1. Creating Numpy arrays

Numpy has many different types of data "containers": lists, dictionaries, tuples etc. However none of them allows for efficient numerical calculation, in particular not in multi-dimensional cases (think e.g. of operations on images). Numpy has been developed exactly to fill this gap. It provides a new data structure, the **numpy array**, and a large library of operations that allow to:

- generate such arrays
- combine arrays in different ways (concatenation, stacking etc.)
- modify such arrays (projection, extraction of sub-arrays etc.)
- apply mathematical operations on them

Numpy is the base of almost the entire Python scientific programming stack. Many libraries build on top of Numpy, either by providing specialized functions to operate on them (e.g. scikit-image for image processing) or by creating more complex data containers on top of it. The data science library Pandas that will also be presented in this course is a good example of this with its dataframe structures.

In [ ]:
```
import numpy as np
from svg import numpy_to_svg
```

1.1 What is an array ?

Let us create the simplest example of an array by transforming a regular Python list into an array (we will see more advanced ways of creating arrays in the next chapters):

In [ ]:
```
mylist = [2,5,3,9,5,2]
```

In [3]:
```
mylist
```

Out[3]:
```
[2, 5, 3, 9, 5, 2]
```

In [4]:
```
myarray = np.array(mylist)
```

In [5]:
```
myarray
```

Out[5]:
```
array([2, 5, 3, 9, 5, 2])
```

In [6]:
```
type(myarray)
```

Out[6]:
```
numpy.ndarray
```

We see that `myarray` is a Numpy array thanks to the `array` specification in the output. The type also says that we have a numpy ndarray (n-dimensional). At this point we don't see a big difference with regular lists, but we'll see in the following sections all the operations we can do with these objects.

We can already see a difference with two basic attributes of arrays: their type and shape.

1.1.1 Array Type

Just like when we create regular variables in Python, arrays receive a type when created. Unlike regular list, **all** elements of an array always have the same type. The type of an array can be recovered through the `.dtype` method:

```
In [7]: myarray.dtype
Out[7]: dtype('int64')
```

Depending on the content of the list, the array will have different types. But the logic of "maximal complexity" is kept. For example if we mix integers and floats, we get a float array:

```
In [8]: myarray2 = np.array([1.2, 6, 7.6, 5])
        myarray2
Out[8]: array([1.2, 6. , 7.6, 5. ])
```

```
In [9]: myarray2.dtype
Out[9]: dtype('float64')
```

In general, we have the possibility to assign a type to an array. This is true here, as well as later when we'll create more complex arrays, and is done via the `dtype` option:

```
In [10]: myarray2 = np.array([1.2, 6, 7.6, 500], dtype=np.uint8)
         myarray2
Out[10]: array([  1,   6,   7, 244], dtype=uint8)
```

The type of the array can also be changed after creation using the `.astype()` method:

```
In [11]: myfloat_array = np.array([1.2, 6, 7.6, 500], dtype=np.float)
         myfloat_array.dtype
Out[11]: dtype('float64')
```

```
In [12]: myint_array = myfloat_array.astype(np.int8)
         myint_array.dtype
Out[12]: dtype('int8')
```

1.1.2 Array shape

A very important property of an array is its **shape** or in other words the dimensions of each axis. That property can be accessed via the `.shape` property:

```
In [13]: myarray
Out[13]: array([2, 5, 3, 9, 5, 2])
```

```
In [14]: myarray.shape
Out[14]: (6,)
```

We see that our simple array has only one dimension of length 6. Now of course we can create more complex arrays. Let's create for example a *list of two lists*:

In [15]:
```
my2d_list = [[1,2,3], [4,5,6]]

my2d_array = np.array(my2d_list)
my2d_array
```

Out[15]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

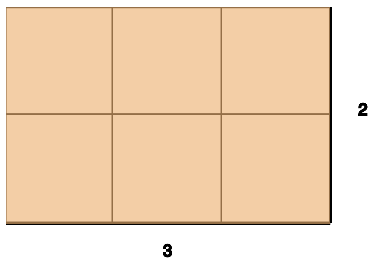
In [16]:
```
my2d_array.shape
```

Out[16]:
```
(2, 3)
```

We see now that the shape of this array is *two-dimensional*. We also see that we have 2 lists of 3 elements. In fact at this point we should forget that we have a list of lists and simply consider this object as a *matrix* with *two rows and three columns*. We'll use the follwing graphical representation to clarify some concepts:

In [17]:
```
numpy_to_svg(my2d_array)
```

Out[17]:



1.2 Creating arrays

We have seen that we can turn regular lists into arrays. However this becomes quickly impractical for larger arrays. Numpy offers several functions to create particular arrays.

