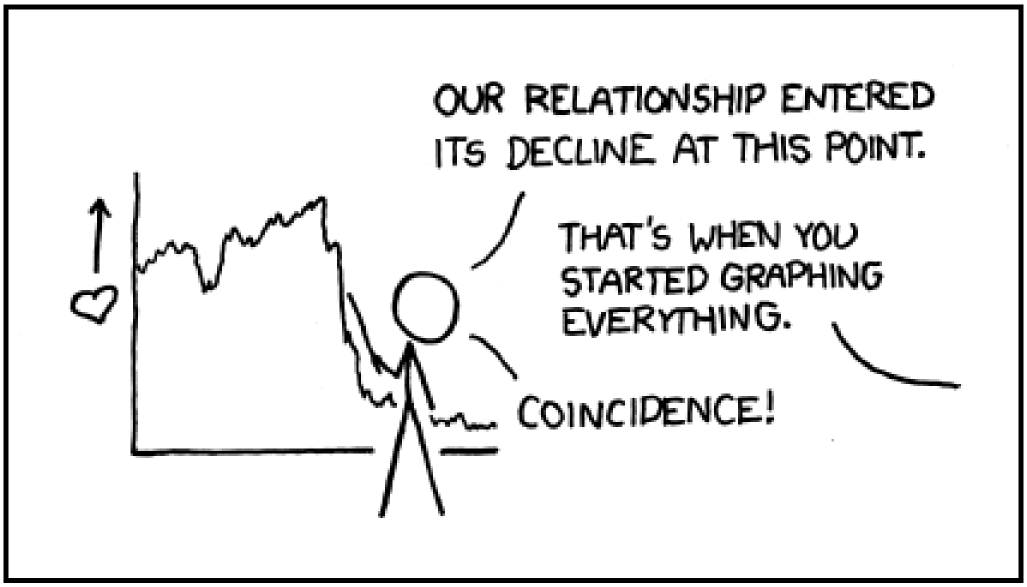
Numerical_methods_lab_2

December 9, 2024

1 Numerical Methods Lab 2: Data & Plotting

xkcd comic # 523: Decline



There are a number of libraries in Python that can assist with reading in data and plotting. The ones we will focus on today are

- numpy
- matplotlib
- pandas

If you don't have any of these libraries installed, you can install them like this:

[ ]:
```
%pip install numpy
%pip install pandas
%pip install matplotlib
```

When importing libraries in Python, you can rename them very easily for your convenience:

[2]:
```
import numpy as np
import pandas as pd
```

This way, you don't have to type out the entire library name every time you use it!

1.0.1 Introducing numpy: arrays

Numpy is a powerful and easy to use package used for scientific computing in Python.

As we saw in Lab 1, lists are useful tools to store all kinds of data, and can be quite fast when dealing with only a few elements in a list. However, depending on your use-case, you may need to perform more complex operations on large amounts of numerical data. This is where numpy arrays come in handy! You can essentially treat these arrays as vectors and matrices, and are created like this:

[3]:
```
vec = np.array([1, 0, 1])

vec
```

[3]:
```
array([1, 0, 1])
```

Accessing items in an array, and slicing arrays are exactly the same as with lists:

[4]:
```
vec[0]
```

[4]:
```
1
```

[5]:
```
vec[1:]
```

[5]:
```
array([0, 1])
```

And of course, this is how we define a matrix:

[6]:
```
mat = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
mat
```

[6]:
```
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

Operations are incredibly simple:

[7]:
```
# addition

vec2 = np.array([2, 1, 1])

vec + vec2
```

[7]:
```
array([3, 1, 2])
```

[8]:
```
# multiplication

vec * vec2
```

[8]: array([2, 0, 1])

[9]:
```
# division

vec / vec2
```

[9]: array([0.5, 0. , 1.])

You can find out the number of dimensions in an array by using *ndim*:

[10]:
```
vec.ndim
```

[10]: 1

[11]:
```
mat.ndim
```

[11]: 2

and find out the number of elements along each dimension using *shape*:

[12]:
```
vec.shape
```

[12]: (3,)

[13]:
```
mat.shape
```

[13]: (3, 3)

Finally, you can find out the total number of elements by using *size*:

[14]:
```
mat.size
```

[14]: 9

There are some other ways of defining arrays in numpy. For instance, if you want to initialise an array consisting of zeros:

[15]:
```
zeros = np.zeros(5)      # this is an array containing 5 elements of zero

zeros
```

[15]: array([0., 0., 0., 0., 0.])

[16]:
```
zeros_mat = np.zeros((3, 2))        # creates a 3x2 matrix of zeros

zeros_mat
```

[16]: array([[0., 0.],
       [0., 0.],
       [0., 0.]])

An array filled with ones:

[17]:
```
ones = np.ones(4)

ones
```

[17]: array([1., 1., 1., 1.])

An array consisting of a range of numbers:

[18]:
```
my_range = np.arange(10)

my_range
```

[18]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

An array containing a range of numbers at even intervals:

[19]:
```
my_range2 = np.arange(0, 20, 5)      # within the brackets are the first␣
↪number, last number, and step size

my_range2
```

[19]: array([0, 5, 10, 15])

An array with values that are spaced linearly in a given interval:

[20]:
```
lin_array = np.linspace(5, 25)          # default 50 elements

print(lin_array)

lin_array2 = np.linspace(5, 25, 10)     # only 10 elements

print(lin_array2)
```

```
[ 5.          5.40816327  5.81632653  6.2244898   6.63265306  7.04081633
  7.44897959  7.85714286  8.26530612  8.67346939  9.08163265  9.48979592
  9.89795918 10.30612245 10.71428571 11.12244898 11.53061224 11.93877551
 12.34693878 12.75510204 13.16326531 13.57142857 13.97959184 14.3877551
 14.79591837 15.20408163 15.6122449  16.02040816 16.42857143 16.83673469
 17.24489796 17.65306122 18.06122449 18.46938776 18.87755102 19.28571429
 19.69387755 20.10204082 20.51020408 20.91836735 21.32653061 21.73469388
 22.14285714 22.55102041 22.95918367 23.36734694 23.7755102  24.18367347
 24.59183673 25.        ]
```

```
[ 5.          7.22222222  9.44444444 11.66666667 13.88888889 16.11111111
 18.33333333 20.55555556 22.77777778 25.        ]
```

An array with values that are spaced logarithmically in a given interval:

[21]:
```
log_array = np.logspace(0, 3, 10)       # an array between 1 and 1000␣
 ↪containing 10 elements

log_array
```

[21]:
```
array([   1.        ,    2.15443469,    4.64158883,   10.        ,
         21.5443469 ,   46.41588834,  100.        ,  215.443469  ,
        464.15888336, 1000.        ])
```

You can find out more about numpy arrays here if you're interested.

1.0.2 Introducing numpy: constants & functions

There are a number of constants that numpy provides for you:

- pi
- e
- a representation of infinity

[22]:
```
print(f"pi:  {np.pi}")

print(f"e:   {np.e}")

print(f"infinity:   {np.inf}")
```

```
pi:  3.141592653589793
e:   2.718281828459045
infinity:   inf
```

A range of mathematical functions are also included within numpy, for example:

- trigonometry
- rounding
- sums, products, differences
- exponents and logarithms

Trigonometry is always performed in radians. Examples of trigonometric functions include:

[23]:
```
# sin
print(np.sin(np.pi / 2))

# cos
print(np.cos(0))

# tan
print(np.tan(np.pi / 3))
```

```python
# arcsin
print(np.arcsin(0))

# arccos
print(np.arccos(0))
```

```
1.0
1.0
1.7320508075688767
0.0
1.5707963267948966
```

There are three main kinds of rounding that can be performed.

Rounding up to the nearest integer using *round*:

[24]:
```python
np.round(3.2)
```

[24]: 3.0

rounding down using *floor*:

[25]:
```python
np.floor(3.7)
```

[25]: 3.0

and rounding up using *ceil*:

[26]:
```python
np.ceil(3.2)
```

[26]: 4.0

You can calculate the sum across an array by using *sum*:

[27]:
```python
array = np.arange(1, 5)

print(array)

np.sum(array)
```

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

[27]: 10

The product across an array by using *prod*:

[28]:
```python
np.prod(array)
```

[28]: 24

The cumulate sum of an array using *cumsum*:

[29]:
```
np.cumsum(array)
```

[29]:
```
array([ 1,  3,  6, 10])
```

The mean of an array using *mean*:

[30]:
```
np.mean(array)

print(np.std(array))
```

```
1.118033988749895
```

You can calculate the exponent, $\exp(x)$, of a single value, or an array by using *exp*:

[31]:
```
np.exp(2)
```

[31]:
```
7.38905609893065
```

[32]:
```
np.exp(array)
```

[32]:
```
array([ 2.71828183,  7.3890561 , 20.08553692, 54.59815003])
```

The natural logarithm, $\log(x)$, of a single value, or an array using *log*:

[33]:
```
np.log(2)
```

[33]:
```
0.6931471805599453
```

[34]:
```
np.log(array)
```

[34]:
```
array([0.        , 0.69314718, 1.09861229, 1.38629436])
```

The base-10 logarithm, $\log_{10}(x)$, of a single value, or an array using *log10*:

[35]:
```
np.log10(100)
```

[35]:
```
2.0
```

[36]:
```
np.log10(array)
```

[36]:
```
array([0.        , 0.30103   , 0.47712125, 0.60205999])
```

For a more comprehensive list of mathematical functions contained within numpy, you can look here.

1.0.3 Randomness with numpy

Sometimes we may need to generate random numbers, for instance in the case of

- sampling data
- generating noise
- simulations & modelling

There are two main methods to produce random numbers. The first relies on physical phenomena that is expected to be random, for instance measuring thermal noise or quantum data. This is known as *true* random number generation. Can you think of any other physical examples that can provide randomness to us?

The second method relies on algorithms that can generate series of numbers that can appear random, however these are ultimately deterministic due to their reliance on an initial condition known as a *seed*. This is called *pseudorandom* number generation. The use of seeds in this method is beneficial as it ensures reproducibility of the random data.

Numpy employs pseudorandom number generators for generating and manipulating random numbers. Let's begin by first setting a seed:

[37]:
```
np.random.seed(1)
```

We can select a random float between 0 and 1 by simply calling

[38]:
```
print(np.random.rand())

print(np.random.rand(3))
```

```
0.417022004702574
[7.20324493e-01 1.14374817e-04 3.02332573e-01]
```

If we wanted to select a random integer from a range:

[39]:
```
print(np.random.randint(10))
print(np.random.randint(10, size=3))
```

