Python for Hydrology

Session 3 – Scientific Computing

Objective:

Manipulate and use in a complementary way the 3 most used scientific libraries of Python, which are NumPy, Matplotlib, and Scipy,

NumPy

Start opening Jupyter Lab and creating a new folder called "Session3" inside it, add a new notebook with the same name.

Numpy has become a standard package for scientific programming in Python. It is based in the object "ndarray"

To start with NumPy, you need to import it: the typical way of importing Python is by calling it "**np**"

```
import numpy as np
```

To **create an array**, you can use the following command, it would return an array of the shape 4x1

```
a = np.array( (1, 2, 3, 4) )
a
```

```
a = np.array( (1, 2, 3, 4) )
a
```

```
array([1, 2, 3, 4])
```

It can also be defined with brackets

```
a = np.array( [1, 2, 3, 4] )
a
```

If you want to get the shape of the array, use "**shape**"

```
a.shape
```

```
a.shape
```

```
(4,)
```

To get the dimensions of the array, use "**ndim**"

```
a.ndim
```

```
a.ndim
```

```
1
```

You can also see the size using "**size**"

```
a.size
```

```
a.size
```

```
4
```

To get the type of the variables inside the array you can use "**dtype**"

```
a.dtype
```

```
a.dtype
```

```
dtype('int32')
```

A **2x2 array** would be of the following way

```
b = np.array( [[1.,2.], [3.,4.]] )
b
```

```
b = np.array( [[1.,2.], [3.,4.]] )
b
```

```
array([[1., 2.],
       [3., 4.]])
```

We can filter the values, similar to slicing in lists, for arrays, commas separate the dimensions if we want to get the value of the first row and the value of the second column.

```
b[0,1]
```

```
b[0,1]
```

```
2.0
```

We can define the **type** of the array with the declaration of the values

```
np.array( [0, 4, -4], dtype=complex)
```

```
np.array( [0, 4, -4], dtype=complex)
```

```
array([ 0.+0.j,  4.+0.j, -4.+0.j])
```

To create an empty array, use "**empty**". It returns an array with zeros and high values in scientific notation.

```
np.empty((3,3))
```

```
np.empty((3,3))
```

```
array([[0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       [0.00000000e+000, 0.00000000e+000, 6.52166653e-321],
       [1.14163374e-311, 1.14163374e-311, 0.00000000e+000]])
```

To create an array of zeros, we can use "**zeros**"

```
np.zeros((3,2))
```

```
np.zeros((3,2))
```

```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

We can also do the same for an array of ones, we use "**ones**" for this.

```
np.ones((3,3), dtype=float)
```

```
np.ones((3,3), dtype=float)
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

We can use the "**ones_like**" function to create an array of ones of the same shape of another array

```
np.ones_like(a)
```

```
np.ones_like(a)
```

```
array([1, 1, 1, 1])
```

In the same way, we can do this for an array of zeros. We use "**zeros_like**" for this.

```
np.zeros_like(a)
```

```
np.zeros_like(a)
```

```
array([0, 0, 0, 0])
```

To create a range of values, we can use "**arange**"

```
np.arange(7)
```

```
np.arange(7)
```

```
array([0, 1, 2, 3, 4, 5, 6])
```

The "arange" option can support the declaration of the starting, ending, and step values.

```
np.arange(1.5, 3., 0.5)
```

```
np.arange(1.5, 3., 0.5)
```

```
array([1.5, 2. , 2.5])
```

Similarly, we can use "**linspace**" to create a range, the difference with "arange" is that "linspace" take the number of values desired instead of the step.

```
np.linspace(1, 20, 5)
```

```
np.linspace(1, 20, 5)
```

```
array([ 1.  ,  5.75, 10.5 , 15.25, 20.  ])
```

The option "linspace" can return a second value regarding the step

```
x, dx = np.linspace(0., 2*np.pi, 100, retstep=True)
x
```

```
x, dx = np.linspace(0., 2*np.pi, 100, retstep=True)
x
```

```
array([0.        , 0.06346652, 0.12693304, 0.19039955, 0.25386607,
       0.31733259, 0.38079911, 0.44426563, 0.50773215, 0.57119866,
       0.63466518, 0.6981317 , 0.76159822, 0.82506474, 0.88853126,
       0.95199777, 1.01546429, 1.07893081, 1.14239733, 1.20586385,
       1.26933037, 1.33279688, 1.3962634 , 1.45972992, 1.52319644,
       1.58666296, 1.65012947, 1.71359599, 1.77706251, 1.84052903,
       1.90399555, 1.96746207, 2.03092858, 2.0943951 , 2.15786162,
       2.22132814, 2.28479466, 2.34826118, 2.41172769, 2.47519421,
       2.53866073, 2.60212725, 2.66559377, 2.72906028, 2.7925268 ,
       2.85599332, 2.91945984, 2.98292636, 3.04639288, 3.10985939,
       3.17332591, 3.23679243, 3.30025895, 3.36372547, 3.42719199,
       3.4906585 , 3.55412502, 3.61759154, 3.68105806, 3.74452458,
       3.8079911 , 3.87145761, 3.93492413, 3.99839065, 4.06185717,
       4.12532369, 4.1887902 , 4.25225672, 4.31572324, 4.37918976,
       4.44265628, 4.5061228 , 4.56958931, 4.63305583, 4.69652235,
       4.75998887, 4.82345539, 4.88692191, 4.95038842, 5.01385494,
       5.07732146, 5.14078798, 5.2042545 , 5.26772102, 5.33118753,
       5.39465405, 5.45812057, 5.52158709, 5.58505361, 5.64852012,
       5.71198664, 5.77545316, 5.83891968, 5.9023862 , 5.96585272,
       6.02931923, 6.09278575, 6.15625227, 6.21971879, 6.28318531])
```

```
dx
```

```
dx
```

```
0.06346651825433926
```

The tool "**fromfunction**" allows us to pass a function and the shape of an array. As a result, it returns an array of the shape mentioned, and the function takes as parameters the position in the array.

```
def f(i, j):
    return 2 * i * j
```

```
np.fromfunction(f,(4,3))
```

```
def f(i, j):
    return 2 * i * j

np.fromfunction(f,(4,3))
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  4.],
       [ 0.,  4.,  8.],
       [ 0.,  6., 12.]])
```

The tool "fromfunction" can take lambda functions too. A lambda function is as follows:

```
f = lambda x: x**2-3*x+2
print(f(4.))
```

```
f = lambda x: x**2-3*x+2
print(f(4.))
```

```
6.0
```

So coupling the same function used before, we get the following script:

```
np.fromfunction(lambda i,j: 2*i*j, (4,3))
```

```
np.fromfunction(lambda i,j: 2*i*j, (4,3))
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  4.],
       [ 0.,  4.,  8.],
       [ 0.,  6., 12.]])
```

An advantage of using NumPy is that we can easily declare **the type** in an easy way. NumPy supports lots of types that can be reviewed in the following link:

https://numpy.org/doc/stable/reference/arrays.dtypes.html

For example, if we want to declare a float of 16 bits

```
arr = np.array([1, 2, 3, 4], dtype='f2')
arr.dtype
```

```
arr = np.array([1, 2, 3, 4], dtype='f2')
arr.dtype
```

```
dtype('float16')
```

A float of 32 bits

```
arr = np.array([1, 2, 3, 4], dtype='f4')
arr.dtype
```

```
arr = np.array([1, 2, 3, 4], dtype='f4')
arr.dtype
```

```
dtype('float32')
```

And a float of 64 bits

```
arr = np.array([1, 2, 3, 4], dtype='f8')
arr.dtype
```

```
arr = np.array([1, 2, 3, 4], dtype='f8')
arr.dtype
```

```
dtype('float64')
```

We can also store the type in a variable, for example:

```
dt = np.dtype('f8')
dt
```

```
dt = np.dtype('f8')
dt
```

```
dtype('float64')
```