1.2.1 Common simple arrays

For example an array full of zeros or ones:

In [18]:
```
one_array = np.ones((2,3))
one_array
```

Out[18]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [19]:
```
zero_array = np.zeros((2,3))
zero_array
```

Out[19]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

One can also create diagonal matrix:

In [20]:
```
np.eye(3)
```
Out[20]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

By default Numpy creates float arrays:

In [21]:
```
one_array.dtype
```
Out[21]:
```
dtype('float64')
```

However as mentioned before, one can impose a type usine the `dtype` option:

In [22]:
```
one_array_int = np.ones((2,3), dtype=np.int8)
one_array_int
```
Out[22]:
```
array([[1, 1, 1],
       [1, 1, 1]], dtype=int8)
```

In [23]:
```
one_array_int.dtype
```
Out[23]:
```
dtype('int8')
```

1.2.2 Copying the shape

Often one needs to create arrays of same shape. This can be done with "like-functions":

In [24]:
```
same_shape_array = np.zeros_like(one_array)
same_shape_array
```
Out[24]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

In [25]:
```
one_array.shape
```
Out[25]:
```
(2, 3)
```

In [26]:
```
same_shape_array.shape
```
Out[26]:
```
(2, 3)
```

In [27]:
```
np.ones_like(one_array)
```
Out[27]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

1.2.3 Complex arrays

We are not limited to create arrays containing ones or zeros. Very common operations involve e.g. the creation of arrays containing regularly arrange numbers. For example a "from-to-by-step" list:

In [28]:
```
np.arange(0, 10, 2)
```
Out[28]:
```
array([0, 2, 4, 6, 8])
```

Or equidistant numbers between boundaries:

```
In [29]: np.linspace(0,1, 10)
```

```
Out[29]: array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
                0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

Numpy offers in particular a `random` submodules that allows one to create arrays containing values from a wide array of distributions. For example, normally distributed:

```
In [30]: normal_array = np.random.normal(loc=10, scale=2, size=(3,4))
         normal_array
```

```
Out[30]: array([[16.64156121, 13.38970093, 11.32772287,  7.93713055],
                [ 8.33365707, 11.27817138,  9.81766403, 11.11541451],
                [12.97743479,  7.1622948 , 12.02417108,  8.64402656]])
```

```
In [31]: np.random.poisson(lam=5, size=(3,4))
```

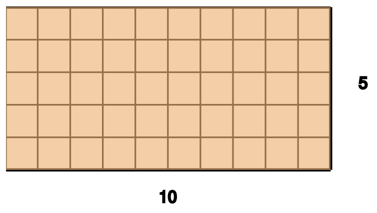
```
Out[31]: array([[4, 4, 2, 4],
                [3, 7, 6, 3],
                [6, 5, 5, 4]])
```

1.2.4 Higher dimensions

Until now we have almost only dealt with 1D or 2D arrays that look like a simple grid:

```
In [32]: myarray = np.ones((5,10))
         numpy_to_svg(myarray)
```

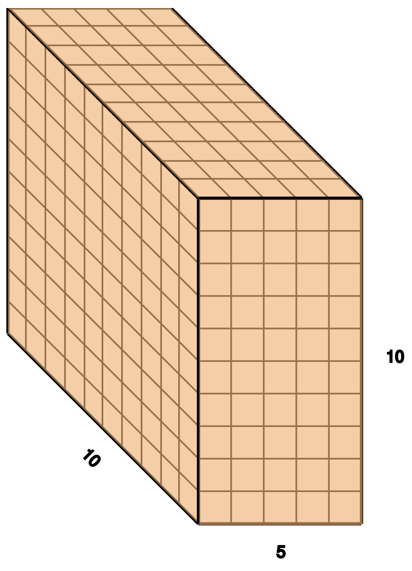
Out[32]:



We are not limited to create 1 or 2 dimensional arrays. We can basically create any-dimension array. For example in microscopy, images can be volumetric and thus they are 3D arrays in Numpy. For example if we acquired 5 planes of a 10px by 10px image, we would have something like:

```
In [33]: array3D = np.ones((10,10,5))
In [34]: numpy_to_svg(array3D)
```

Out[34]:



All the functions and properties that we have seen until now are N-dimensional, i.e. they work in the same way irrespective of the array size.

1.3 Importing arrays

We have seen until now multiple ways to create arrays. However, most of the time, you will *import* data from some source, either directly as arrays or as lists, and use these data in your analysis.

1.3.1 Loading and saving arrays

Numpy can efficiently save and load arrays in its own format `.npy` . Let's create an array and save it:

```
In [35]: array_to_save = np.random.normal(10, 2, (4,5))
         array_to_save
```

```
Out[35]: array([[ 5.41052227, 11.78370736,  9.22402365,  9.91645679,  9.48495895],
                [10.10853493,  8.75839699,  8.26026504, 12.51736441,  9.80407577],
                [10.09084097,  7.27962072, 11.05963249, 14.37978527,  9.00654627],
                [ 6.01521954, 10.25115807, 10.28647927, 10.12389832,  8.91184397]])
```