```
0
[0 1 7]
```

Randomly selected data can be done by using *choice*:

[40]:
```
my_da = np.random.randint(10, size=5)

print(my_da)

print(np.random.choice(my_da))

print(np.random.choice(my_da, size=3))

# with repacement
print(np.random.choice(my_da, size=3, replace=True))
```

```
[6 9 2 4 5]
2
```

```
[5 4 5]
[2 5 2]
```

Oftentimes, you may want to generate random numbers from a probability distribution. The most basic is the uniform distribution

$$p(x) = \frac{1}{b-a}$$

where a and b define the interval $[a, b)$. In numpy, the default values are $a = 0$, and $b = 1$.

[41]:
```python
print(np.random.uniform())
```

```
0.9139620245792329
```

We can very easily define the interval:

[42]:
```python
# between 0-10
print(np.random.uniform(10))

# between 2-5
print(np.random.uniform(2,5))
```

```
5.8851567281171056
3.2920957015541252
```

and the number of points

[43]:
```python
print(np.random.uniform(10, 20, size=5))
```

```
[19.39127789 17.78389236 17.15970516 18.02757504 10.92800809]
```

The probability distribution that most occurs in nature is the normal distribution (also known as the Gaussian):

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

with mean μ and standard deviation σ. Recall that the square of the standard deviation σ^2 is the variance. In numpy the default values are $\mu = 0$ and $\sigma = 1$.

[44]:
```python
print(np.random.normal())
```

```
1.118296977299658
```

and of course, you can define your own mean and standard deviation

[45]:
```python
print(np.random.normal(10, 3))

print(np.random.normal(10, 3, size=5))
```

```
10.166839569875433
[11.12996991 11.12840506  9.50742769  9.74636965 12.79467449]
```

[46]:
```python
my_norm = np.random.normal(10, 5, size=100000)

print(abs(10 - np.mean(my_norm)))

print(abs(5 - np.std(my_norm)))
```

```
0.008979279218179315
0.00441481355197304
```

There are many distributions that can be sampled using numpy.random, for a more comprehensive list you can see here

1.0.4 Plotting with matplotlib

The matplotlib library contains the *pyplot* module which is used for plotting. We can import it in like this:

[47]:
```python
import matplotlib.pyplot as plt
```

Matplotlib.pyplot graphs data on *Figures*, which can contain multiple *axes*. Figures are like a canvas for the plots to be created on, and you can specify a range of features including:

- size of the figure
- the background colour
- titles and legends
- subfigures and subplots
- colourbars
- … and many more!

Axes are essentially where the plots are created within the figure, and you can specify a range of features including:

- axis labels
- error bars
- the type of plot (e.g. scatterplot, histogram..)
- vertical and horizontal lines
- grid lines
- axis limits
- legends
- … and many more!

A great way to visualise this is shown below:

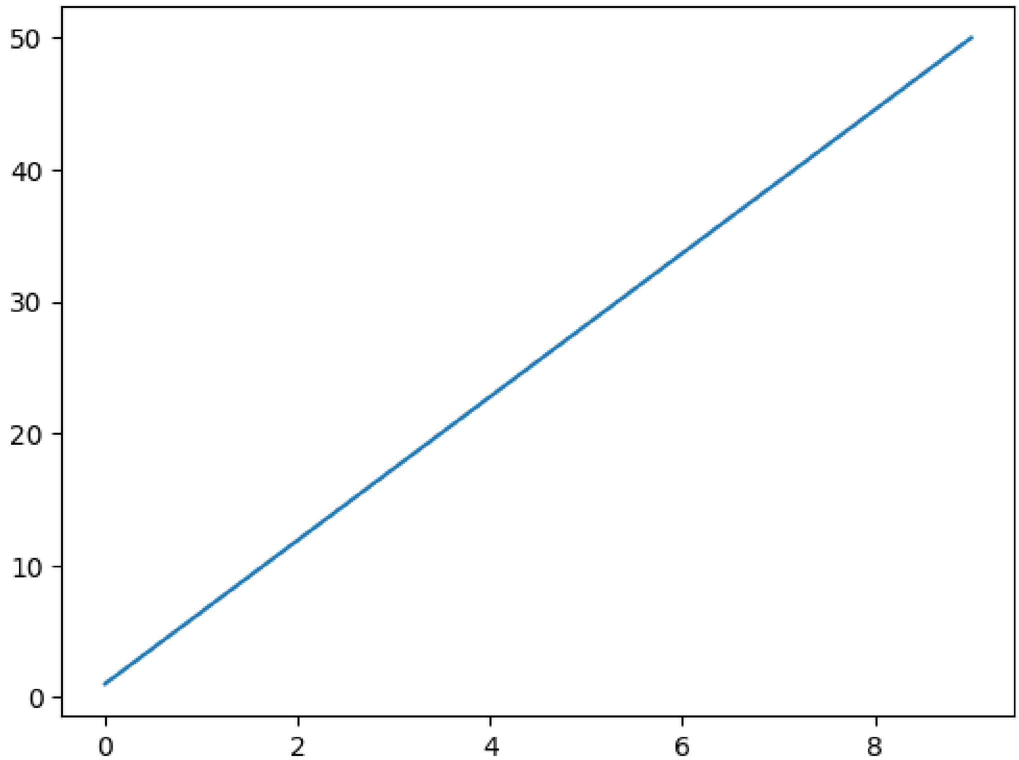
source: https://matplotlib.org

Let's first create some dummy data for plotting. When plotting, it is important that your x- and y- datasets are of the same length, otherwise an error will occur!

```
[48]: x = np.arange(10)
      y = np.linspace(1, 50, 10)
```

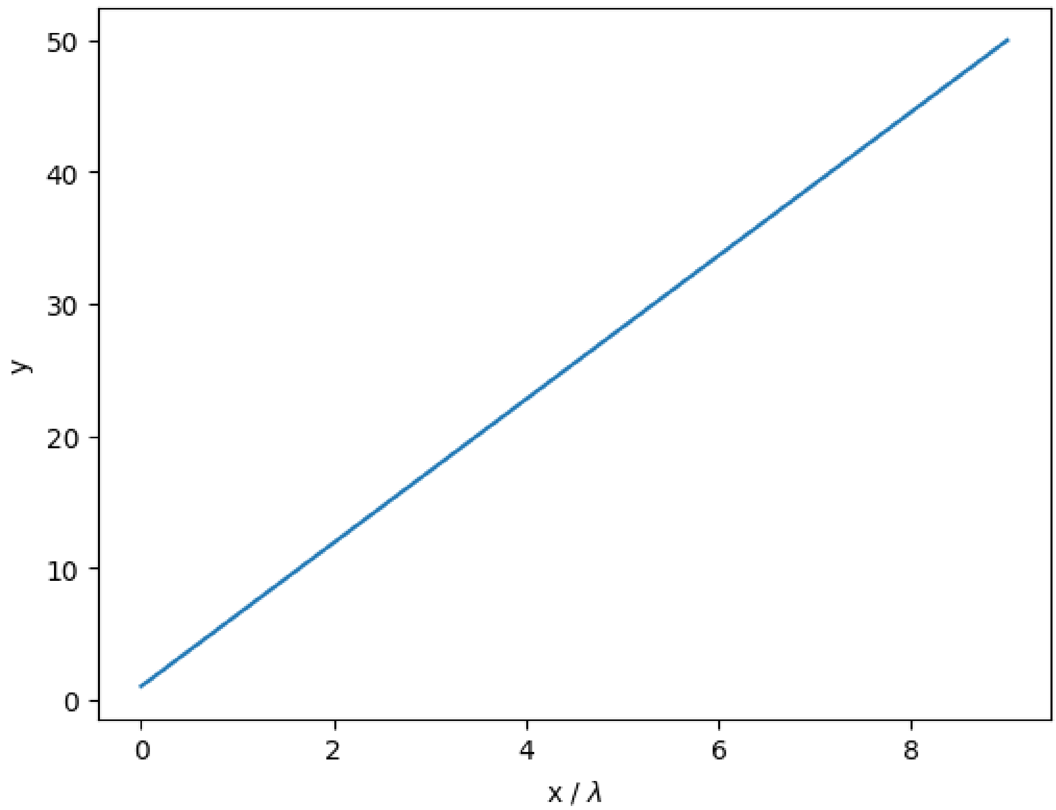
There are a number of ways to initialise plots using pyplot. The most basic way is like this:

