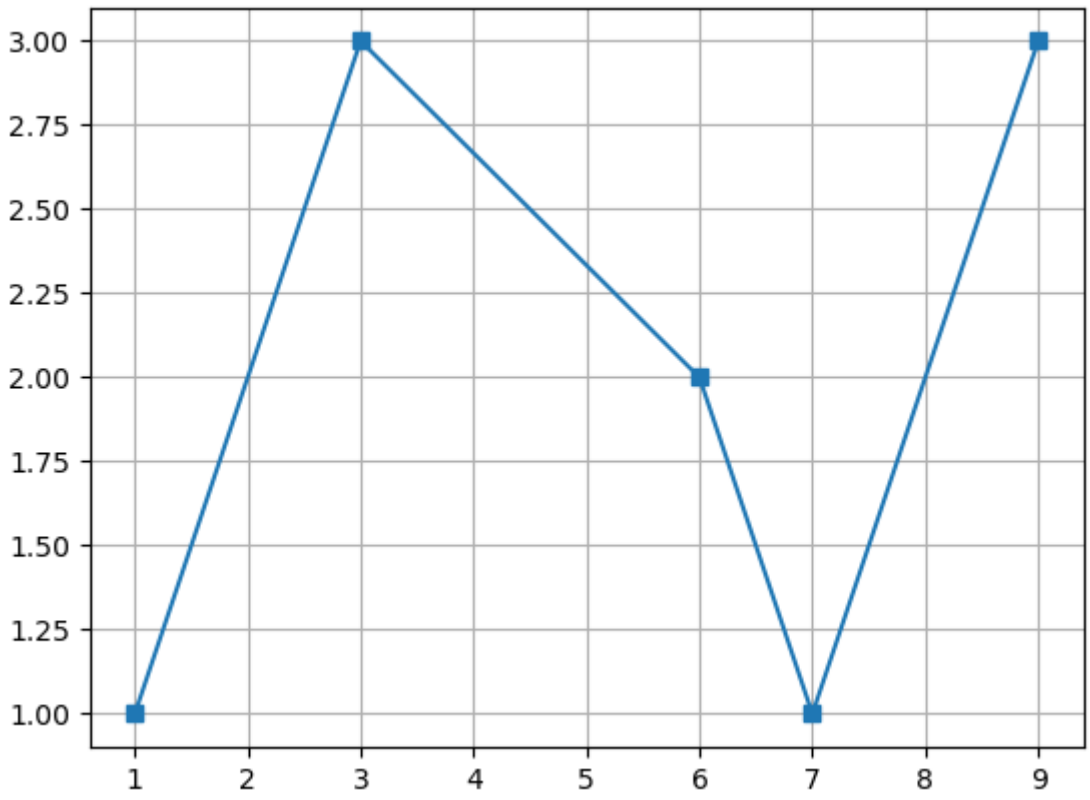
> Note: There are many more marker types available. See: https://matplotlib.org/3.3.3/api/markers_api.html

```
plt.plot(x, y, 'D')
```

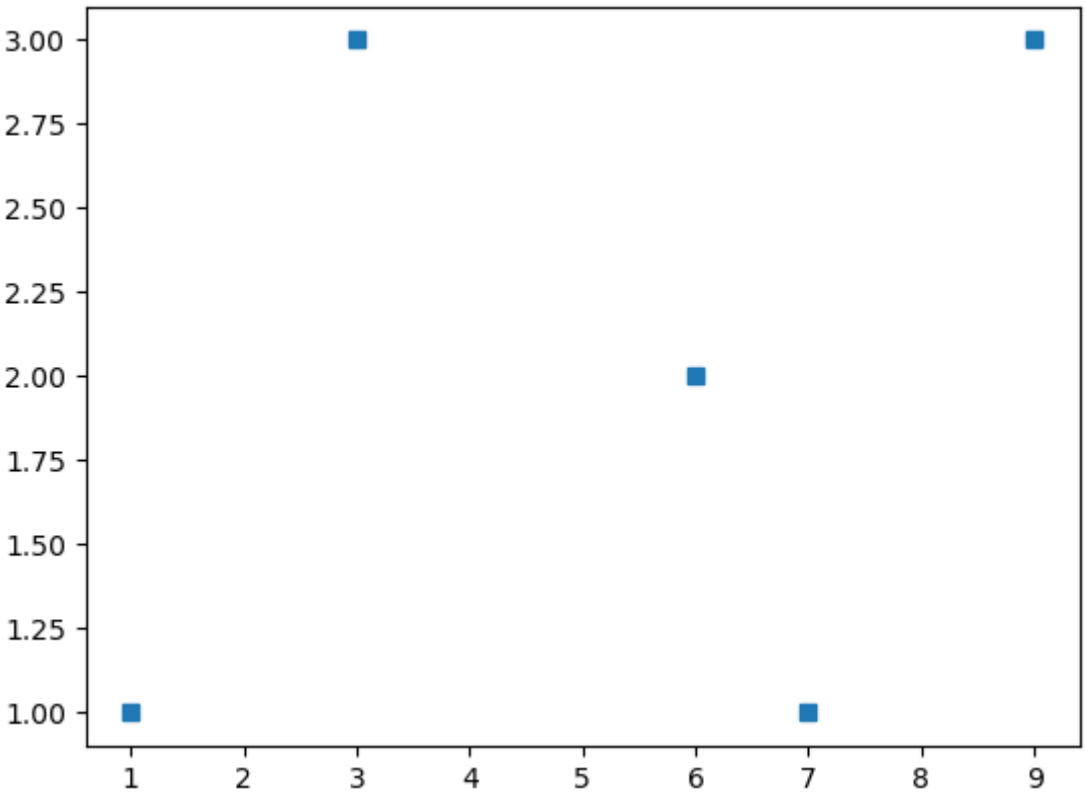
```
plt.grid()

plt.plot(x, y, marker='s')
```

```
[<matplotlib.lines.Line2D at 0x7fffc86785b0>]
```




```
plt.plot(x, y, linestyle='', marker='s');
```



Exercise

1. Use the marker parameter instead the short code.
2. What do you have to do to get the same plot as above?

```

```

That looks quite nice but we want to use another **color**, maybe red.

The following color abbreviations (short codes) are available:

```
Color   code
-------------
black    'k'
white    'w'
red      'r'
green    'g'
blue     'b'
cyan     'c'
magenta  'm'
yellow   'y'
```

> Note: There are many more named colors available. See:
> https://matplotlib.org/3.1.0/gallery/color/named_colors.html

Matplotlib accepts combinations of multiple parameter abbreviations, e.g.

`'Dr'` is the same as `linestyle='None', marker='D', color='r'`

==>

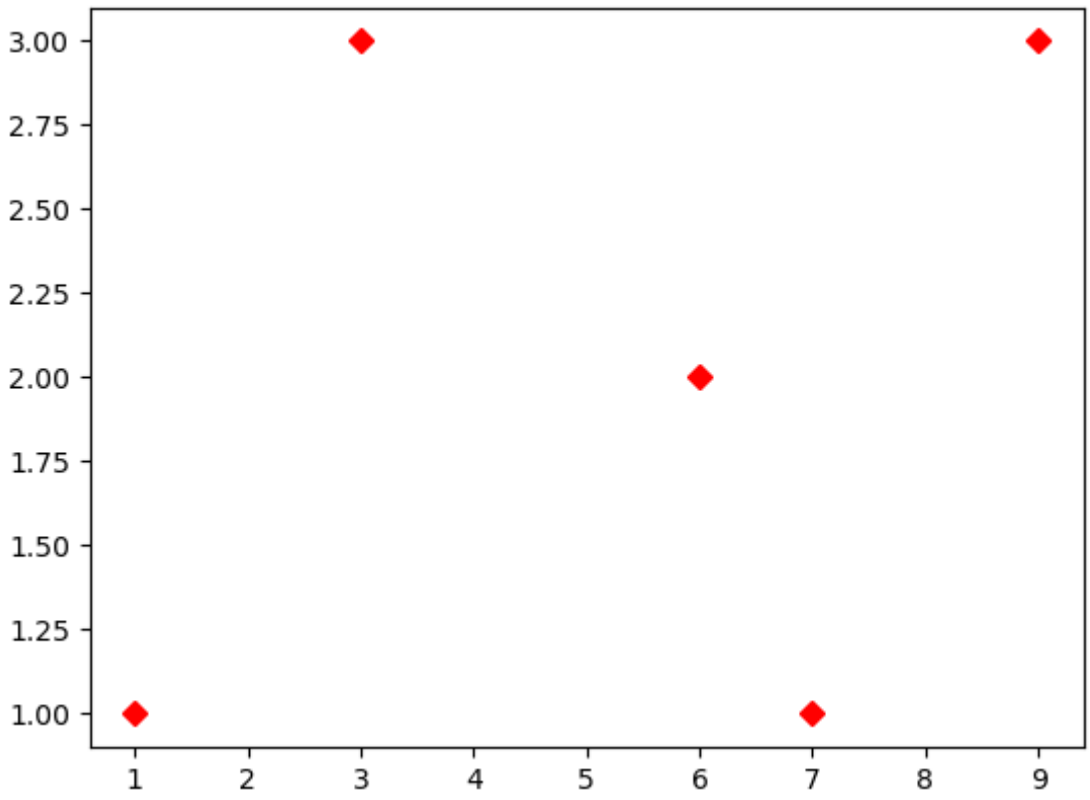
```
plt.plot(x, y, 'Dr')
```

is the same as

```
plt.plot(x, y, linestyle='None', marker='D', color='r')
```

```
#plt.plot(x, y, 'rD')
plt.plot(x, y, linestyle='None', marker='D', color='r')
```

```
[<matplotlib.lines.Line2D at 0x7fffbd208a30>]
```



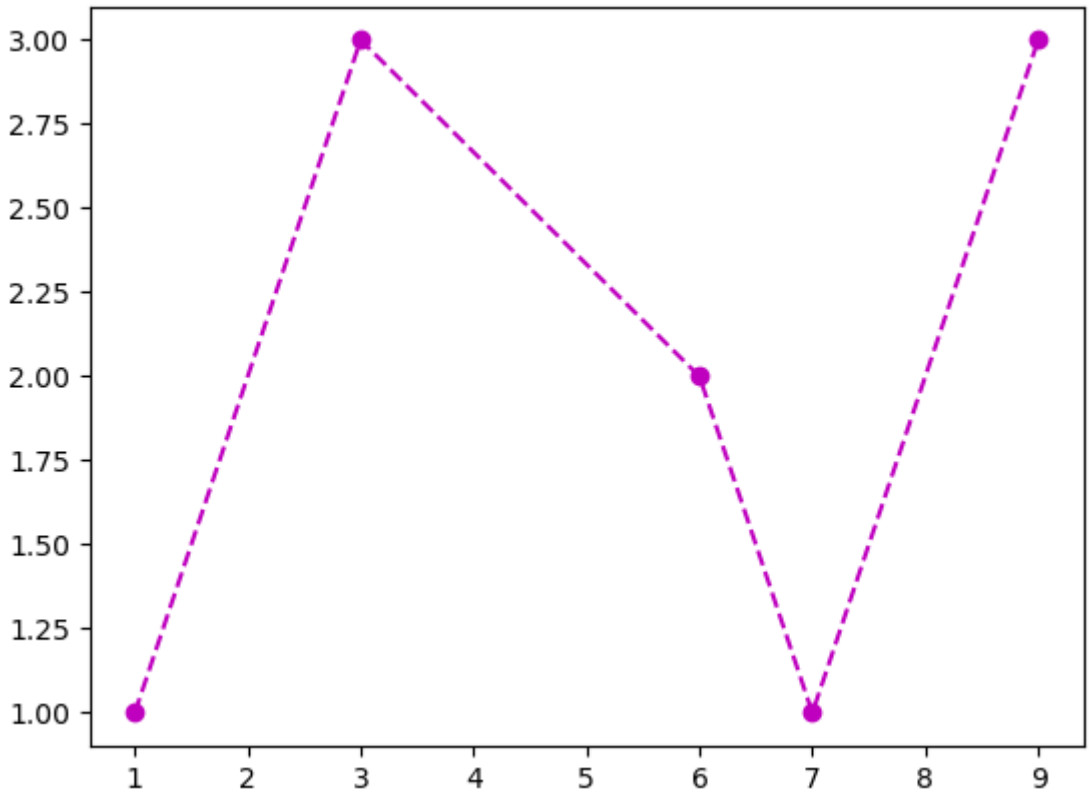
Let's create a dashed line plot with circle markers in magenta.

```
plt.plot(x, y, '--om')

#plt.plot(x, y, linestyle='--', marker='o', color='m')
```

```
#plt.plot(x, y, 'om--')
plt.plot(x, y, linestyle='--', marker='o', color='m')
```

```
[<matplotlib.lines.Line2D at 0x7fffbd06f040>]
```