And we can use it later

```
a = np.array([0., 1., -2.], dtype=dt)
```

We can print its values as if it were a string, as default NumPy assign the numerical types as little-endian "<" (endian refer to how the value is read)

```
dt.str
```

```
dt.str
```

```
'<f8'
```

Another way of printing the type

```
dt.name
```

```
dt.name
```

```
'float64'
```

To get the size of the type:

```
dt.itemsize
```

```
dt.itemsize
```

```
8
```

If we want to, we can change the type of the array after it is declared

```
a.astype('float32')
```

```
a.astype('float32')
```

```
array([ 0.,  1., -2.], dtype=float32)
```

Or as an unsigned integer of 64 bits

```
a.astype(np.uint8)
```

```
a.astype(np.uint8)
```

```
array([  0,   1, 254], dtype=uint8)
```

Mathematical operations can be performed in arrays. They do not act as a matrix; the calculation is made one item by one

```
a = np.array( ((1,2), (3,4)) )
b = a
a*b
```

```
a = np.array( ((1,2), (3,4)) )
b = a
a*b
```

```
array([[ 1,  4],
       [ 9, 16]])
```

You can also make the operation directly to the assignation of a variable

```
a = np.linspace(1,6,6)**3
a
```

```
a = np.linspace(1,6,6)**3
a
```

```
array([  1.,   8.,  27.,  64., 125., 216.])
```

You can perform **conditional filters** that return Boolean values

```
a > 100
```

```
a > 100
```

```
array([False, False, False, False,  True,  True])
```

You can perform more elaborated filters too

```
(a < 10) | (a > 100)
```

```
(a < 10) | (a > 100)
```

```
array([ True,  True, False, False,  True,  True])
```

If we divide by 0 a value, we get a nan value. The nan values are useful when you want empty values.

```
a= np.arange(4)
a/a
```

```
a= np.arange(4)
a/a
```

```
<ipython-input-38-236fcd99ad92>:2: RuntimeWarning: invalid value encountered in true_divide
  a/a
array([nan,  1.,  1.,  1.])
```

We can find the **nan values** in an array

```
np.isnan(a/a)
```

```
np.isnan(a/a)
```

```
<ipython-input-39-01e6eb20ffaf>:1: RuntimeWarning: invalid value encountered in true_divide
  np.isnan(a/a)
array([ True, False, False, False])
```

We can modify the structure of the array, so having the following array:

```
a=np.array([[0,1],[2,3]])
a
```

```
a=np.array([[0,1],[2,3]])
a
```

```
array([[0, 1],
       [2, 3]])
```

We can transform into an array of one line with "**flatten**"

```
a.flatten()
```

```
a.flatten()
```

```
array([0, 1, 2, 3])
```

And you can resize any array with the "**resize**" tool

```
(a.flatten()).resize(2,2)
a
```

```
(a.flatten()).resize(2,2)
a
```

```
array([[0, 1],
       [2, 3]])
```

If you resize and the array has not the shape, NumPy automatically adjusts it.

```
np.resize(a,(2,3))
```

```
np.resize(a,(2,3))
```

```
array([[0, 1, 2],
       [3, 0, 1]])
```

You can transpose the array with "**transpose**"

```
a.transpose()
```

```
a.transpose()
```

```
array([[0, 2],
       [1, 3]])
```

You can filter the values of the array based on a step

```
a = np.linspace(1,6,6)
a[1:4:2]
```

```
a = np.linspace(1,6,6)
a[1:4:2]
```

```
array([1., 2., 3., 4., 5., 6.])
```

Another way of filtering when you want only a column of the array

```
a = np.linspace(1,12,12).reshape(4,3)
a[:,0]
```

```
a = np.linspace(1,12,12).reshape(4,3)
a[:,0]
```

```
array([ 1.,  4.,  7., 10.])
```

NumPy allows you to make some quick statistics of your data.

To do a mean, use "**mean**"

```
np.mean(a)
```

```
np.mean(a)
```

```
6.5
```

To calculate the median, use "**median**"

```
np.median(a)
```

```
np.median(a)
```

```
6.5
```

To calculate the standard deviation, use "**std**"

```
np.std(a)
```

```
np.std(a)
```

```
3.452052529534663
```

For the correlation coefficient, use "**corrcoef**"

```
x = np.array([1., 2., 3., 4., 5.])
y = np.array([0.08, 0.31, 0.41, 0.48, 0.62])

np.corrcoef(x,y)
```

```
x = np.array([1., 2., 3., 4., 5.])
y = np.array([0.08, 0.31, 0.41, 0.48, 0.62])

np.corrcoef(x,y)
```

```
array([[1.        , 0.97787645],
       [0.97787645, 1.        ]])
```

Matplotlib

To use Matplotlib, start **importing** the library

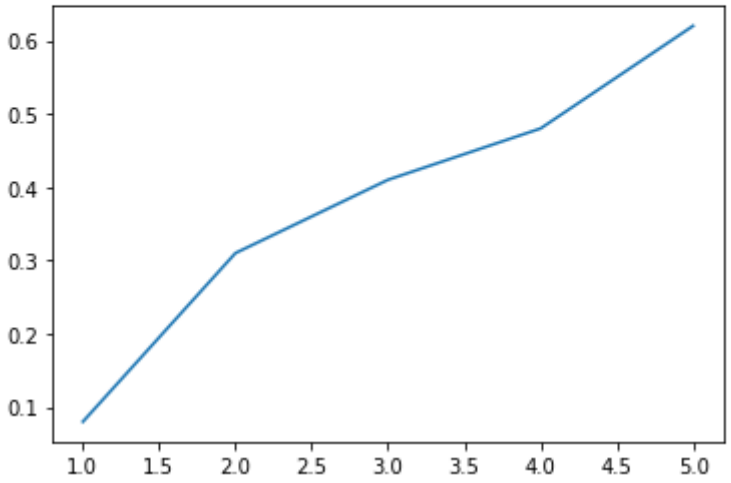
```
import matplotlib.pyplot as plt
```

Arrays can be used to plot, a **simple plot** would be as follows:

```
x = np.array([1., 2., 3., 4., 5.])
y = np.array([0.08, 0.31, 0.41, 0.48, 0.62])

plt.plot(x,y)
```

```
x = np.array([1., 2., 3., 4., 5.])
y = np.array([0.08, 0.31, 0.41, 0.48, 0.62])

plt.plot(x,y)
```

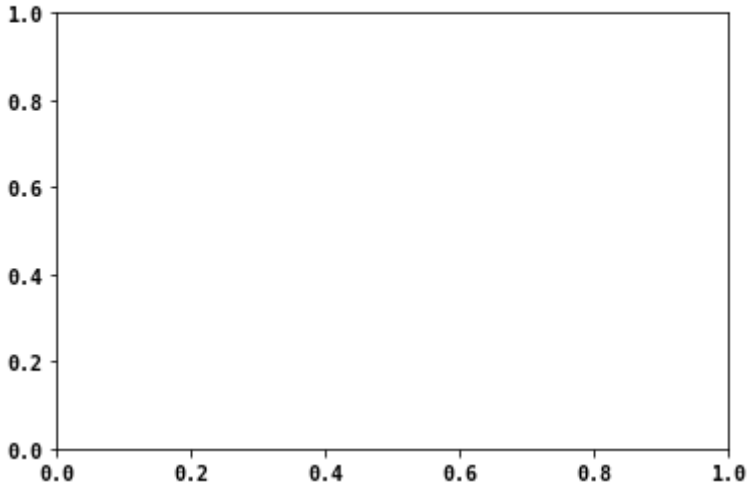
```
[<matplotlib.lines.Line2D at 0x21a32f41040>]
```



But the previous plot has not a good organization, a better way of plotting is using "**figs**"