```
In [36]: np.save('my_saved_array.npy', array_to_save)
In [37]: ls
```

```
01-DA_Numpy_arrays_creation.ipynb   98-DA_Numpy_Solutions.ipynb
02-DA_Numpy_array_maths.ipynb       99-DA_Pandas_Exercises.ipynb
03-DA_Numpy_matplotlib.ipynb        99-DA_Pandas_Solutions.ipynb
04-DA_Numpy_indexing.ipynb          My_first_plot.png
05-DA_Numpy_combining_arrays.ipynb  SNSF_data.ipynb
06-DA_Pandas_introduction.ipynb     Untitled.ipynb
07-DA_Pandas_structures.ipynb       __pycache__/
08-DA_Pandas_import.ipynb           ipyleaflet.ipynb
09-DA_Pandas_operations.ipynb       multiple_arrays.npz
10-DA_Pandas_combine.ipynb          my_saved_array.npy
11-DA_Pandas_splitting.ipynb        raw.githubusercontent.com/
12-DA_Pandas_plotting.ipynb         svg.py
13-DA_Pandas_ML.ipynb               unused/
98-DA_Numpy_Exercises.ipynb
```

Now that this array is saved on disk, we can load it again using `np.load` :

```
In [38]: new_array = np.load('my_saved_array.npy')
         new_array
```

```
Out[38]: array([[ 5.41052227, 11.78370736,  9.22402365,  9.91645679,  9.48495895],
                [10.10853493,  8.75839699,  8.26026504, 12.51736441,  9.80407577],
                [10.09084097,  7.27962072, 11.05963249, 14.37978527,  9.00654627],
                [ 6.01521954, 10.25115807, 10.28647927, 10.12389832,  8.91184397]])
```

If you have several arrays that belong together, you can also save them in a single file using `np.savez` in `npz` format. Let's create a second array:

```
In [39]: array_to_save2 = np.random.normal(10, 2, (1,2))
         array_to_save2
```

```
Out[39]: array([[14.57759687,  7.62340049]])
```

```
In [40]: np.savez('multiple_arrays.npz', array_to_save=array_to_save, array_to_save2=
         array_to_save2)
```

```
In [41]: ls
```

```
01-DA_Numpy_arrays_creation.ipynb   98-DA_Numpy_Solutions.ipynb
02-DA_Numpy_array_maths.ipynb       99-DA_Pandas_Exercises.ipynb
03-DA_Numpy_matplotlib.ipynb        99-DA_Pandas_Solutions.ipynb
04-DA_Numpy_indexing.ipynb          My_first_plot.png
05-DA_Numpy_combining_arrays.ipynb  SNSF_data.ipynb
06-DA_Pandas_introduction.ipynb     Untitled.ipynb
07-DA_Pandas_structures.ipynb       __pycache__/
08-DA_Pandas_import.ipynb           ipyleaflet.ipynb
09-DA_Pandas_operations.ipynb       multiple_arrays.npz
10-DA_Pandas_combine.ipynb          my_saved_array.npy
11-DA_Pandas_splitting.ipynb        raw.githubusercontent.com/
12-DA_Pandas_plotting.ipynb         svg.py
13-DA_Pandas_ML.ipynb               unused/
98-DA_Numpy_Exercises.ipynb
```

And when we load it again:

```
In [42]: load_multiple = np.load('multiple_arrays.npz')
         type(load_multiple)

Out[42]: numpy.lib.npyio.NpzFile
```

We get here an `NpzFile` *object* from which we can read our data. Note that when we load an `npz` file, it is only loaded *lazily*, i.e. data are not actually read, but the content is parsed. This is very useful if you need to store large amounts of data but don't always need to re-load all of them. We can use methods to actually access the data:

```
In [43]: load_multiple.files

Out[43]: ['array_to_save', 'array_to_save2']

In [44]: load_multiple.get('array_to_save2')

Out[44]: array([[14.57759687,  7.62340049]])
```

1.3.2 Importing data as arrays

Images are a typical example of data that are array-like (matrix of pixels) and that can be imported directly as arrays. Of course, each domain will have it's own *importing libraries*. For example in the area of imaging, the scikit-image package is one of the main libraries, and it offers and importer of images as arrays which works both with local files and web addresses:

```
In [45]: import skimage.io

         image = skimage.io.imread('https://upload.wikimedia.org/wikipedia/commons/f/
         fd/%27%C3%9Cbermut_Exub%C3%A9rance%27_by_Paul_Klee%2C_1939.jpg')
```

We can briefly explore that image:

```
In [46]: type(image)

Out[46]: numpy.ndarray

In [47]: image.dtype

Out[47]: dtype('uint8')

In [48]: image.shape

Out[48]: (584, 756, 3)
```

We see that we have an array of integeres with 3 dimensions. Since we imported a jpg image, we know that the thrid dimension corresponds to three color channels Red, Green, Blue (RGB).

You can also read regular CSV files directly as Numpy arrays. This is more commonly done using Pandas, so we don't spend much time on this, but here is an example on importing data from the web:

```
In [49]: oilprice = np.loadtxt('https://raw.githubusercontent.com/guiwitz/Rdatasets/m
         aster/csv/quantreg/gasprice.csv',
                   delimiter=',', usecols=range(2,3), skiprows=1)
```

```
In [50]: oilprice
```

```
Out[50]: array([126.6, 127.2, 132.1, 133.3, 133.9, 134.5, 133.9, 133.4, 132.8,
                132.3, 131.1, 134.1, 119.2, 116.8, 113.9, 110.6, 107.8, 105.4,
                102.5, 104.5, 104.3, 104.7, 105.2, 106.6, 106.9, 109. , 110.4,
                111.3, 112.1, 112.9, 114. , 113.8, 113.5, 112.6, 111.4, 110.4,
                109.8, 109.4, 109.1, 109.1, 109.9, 111.2, 112.4, 112.4, 112.7,
                112. , 111. , 109.7, 109.2, 108.9, 108.4, 108.8, 109.1, 109.1,
                110.2, 110.4, 109.9, 109.9, 109.1, 107.5, 106.3, 105.3, 104.2,
                102.6, 101.4, 100.6,  99.5, 100.4, 101.1, 101.4, 101.2, 101.3,
                101. , 101.5, 101.3, 102.6, 105.1, 105.8, 107.2, 108.9, 110.2,
                111.8, 112. , 112.8, 114.3, 115.1, 115.3, 114.9, 114.7, 113.9,
                113.2, 112.8, 112.6, 112.3, 111.6, 112.3, 112.1, 112.1, 112.4,
                112.3, 111.8, 111.5, 111.5, 111.3, 111.3, 112. , 112. , 111.2,
                110.6, 109.8, 108.9, 107.8, 107.4, 106.9, 106.5, 106.6, 106.1,
                105.5, 105.5, 106.2, 105.3, 104.7, 104.2, 104.8, 105.8, 105.6,
                105.7, 106.8, 107.9, 107.9, 108.6, 108.6, 109.7, 110.6, 110.6,
                110.7, 110.4, 110.1, 109.5, 108.9, 108.6, 108.1, 107.5, 106.9,
                106.2, 106. , 105.9, 106.5, 106.2, 105.5, 105.1, 104.5, 104.7,
                109.2, 109. , 109.3, 109.2, 108.4, 107.5, 106.4, 105.8, 105.1,
                103.6, 101.8, 100.3,  99.9,  99.2,  99.5, 100.1,  99.9, 100.5,
                100.7, 101.6, 100.9, 100.4, 100.7, 100.5, 100.7, 101.2, 101.1,
                102.8, 103.3, 103.7, 104. , 104.5, 104.6, 105. , 105.6, 106.5,
                107.3, 107.9, 109.5, 109.7, 110.3, 110.9, 111.4, 113. , 115.7,
                116.1, 116.5, 116.1, 115.6, 115. , 114. , 112.9, 112. , 111.4,
                110.6, 110.7, 112.1, 112.3, 112.2, 111.3, 108.2, 107.5, 106.4,
                105.6, 104.4, 106.3, 107. , 106.2, 106.8, 106.8, 106.2, 105.8,
                105.2, 106. , 106.3, 105.6, 105.5, 106.3, 107.7, 109.4, 111. ,
                113.3, 114.1, 116.4, 117.3, 119.1, 119.3, 119.4, 119. , 118.3,
                117.7, 116.9, 115.9, 114.8, 113.8, 112.6, 112.4, 112.1, 112.2,
                111.3, 111.1, 110.7, 110.6, 110.6, 110. , 109.2, 108.1, 107.3,
                106.2, 106. , 105.9, 105.6, 105.7, 105.8, 105.7, 107.2, 107.5,
                107.7, 108.6, 109.2, 108.4, 107.9, 107.6, 107.3, 107.8, 109.9,
                111.5, 111.6, 112.8, 115.8, 117.2, 119.5, 123.4, 124.3, 125.7,
                125.9, 126.2, 126.9, 126. , 125.2, 124.7, 124.1, 123. , 121.9,
                121.7, 121.5, 121.5, 120.9, 119.9, 119.6, 119.9, 120.1, 119.3,
                120.1, 120.3, 120.3, 119.9, 119.1, 120.3, 120.5, 121.7, 122.5,
                122.9, 123.8, 124.6, 124.2, 124.1, 123.3, 122.7, 122.4, 122. ,
                123.5, 123.6, 123.2, 123. , 122.7, 122. , 121.7, 120.8, 119.9,
                119.1, 119.6, 119.1, 119.2, 118.7, 118.8, 118.5, 118.2, 118.2,
                119.5, 120.4, 120.6, 119.8, 118.9, 117.9, 117.1, 116.9, 116.5,