Customize markers and lines

You can customize the plot further by changing

- marker size
- marker facecolor
- marker edgecolor
- linewidth

In the following plot, we generate larger red markers with black edges. For the line, we use the default linestyle and linecolor, but we increase the linewidth to 2.

```
plt.plot?
```

```
□[0;31mSignature:□[0m
□[0mplt□[0m□[0;34m.□[0m□[0mplot□[0m□[0;34m(□[0m□[0;34m*□[0m□[0margs□[0m□[0;34
m,□[0m □[0mscalex□[0m□[0;34m=□[0m□[0;32mTrue□[0m□[0;34m,□[0m
□[0mscaley□[0m□[0;34m=□[0m□[0;32mTrue□[0m□[0;34m,□[0m
□[0mdata□[0m□[0;34m=□[0m□[0;32mNone□[0m□[0;34m,□[0m
□[0;34m**□[0m□[0mkwargs□[0m□[0;34m)□[0m□[0;34m□[0m□[0;34m□[0m□[0m
□[0;31mDocstring:□[0m
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 `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
    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'}
    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]
    path_effects: `.AbstractPathEffect`
    picker: float or callable[[Artist, Event], tuple[bool, dict]]
    pickradius: float
    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

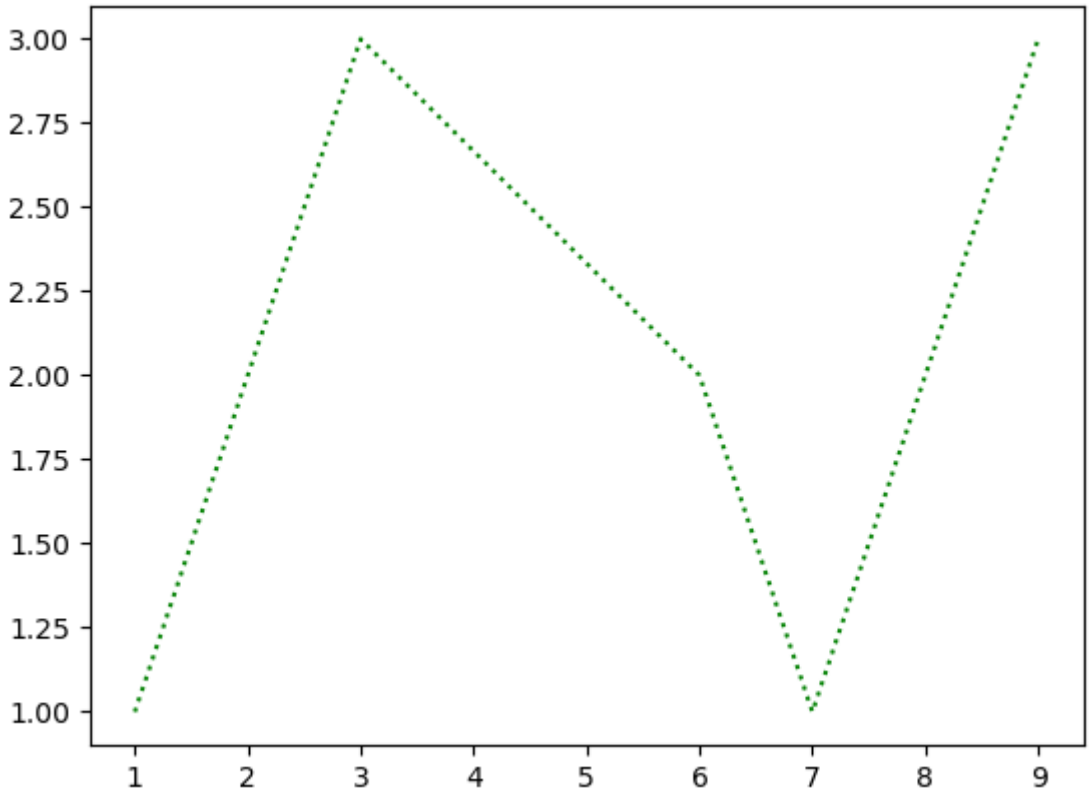
=============    ================================
character        color
=============    ================================
``'b'``          blue
``'g'``          green
``'r'``          red
``'c'``          cyan
``'m'``          magenta
``'y'``          yellow
``'k'``          black
``'w'``          white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

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'``).
□[0;31mFile:□[0m      ~/.conda/envs/cygnss-d/lib/python3.9/site-
packages/matplotlib/pyplot.py
□[0;31mType:□[0m      function
```

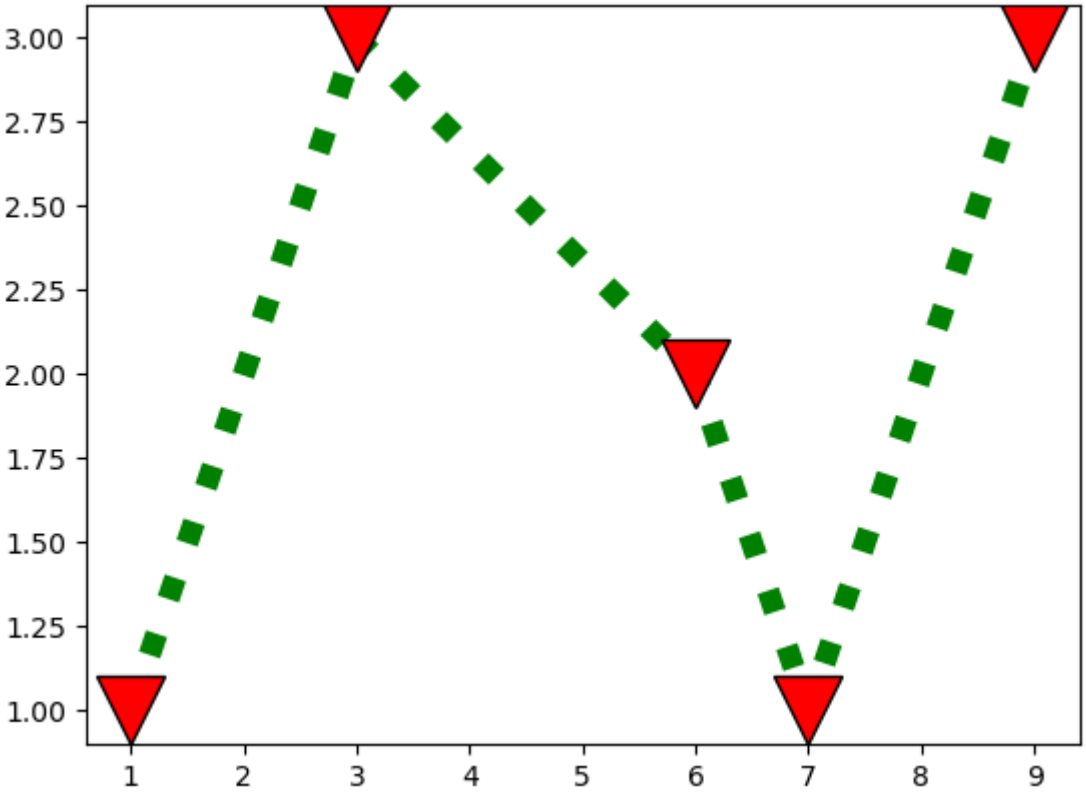
```
plt.plot(x, y, 'g:')
```

```
[<matplotlib.lines.Line2D at 0x7fffb8d71250>]
```




```
plt.plot(x, y, linewidth=8, linestyle=':', color='g', marker='v', markersize=24,
markerfacecolor='red', markeredgecolor='black')
```

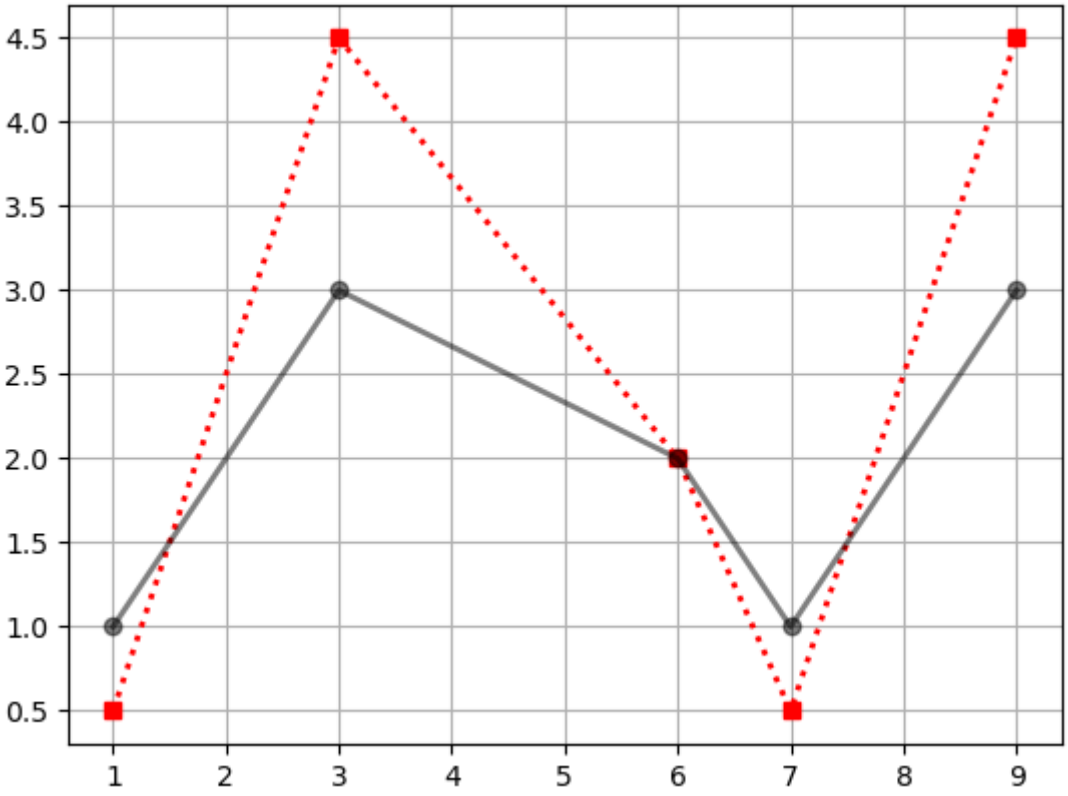
```
[<matplotlib.lines.Line2D at 0x7fffb8d8d070>]
```



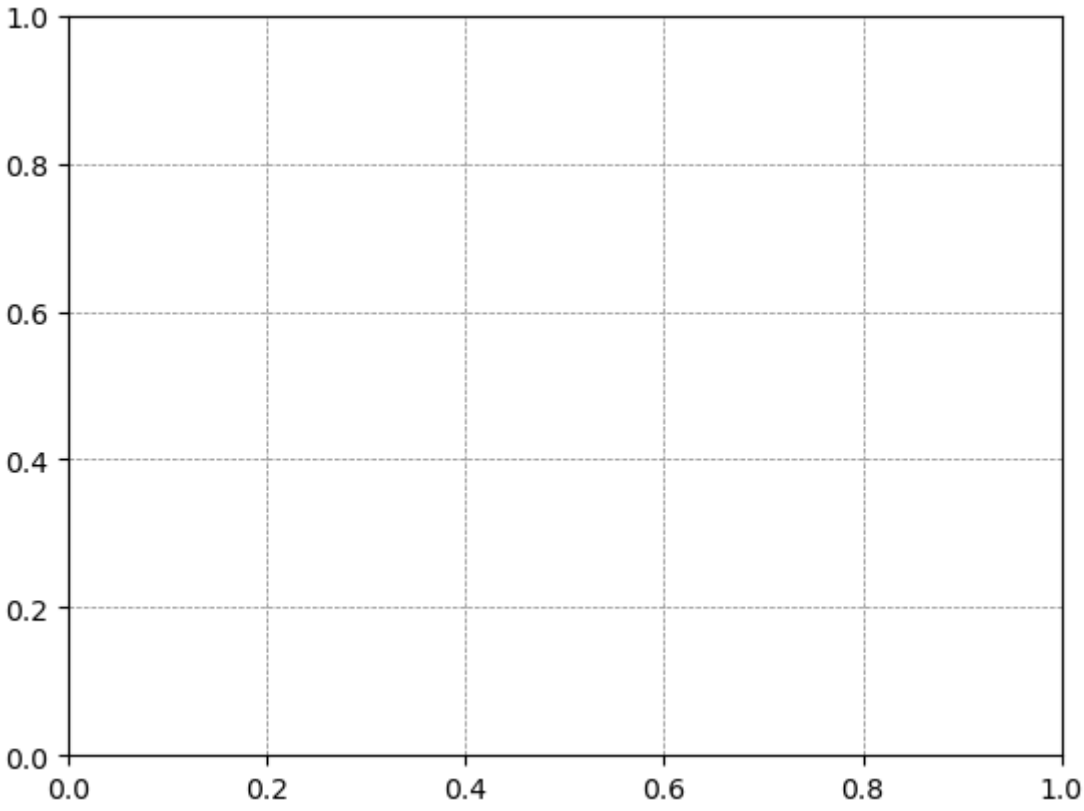
Sometimes you want to overlay different components of your plots using transparency. In matplotlib this is achieved by setting the transparency parameter `alpha`. Here, `alpha=1.` corresponds to full **opacity**, and `alpha=0.` to **full transparency**. In the following plot, we draw a black transparent line on top of a solid red line:

```
plt.grid()
plt.plot(x, [v**2/2 for v in y], 'rs:', linewidth=2)
plt.plot(x, y, 'ko-', linewidth=2, alpha=0.5)
```

```
[<matplotlib.lines.Line2D at 0x7fffb8b592e0>]
```




```
plt.grid(color='gray', lw=0.5, ls='--')
```



Exercise

1. Import numpy `import numpy as np`

2. Create data, e.g.

```
x = np.arange(1.,100.,4)
y = np.sin(3.14159*(x+.5)/50.)
```

3. Create a line plot with the following attributes

   - set line style to dotted
   - set line color to red
   - set line width to 1
   - add plus markers in black with marker size 8 to the plot

4. Create another line plot with the following attributes:

   - set line style to solid
   - set line color to red
   - set line width to 2
   - add triangle markers in green with marker size 12 and marker edgecolor black
   - make the line plot transparent with 75% transparency