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

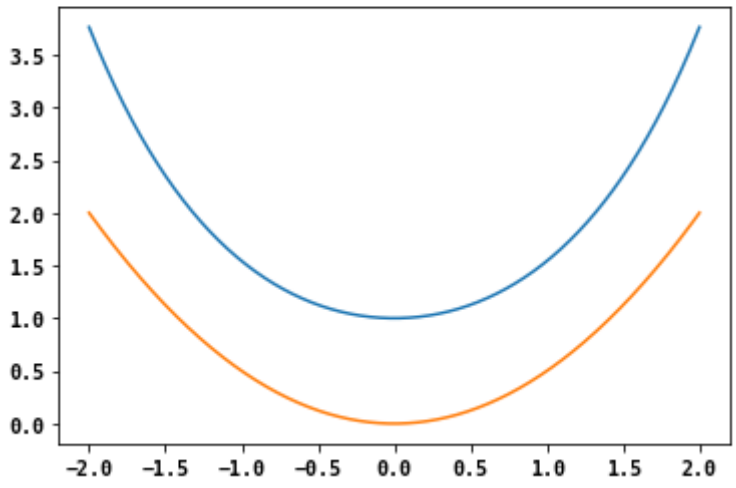
```
fig, ax = plt.subplots()
```



Creating figures, we use "**axis**" as plotting object

```
fig, ax = plt.subplots()
x = np.linspace(-2, 2, 1000)
ax.plot(x, np.cosh(x))
ax.plot(x, x**2 / 2)
plt.show()
```

```
fig, ax = plt.subplots()
x = np.linspace(-2, 2, 1000)
ax.plot(x, np.cosh(x))
ax.plot(x, x**2 / 2)
plt.show()
```

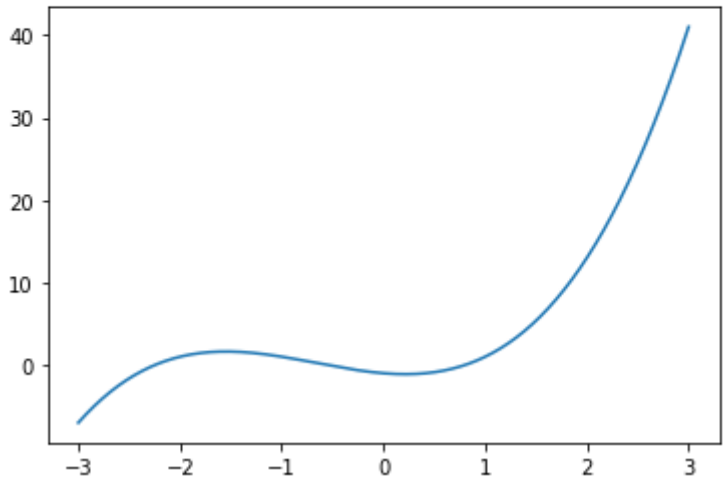


Another simple way of structuring our plot:

```
x = np.linspace(-3,3,1000)
y = x**3 + 2 * x**2 - x - 1
fig, ax = plt.subplots()
ax.plot(x,y)
```

```
x = np.linspace(-3,3,1000)
y = x**3 + 2 * x**2 - x - 1
fig, ax = plt.subplots()
ax.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x21a3e216520>]
```

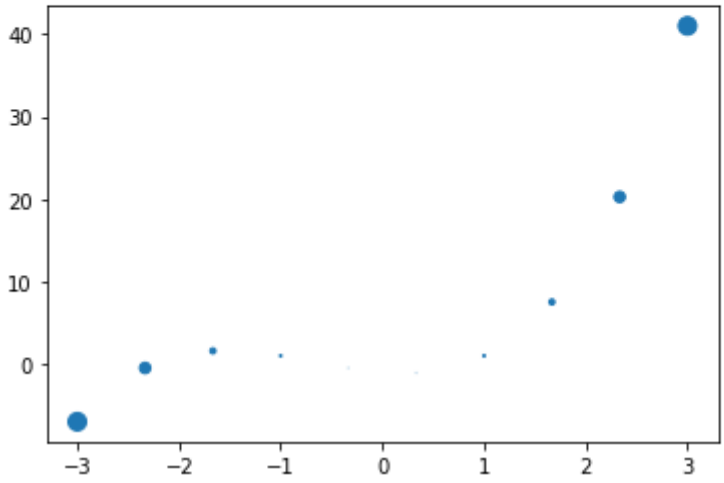


For a **scatter** plot, we can specify an extra parameter that corresponds to the size of the point plotted.

```
x = np.linspace(-3,3,10)
y = x**3 + 2 * x**2 - x - 1
z = x**4
fig, ax = plt.subplots()
ax.scatter(x,y,s=z)
```

```
x = np.linspace(-3,3,10)
y = x**3 + 2 * x**2 - x - 1
z = x**4
fig, ax = plt.subplots()
ax.scatter(x,y,s=z)
```

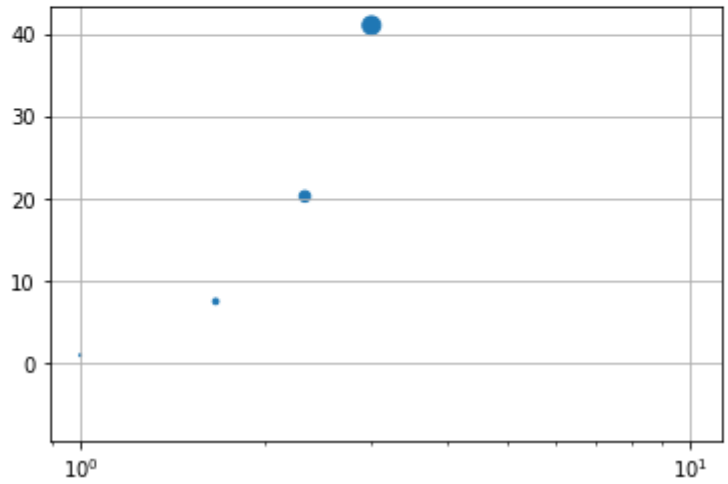
```
<matplotlib.collections.PathCollection at 0x21a3e264f70>
```



You can add the grids with the "**yaxis**" and "**xaxis**" properties, moreover we can adjust the axis to transform it to a log scale with "**set_xscale**" or "**set_yscale**"

```
x = np.linspace(-3,3,10)
y = x**3 + 2 * x**2 - x - 1
z = x**4
fig, ax = plt.subplots()
ax.scatter(x,y,s=z)
ax.yaxis.grid(True)
ax.xaxis.grid(True)
ax.set_xscale('log')
```

```
x = np.linspace(-3,3,10)
y = x**3 + 2 * x**2 - x - 1
z = x**4
fig, ax = plt.subplots()
ax.scatter(x,y,s=z)
ax.yaxis.grid(True)
ax.xaxis.grid(True)
ax.set_xscale('log')
```



In the folder of the session 3 you can find a folder called "**Data**" that contains a file called "**temp.txt**", NumPy has the possibility of loading some kind of formatted files.

```
Min and Max values of temperature in celcius degrees
Source: Hatari
Month    Min      Max
1        3.89     9.81
2        5.68     12.54
3        7.79     15.39
4        11.21    19.47
5        14.10    24.65
6        18.52    28.37
7        20.79    29.17
8        21.89    32.28
9        17.92    26.47
10       14.84    21.45
11       8.51     16.31
12       6.55     13.13
```

To open the file, we use "**loadtxt**"

```
data = np.loadtxt('temp.txt', unpack=True,skiprows=3)
data
```