```
[49]: fig, ax = plt.subplots()          # initialise the plot
      ax.plot(x, y)                     # plot y vs. x as a line
      plt.show()                        # display the plot
```



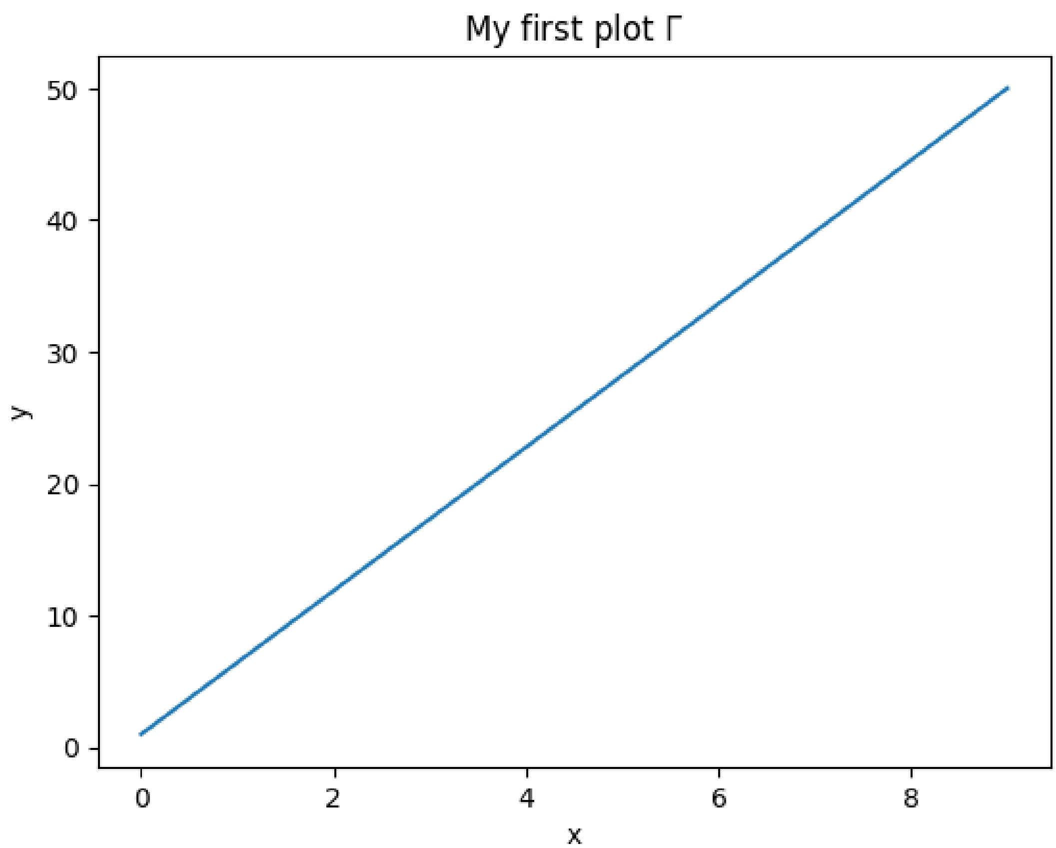
You can specify axis labels like this:

```
[50]: fig, ax = plt.subplots()
      ax.plot(x, y)
      ax.set_xlabel(r"x / $\lambda$")       # sets the x-axis label to be "x"
      ax.set_ylabel("y")      # sets the y-axis label to be "y"
      plt.show()
```



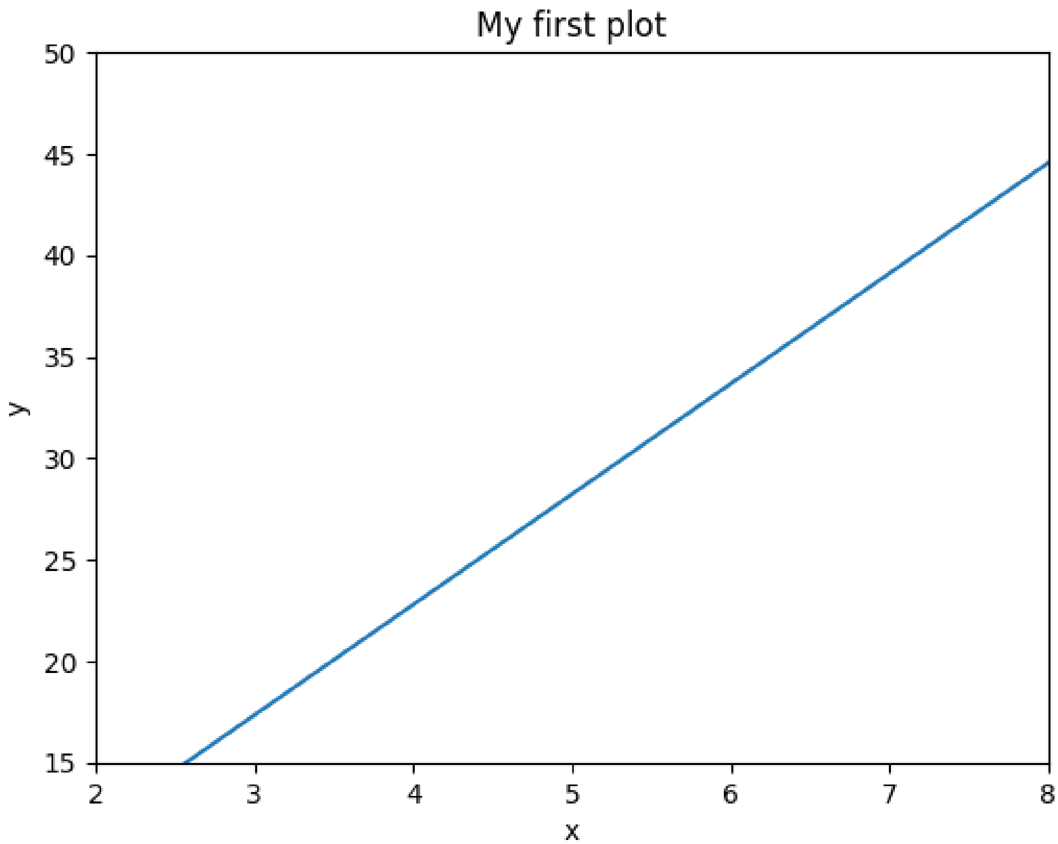
You can even add a title to your plot!

[51]:
```
fig, ax = plt.subplots()
plt.plot(x, y)
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.title(r"My first plot $\Gamma$")      # creates a title called "My first␣
↪plot"
plt.show()
```



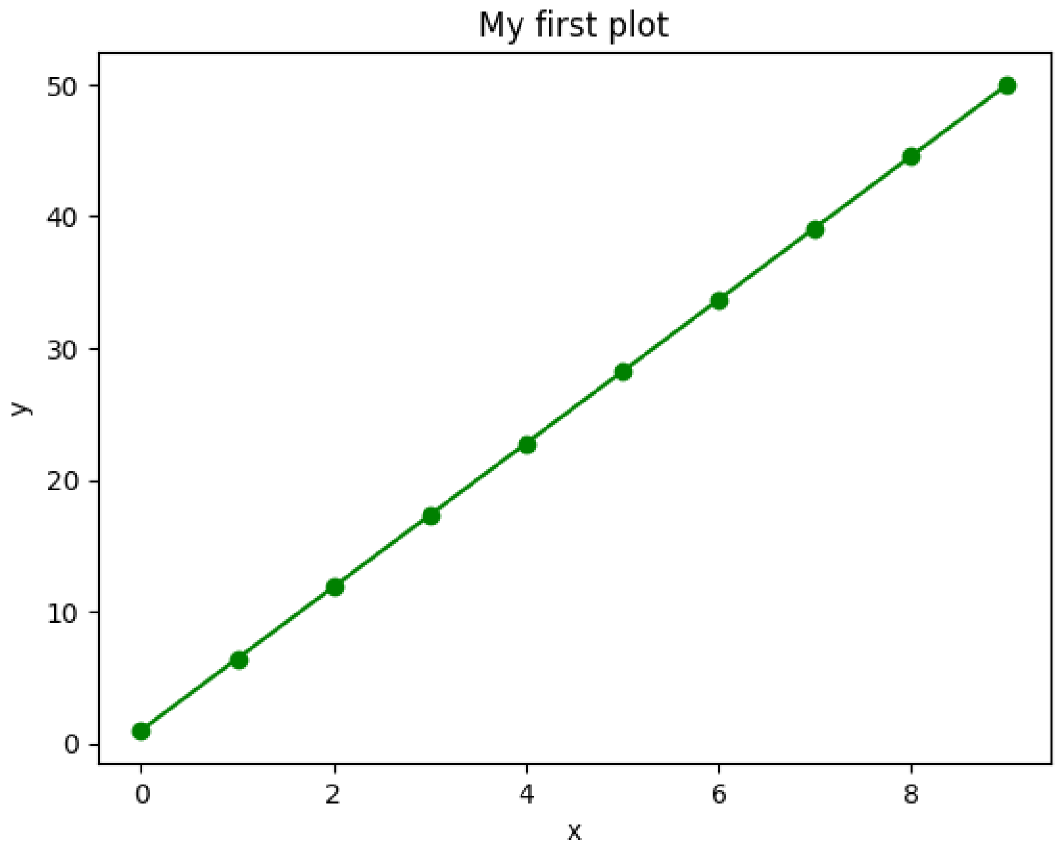
You can define the axis range using *xlim* and *ylim*:

[52]:
```
fig, ax = plt.subplots()
plt.plot(x, y)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set(xlim=[2, 8])
ax.set(ylim=[15, 50])
plt.title("My first plot")      # creates a title called "My first plot"
plt.show()
```



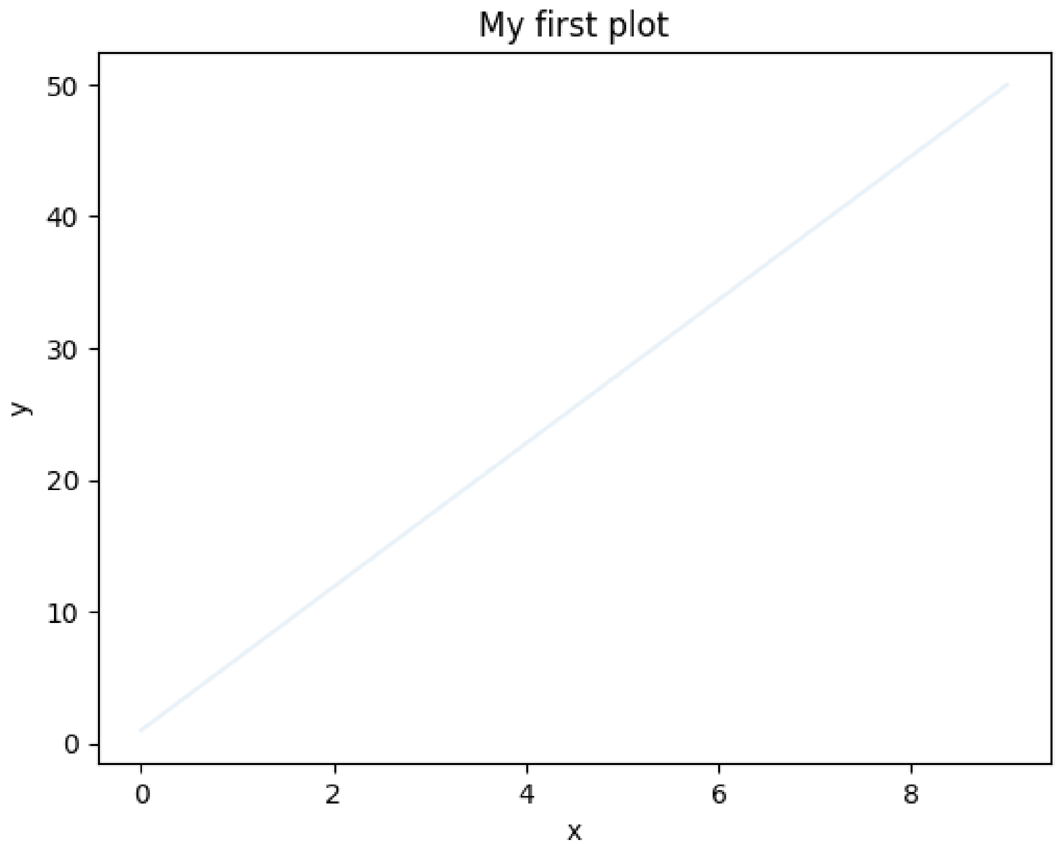
We can also specify more plot parameters within plt.plot:

[53]:
```
fig, ax = plt.subplots()
plt.plot(x, y, marker='o', color='g')        # includes markers for the data␣
 ↪points, and changes the line to green
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.title("My first plot")
plt.show()
```



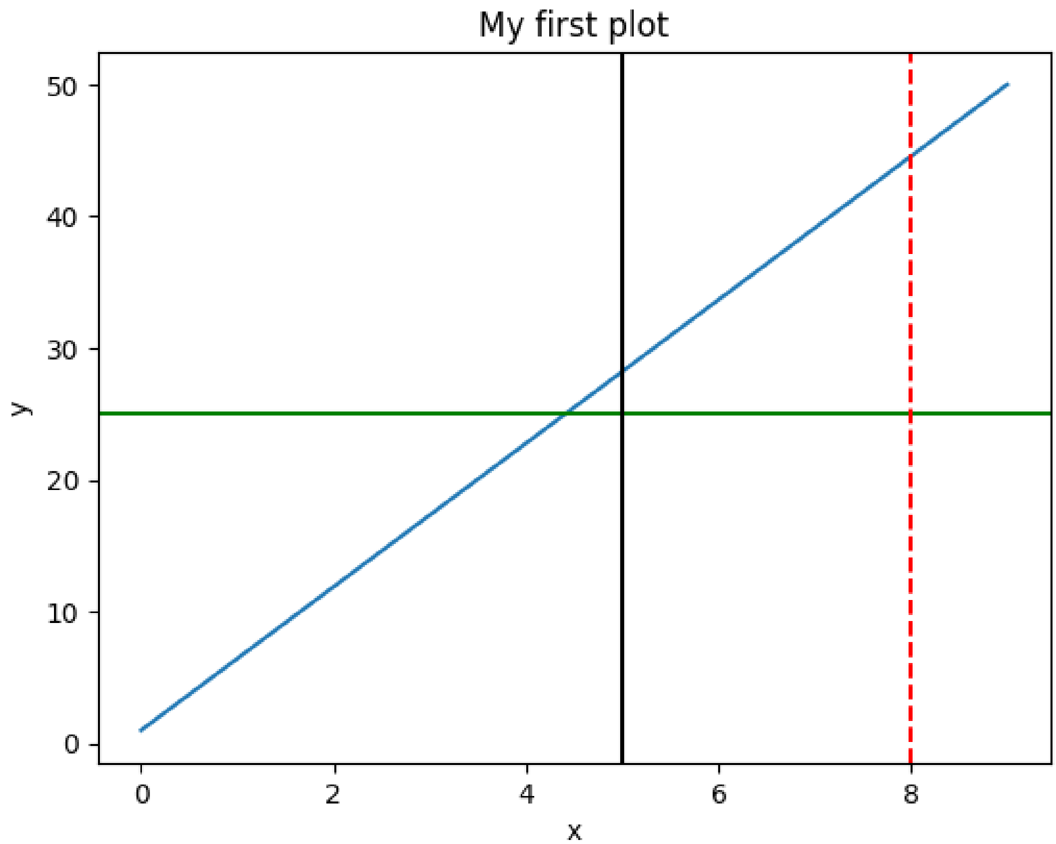
You can also change the transparency of the data by using *alpha*

[54]:
```
fig, ax = plt.subplots()
plt.plot(x, y, alpha=0.1)
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.title("My first plot")
plt.show()
```



Horizontal and vertical lines:

[55]:
```
fig, ax = plt.subplots()
plt.plot(x, y)
ax.axhline(25, color='g')        # specifies the colour of this line to be green
ax.axvline(5, color='k')         # specifies the colour of this line to be black
ax.axvline(8, color='r', linestyle='--')        # specifies the colour of this␣
↪line to be red, and the line to be dashed
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.title("My first plot")
plt.show()
```

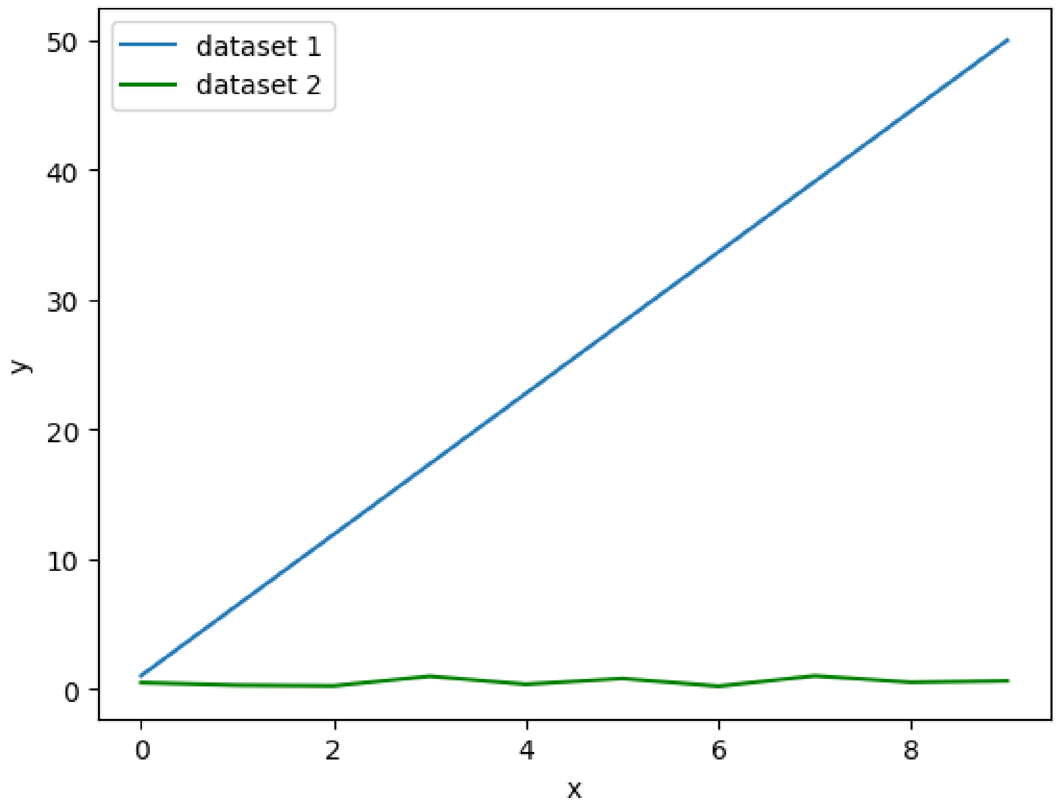


You can include as much data as you like within plots. Let's generate some more dummy data:

[56]:
```
# create your own data here:

x2 = np.arange(10)