Hint: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html

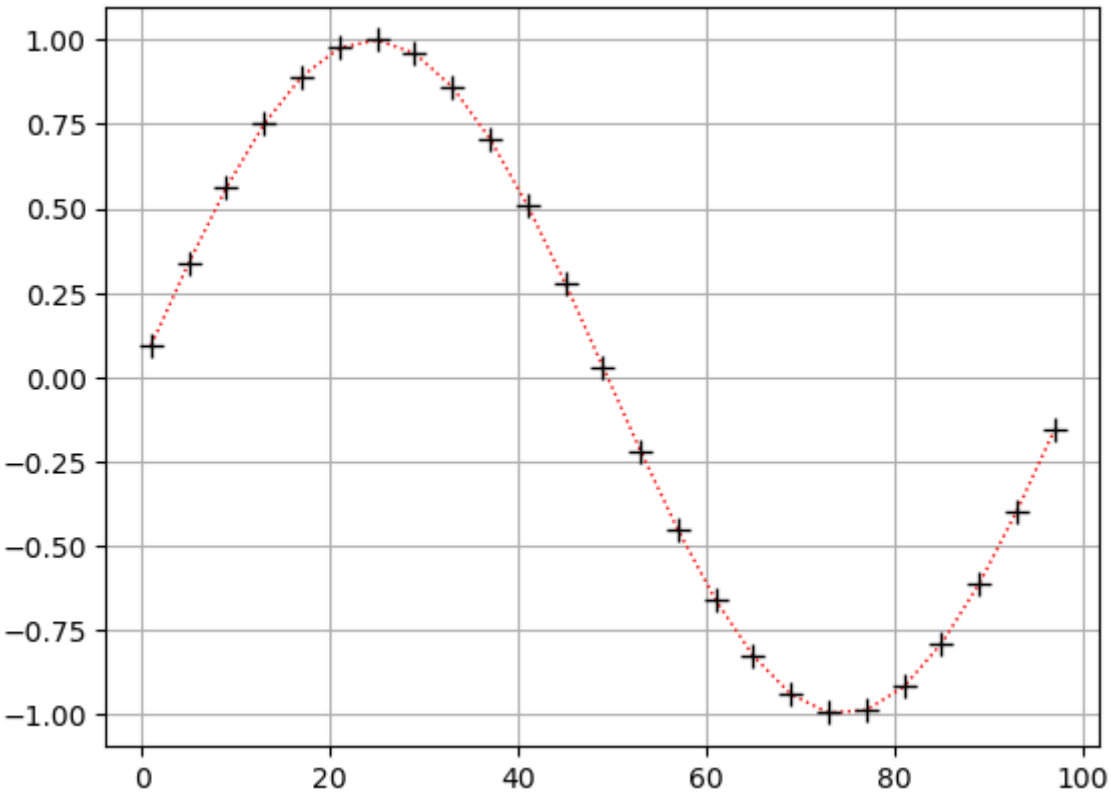
```
# 1.
import numpy as np
```

```
# 2.
x = np.arange(1.,100.,4)
y = np.sin(3.14159*(x+.5)/50.)
```

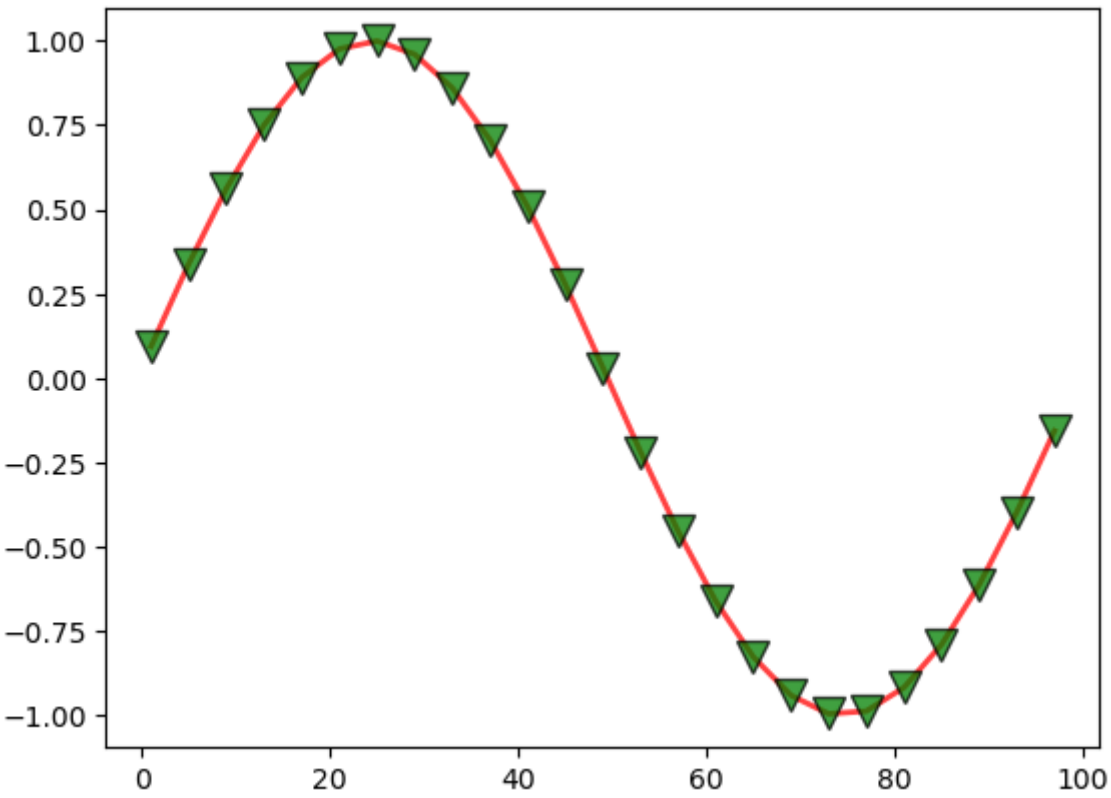
```
# 3.
plt.grid()
#plt.plot(x,y,':r', lw=1, marker='+',ms='8', markeredgecolor='k')

plt.plot(x, y, ':r+', lw=1, ms='8', markeredgecolor='k')
```

```
[<matplotlib.lines.Line2D at 0x7fffb8a997c0>]
```




```
# 4.
plt.plot(x, y, linewidth=2, linestyle='-', color='r', alpha=0.75, marker='v',
markersize=12, markerfacecolor='g', markeredgecolor='k');
```



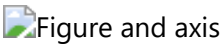
But of course you can still adjust the various settings manually, such as size, color, output resolution, line width, text settings, etc..

Figure and axes

To make the best use of Matplotlib, you should understand the base plot concept. Matplotlib provides two objects to control the plots

- **figure** object
- **axis** object

Figure can be considered as the workspace and axis as the actual plot. You can create a figure that contains one or more plots.

 Figure and axis

The left part of the image shows in gray a figure containing one plot (axis). In the middle we see in gray one figure containing two plots (axis with 2 elements) and on the right side we see in gray a figure containing four plots (axis with 4 elements). In the next section, the use of figure and axis will be shown in more detail.

It is very convenient to create figure and axis objects at once using the Matplotlib `pyplot.subplots` function. Furthermore, further setting options for the axis are introduced, which are mostly self-explanatory.

The next example demonstrates the use of fig and axis for a plot where we also add a title and the x- and y-axis titles.

```
fig, ax = plt.subplots(figsize=(6,4))

ax.set_title('Example A', fontsize=24)
ax.set_ylabel('y axis label', fontsize=16)
ax.set_xlabel('x axis label', fontsize=16)
ax.tick_params(labelsize=16)

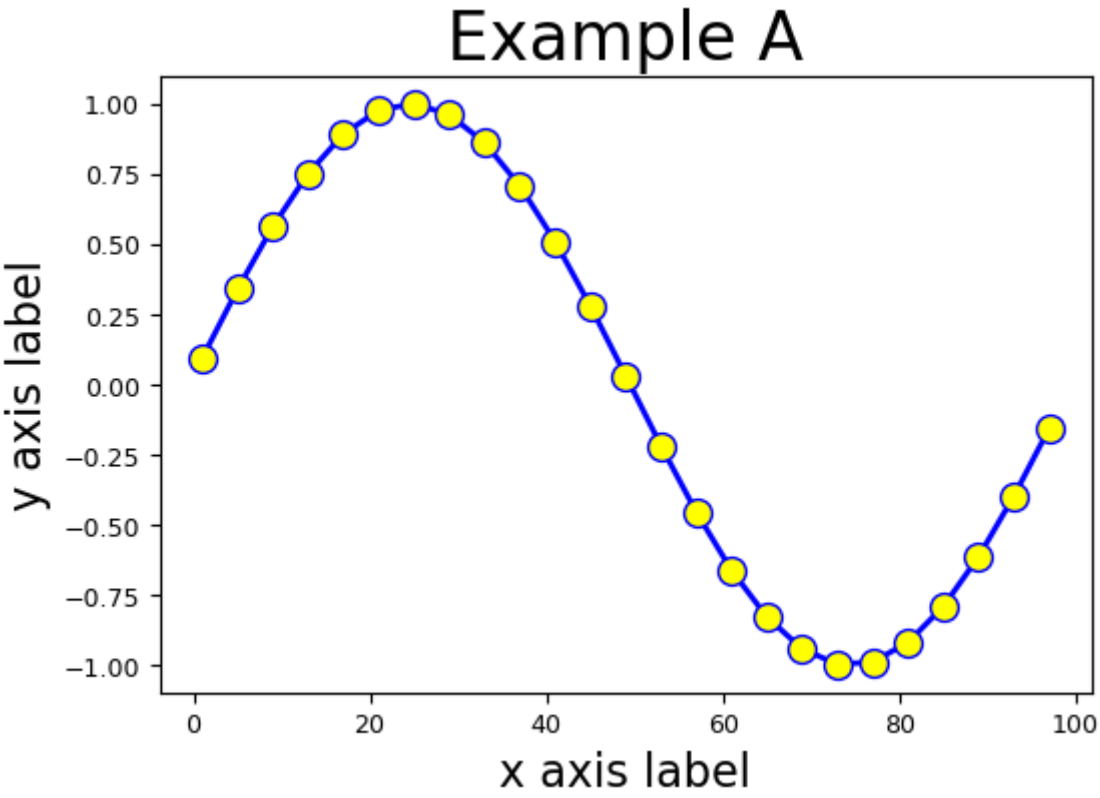
ax.plot(x, y, linestyle="solid", linewidth=2, color="blue",
              marker="o", markersize="10", markerfacecolor="yellow")
```

```
fig, ax = plt.subplots(figsize=(6,4))

ax.set_title('Example A', fontsize=24)
ax.set_ylabel('y axis label', fontsize=16)
ax.set_xlabel('x axis label', fontsize=16)
ax.tick_params(labelsize=9)
```

```
ax.plot(x, y, linestyle="solid", linewidth=2, color="blue",
              marker="o", markersize="10", markerfacecolor="yellow")
```

```
[<matplotlib.lines.Line2D at 0x7fffb87db670>]
```



Multiple plots

In the next section we demonstrate how to use `pyplot.subplots` functionality to create two plots in one workspace.

1. two plots in one row
2. common title on top of the figure

Note that `axs` is now an array, and we access the individual axes by indexing it.

```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(8,4))
print(axs.shape)

fig.suptitle('Panel plot', fontsize=24)

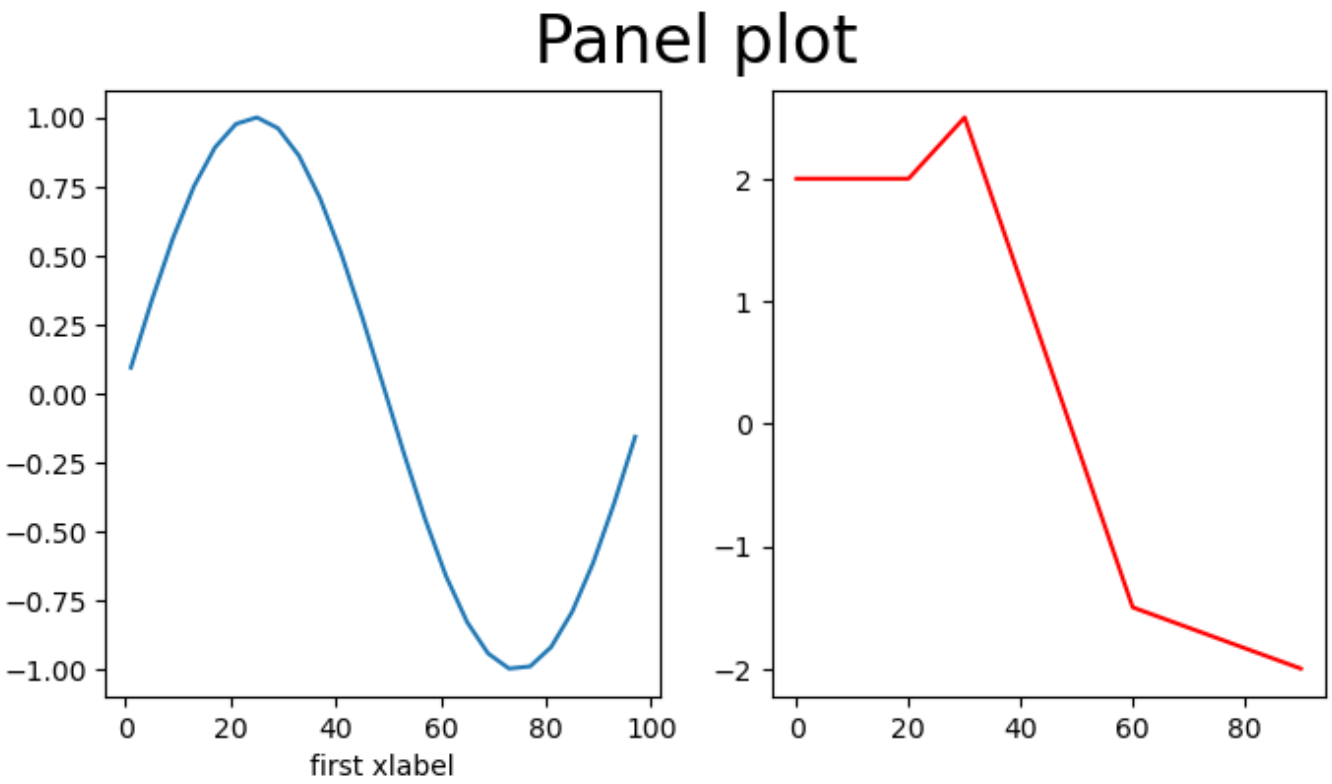
x2 = [0, 20, 30,   60,   90]
y2 = [ 2, 2, 2.5, -1.5, -2]

axs[0].plot(x, y)
axs[1].plot(x2, y2, "r")
```

```
axs[0].set_xlabel('first xlabel')
```

```
(2,)
```

```
Text(0.5, 0, 'first xlabel')
```