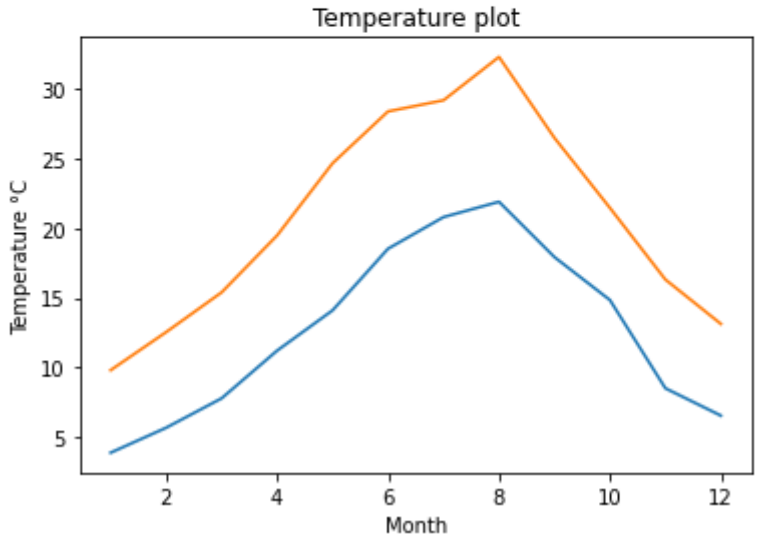
```
data = np.loadtxt('temp.txt', unpack=True,skiprows=3)
data
```

```
array([[ 1.  ,  2.  ,  3.  ,  4.  ,  5.  ,  6.  ,  7.  ,  8.  ,  9.  ,
        10.  , 11.  , 12.  ],
       [ 3.89,  5.68,  7.79, 11.21, 14.1 , 18.52, 20.79, 21.89, 17.92,
        14.84,  8.51,  6.55],
       [ 9.81, 12.54, 15.39, 19.47, 24.65, 28.37, 29.17, 32.28, 26.47,
        21.45, 16.31, 13.13]])
```

With the data loaded to the variable "data" we can make a plot of the values, and add labels to the plot

```
data = np.loadtxt('temp.txt', unpack=True,skiprows=3)
fig, ax = plt.subplots()
ax.plot(data[0,:], data[1,:])
ax.plot(data[0,:], data[2,:])
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
plt.show()
```

```
data = np.loadtxt('temp.txt', unpack=True,skiprows=3)
fig, ax = plt.subplots()
ax.plot(data[0,:], data[1,:])
ax.plot(data[0,:], data[2,:])
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
plt.show()
```



We can use the parameter "**label**" inside the plot and enable the legend feature in the axis to have a **legend**

```
data = np.loadtxt('temp.txt', unpack=True,skiprows=3)
fig, ax = plt.subplots()
ax.plot(data[0,:], data[1,:], label='min T')
ax.plot(data[0,:], data[2,:], label='max T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
plt.show()
```

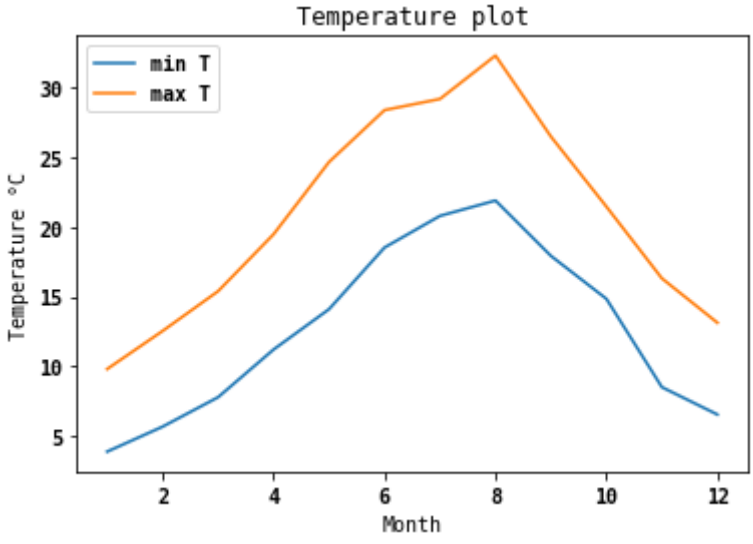
We can use the function "**rc**" to change the font properties of all the graph, the properties can be stored in a variable, for example:

```
from matplotlib import rc
```

```
font_properties = {'family' : 'monospace',
```

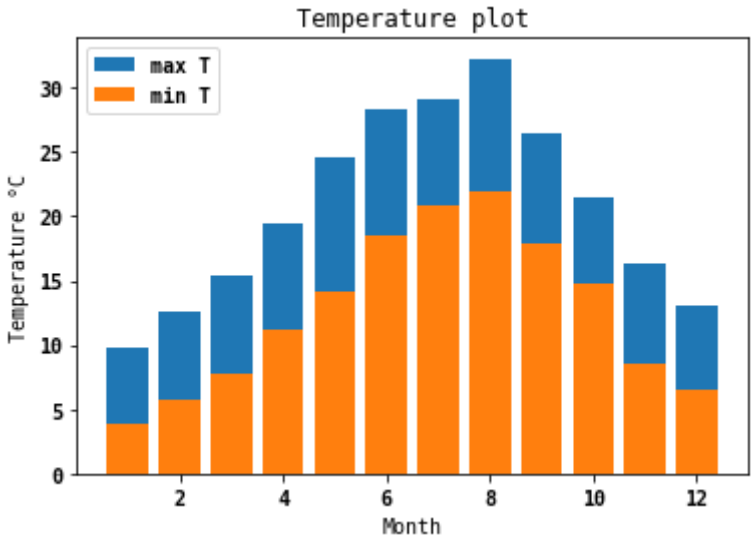
```
'weight' : 'bold',
'size' : 10}
rc('font', **font_properties)data = np.loadtxt('temp.txt',
unpack=True,skiprows=3)
fig, ax = plt.subplots()
ax.plot(data[0,:], data[1,:], label='min T')
ax.plot(data[0,:], data[2,:], label='max T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
plt.show()
```

```
from matplotlib import rc
font_properties = {'family' : 'monospace',
'weight' : 'bold',
'size' : 10}
rc('font', **font_properties)
data = np.loadtxt('temp.txt', unpack=True,skiprows=3)
fig, ax = plt.subplots()
ax.plot(data[0,:], data[1,:], label='min T')
ax.plot(data[0,:], data[2,:], label='max T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
plt.show()
```



We can make a bar-plot, to do this, instead of using "plot" we must use "bar"

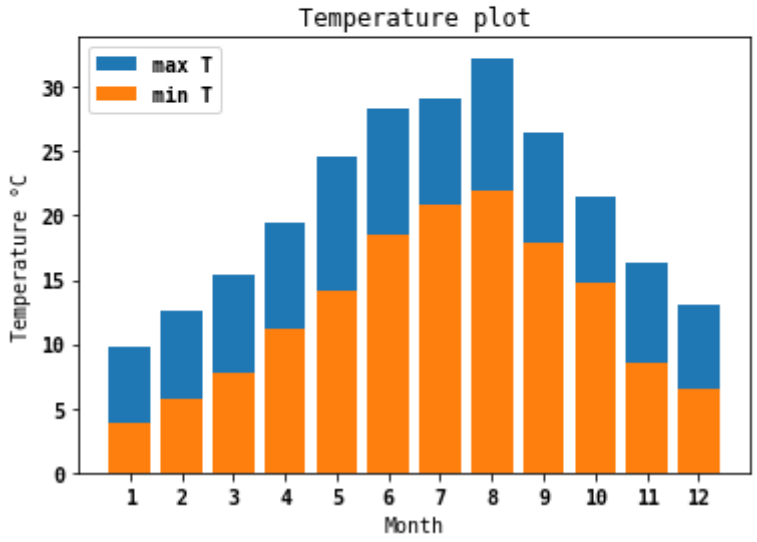
```
fig, ax = plt.subplots()
ax.bar(data[0,:], data[1,:], label='min T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
plt.show()
```



We can modify the X-axis label to see the complete range of months. To do this, we use "**set_xticks**" and we pass a range through it.

```
fig, ax = plt.subplots()
ax.bar(data[0,:], data[2,:], label='max T')
ax.bar(data[0,:], data[1,:], label='min T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
ax.set_xticks(np.arange(1,13))
plt.show()
```