                117. , 116.4, 118.5, 121.9, 121.8, 123. , 122.9, 122.7, 121.9,
                120.8, 119.5, 119.5, 118.7, 117.8, 116.8, 116.3, 116.4, 115.6,
                115. , 114. , 112.8, 111.8, 110.8, 109.9, 108.9, 108.3, 107.2,
                105.5, 105.1, 104.5, 103.2, 103.8, 102.5, 101.7, 100.6,  99.8,
                102.6, 102.3, 101.8, 102.1, 103.2, 103.8, 105.2, 105.5, 105.2,
                104.7, 106. , 104.9, 104.1, 104.2, 104.1, 103.7, 104.4, 103.5,
                102.3, 101.8, 101.1, 100.4,  99.8,  99.1,  98.7,  99.9,  99.9,
                100.6, 101. , 100.7, 100.1,  99.7,  99.4,  98.1,  97.1,  95.4,
                 93.3,  92.3,  92.1,  91.4,  91.3,  92. ,  92.1,  91.3,  90.8,
                 90.7,  89.9,  88.5,  89.1,  90. ,  95.8,  99.9, 105.5, 108.7,
                110.7, 110.3, 109.9, 110.7, 110.9, 111.2, 110.1, 108.8, 109.2,
                108.8, 110.5, 109.5, 111. , 112.3, 114.8, 117.2, 117.2, 118.3,
                121.4, 121.2, 121.4, 122.3, 123.4, 125.2, 124.8, 124.2, 123.4,
                122. , 122.5, 121.8, 122.2, 124. , 125.8, 126.2, 126. , 126.3,
                125.7, 126.3, 126. , 125.2, 126.8, 130.7, 130.7, 131.9, 135. ,
                140. , 141.3, 149. , 151.1, 150.8, 148.4, 147.8, 144.7, 141.5,
                140.6, 138.6, 142.7, 146.6, 149.4, 150.9, 153.5, 160.7, 166.4,
                164.1, 160.6, 157.1, 152.1, 149.9, 144.7, 143.7, 142. , 144.4,
                145.6, 150.2, 153.5, 153.9, 152.5, 149.8, 147.3, 151.6, 153.2,
                152.3, 150.2, 150.1, 148.7, 148.9, 146.4, 142.5, 139.6, 138.8,
                137.7, 140. , 145.8, 145.6, 144.6, 142.6, 146. , 142.9, 141. ,
                139.3, 138.7, 137.7, 137.9, 141.1, 146.9, 153.5, 158.6, 158.5,
                165.9, 166.3, 163.7, 165.6, 163. , 158. , 152.6, 145.4, 138.4,
                135. , 133. , 131.8, 131.9, 131.9, 134.7, 139.9, 148. , 153.8,
                151.1, 151.6, 146. , 138.1, 131. , 126.4, 122.1, 119.3, 117. ,
                114.7, 114. , 109.7, 108.4, 107.5, 104.2, 106.3, 109.6, 110.9,
                109.9, 108.7, 108.1, 109.8, 108.5, 108.9, 108.7, 111.8, 119.4,
                126.2, 130.8, 133.9, 138.2, 136.8, 136.7, 135.3, 135.6, 134.9,
                136. , 134.8, 135.3, 133.2, 133.5, 134.2, 135.7, 134.5, 136.1,
                138.1, 137.6, 135.5, 135.5, 135.7, 136.5, 135.3, 135.5, 136.7,
                135.7, 138.5, 141.6, 142.2, 144.3, 142.7, 142.7, 140.6, 137. ,
                133.6, 131.6, 131.6, 132.2, 137.1, 141.7, 141.2, 142.3, 142.2,
                143.7, 149.9, 158.2, 163. , 161.7, 164.1, 166.3, 167.3, 162.6,
                157.7, 155.7, 152.1, 150.4, 148.6, 144.1, 142.7, 144.4, 143.9,
                142.8, 145.6, 148. , 145.1, 144.3, 144.8, 148.9, 149.6, 148.8,
                151.6, 155. , 159.4, 169.3, 168.8, 165.3, 163.6, 158. , 152.4,
                151.1, 151.5, 152.7, 149.9, 149.4, 146.4, 145.9, 147.8, 145.4,
                144.1, 143.3, 145.9, 145.4, 149.2, 154.4, 157.9, 160.4, 159.1,
                160.9, 161.7])
```

In [ ]:

2. Mathematics with arrays

One of the great advantages of Numpy arrays is that they allow one to very easily apply mathematical operations to entire arrays effortlessly. We are presenting here 3 ways in which this can be done.