If we want to use the same axes ranges we can set the `sharex` and `sharey` parameter to *True*.

```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(8,4), sharex=True,
sharey=False)
fig.suptitle('Panel plot', fontsize=24)

x2 = [0, 20, 30,   60,   90]
y2 = [ 2, 2, 2.5, -1.5, -2]

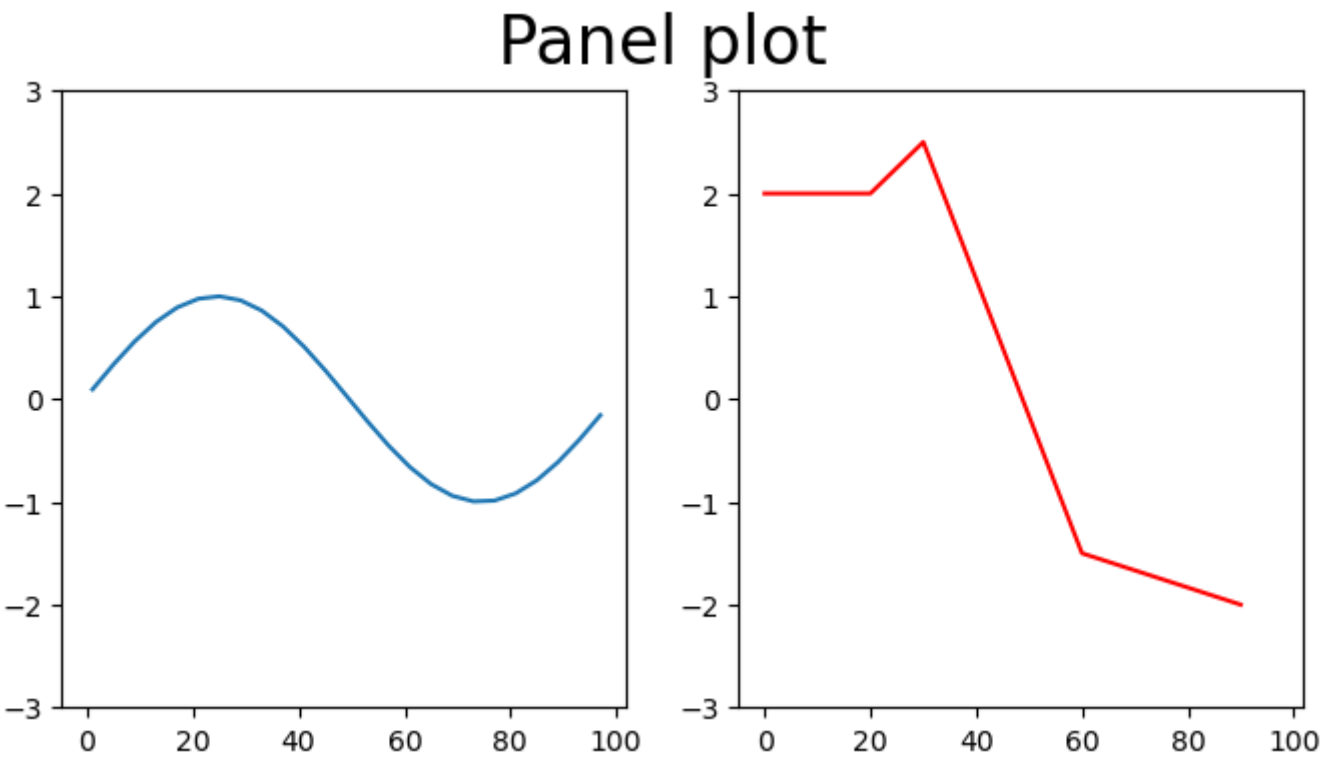
axs[0].plot(x, y)
axs[1].plot(x2, y2, "r")

axs[0].set_ylim(-3, 3)
axs[1].set_ylim(-3, 3)
```

```
#axs[1].yaxis.set_ticks_position('both')

#axs[1].yaxis.tick_right()
```

```
(-3.0, 3.0)
```



Exercise

- define a frame that contains two plots, one plot per row
- write a title on top of the figure
- create upper plot with magenta line color
- create lower plot with green line color

```
fig, axs = plt.subplots(2, 1, figsize=(4,8))
fig.suptitle('Panel plot', fontsize=24)

x2 = [0, 20, 30,   60,   90]
y2 = [ 2, 2, 2.5, -1.5, -2]

axs[0].plot(x, y, "m")
axs[1].plot(x2, y2, "g")

axs[0].set_y2ticks([-1, 0, 1])
```

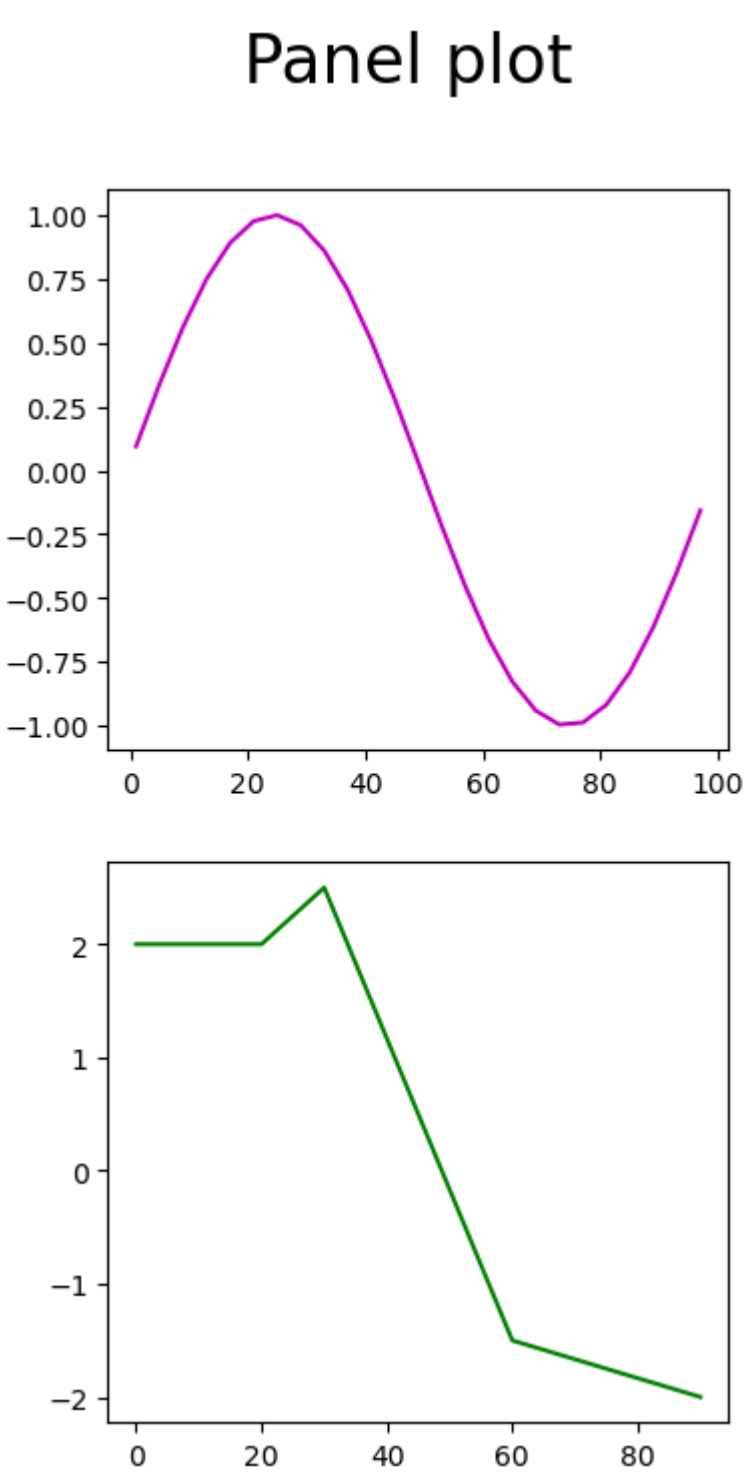
```
fig.tight_layout()
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

Cell In [74], line 10
      7 axs[0].plot(x, y, "m")
      8 axs[1].plot(x2, y2, "g")
---> 10 axs[0].set_y2ticks([-1, 0, 1])
     12 fig.tight_layout()


AttributeError: 'AxesSubplot' object has no attribute 'set_y2ticks'
```



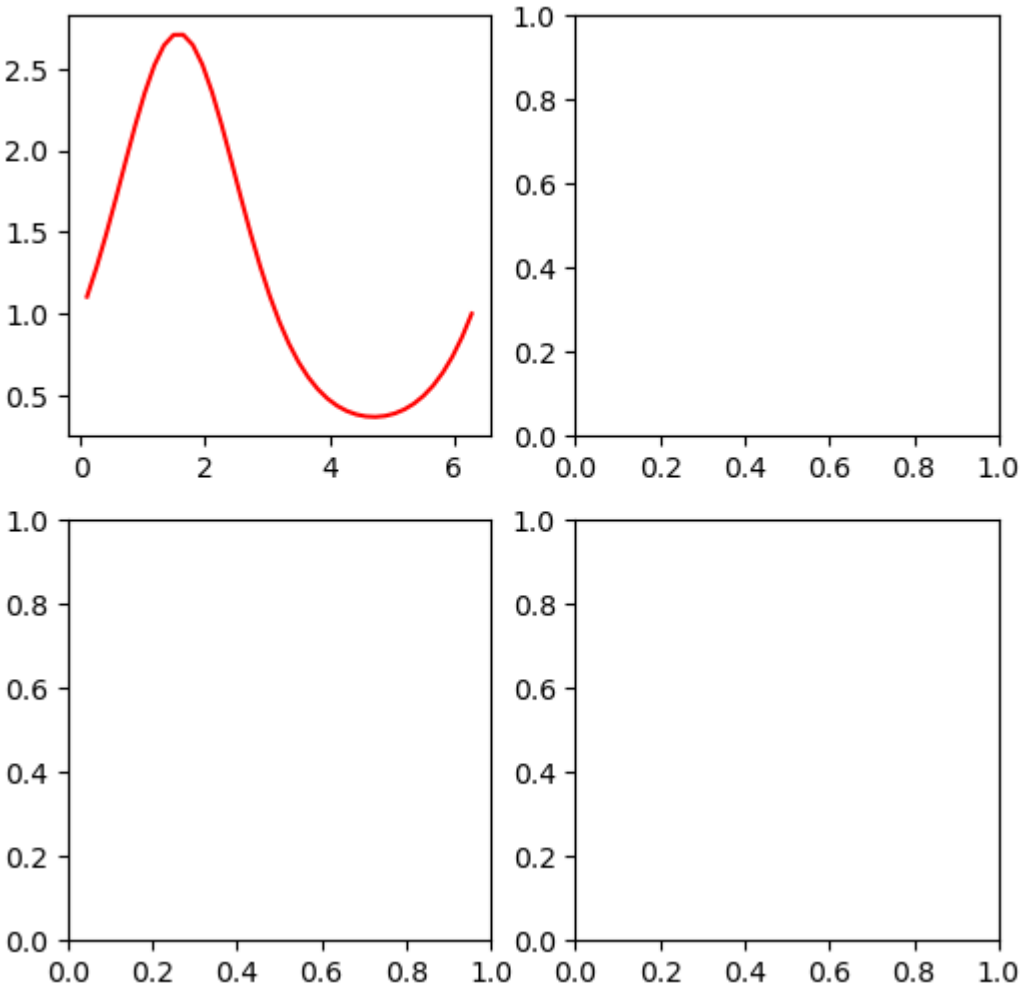

```
fig, ax = plt.subplots(2, 2, figsize=(6, 6))
print(ax.shape)

ax[0, 0].plot(x, y, color='red')

#ax[0].plot(x, y)
```

```
(2, 2)
```

```
[<matplotlib.lines.Line2D at 0x7fffb56eb0d0>]
```



Plot types

Line plots are a little bit boring that's why we change the plot type and see what we get.