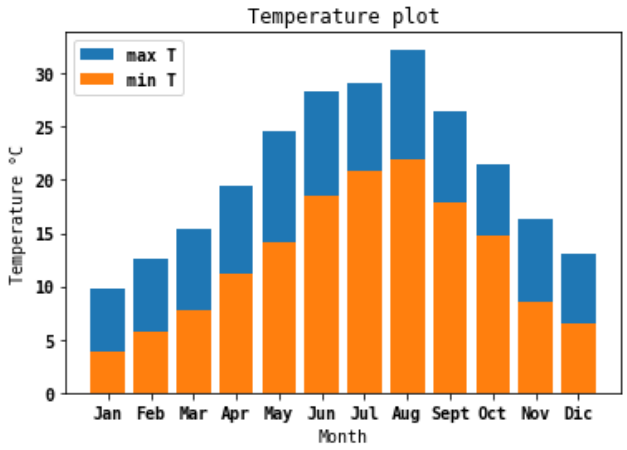
```
fig, ax = plt.subplots()
ax.bar(data[0,:], data[2,:], label='max T')
ax.bar(data[0,:], data[1,:], label='min T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
ax.set_xticks(np.arange(1,13))
plt.show()
```



If you want to change the labels, you need to use "**set_xticklabels**"

```
font_properties = {'family' : 'monospace',
'weight' : 'bold',
'size' : 10}
rc('font', **font_properties)
data = np.loadtxt('temp.txt', unpack=True,skiprows=3)
fig, ax = plt.subplots()
ax.bar(data[0,:], data[2,:], label='max T')
ax.bar(data[0,:], data[1,:], label='min T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
ax.set_xticks(np.arange(1,13))
ax.set_xticklabels(['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','S
ept','Oct','Nov','Dic'])
plt.show()
```

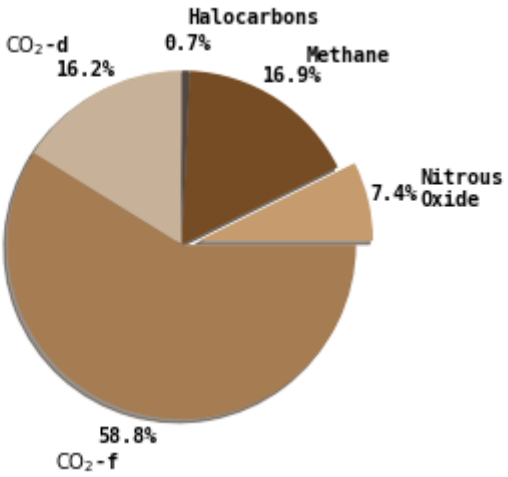
```
font_properties = {'family' : 'monospace',
'weight' : 'bold',
'size' : 10}
rc('font', **font_properties)
data = np.loadtxt('temp.txt', unpack=True,skiprows=3)
fig, ax = plt.subplots()
ax.bar(data[0,:], data[2,:], label='max T')
ax.bar(data[0,:], data[1,:], label='min T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
ax.set_xticks(np.arange(1,13))
ax.set_xticklabels(['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sept','Oct','Nov','Dic'])
plt.show()
```



In the same way, we can make a **pie-plot**, we can pass an array with one column of strings as keys and one column of data, we can add colors a explode option to highlight some specific value. This way of using arrays with keys is useful for organizing your data.

```
gas_emissions = np.array([(r'$\mathrm{CO_2}$-d', 2.2),
                          (r'$\mathrm{CO_2}$-f', 8.0),
                          ('Nitrous\nOxide', 1.0),
                          ('Methane', 2.3),
                          ('Halocarbons', 0.1)],
dtype=[('source', 'U17'), ('emission', 'f4')])
colors = ['#C7B299', '#A67C52', '#C69C6E', '#754C24', '#534741']
explode = [0, 0, 0.1, 0, 0]
fig, ax = plt.subplots()
ax.axis('equal') # So our pie looks round!
ax.pie(gas_emissions['emission'], colors=colors, shadow=True,
startangle=90,
 explode=explode, labels=gas_emissions['source'], autopct='%.1f%%',
pctdistance=1.15, labeldistance=1.3)
plt.show()
```

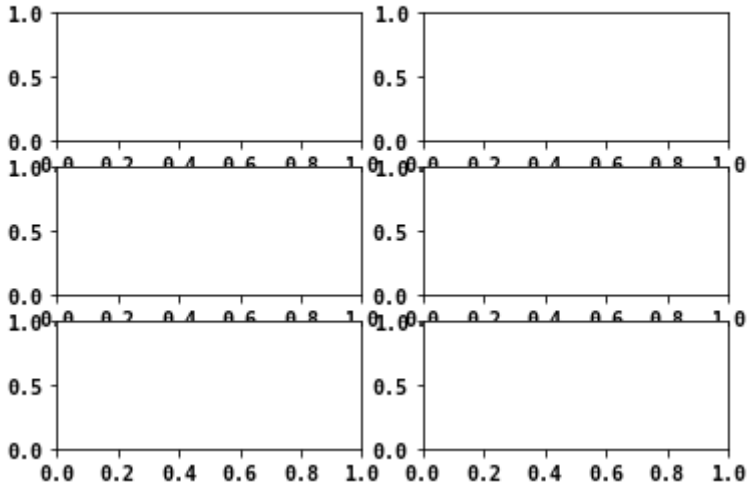
```
gas_emissions = np.array([(r'$\mathrm{CO_2}$-d', 2.2),
                          (r'$\mathrm{CO_2}$-f', 8.0),
                          ('Nitrous\nOxide', 1.0),
                          ('Methane', 2.3),
                          ('Halocarbons', 0.1)],
dtype=[('source', 'U17'), ('emission', 'f4')])
colors = ['#C7B299', '#A67C52', '#C69C6E', '#754C24', '#534741']
explode = [0, 0, 0.1, 0, 0]
fig, ax = plt.subplots()
ax.axis('equal') # So our pie looks round!
ax.pie(gas_emissions['emission'], colors=colors, shadow=True, startangle=90,
 explode=explode, labels=gas_emissions['source'], autopct='%.1f%%',
pctdistance=1.15, labeldistance=1.3)
plt.show()
```



If we want to create more **sub axes**, we need to define the number of rows and cols of the plot, the basic structure is as follows:

```
fig, axes = plt.subplots(nrows=3, ncols=2)
```

```
fig, axes = plt.subplots(nrows=3, ncols=2)
```

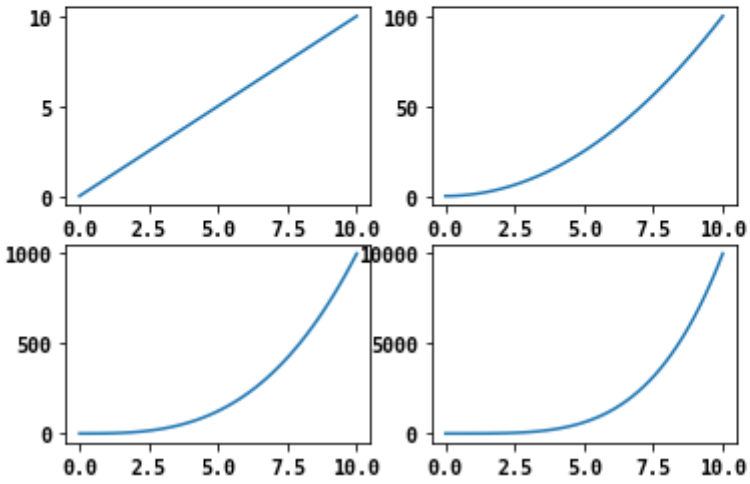


When you make the plot, you need to define the same structure of the rows and cols defined with "axes"

```
x = np.linspace(0,10)
fig, ((ax1,ax2),(ax3,ax4)) = plt.subplots(nrows=2, ncols=2)
ax1.plot(x,x)
ax2.plot(x,x**2)
ax3.plot(x,x**3)
ax4.plot(x,x**4)
```

```
x = np.linspace(0,10)
fig, ((ax1,ax2),(ax3,ax4)) = plt.subplots(nrows=2, ncols=2)
ax1.plot(x,x)
ax2.plot(x,x**2)
ax3.plot(x,x**3)
ax4.plot(x,x**4)
```