```
In [1]: import numpy as np
```

2.1 Simple calculus

To illustrate how arrays are useful, let's first consider the following problem. You have a list:

```
In [2]: mylist = [1,2,3,4,5]
```

And now you wish to add to each element of that list the value 3. If we write:

```
In [3]: mylist + 3

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-3-ecae2962d7b1> in <module>
        ----> 1 mylist + 3

        TypeError: can only concatenate list (not "int") to list
```

We receive an error because Python doesn't know how to combine a list with a simple integer. In this case we would have to use a for loop or a comprehension list, which is cumbersome.

```
In [4]: [x + 3 for x in mylist]

Out[4]: [4, 5, 6, 7, 8]
```

Let's see now how this works for an array:

```
In [5]: myarray = np.array(mylist)

In [6]: myarray + 3

Out[6]: array([4, 5, 6, 7, 8])
```

Numpy understands without trouble that our goal is to add the value 3 to *each element* in our list. Naturally this is dimension independent e.g.:

```
In [7]: my2d_array = np.ones((3,6))
        my2d_array

Out[7]: array([[1., 1., 1., 1., 1., 1.],
               [1., 1., 1., 1., 1., 1.],
               [1., 1., 1., 1., 1., 1.]])
```

```
In [8]:  my2d_array + 3

Out[8]:  array([[4., 4., 4., 4., 4., 4.],
                [4., 4., 4., 4., 4., 4.],
                [4., 4., 4., 4., 4., 4.]])
```

Of course as long as we don't reassign this new state to our variable it remains unchanged:

```
In [9]:  my2d_array

Out[9]:  array([[1., 1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1., 1.],
                [1., 1., 1., 1., 1., 1.]])
```

We have to write:

```
In [10]: my2d_array = my2d_array + 3

In [11]: my2d_array

Out[11]: array([[4., 4., 4., 4., 4., 4.],
                [4., 4., 4., 4., 4., 4.],
                [4., 4., 4., 4., 4., 4.]])
```

Naturally all basic operations work:

```
In [12]: my2d_array * 4

Out[12]: array([[16., 16., 16., 16., 16., 16.],
                [16., 16., 16., 16., 16., 16.],
                [16., 16., 16., 16., 16., 16.]])

In [13]: my2d_array / 5

Out[13]: array([[0.8, 0.8, 0.8, 0.8, 0.8, 0.8],
                [0.8, 0.8, 0.8, 0.8, 0.8, 0.8],
                [0.8, 0.8, 0.8, 0.8, 0.8, 0.8]])

In [14]: my2d_array ** 5

Out[14]: array([[1024., 1024., 1024., 1024., 1024., 1024.],
                [1024., 1024., 1024., 1024., 1024., 1024.],
                [1024., 1024., 1024., 1024., 1024., 1024.]])
```

2.2 Mathematical functions

In addition to simple arithmetic, Numpy offers a vast choice of functions that can be directly applied to arrays. For example trigonometry:

```
In [15]: np.cos(myarray)

Out[15]: array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362,  0.28366219])
```

Exponentials and logs:

```
In [16]: np.exp(myarray)

Out[16]: array([  2.71828183,   7.3890561 ,  20.08553692,  54.59815003,
                148.4131591 ])
```

```
In [17]: np.log10(myarray)

Out[17]: array([0.        , 0.30103   , 0.47712125, 0.60205999, 0.69897   ])
```

2.3 Logical operations

If we use a logical comparison on a regular variable, the output is a *boolean* (True or False) that describes the outcome of the comparison:

```
In [18]: a = 3
         b = 2
         a > 3

Out[18]: False
```

We can do exactly the same thing with arrays. When we added 3 to an array, that value was automatically added to each element of the array. With logical operations, the comparison is also done for each element in the array resulting in a boolean array:

```
In [19]: myarray = np.zeros((4,4))
         myarray[2,3] = 1
         myarray

Out[19]: array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 1.],
                [0., 0., 0., 0.]])
```

```
In [20]: myarray > 0

Out[20]: array([[False, False, False, False],
                [False, False, False, False],
                [False, False, False,  True],
                [False, False, False, False]])
```

Exactly as for simple variables, we can assign this boolean array to a new variable directly:

```
In [21]: myboolean = myarray > 0
```

```
In [22]: myboolean

Out[22]: array([[False, False, False, False],
                [False, False, False, False],
                [False, False, False,  True],
                [False, False, False, False]])
```

2.4 Methods modifying array dimensions

The operations described above were applied *element-wise*. However sometimes we need to do operations either at the array level or some of its axes. For example, we need very commonly statistics on an array (mean, sum etc.)

In [23]:
```
nd_array = np.random.normal(10, 2, (3,4))
nd_array
```

Out[23]:
```
array([[ 8.22235922, 10.86316749,  8.97190654, 12.16211971],
       [11.31745909,  9.80774793, 11.2873836 ,  6.77945745],
       [10.20776894,  8.78011512,  6.96723135, 11.77819806]])
```

In [24]:
```
np.mean(nd_array)
```

Out[24]:
```
9.762076209457817
```

In [25]:
```
np.std(nd_array)
```

Out[25]:
```
1.747626512794281
```

Or the maximum value:

In [26]:
```
np.max(nd_array)
```

Out[26]:
```
12.162119714449235
```

Note that several of these functions can be called as array methods instead of numpy functions:

In [27]:
```
nd_array.mean()
```

Out[27]:
```
9.762076209457817
```

In [28]:
```
nd_array.max()
```

Out[28]:
```
12.162119714449235
```

Note that most functions can be applied to specific axes. Let's remember that our arrays is:

In [29]:
```
nd_array
```

Out[29]:
```
array([[ 8.22235922, 10.86316749,  8.97190654, 12.16211971],
       [11.31745909,  9.80774793, 11.2873836 ,  6.77945745],
       [10.20776894,  8.78011512,  6.96723135, 11.77819806]])
```

We can for example do a maximum projection along the first axis (rows): the maximum value of eadch column is kept:

In [30]:
```
proj0 = nd_array.max(axis=0)
proj0
```

Out[30]:
```
array([11.31745909, 10.86316749, 11.2873836 , 12.16211971])
```

In [31]:
```
proj0.shape
```

Out[31]:
```
(4,)
```

We can of course do the same operation for the second axis:

```
In [32]: proj1 = nd_array.max(axis=1)
         proj1

Out[32]: array([12.16211971, 11.31745909, 11.77819806])
```

```
In [33]: proj1.shape

Out[33]: (3,)
```

There are of course more advanced functions. For example a cumulative sum:

```
In [34]: np.cumsum(nd_array)

Out[34]: array([  8.22235922,  19.08552671,  28.05743325,  40.21955296,
                 51.53701205,  61.34475998,  72.63214358,  79.41160103,
                 89.61936998,  98.3994851 , 105.36671645, 117.14491451])
```

3. Plotting arrays

Arrays can represent any type of numeric data, typical examples being e.g. time-series (1D), images (2D) etc. Very often it is helpful to visualize such arrays either while developing an analysis pipeline or as an end-result. We show here briefly how this visualization can be done using the Matplotlib library. That library has extensive capabilities and we present here a minimal set of examples to help you getting started. Note that we will see other libraries when exploring Pandas in the next chapters that are more specifically dedicated to data science.

All the necessary plotting functions reside in the `pyplot` module of Matplotlib. `plt` contains for example all the functions for various plot types:

- plot an image: `plt.imshow()`
- line plot: `plt.plot`
- plot a histogram: `plt.hist()`
- etc.

Let's import it with it's standard abbreviation `plt` (as well as numpy):

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

3.1 Data

We will use here Numpy to generate synthetic data to demonstrate plotting. We create an array for time, and then transform that array with a sine function. Finally we make a second version where we add some noise to the data:

In [2]:
```
# time array
time = np.arange(0,20,0.5)
# sine function
time_series = np.sin(time)
# sine function plus noise
time_series_noisy = time_series + np.random.normal(0,0.5,len(time_series))
```

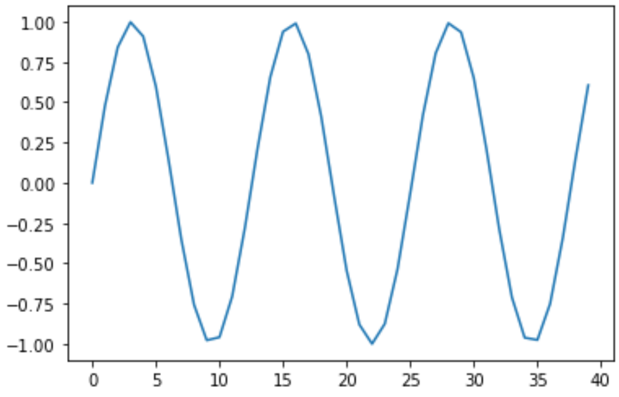
3.2 General concepts

We are going to see in the next sections a few example of important plots and how to customize them. However we start here by explaining here the basic concept of Matplotlib using a simple line plot (see next section for details on line plot).