You can find the list of different pyplot functions at https://matplotlib.org/stable/api/pyplot_summary.html

Next, we use arrays as input, which we create using the Numpy module (plots just looks nice 😉). Here are just a few examples of the different plot functions.

```
import numpy as np

x = np.linspace(0.1, 2 * np.pi, 41)
y = np.exp(np.sin(x))

fig, axs = plt.subplots(nrows=1, ncols=4, figsize=(16,4))

axs[0].plot(x, y)
axs[1].scatter(x, y)
```

```
axs[2].bar(x, y)
axs[3].stem(x, y)
```

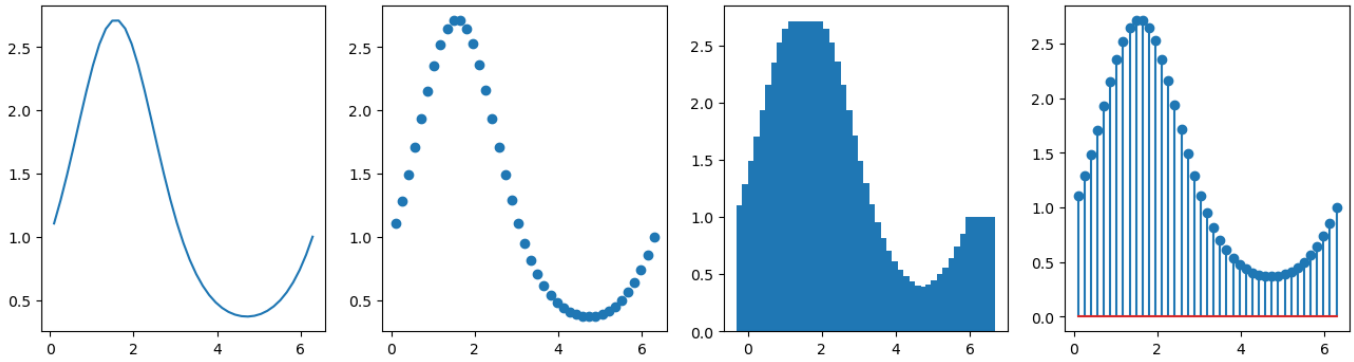
```
import numpy as np

x = np.linspace(0.1, 2 * np.pi, 41)
y = np.exp(np.sin(x))

fig, axs = plt.subplots(nrows=1, ncols=4, figsize=(16,4))

axs[0].plot(x, y)
axs[1].scatter(x, y)
axs[2].bar(x, y)
axs[3].stem(x, y)
```

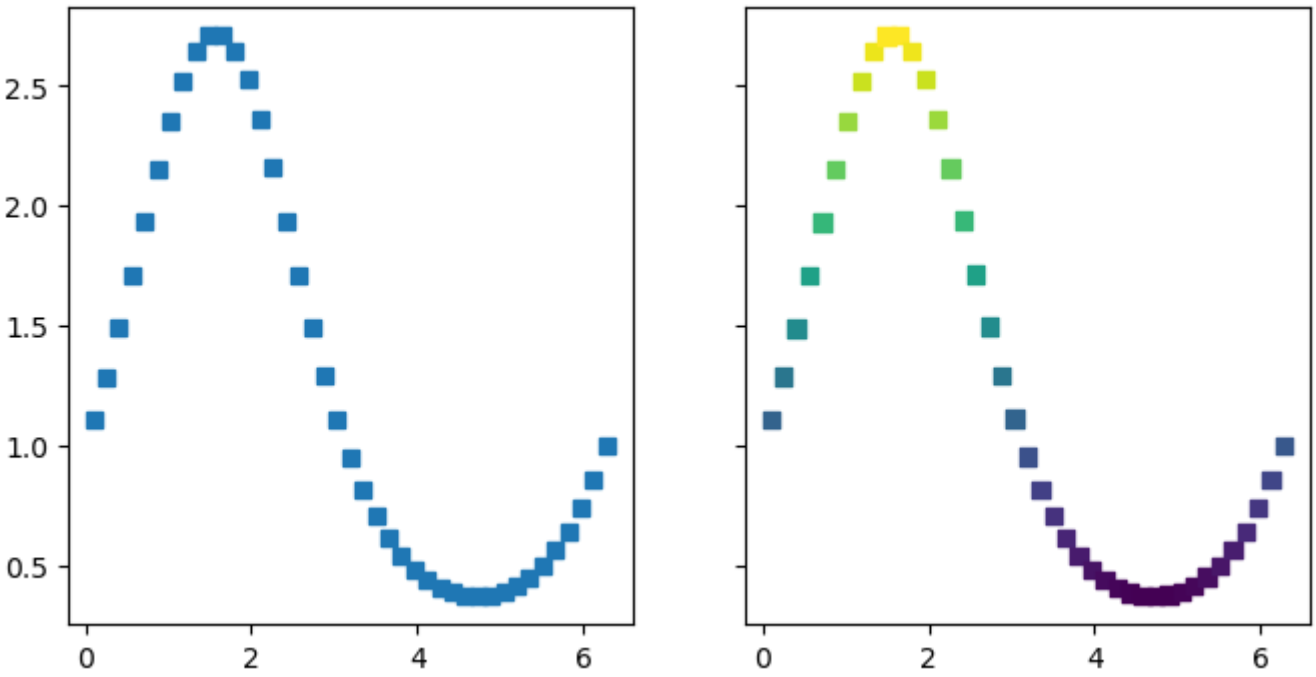
```
<StemContainer object of 3 artists>
```



```
fig, axs = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(8, 4))

axs[0].plot(x, y, 's')
axs[1].scatter(x, y, c=y, marker='s')
```

```
<matplotlib.collections.PathCollection at 0x7fffb5efc9a0>
```



Text annotations

You can customize the titles and axis labels with the ax.set() method in one line.

The next example adds

1. a multiline title above the plot
2. multiline labels at x- and y-axis

```
fig, ax = plt.subplots(figsize=(6,4))

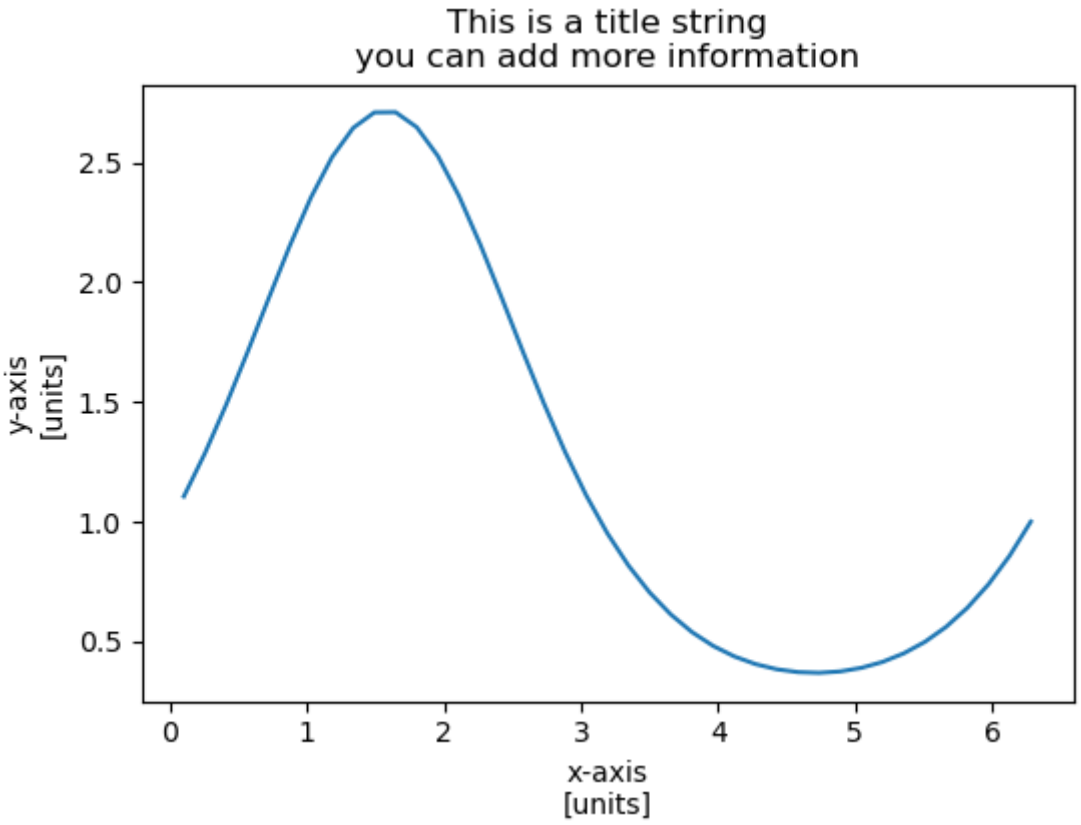
ax.plot(x, y)

ax.set(title='This is a title string\nyou can add more information',
       xlabel='x-axis\n[units]',
       ylabel='y-axis\n[units]')
```

```
fig, ax = plt.subplots(figsize=(6,4))

ax.plot(x, y)

ax.set(title='This is a title string\nyou can add more information',
       xlabel='x-axis\n[units]',
       ylabel='y-axis\n[units]');
```



You can also write additional text in a plot and of course change the font, e.g. Greek, and size.

```
fig, ax = plt.subplots(figsize=(6,4))

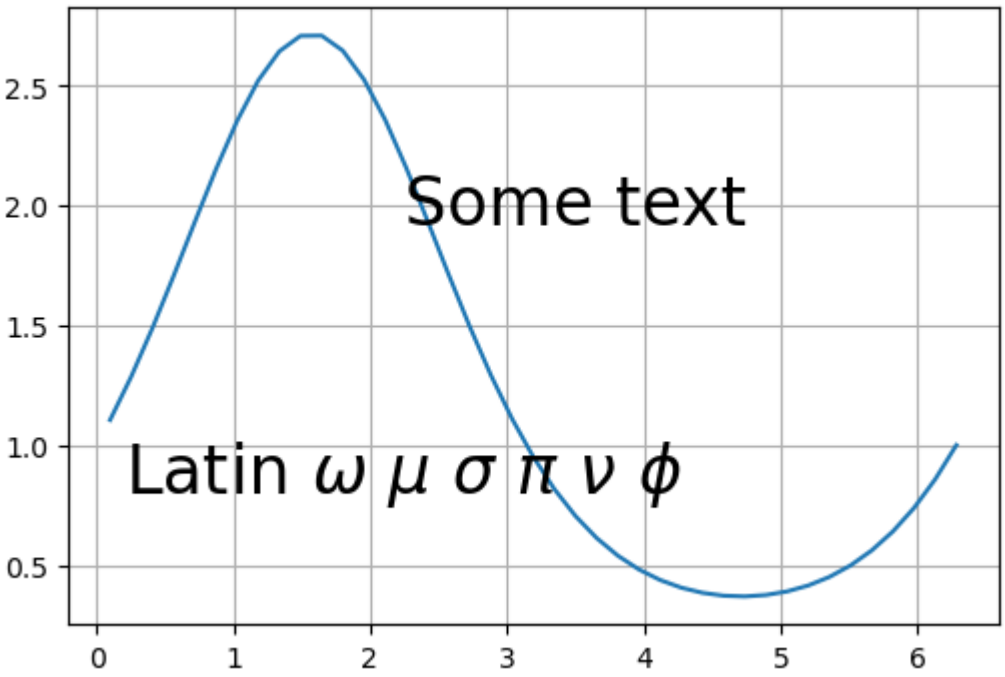
ax.plot(x, y)

ax.text(3.5, 2.0, 'Some text',fontsize=24)
ax.text(0.2, 0.8, r'$\omega\ \mu\ \sigma\ \pi\ \nu\ \phi$', fontsize=24)
```

```
fig, ax = plt.subplots(figsize=(6,4))
ax.grid()
ax.plot(x, y)

ax.text(3.5, 2.0, 'Some text',fontsize=24, ha='center', va='center')
ax.text(0.2, 0.8, r'Latin $\omega\ \mu\ \sigma\ \pi\ \nu\ \phi$', fontsize=24)
```

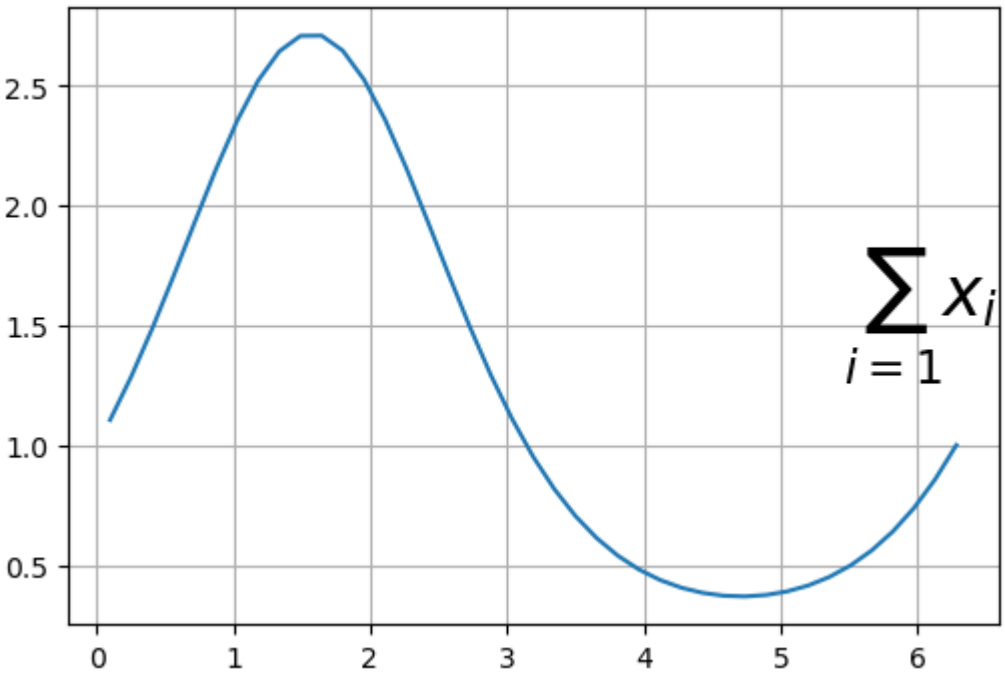
```
Text(0.2, 0.8, 'Latin $\\omega\\ \\mu\\ \\sigma\\ \\pi\\ \\nu\\ \\phi$')
```




```
fig, ax = plt.subplots(figsize=(6,4))
ax.grid()
ax.plot(x, y)

ax.text(1, 0.5, r'$\sum_{i=1}x_i$',fontsize=24, ha='right', va='center',
transform=ax.transAxes)
```

```
Text(1, 0.5, '$\\sum_{i=1}x_i$')
```