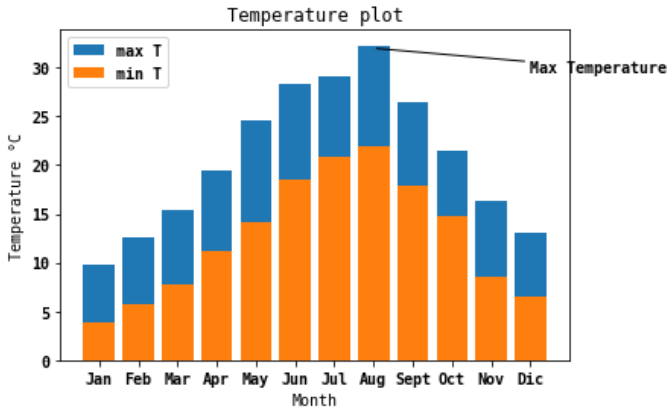
```
[<matplotlib.lines.Line2D at 0x21a3f795f10>]
```



To create an arrow annotation, you can use "**annotate.**", you can add it properties of an arrow too.

```
fig, ax = plt.subplots()
ax.bar(data[0,:], data[2,:], label='max T')
ax.bar(data[0,:], data[1,:], label='min T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
ax.set_xticks(np.arange(1,13))
ax.set_xticklabels(['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','S
ept','Oct','Nov','Dic'])
ax.annotate('Max Temperature', xy=(8,32),
xytext=(12,30),arrowprops={'arrowstyle': '-'}, va='center')
plt.show()
```

```
fig, ax = plt.subplots()
ax.bar(data[0,:], data[2,:], label='max T')
ax.bar(data[0,:], data[1,:], label='min T')
ax.set_xlabel('Month')
ax.set_ylabel('Temperature °C')
ax.set_title('Temperature plot')
ax.legend(loc='upper left')
ax.set_xticks(np.arange(1,13))
ax.set_xticklabels(['Jan','Feb','Mar','Apr','May','Jun','Jul','Aug','Sept','Oct','Nov','Dic'])
ax.annotate('Max Temperature', xy=(8,32), xytext=(12,30),arrowprops={'arrowstyle': '-'}, va='center')
plt.show()
```

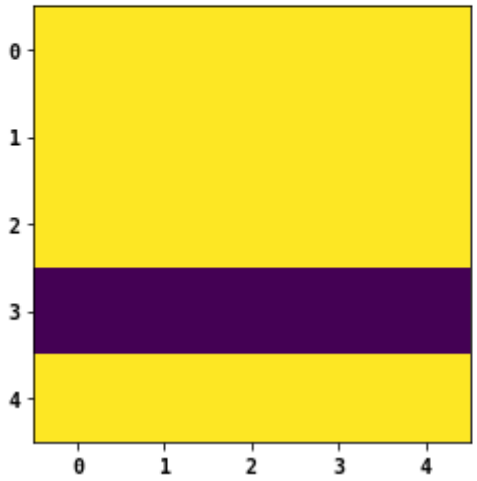


Another useful tool is the ability to plot arrays. This allows you to track the distribution of the array when you make changes in it.

```
a = np.ones((5,5))
a[3,:] = 0
plt.imshow(a)
```

```
a = np.ones((5,5))
a[3,:] = 0
plt.imshow(a)
```

```
<matplotlib.image.AxesImage at 0x21a3f8c8d90>
```



Scipy

SciPy has many tools; most of the tools are imported differently. For example, SciPy has compiled many **constants**. To use them, import constants as follow:

```
import scipy.constants as pc
```

If we want to get the **Avogadro constant**

```
pc.physical_constants['Avogadro constant']
```

```
pc.physical_constants['Avogadro constant']
```

```
(6.02214076e+23, 'mol^-1', 0.0)
```

For the **gas constant**

```
pc.value('molar gas constant')
```

```
pc.value('molar gas constant')
```

```
8.314462618
```

You can import the previous constant directly into a variable

```
from scipy.constants import R
R
```

```
from scipy.constants import R

R
```

```
8.314462618
```

Some constants are included as direct functions in Scipy.

```
pc.atm  #atm in Pascals
```

```
pc.atm  #atm in Pascals
```

```
101325.0
```

Scipy has tools to integrate functions; most of these tools are coupled from Fortran libraries.

For example, a **quadratic integration** can be solved importing "quad":

```
from scipy.integrate import quad
```

Given the following equation:

$$\int_1^4 x^{-2}\,dx = \frac{3}{4}$$

We would like to solve it with quadratic integration. First, we need to create the function related to that integral, then run "quad" with its limits. The result is the integral of the equation and a error associated with the calulation

```
f = lambda x: 1/x**2
quad(f, 1, 4)
```

```
f = lambda x: 1/x**2 #ITEGRAL DE X**-2 DX
quad(f, 1, 4) #error
```

```
(0.7500000000000002, 1.913234548258993e-09)
```

Another example of using "quad"

```
quad(lambda x: np.exp(-x**2), 0, np.inf)
```

```
quad(lambda x: np.exp(-x**2), 0, np.inf)
```

```
(0.8862269254527579, 7.101318378329813e-09)
```

Scipy has tools for solving **Ordinary Differential Equations (ODE)**. To calculate ODEs, we need to import "odeint" which is based in the LSODA package of Fortran, this is not the only method, but it is one of the most used.

```
from scipy.integrate import odeint
```

We are going to solve a first-order reaction rate equation:

$$\frac{\mathrm{d}[\mathrm{A}]}{\mathrm{d}t} = -k[\mathrm{A}].$$

The value of the first-order reaction "k" would be equal to 0.2, and the starting concentration of "A" equal to 100; we also need an array with the time "t"

```
k = 0.2
y0 = 100
t = np.linspace(0, 20, 20)
```

"odeint" requires a function to solve the ODE, the function for solving the equation would be as follows:

```
def dydt(y, t):
    return -k * y
```

So coupling all the previous script and adding the "odeint" function, we get the following script:

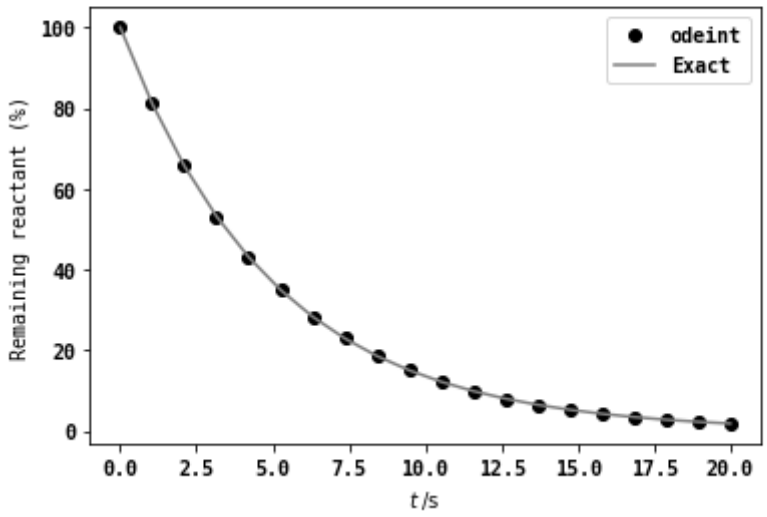
```
from scipy.integrate import odeint
k = 0.2
y0 = 100
```

```
t = np.linspace(0, 20, 20)
def dydt(y, t):
    return -k * y
y = odeint(dydt, y0, t)
```