y2 = np.random.uniform(size=10)
```

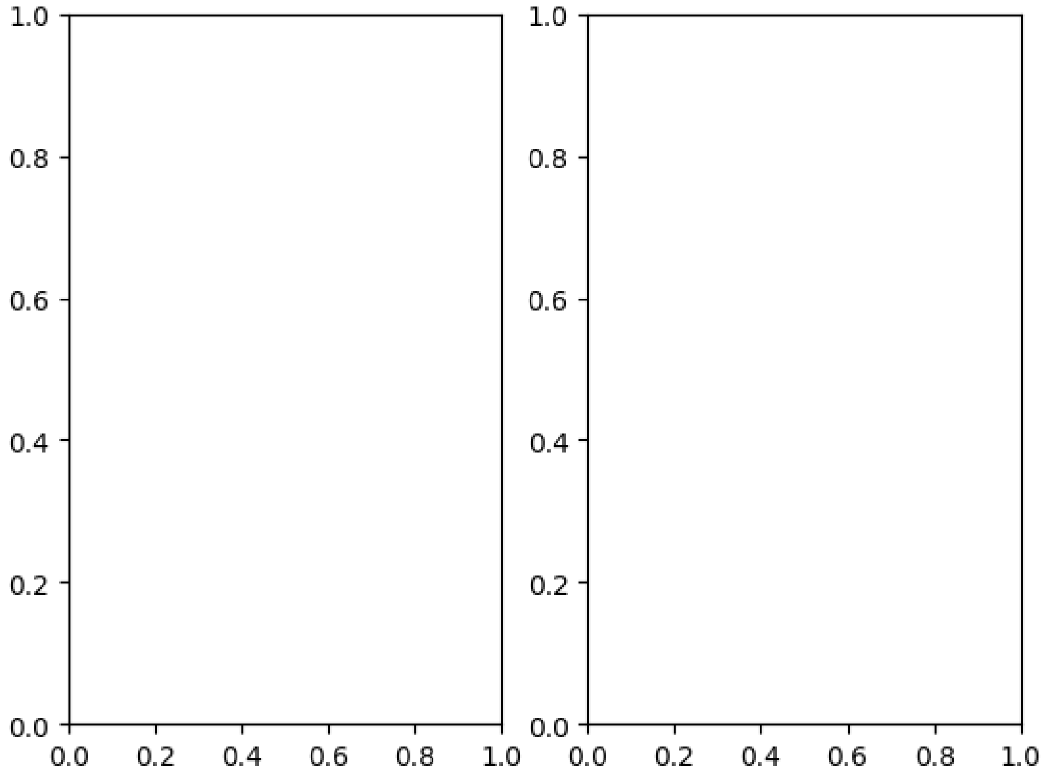
Let's plot this data, along with a legend!

[57]:
```
fig, ax = plt.subplots()
plt.plot(x, y, label="dataset 1")            # sets the label of the data
plt.plot(x2, y2, color='g',label="dataset 2")
ax.set_xlabel("x")
ax.set_ylabel("y")
plt.legend()                                 # creates the legend
plt.show()
```

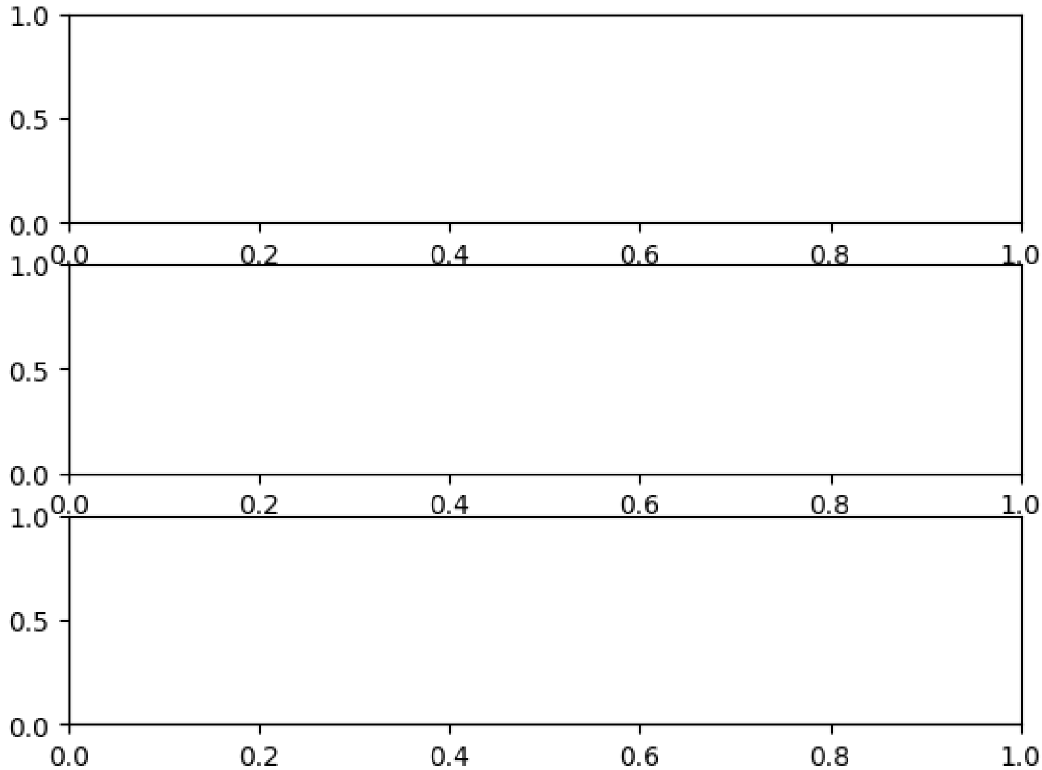


We can have as many subplots as we'd like:

[58]:
```
fig, (ax1, ax2) = plt.subplots(1, 2)
plt.show()
```

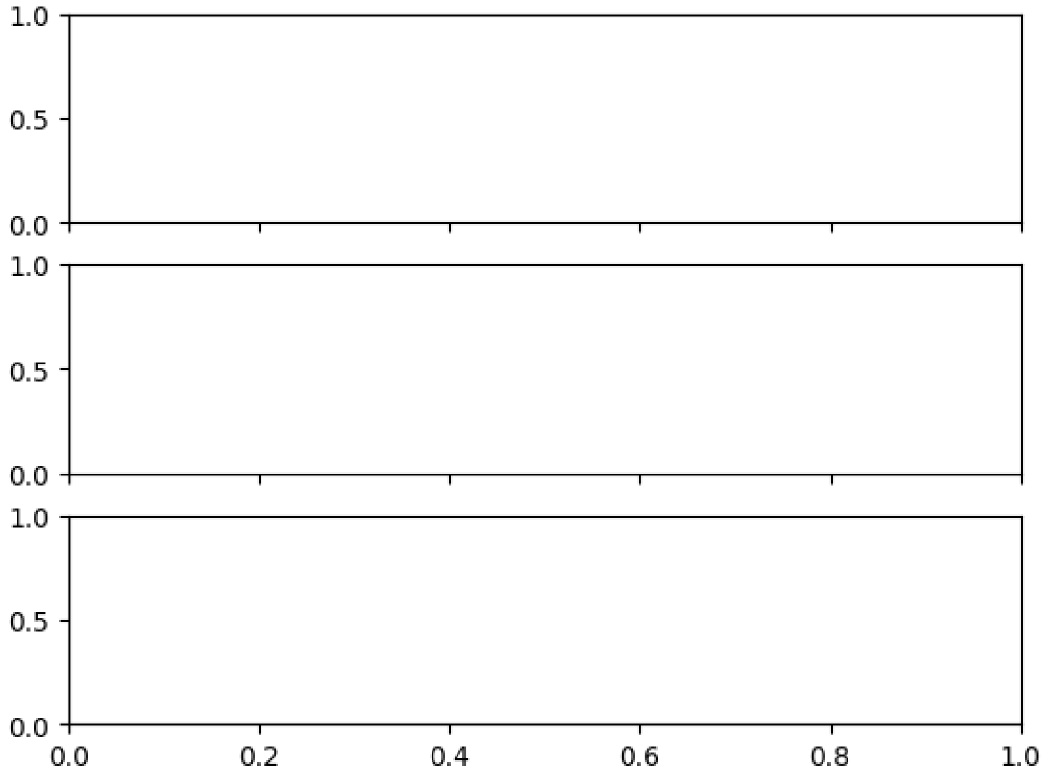


[59]:
```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1)
plt.show()
```



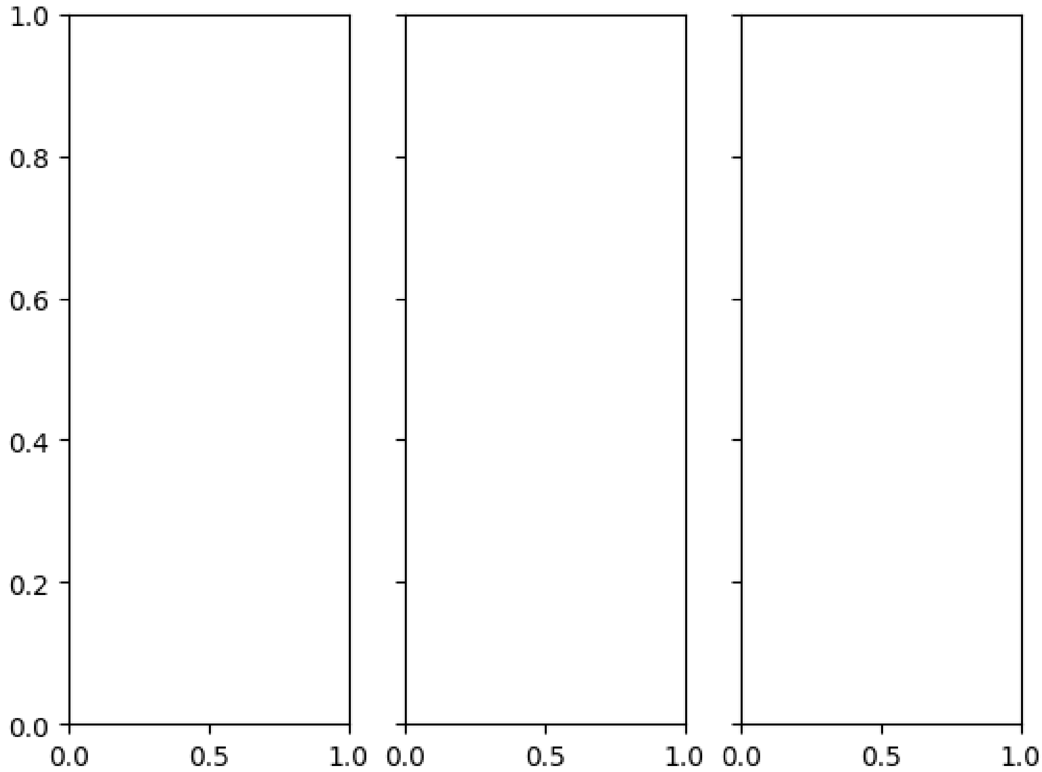
If we want our plots to share an axis, this can be done very easily

[60]:
```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, sharex=True)
plt.show()
```



[61]:

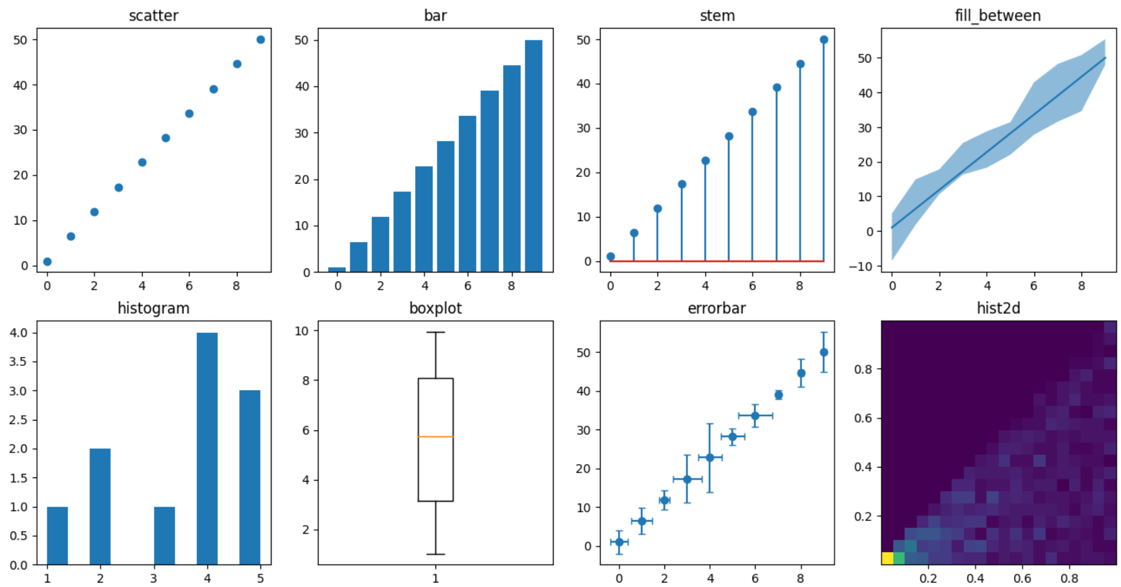
```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True)
plt.show()
```



There are many more types of plotting. Here are some examples of more plots:

[62]:
```
# generate some more dummy data for plotting
y_top = [i + np.random.uniform(10) for i in y]                           # this␣
 ↪data defines the top boundary of the fill_between plot
y_bottom = [i - np.random.uniform(10) for i in y]                        # this␣
 ↪data defines the bottom boundary of the fill_between plot
x_hist = [5, 5, 5, 3, 2, 2, 1, 4, 4, 4, 4]                               # this␣
 ↪data is for the histogram plot
x_box = np.random.uniform(1, 10, 100)                                    # this␣
 ↪data is for the boxplot plot
x_err = np.random.uniform(0, 1, 10)                                      # this␣
 ↪data is for the errorbar plot
y_err = np.random.uniform(1, 10, 10)                                     # this␣
 ↪data is for the errorbar plot
x_hist2d = np.random.rand(1000)                                          # this␣
 ↪data is for the hist2d plot
y_hist2d = x_hist2d*np.random.rand(1000)                                 # this␣
 ↪data is for the hist2d plot
```