We can also add text to the workspace outside the plot, e.g. a copyright text. The figure size will be extended when the text string won't fit and the plot size will be unchanged.

```
fig, ax = plt.subplots(figsize=(6,4))

ax.plot(x, y)

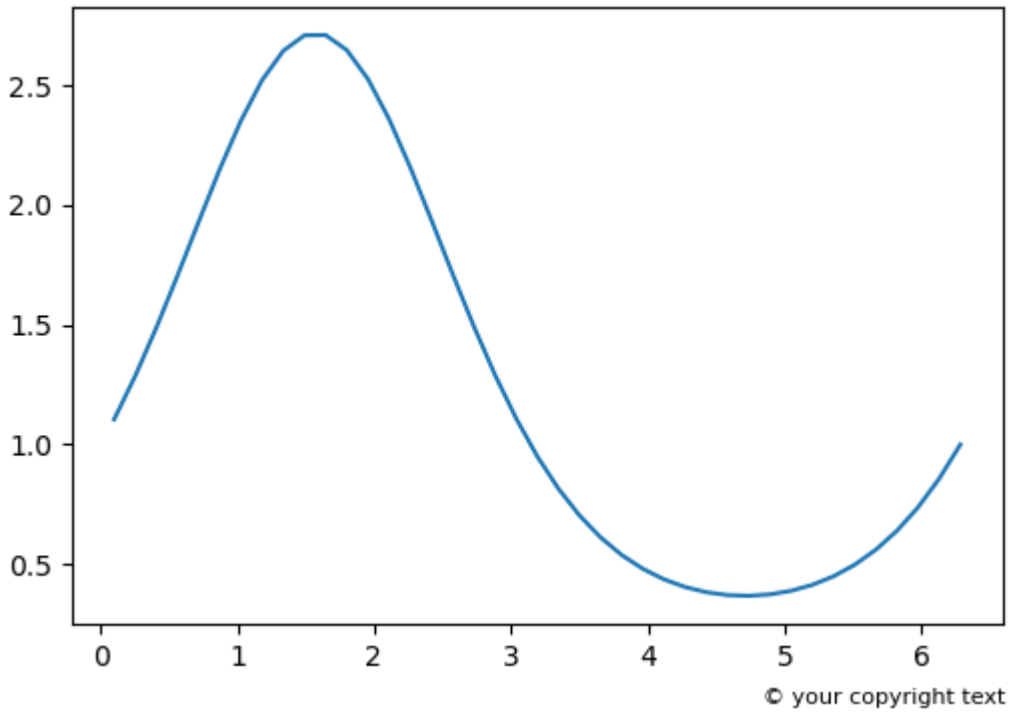
plt.text(0.7, 0.01, '© your copyright text', fontsize=8,
transform=plt.gcf().transFigure)
```

```
fig, ax = plt.subplots(figsize=(6,4))

ax.plot(x, y)

plt.text(0.7, 0.01, '© your copyright text', fontsize=8,
transform=fig.transFigure)
```

```
Text(0.7, 0.01, '© your copyright text')
```



Exercise

1. Move the copyright text to the right side in vertical mode.
2. Choose a different font (e.g. Times New Roman).
3. Choose cursive font style.

4. Choose a different font size.
5. Choose a different font color.

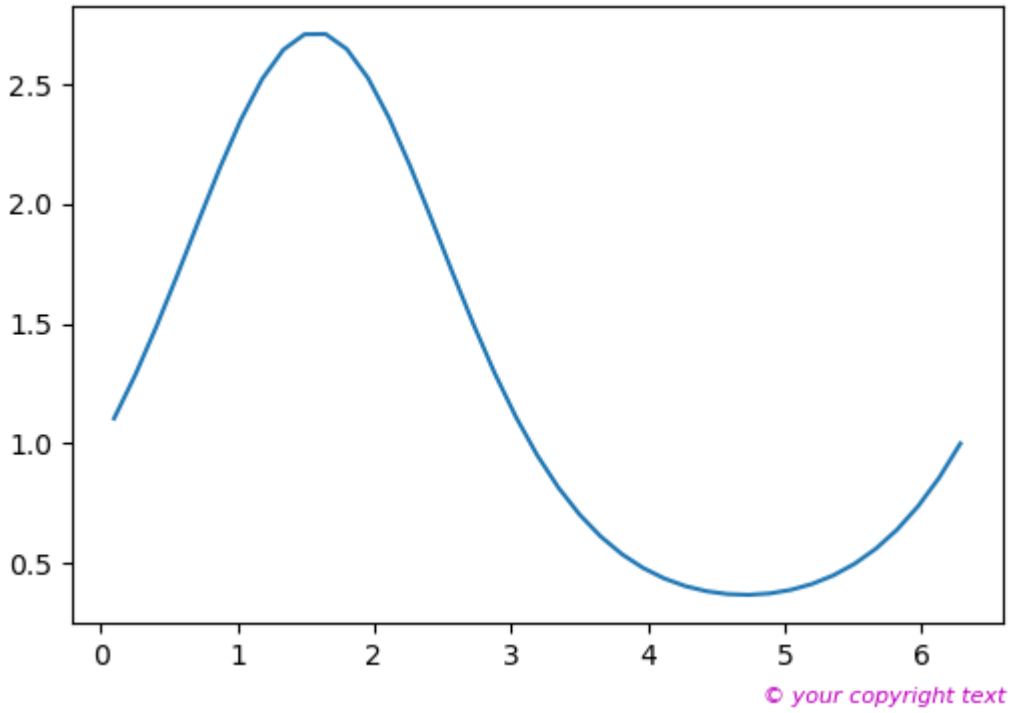
Hint: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html

```
fig, ax = plt.subplots(figsize=(6,4))

ax.plot(x, y)

plt.text(0.7, 0.01, '© your copyright text', fontsize=8,
transform=fig.transFigure, fontstyle='italic', color='m')
```

```
Text(0.7, 0.01, '© your copyright text')
```




```
plt.text?
```

```
□[0;31mSignature:□[0m
□[0mplt□[0m□[0;34m.□[0m□[0mtext□[0m□[0;34m(□[0m□[0mx□[0m□[0;34m,□[0m
□[0my□[0m□[0;34m,□[0m □[0ms□[0m□[0;34m,□[0m
□[0mfontdict□[0m□[0;34m=□[0m□[0;32mNone□[0m□[0;34m,□[0m
□[0;34m**□[0m□[0mkwargs□[0m□[0;34m)□[0m□[0;34m□[0m□[0;34m□[0m□[0m
□[0;31mDocstring:□[0m
Add text to the Axes.
```

```
Add the text *s* to the Axes at location *x*, *y* in data coordinates.

Parameters
----------
x, y : float
    The position to place the text. By default, this is in data
    coordinates. The coordinate system can be changed using the
    *transform* parameter.

s : str
    The text.

fontdict : dict, default: None
    A dictionary to override the default text properties. If fontdict
    is None, the defaults are determined by `.rcParams`.

Returns
-------
`.Text`
    The created `.Text` instance.

Other Parameters
----------------
**kwargs : `~matplotlib.text.Text` properties.
    Other miscellaneous text parameters.

    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
    alpha: scalar or None
    animated: bool
    backgroundcolor: color
    bbox: dict with properties for `.patches.FancyBboxPatch`
    clip_box: unknown
    clip_on: unknown
    clip_path: unknown
    color or c: color
    figure: `.Figure`
    fontfamily or family: {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy',
'monospace'}
    fontproperties or font or font_properties: `.font_manager.FontProperties` or
`str` or `pathlib.Path`
    fontsize or size: float or {'xx-small', 'x-small', 'small', 'medium', 'large',
'x-large', 'xx-large'}
    fontstretch or stretch: {a numeric value in range 0-1000, 'ultra-condensed',
'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded',
'expanded', 'extra-expanded', 'ultra-expanded'}
    fontstyle or style: {'normal', 'italic', 'oblique'}
    fontvariant or variant: {'normal', 'small-caps'}
    fontweight or weight: {a numeric value in range 0-1000, 'ultralight', 'light',
```

```
'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi',
'bold', 'heavy', 'extra bold', 'black'}
    gid: str
    horizontalalignment or ha: {'center', 'right', 'left'}
    in_layout: bool
    label: object
    linespacing: float (multiple of font size)
    math_fontfamily: str
    multialignment or ma: {'left', 'right', 'center'}
    parse_math: bool
    path_effects: `.AbstractPathEffect`
    picker: None or bool or float or callable
    position: (float, float)
    rasterized: bool
    rotation: float or {'vertical', 'horizontal'}
    rotation_mode: {None, 'default', 'anchor'}
    sketch_params: (scale: float, length: float, randomness: float)
    snap: bool or None
    text: object
    transform: `.Transform`
    transform_rotates_text: bool
    url: str
    usetex: bool or None
    verticalalignment or va: {'center', 'top', 'bottom', 'baseline',
'center_baseline'}
    visible: bool
    wrap: bool
    x: float
    y: float
    zorder: float

Examples
--------
Individual keyword arguments can be used to override any given
parameter::

    >>> text(x, y, s, fontsize=12)

The default transform specifies that text is in data coords,
alternatively, you can specify text in axis coords ((0, 0) is
lower-left and (1, 1) is upper-right).  The example below places
text in the center of the Axes::

    >>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
    ...      verticalalignment='center', transform=ax.transAxes)

You can put a rectangular box around the text instance (e.g., to
set a background color) by using the keyword *bbox*.  *bbox* is
a dictionary of `~matplotlib.patches.Rectangle`
properties.  For example::
```

```
    >>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
□[0;31mFile:□[0m      ~/.conda/envs/cygnss-d/lib/python3.9/site-
packages/matplotlib/pyplot.py
□[0;31mType:□[0m      function
```

```

```

Plot settings

Next, as an example, we want to plot annual average temperature data from Germany (from DWD), coloring the data points according to their values, add s title string, and set the axis titles as well. To display the data as individual points we use the scatter function as shown above.

Used parameters of the scatter plot command:

```
    c=temp           use variable temp for coloring the marker

    cmap='afmhot_r'  use colormap afmhot but reversed

    s=80             increase the size of the markers
```

Example:

```
years = np.arange(1951,2021)

temp = np.array([3.02, 7.91, 5.08, 2.53, 0.93, 0.58, 6.85, 1.58, 5.63, 1.30, 3.97,
\
                 2.34, 4.50, 9.93, 1.18, 2.64, 3.95, 2.38, 6.63, 2.01, 7.08, 4.70,
\
                 5.78, 2.75, 6.63, 10.20, 1.25, 1.89, 2.29, 1.45, 2.19, 6.54,
9.93, \
                 3.30, 2.63, 4.31, 1.64, 2.10, 4.67, 5.91, 5.27, 9.64, 1.89,
16.27, \
                 10.54, 3.00, 5.16, 7.08, 5.16, 5.67, 7.36, 6.23, 19.01, 4.70,
7.57, \
                 14.34, 4.82, 7.24, 4.58, 10.63, 4.27, 7.62, 10.47, 6.14, 17.60, \
                 9.21, 6.80, 20.37, 16.97, 11.39])

fig, ax = plt.subplots(figsize=(6,4))

ax.set_title('Annual average temperature', fontsize=24)
```

```
ax.set_ylabel('Temperature\n[deg C]', fontsize=16)
ax.set_xlabel('year', fontsize=16)
ax.tick_params(labelsize=16)

ax.plot(years, temp, color='lightgray')
ax.scatter(years, temp, c=temp, cmap='afmhot_r', s=80)
```

```
years = np.arange(1951,2021)

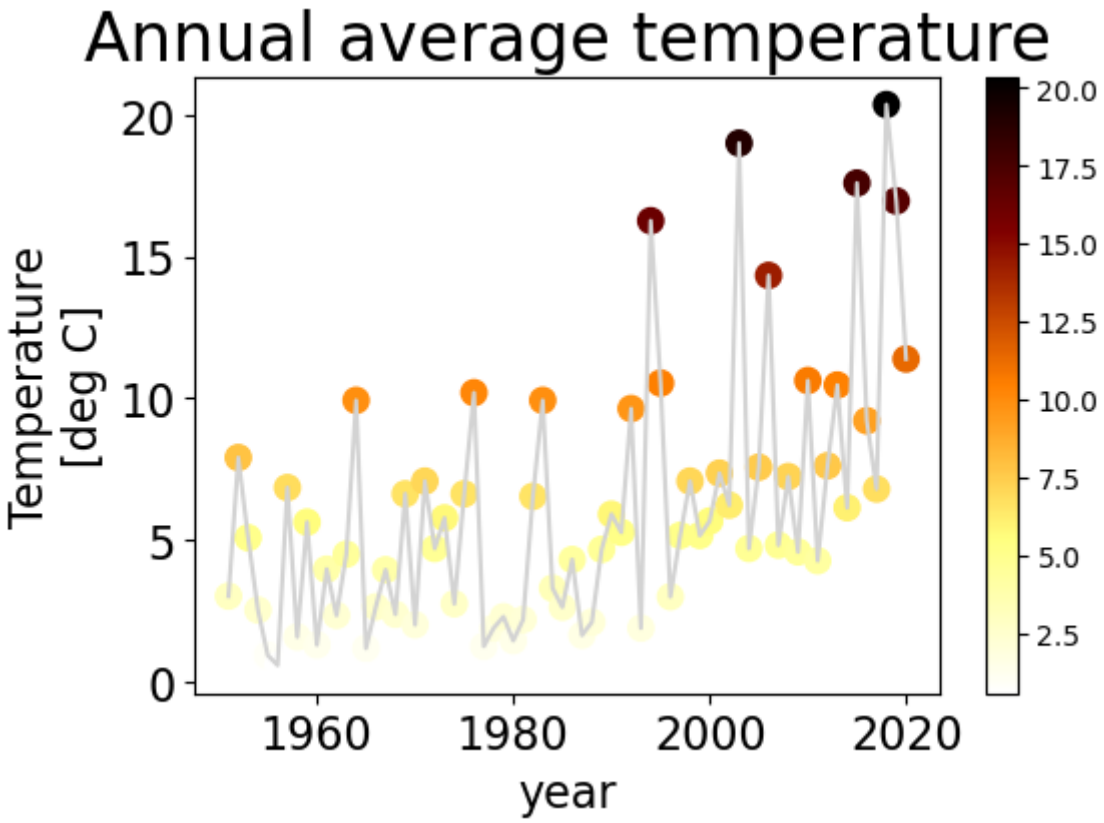
temp = np.array([3.02, 7.91, 5.08, 2.53, 0.93, 0.58, 6.85, 1.58, 5.63, 1.30, 3.97,
\
                 2.34, 4.50, 9.93, 1.18, 2.64, 3.95, 2.38, 6.63, 2.01, 7.08, 4.70,
\
                 5.78, 2.75, 6.63, 10.20, 1.25, 1.89, 2.29, 1.45, 2.19, 6.54,
9.93, \
                 3.30, 2.63, 4.31, 1.64, 2.10, 4.67, 5.91, 5.27, 9.64, 1.89,
16.27, \
                 10.54, 3.00, 5.16, 7.08, 5.16, 5.67, 7.36, 6.23, 19.01, 4.70,
7.57, \
                 14.34, 4.82, 7.24, 4.58, 10.63, 4.27, 7.62, 10.47, 6.14, 17.60, \
                 9.21, 6.80, 20.37, 16.97, 11.39])

fig, ax = plt.subplots(figsize=(6,4))

ax.set_title('Annual average temperature', fontsize=24)
ax.set_ylabel('Temperature\n[deg C]', fontsize=16)
ax.set_xlabel('year', fontsize=16)
ax.tick_params(labelsize=16)

ax.plot(years, temp, color='lightgray')
img = ax.scatter(years, temp, c=temp, cmap='afmhot_r', s=80)
plt.colorbar(img)
```

```
<matplotlib.colorbar.Colorbar at 0x7fffb2cc7e80>
```