We can make a plot of the result, comparing the exact solution and the solution related to "odeint"

```
fig, ax = plt.subplots()
ax.plot(t, y, 'o', color='k', label=r'odeint')
ax.plot(t, y0 * np.exp(-k*t), color='gray', label='Exact')
ax.set_xlabel(r'$t\;/\mathrm{s}$')
ax.set_ylabel('Remaining reactant (%)')
ax.legend()
plt.show()
```

```
fig, ax = plt.subplots()
ax.plot(t, y, 'o', color='k', label=r'odeint')
ax.plot(t, y0 * np.exp(-k*t), color='gray', label='Exact')
ax.set_xlabel(r'$t\;/\mathrm{s}$')
ax.set_ylabel('Remaining reactant (%)')
ax.legend()
plt.show()
```



A more complicated problem would be a reaction in which we have that "P" is produced by "B" and "B" is produced by "A"

$$A \rightarrow B \rightarrow P$$

The previous reactions are defined with the following equation:

$$\frac{d[A]}{dt} = -k_1[A]$$
$$\frac{d[B]}{dt} = k_1[A] - k_2[B]$$

Similarly, we input the reaction constants, the initial values, and the array with the times.

```
k1, k2 = 0.2, 0.8
A0, B0 = 100, 0
t = np.linspace(0, 20, 100)
```

Then we create the function:

```
def dydt(y, t, k1, k2):
    y1, y2 = y
    dy1dt = -k1 * y1
    dy2dt = k1 * y1 - k2 * y2
    return dy1dt, dy2dt
```

As in the function we have "Y" with two values, we lump together the initial concentrations

```
y0 = A0, B0
```

So, till this point, we have the following script, in which we calculate "P" as the product of the decomposition of "A" and "B"

```
k1, k2 = 0.2, 0.8
A0, B0 = 100, 0
t = np.linspace(0, 20, 100)
def dydt(y, t, k1, k2):
    y1, y2 = y
    dy1dt = -k1 * y1
```

```
    dy2dt = k1 * y1 - k2 * y2
    return dy1dt, dy2dt
y0 = A0, B0
y1, y2 = odeint(dydt, y0, t, args=(k1, k2)).T
A, B = y1, y2
P = A0 - A - B
```

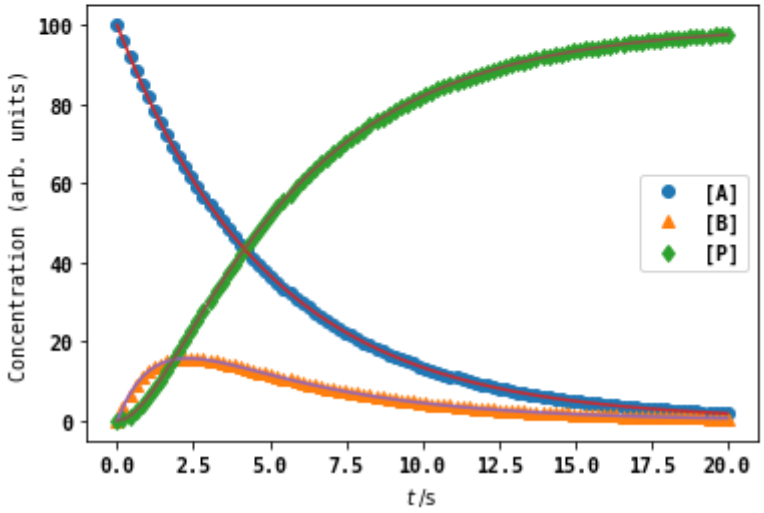
An analytical solution for this would be:

```
# Analytical result
Aexact = A0 * np.exp(-k1*t)
Bexact = A0 * k1/(k2-k1) * (np.exp(-k1*t) - np.exp(-k2*t))
Pexact = A0 - Aexact - Bexact
```

And if we plot to compare we get the following graph:

```
fig, ax = plt.subplots()
ax.plot(t, A, 'o', label='[A]')
ax.plot(t, B, '^', label='[B]')
ax.plot(t, P, 'd', label='[P]')
ax.plot(t, Aexact)
ax.plot(t, Bexact)
ax.plot(t, Pexact)
ax.set_xlabel(r'$t\;/\mathrm{s}$')
ax.set_ylabel('Concentration (arb. units)')
ax.legend()
plt.show()
```

```
fig, ax = plt.subplots()
ax.plot(t, A, 'o', label='[A]')
ax.plot(t, B, '^', label='[B]')
ax.plot(t, P, 'd', label='[P]')
ax.plot(t, Aexact)
ax.plot(t, Bexact)
ax.plot(t, Pexact)
ax.set_xlabel(r'$t\;/\mathrm{s}$')
ax.set_ylabel('Concentration (arb. units)')
ax.legend()
plt.show()
```



Another functionality that Scipy has is the interpolation.

Let's import the **interpolation** tool "interp1d"

```
from scipy.interpolate import interp1d
```

For the demonstration of interpolation, we follow these steps:

- Create a function with the constants "A", "nu", and "k"
- We create a range of values "x"
- We solve the function with the previous range and store the results in the variable "y"
- We solve the interpolation with the nearest neighbor method "f_nearest", linear interpolation "f_linear" and cubic interpolation "f_cubic"

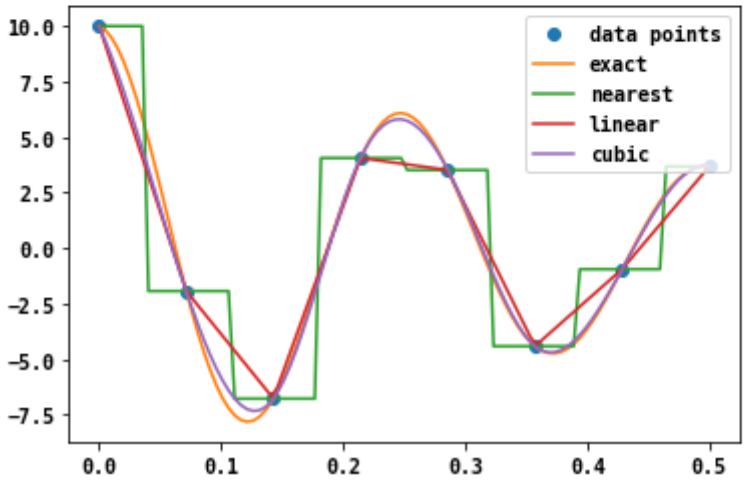
```
A, nu, k = 10, 4, 2
def f(x, A, nu, k):
```

```
    return A * np.exp(-k*x) * np.cos(2*np.pi * nu * x)
xmax, nx = 0.5, 8
x = np.linspace(0, xmax, nx)
y = f(x, A, nu, k)
f_nearest = interp1d(x, y, kind='nearest')
f_linear = interp1d(x, y)
f_cubic = interp1d(x, y, kind='cubic')
```

If we do the previous step, we get an object from the interpolations, so we can create a more discrete range of values and plot them based on that range as follows:

```
x2 = np.linspace(0, xmax, 100)

fig, ax = plt.subplots()
ax.plot(x, y, 'o', label='data points')
ax.plot(x2, f(x2, A, nu, k), label='exact')
ax.plot(x2, f_nearest(x2), label='nearest')
ax.plot(x2, f_linear(x2), label='linear')
ax.plot(x2, f_cubic(x2), label='cubic')
ax.legend()
plt.show()
```

```
A, nu, k = 10, 4, 2
def f(x, A, nu, k):
    return A * np.exp(-k*x) * np.cos(2*np.pi * nu * x)
xmax, nx = 0.5, 8
x = np.linspace(0, xmax, nx)
y = f(x, A, nu, k)
f_nearest = interp1d(x, y, kind='nearest')
f_linear = interp1d(x, y)
f_cubic = interp1d(x, y, kind='cubic')

x2 = np.linspace(0, xmax, 100)

fig, ax = plt.subplots()
ax.plot(x, y, 'o', label='data points')
ax.plot(x2, f(x2, A, nu, k), label='exact')
ax.plot(x2, f_nearest(x2), label='nearest')
ax.plot(x2, f_linear(x2), label='linear')
ax.plot(x2, f_cubic(x2), label='cubic')
ax.legend()
plt.show()
```