```
fig, axs = plt.subplots(2, 4, figsize=(16,8))       # the figsize changes the size of the figure in the (x, y) direction

# first row, first plot
axs[0, 0].scatter(x, y)
axs[0, 0].set_title("scatter")

# first row, second plot
axs[0, 1].bar(x, y)
axs[0, 1].set_title("bar")

# first row, third plot
axs[0, 2].stem(x, y)
axs[0, 2].set_title("stem")

# first row, fourth plot
axs[0, 3].plot(x, y)
axs[0, 3].fill_between(x, y_top, y_bottom, alpha=0.5)
axs[0, 3].set_title("fill_between")

# second row, first plot
axs[1, 0].hist(x_hist)
axs[1, 0].set_title("histogram")

# second row, second plot
axs[1, 1].boxplot(x_box)
axs[1, 1].set_title("boxplot")

# second row, third plot
axs[1, 2].errorbar(x, y, y_err, x_err, fmt='o', capsize=3)   # fmt='o' set the markers to be points, and capsize=3 sets the length of the top of the errorbars
axs[1, 2].set_title("errorbar")

# second row, fourth plot
axs[1, 3].hist2d(x_hist2d, y_hist2d, bins=20)
axs[1, 3].set_title("hist2d")

plt.show()
```



1.0.5 Reading in data with pandas

Pandas is a useful library that makes working with datasets very easily. There are in-built functions for, e.g., analysing, cleaning, and manipulating data.

Let's begin by reading in a rainfall dataset from the Deutscher Wetterdienst. We can read in .csv and .txt files by using the function *read_csv*:

[63]:
```
data = pd.read_csv("produkt_nieder_tag_20230430_20241030_00006.txt", sep=';')
```

The *sep* keyword allows us to specify how the different columns of data are separated. In this dataset, each column is separated by a semicolon.

Let's have a look at the data we have read in below:

[64]:
```
data
```

[64]:

	STATIONS_ID	MESS_DATUM	QN_6	RS	RSF	SH_TAG	NSH_TAG	eor
0	6	20230430	3	0.0	6	0	0	eor
1	6	20230501	3	0.4	6	0	0	eor
2	6	20230502	3	0.9	6	0	0	eor
3	6	20230503	3	0.0	0	0	0	eor
4	6	20230504	3	0.0	0	0	0	eor
..	…	…	…	…	…	…	…	…
545	6	20241026	1	0.0	0	0	0	eor
546	6	20241027	1	0.0	0	0	0	eor
547	6	20241028	1	0.0	0	0	0	eor
548	6	20241029	1	0.0	0	0	0	eor
549	6	20241030	1	0.0	0	0	0	eor

```
[550 rows x 8 columns]
```

The various columns are:

- STATIONS_ID: the ID of the Deutscher Wetterdienst station
- MESS_DATUM: the reference date of the measurement
- QN_6: the type of the quality level check of the data
- RS: the daily precipitation depth in mm
- RSF: numerical code for the type of precipitation
- SH_TAG: the snow depth in cm
- NSH_TAG: the new snow depth in cm

If we only wanted to display the first 10 rows of the data, we can use *head*:

[65]:
```
data.head(6)
```

[65]:
```
   STATIONS_ID  MESS_DATUM  QN_6   RS  RSF  SH_TAG  NSH_TAG  eor
0            6    20230430     3  0.0    6       0        0  eor
1            6    20230501     3  0.4    6       0        0  eor
2            6    20230502     3  0.9    6       0        0  eor
3            6    20230503     3  0.0    0       0        0  eor
4            6    20230504     3  0.0    0       0        0  eor
5            6    20230505     3  8.1    6       0        0  eor
```

The last 5 rows using *tail*:

[66]:
```
data.tail(5)
```

[66]:
```
     STATIONS_ID  MESS_DATUM  QN_6   RS  RSF  SH_TAG  NSH_TAG  eor
545            6    20241026     1  0.0    0       0        0  eor
546            6    20241027     1  0.0    0       0        0  eor
547            6    20241028     1  0.0    0       0        0  eor
548            6    20241029     1  0.0    0       0        0  eor
549            6    20241030     1  0.0    0       0        0  eor
```

For a quick statistical summary of the data, you can use *describe*

[67]:
```
data.describe()
```

[67]:
```
       STATIONS_ID    MESS_DATUM        QN_6          RS         RSF  \
count        550.0  5.500000e+02  550.000000  550.000000  550.000000
mean           6.0  2.023623e+07    2.734545    3.006545    3.752727
std            0.0  4.835458e+03    0.679177    5.887286    3.007692
min            6.0  2.023043e+07    1.000000    0.000000    0.000000
25%            6.0  2.023091e+07    3.000000    0.000000    0.000000
50%            6.0  2.024013e+07    3.000000    0.100000    6.000000
75%            6.0  2.024061e+07    3.000000    3.600000    6.000000
max            6.0  2.024103e+07    3.000000   52.500000    8.000000
```

```
           SH_TAG     NSH_TAG
count  550.000000  550.000000
mean     0.112727    0.047273
std      0.964781    0.656717
min      0.000000    0.000000
25%      0.000000    0.000000
50%      0.000000    0.000000
75%      0.000000    0.000000
max     14.000000   14.000000
```

We can transpose the data by using $.T$:

[68]:
```
data.T
```

[68]:
```
                   0         1         2         3         4         5    \
STATIONS_ID        6         6         6         6         6         6
MESS_DATUM  20230430  20230501  20230502  20230503  20230504  20230505
QN_6               3         3         3         3         3         3
RS               0.0       0.4       0.9       0.0       0.0       8.1
RSF                6         6         6         0         0         6
SH_TAG             0         0         0         0         0         0
NSH_TAG            0         0         0         0         0         0
eor              eor       eor       eor       eor       eor       eor

                   6         7         8         9    …       540       541  \
STATIONS_ID        6         6         6         6    …         6         6
MESS_DATUM  20230506  20230507  20230508  20230509    …  20241021  20241022
QN_6               3         3         3         3    …         1         1
RS               0.0       4.9       1.0      17.7    …       0.0       1.0
RSF                0         6         6         6    …         6         6
SH_TAG             0         0         0         0    …         0         0
NSH_TAG            0         0         0         0    …         0         0
eor              eor       eor       eor       eor    …       eor       eor

                 542       543       544       545       546       547  \
STATIONS_ID        6         6         6         6         6         6
MESS_DATUM  20241023  20241024  20241025  20241026  20241027  20241028
QN_6               1         1         1         1         1         1
RS               0.0       0.0       0.0       0.0       0.0       0.0
RSF                6         0         0         0         0         0
SH_TAG             0         0         0         0         0         0
NSH_TAG            0         0         0         0         0         0
eor              eor       eor       eor       eor       eor       eor

                 548       549
STATIONS_ID        6         6
```

```
MESS_DATUM   20241029  20241030
QN_6                1         1
RS                0.0       0.0
RSF                 0         0
SH_TAG              0         0
NSH_TAG             0         0
eor               eor       eor

[8 rows x 550 columns]
```

Sort by values, e.g. by the rainfall amount from largest to smallest:

[69]:
```
data.sort_values(by='RS', ascending=False)
```

[69]:
```
     STATIONS_ID  MESS_DATUM  QN_6    RS  RSF  SH_TAG  NSH_TAG  eor
397            6    20240531     3  52.5    6       0        0  eor
398            6    20240601     3  43.2    6       0        0  eor
200            6    20231116     3  28.5    6       0        0  eor
96             6    20230804     3  27.0    6       0        0  eor
90             6    20230729     3  26.1    6       0        0  eor
..           …         …    …   …  …      …   …
231            6    20231217     3   0.0    0       0        0  eor
230            6    20231216     3   0.0    6       0        0  eor
217            6    20231203     3   0.0    0      11        1  eor
206            6    20231122     3   0.0    0       0        0  eor
549            6    20241030     1   0.0    0       0        0  eor