Note: The draw order is first marker and than line, which doen't look quite nice. To change the draw order we can use the **zorder** parameter.

Example:

```
fig, ax = plt.subplots(figsize=(6,4))

ax.set_title('Annual average temperature', fontsize=24)
ax.set_ylabel('Temperature\n[deg C]', fontsize=16)
ax.set_xlabel('year', fontsize=16)
ax.tick_params(labelsize=16)

ax.plot(years, temp, color='lightgray', zorder=1)
ax.scatter(years, temp, c=temp, cmap='afmhot_r', s=80, zorder=2)
```

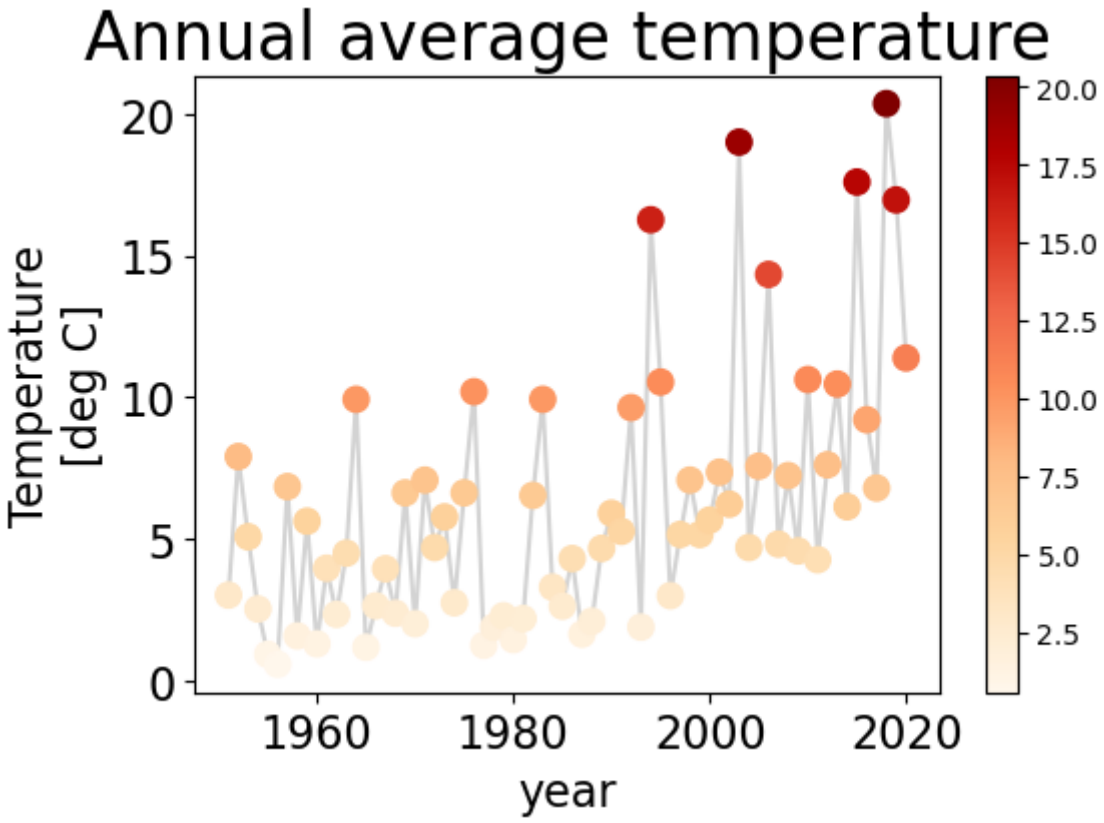
```
fig, ax = plt.subplots(figsize=(6,4))

ax.set_title('Annual average temperature', fontsize=24)
ax.set_ylabel('Temperature\n[deg C]', fontsize=16)
ax.set_xlabel('year', fontsize=16)
ax.tick_params(labelsize=16)

ax.plot(years, temp, color='lightgray', zorder=1)
```

```
img=ax.scatter(years, temp, c=temp, cmap='OrRd', s=80, zorder=2)
plt.colorbar(img)

plt.savefig("Germany_average_annual_temperature_1951-2020.png",
bbox_inches="tight", dpi=300)
```




```
cond = (years >= 1980) & (years <= 2000)
print(years[cond])
```

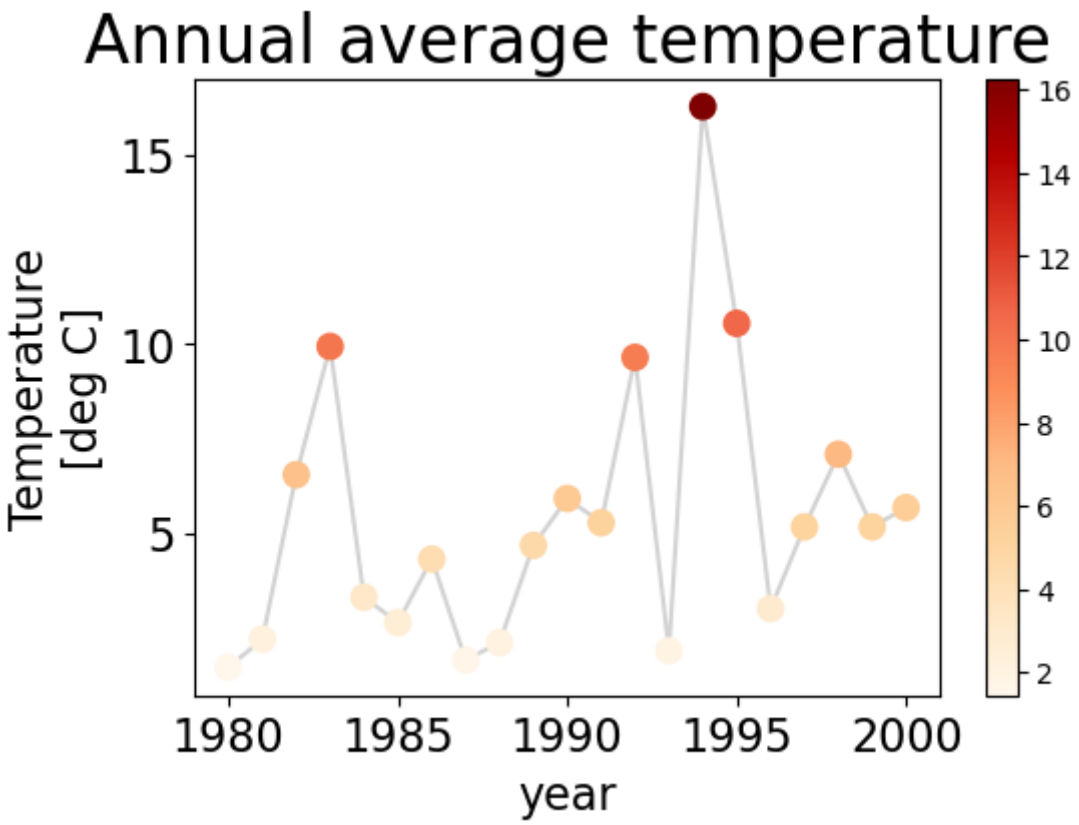
```
[1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993
 1994 1995 1996 1997 1998 1999 2000]
```

```
fig, ax = plt.subplots(figsize=(6,4))

ax.set_title('Annual average temperature', fontsize=24)
ax.set_ylabel('Temperature\n[deg C]', fontsize=16)
ax.set_xlabel('year', fontsize=16)
ax.tick_params(labelsize=16)

ax.plot(years[cond], temp[cond], color='lightgray', zorder=1)
img=ax.scatter(years[cond], temp[cond], c=temp[cond], cmap='OrRd', s=80, zorder=2)
plt.colorbar(img)
```

```
<matplotlib.colorbar.Colorbar at 0x7fffb05e2c10>
```



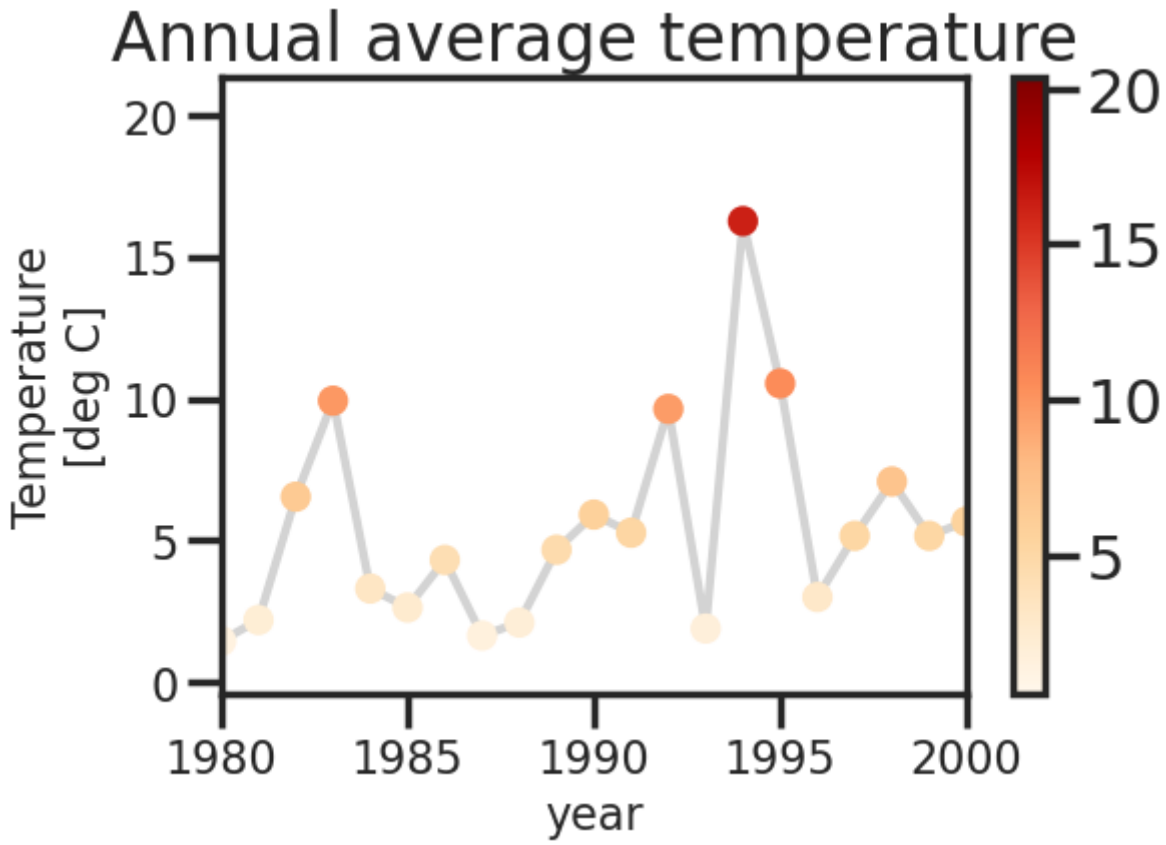

```
fig, ax = plt.subplots(figsize=(6,4))

ax.set_title('Annual average temperature', fontsize=24)
ax.set_ylabel('Temperature\n[deg C]', fontsize=16)
ax.set_xlabel('year', fontsize=16)
ax.tick_params(labelsize=16)

ax.plot(years, temp, color='lightgray', zorder=1)
img=ax.scatter(years, temp, c=temp, cmap='OrRd', s=80, zorder=2)
plt.colorbar(img)

ax.set_xlim(1980, 2000)

plt.savefig('new_fig.png', dpi=90)
```



Matplotlib offers a wide selection of colormaps. You can select colormaps for sequential and for categorical data. For hints which colormap to choose, see https://matplotlib.org/stable/tutorials/colors/colormaps.html.