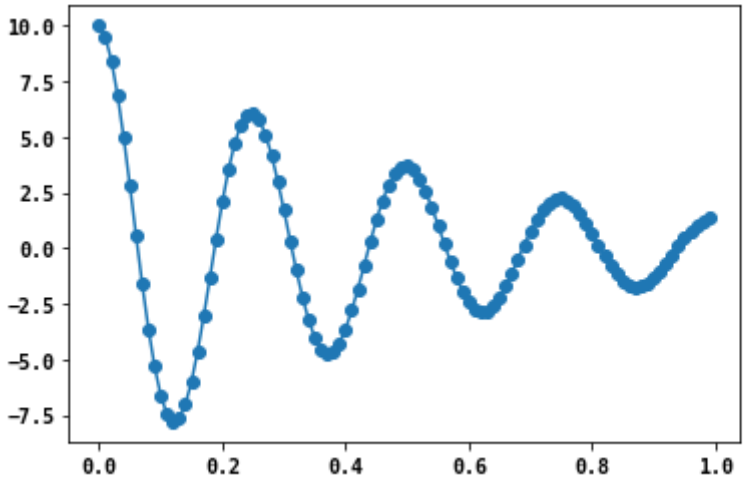
Scipy allows us to **fit curves**, and example of this is using "leastsq", which try to gets the least square residual

First, create a distribution of random points. From the previous equation, we can have an analytical solution as follows:

```
A, freq, tau = 10, 4, 0.5
def f(t, A, freq, tau):
    return A * np.exp(-t/tau) * np.cos(2*np.pi * freq * t)
tmax, dt = 1, 0.01
t = np.arange(0, tmax, dt)
```

```
yexact = f(t, A, freq, tau)
y = yexact
fig, ax = plt.subplots()
ax.plot(t, yexact)
ax.scatter(t, y)
plt.show()
```

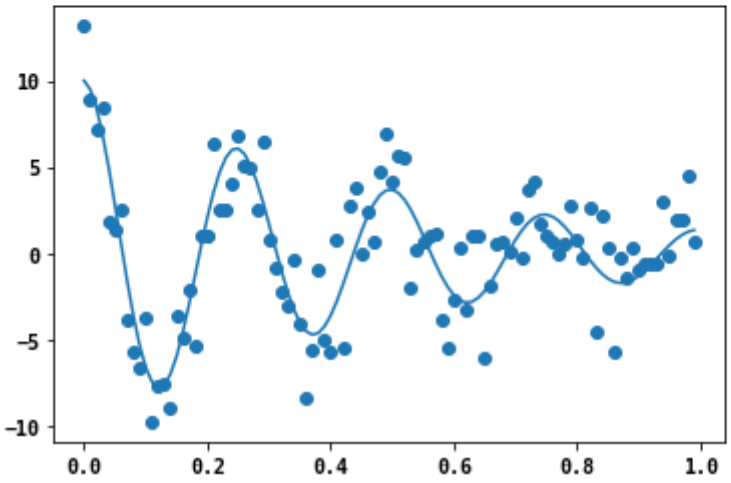
```
A, freq, tau = 10, 4, 0.5
def f(t, A, freq, tau):
    return A * np.exp(-t/tau) * np.cos(2*np.pi * freq * t)
tmax, dt = 1, 0.01
t = np.arange(0, tmax, dt)
yexact = f(t, A, freq, tau)
y = yexact
fig, ax = plt.subplots()
ax.plot(t, yexact)
ax.scatter(t, y)
plt.show()
```



We can convert this solution in a dispersed one by adding random values to the actual points.

```
A, freq, tau = 10, 4, 0.5
def f(t, A, freq, tau):
    return A * np.exp(-t/tau) * np.cos(2*np.pi * freq * t)
tmax, dt = 1, 0.01
t = np.arange(0, tmax, dt)
yexact = f(t, A, freq, tau)
y = yexact + np.random.randn(len(yexact))*2fig, ax = plt.subplots()
ax.plot(t, yexact)
ax.scatter(t, y)
plt.show()
```

```
A, freq, tau = 10, 4, 0.5
def f(t, A, freq, tau):
    return A * np.exp(-t/tau) * np.cos(2*np.pi * freq * t)
tmax, dt = 1, 0.01
t = np.arange(0, tmax, dt)
yexact = f(t, A, freq, tau)
y = yexact + np.random.randn(len(yexact))*2
fig, ax = plt.subplots()
ax.plot(t, yexact)
ax.scatter(t, y)
plt.show()
```



To calculate the curve fitting with the least square, we need to import the library:

```
from scipy.optimize import leastsq
```

The "leastsq" function returns the parameters you input optimized. To solve it, we follow these steps:

- Create a function that returns the residuals of the function
- "p0" correspond to initial values to start the iteration; they don't have a significant impact on the solution; you could try "1,1,1"
- And finally, we pass to it some arguments related to the time and the scattered points.

```
def residuals(p, y, t):
    A, freq, tau = p
    return y - f(t, A, freq, tau)
```

```
p0 = 5, 5, 1
plsq = leastsq(residuals, p0, args=(y, t))
plsq[0]
```

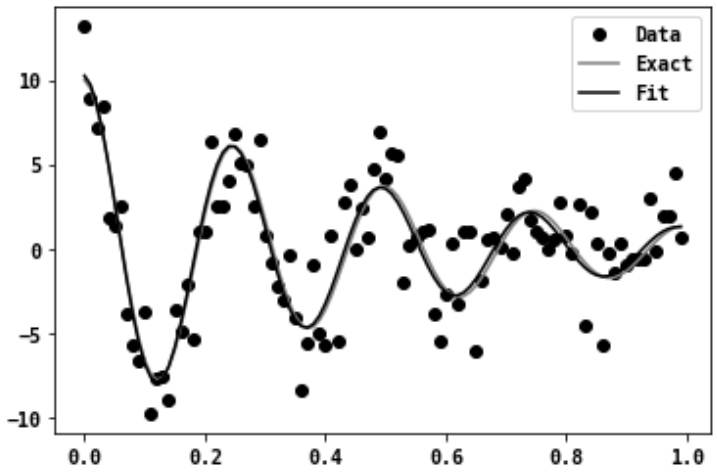
```
from scipy.optimize import leastsq
def residuals(p, y, t):
    A, freq, tau = p
    return y - f(t, A, freq, tau)

p0 = (1,1,1)
plsq = leastsq(residuals, p0, args=(y, t))
plsq[0]
```

```
array([10.25050424,  4.04337564,  0.47510045])
```

If we plot the "f" function defined before with the "optimized" parameters, we would see that we get a solution that is approximate to the exact one, even though we have dispersed the data.

```
fig, ax = plt.subplots()
ax.plot(t, y, 'o', c='k', label='Data')
ax.plot(t,yexact,c='gray', label='Exact')
ax.plot(t,f(t, 10.25050119, 4.04337629, 0.47510076),c='k',
label='Fit')
ax.legend()
plt.show()
```

```
fig, ax = plt.subplots()
ax.plot(t, y, 'o', c='k', label='Data')
ax.plot(t,yexact,c='gray', label='Exact')
ax.plot(t,f(t, 10.25050119, 4.04337629, 0.47510076),c='k', label='Fit')
ax.legend()
plt.show()
```



Reference links:

There are no better links that the ones from the documentation; they are always up-to-date.

- For Numpy:
  https://numpy.org/doc/
- For matplotlib:
  https://matplotlib.org/3.3.2/contents.html
- For Scipy:

  https://docs.scipy.org/doc/scipy/reference/index.html