[550 rows x 8 columns]
```

The syntax of accessing data within pandas is incredibly similar to that of lists and arrays that we have seen earlier. To access by index, you need to use *iloc* along with the index number or slice. For example, if we wanted data from the first row of the dataset:

[70]:
```
data.iloc[0]
```

[70]:
```
STATIONS_ID          6
MESS_DATUM    20230430
QN_6                 3
RS                 0.0
RSF                  6
SH_TAG               0
NSH_TAG              0
eor                eor
Name: 0, dtype: object
```

Data from rows 7-10:

[71]:
```
data.iloc[6:10]
```

[71]:

	STATIONS_ID	MESS_DATUM	QN_6	RS	RSF	SH_TAG	NSH_TAG	eor
6	6	20230506	3	0.0	0	0	0	eor
7	6	20230507	3	4.9	6	0	0	eor
8	6	20230508	3	1.0	6	0	0	eor
9	6	20230509	3	17.7	6	0	0	eor

Data from column 4:

[72]:
```
data.iloc[:,3]
```

[72]:
```
0      0.0
1      0.4
2      0.9
3      0.0
4      0.0
      …
545    0.0
546    0.0
547    0.0
548    0.0
549    0.0
Name: RS, Length: 550, dtype: float64
```

One main difference is that we can also access points by column name. There are three methods to do so:

[73]:
```
data.loc[:,"MESS_DATUM"]
```

[73]:
```
0      20230430
1      20230501
2      20230502
3      20230503
4      20230504
         …
545    20241026
546    20241027
547    20241028
548    20241029
549    20241030
Name: MESS_DATUM, Length: 550, dtype: int64
```

[74]:
```
data["MESS_DATUM"]
```

[74]:
```
0      20230430
1      20230501
2      20230502
3      20230503
4      20230504
```

```
        …
545     20241026
546     20241027
547     20241028
548     20241029
549     20241030
Name: MESS_DATUM, Length: 550, dtype: int64
```

[75]:
```
data.MESS_DATUM
```

[75]:
```
0       20230430
1       20230501
2       20230502
3       20230503
4       20230504
        …
545     20241026
546     20241027
547     20241028
548     20241029
549     20241030
Name: MESS_DATUM, Length: 550, dtype: int64
```

These methods are equivalent, and depend on your preference.

If we wanted more than one column, e.g. date and rainfall, then there are only two methods to do so:

[76]:
```
data[["MESS_DATUM", "RS"]]
```

[76]:
```
     MESS_DATUM   RS
0      20230430  0.0
1      20230501  0.4
2      20230502  0.9
3      20230503  0.0
4      20230504  0.0
..          …    …
545    20241026  0.0
546    20241027  0.0
547    20241028  0.0
548    20241029  0.0
549    20241030  0.0

[550 rows x 2 columns]
```

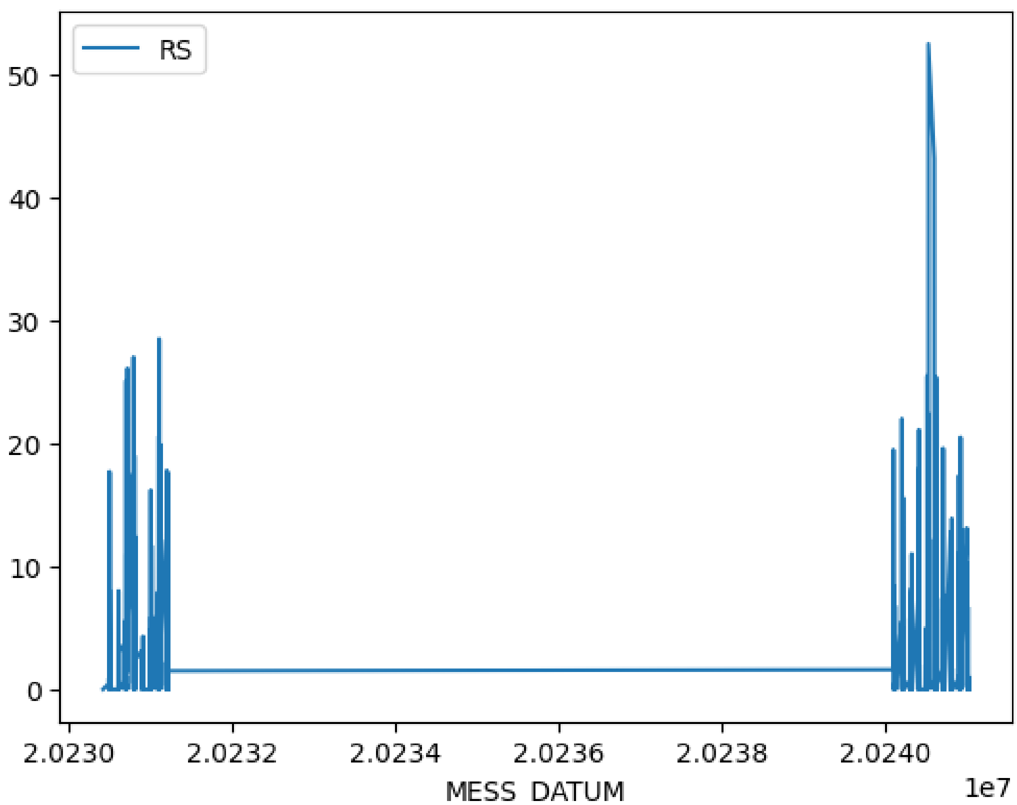
[77]:
```
data.loc[:,["MESS_DATUM", "RS"]]
```

[77]:
```
     MESS_DATUM   RS
0      20230430  0.0
1      20230501  0.4
2      20230502  0.9
3      20230503  0.0
4      20230504  0.0
..          ...  ...
545    20241026  0.0
546    20241027  0.0
547    20241028  0.0
548    20241029  0.0
549    20241030  0.0