Save the plot to PNG

Since we do not only want to have the plots as an inline image, but also save them as a PNG file, we show here how to do that. Therefore, we add a line with pyplots function `savefig` at the end of the script snippet.

```
plt.savefig("Germany_average_annual_temperature_1951-2020.png")
```

Beautiful plots with Seaborn

Seaborn is a high-level interface to matplotlib, specifically aimed at statistical plots. It is very useful for generating beautiful plots that are easy to understand. Documentation: https://seaborn.pydata.org/

With the `set_style` function we can select one of five predefined styles (`whitegrid`, `white`, `darkgrid`, `dark`, `ticks`). Seaborn automatically adjusts the grid and figure background to the preferences and chooses reasonable values for tick and label sizes. We now make a lineplot with Seaborn, using dashed red lines and square markers:

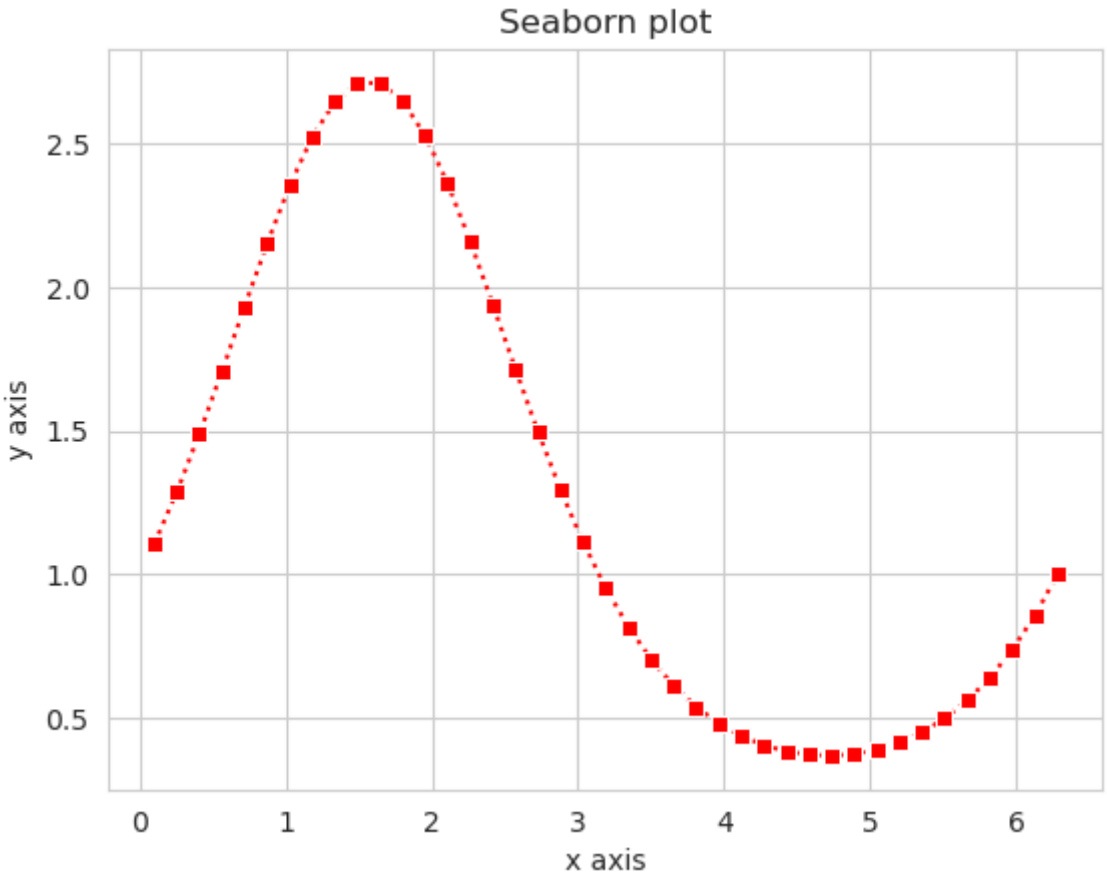
```
import seaborn as sns
sns.set_style('whitegrid')
```

```
sns.lineplot(x=x, y=y, color='red', marker='s', linestyle=':')
plt.title('Seaborn plot')
plt.xlabel('x axis')
plt.ylabel('y axis')
```

```
import seaborn as sns

sns.set_style('whitegrid')

sns.lineplot(x=x, y=y, color='red', marker='s', linestyle=':')
plt.title('Seaborn plot')
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.show()
```



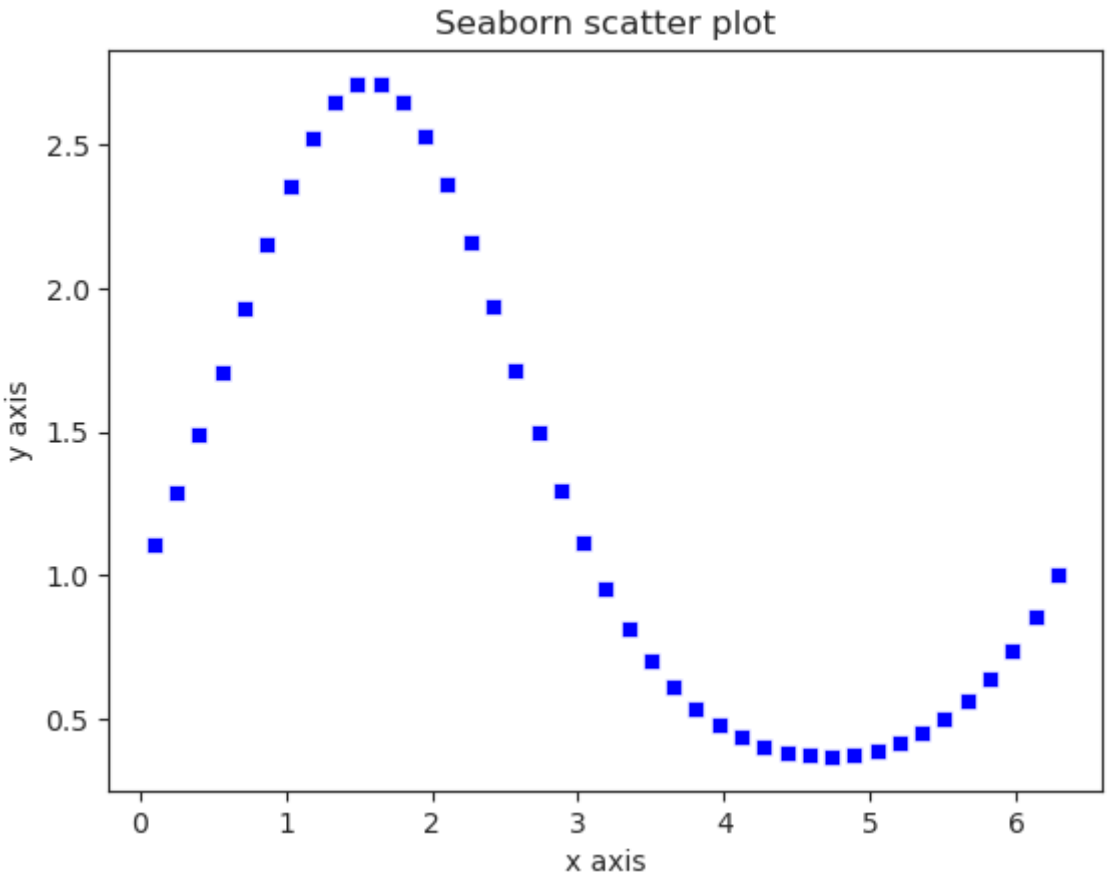
Scatter plot with Seaborn, this time using the 'ticks' style:

```
sns.set_style('ticks')
sns.scatterplot(x=x, y=y, color='blue', marker='s')
plt.title('Seaborn plot')
```

```
plt.xlabel('x axis')
plt.ylabel('y axis')
```

```
sns.set_style('ticks')
sns.scatterplot(x=x, y=y, color='blue', marker='s')
plt.title('Seaborn scatter plot')
plt.xlabel('x axis')
plt.ylabel('y axis')
```

```
Text(0, 0.5, 'y axis')
```



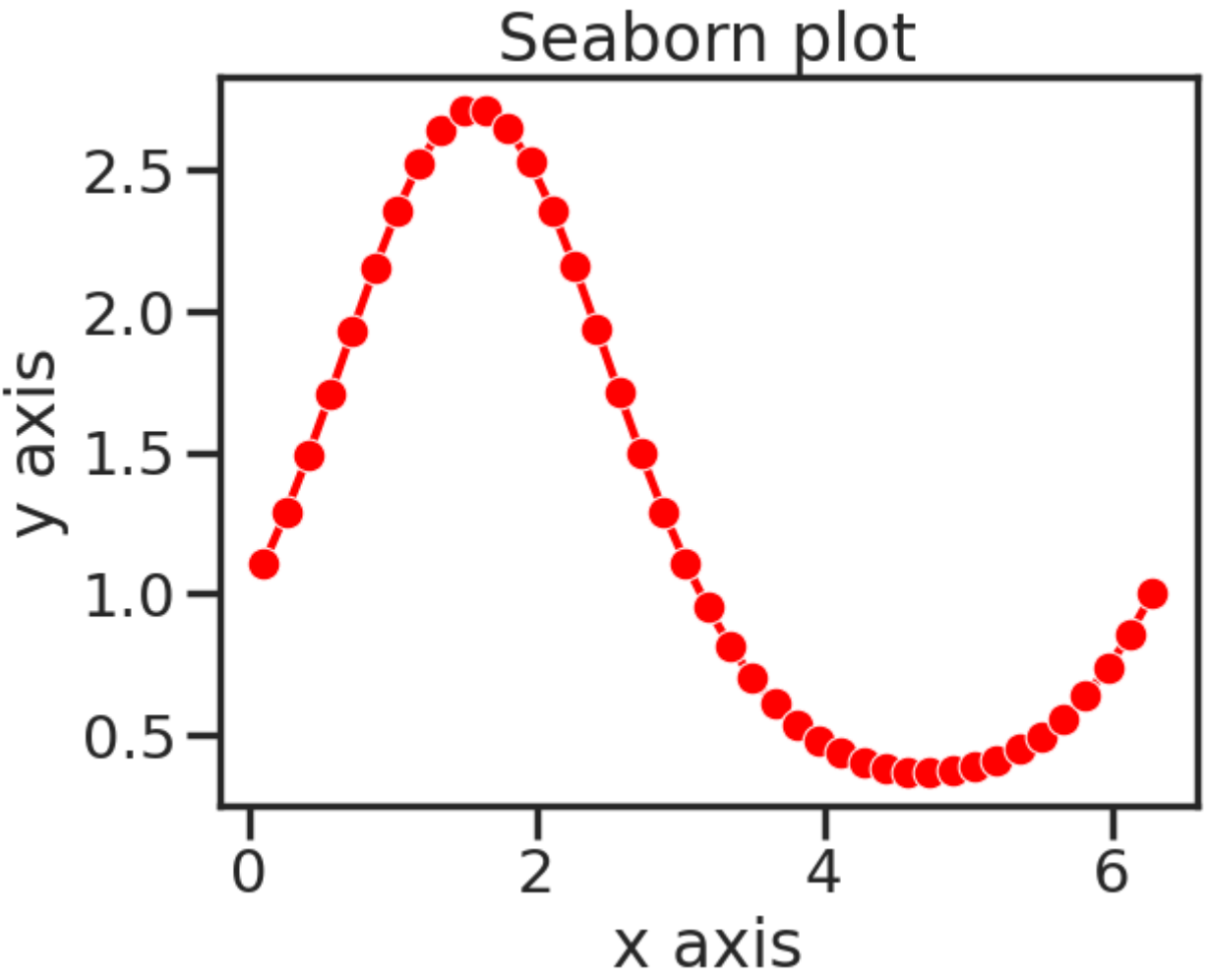
Depending where you want to use your figure, the settings for the plot elements need to be different. For example, in a paper you might need to use small fontsizes, while for a poster presentation you need to produce plots that are easy to read from a distance. Seaborn offers the function `set_context` that can take any value of `notebook` (default), `paper`, `talk`, and `poster`. We repeat the line plot, this time in the `poster` context and using round red markers.

```
```

```
sns.set_context('poster')
sns.set_style('ticks')
sns.lineplot(x=x, y=y, color='red', marker='o')
plt.title('Seaborn plot')
plt.xlabel('x axis')
plt.ylabel('y axis')
```

```
sns.set_context('poster') # default context: sns.set_context("notebook")
sns.set_style('ticks')
sns.lineplot(x=x, y=y, color='red', marker='o')
plt.title('Seaborn plot')
plt.xlabel('x axis')
plt.ylabel('y axis')
```

```
Text(0, 0.5, 'y axis')
```



Exercise

Use seaborn to create the annual average example plot from the 'Plot settings' section above, but

1. use darkgrid style
2. use line width 1
3. use marker size 40
4. draw the marker on top of the line

```

```

Play time - 15 minutes

Now, it is on you to play with matplotlib's and seaborn's capabilities. 😃

> **Note:**
>
> To reset the style settings from seaborn you can do the following:
>
> ```
> sns.reset_orig()
> ```

```

```