[550 rows x 2 columns]
```

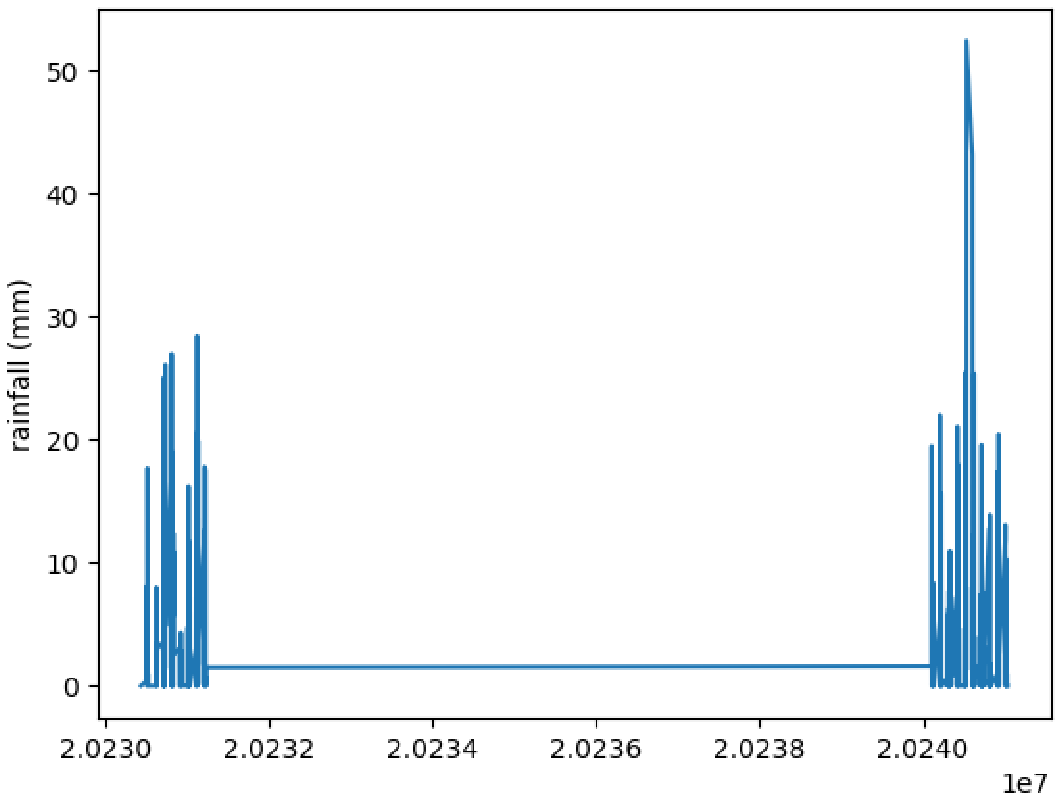
Let's create a plot of the rainfall data! You can use matplotlib.pyplot with pandas data super easily. The downside is that it is not as customisable.

[78]:
```
data.plot("MESS_DATUM", "RS")
```

[78]: <AxesSubplot: xlabel='MESS_DATUM'>



[79]:
```
fig, ax = plt.subplots()
plt.plot(data.MESS_DATUM, data.RS)
ax.set_ylabel("rainfall (mm)")
plt.show()
```



Notice how in these plots, our dates are in the wrong format. This can be easily fixed using the datetime library. We can install it by

[ ]:
```
%pip install datetime
```

and load it

[81]:
```
import datetime
```

Let's keep the same format of YYYY-MM-DD, but instead convert it into a datetime object. The most commonly used formatting codes are

- %Y for a 4-digit year
- %y for a 2-digit year
- %m for a 2-digit month
- %d for a 2-digit day
- %H for a 2-digit hour
- %M for a 2-digit minute

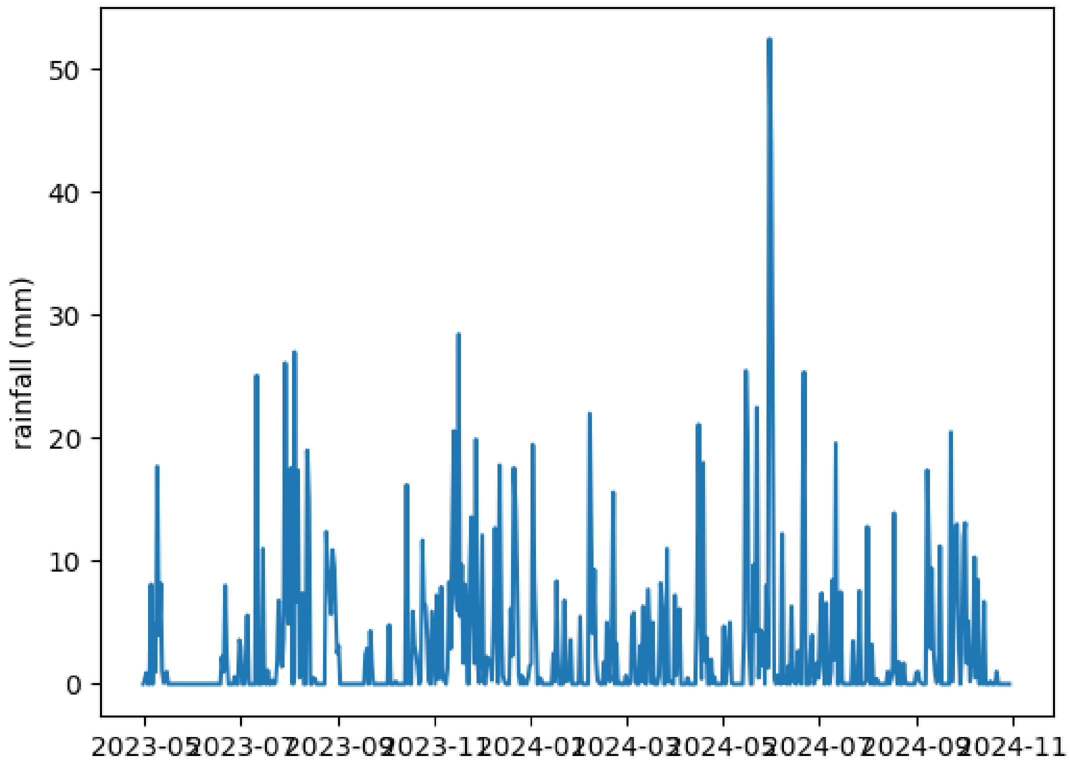
- %S for a 2-digit second

We can use the *datetime.strptime* function that takes a date represented as a string, and converts it into a datetime object. We can convert the data values into a string by using *str()*

[ ]:
```
date_format = '%Y%m%d'
dates = [datetime.datetime.strptime(str(i), date_format) for i in data.
  ↪MESS_DATUM]
```

Now we can replot the data:

[83]:
```
fig, ax = plt.subplots()
plt.plot(dates, data.RS)
ax.set_ylabel("rainfall (mm)")
plt.show()
```

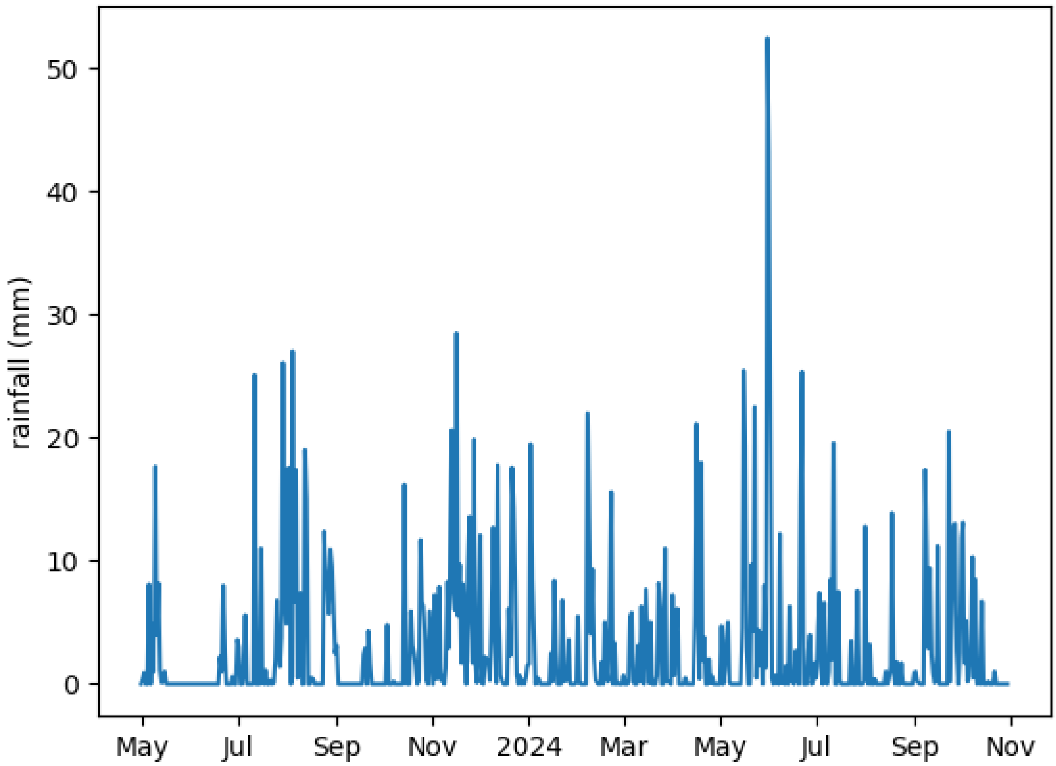


Notice how it's difficult to read the dates on the x-axis. There are two ways that we can make it more readable by using matplotlib's mdates:

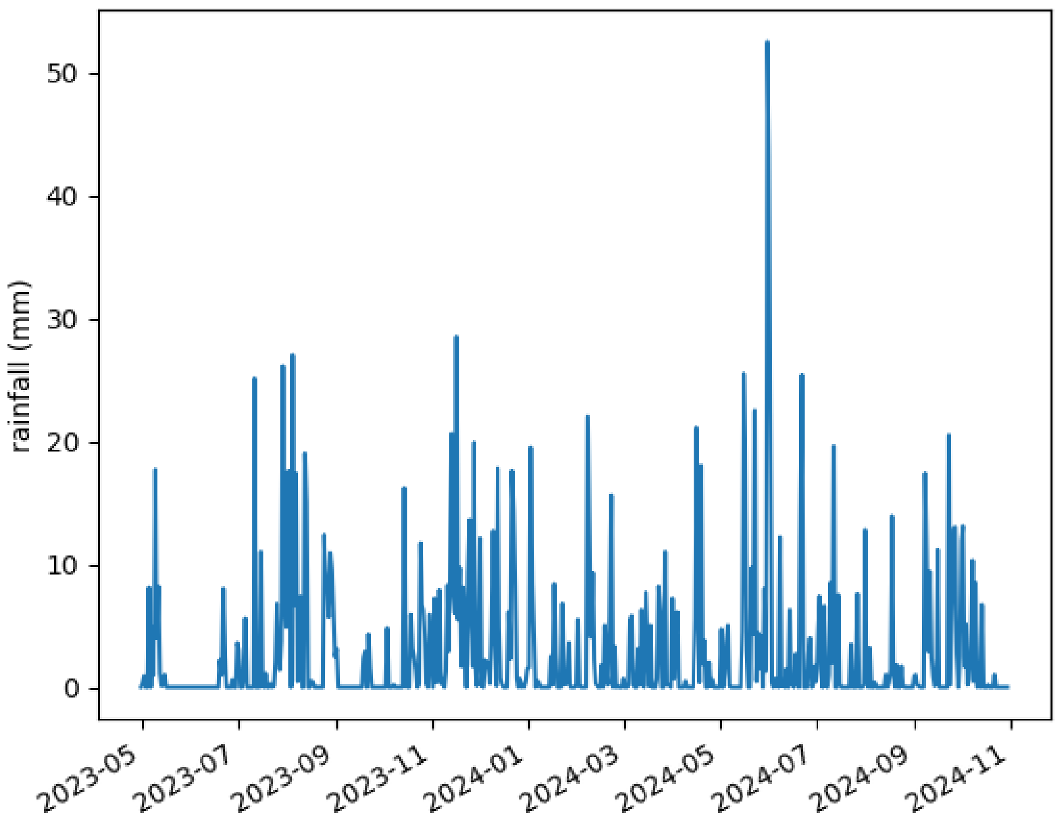
[84]:
```
import matplotlib.dates as mdates
```

[85]:
```
fig, ax = plt.subplots()
plt.plot(dates, data.RS)
```

```
ax.set_ylabel("rainfall (mm)")
ax.xaxis.set_major_formatter(mdates.ConciseDateFormatter(ax.xaxis.
    ↪get_major_locator()))
plt.show()
```



[86]:
```
fig, ax = plt.subplots()
plt.plot(dates, data.RS)
ax.set_ylabel("rainfall (mm)")
for label in ax.get_xticklabels(which='major'):
    label.set(rotation=30, horizontalalignment='right')
plt.show()
```



2 Exercises

2.0.1 Exercise 1

Recall that the normal distribution is defined as

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

Create your own function that returns the probability of a given value x for some input mean μ and standard deviation σ. Using this function, create some dummy x-data and plot the x-values vs. their probability.

[87]:
```
# your code here
```

2.0.2 Exercise 2

Using the Deutscher Wetterdienst data from earlier, plot the culmulative rainfall over the date range. What is the total amount of rain in mm over the date range?

[88]:
```
# your code here
```

2.0.3 Exercise 3

Using the Deutscher Wetterdienst data from earlier, plot the snow depth for December 2023, as well as a horizontal line indicating the mean snow depth for December.

[89]:
```
# your code here
```