3.2.1 One-line plot

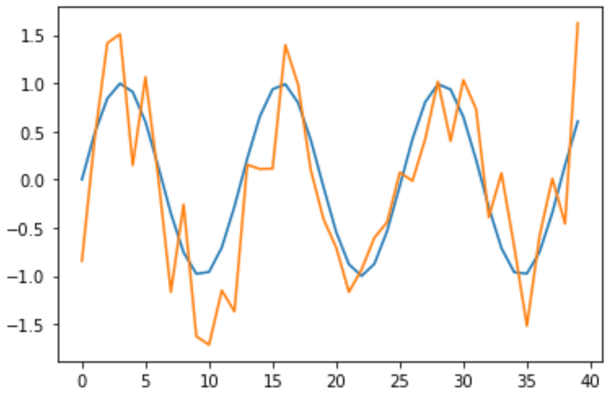
The simplest way to create a plot, is just to directly call the relevant function, e.g. `plt.plot()` for a line plot:

In [3]:
```
plt.plot(time_series);
```



If we need to plot multiple datasets one the same plot, we can just keep adding plots on top of each other:

In [4]:
```
plt.plot(time_series);
plt.plot(time_series_noisy);
```



As you can see Matplotlib automatically knows that you want to combine different signals, and by default colors them. From here, we can further customize each plot individually, but we are very quickly going to see limits for how to adjust the figure settings. What we really need here is a *handle* for the figure and each plot.

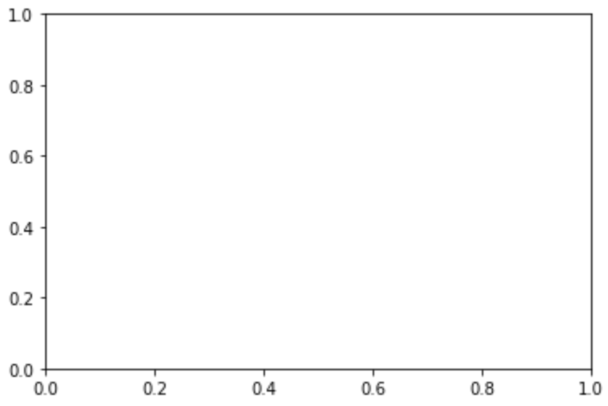
3.2.2 Object-based plots

In order to gain more control on the plot, we need to gain control on the elements that constitute it. Those are:

- The `Figure` object which contains all elements of the figure
- The `Axes` object, the actual plots that belong to a figure object

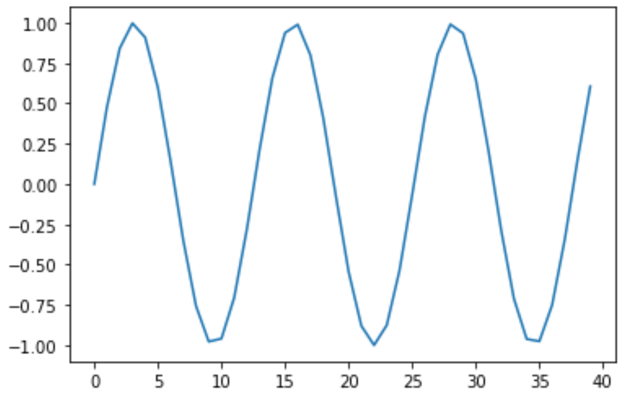
We can gain this control by explicity creating these objects via the `subplots()` function which returns a figure and an axis object:

In [5]:
```
fig, ax = plt.subplots()
```



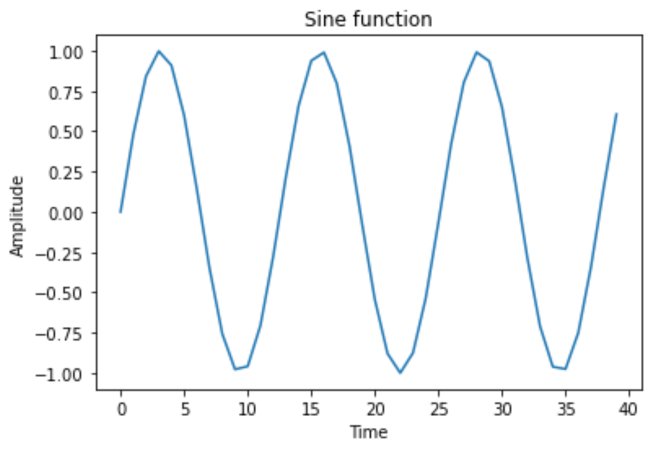
We see that we just get an empty figure with axes that we should now fill. For example the `ax` object can create an image plot on its own:

In [6]:
```
fig, ax = plt.subplots()
ax.plot(time_series);
```



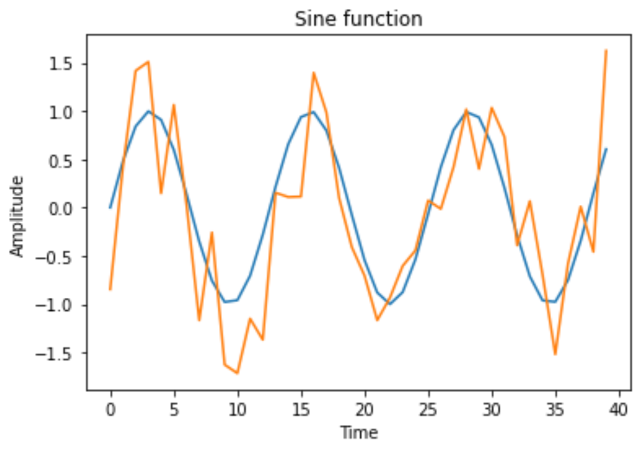
We can go further and customize other elements of the plot. Again, this is only possible because we have reference to the "plot-objects". For example we can add labels:

In [7]:
```
fig, ax = plt.subplots()
plt.plot(time_series);
ax.set_xlabel('Time')
ax.set_ylabel('Amplitude');
ax.set_title('Sine function');
```



We can also superpose multiple plots. As we want all of them to share the same axis, we use the same `ax` reference. For example we can add a line plot:

In [8]:
```
fig, ax = plt.subplots()
ax.plot(time_series);
ax.plot(time_series_noisy);
ax.set_xlabel('Time')
ax.set_ylabel('Amplitude');
ax.set_title('Sine function');
```



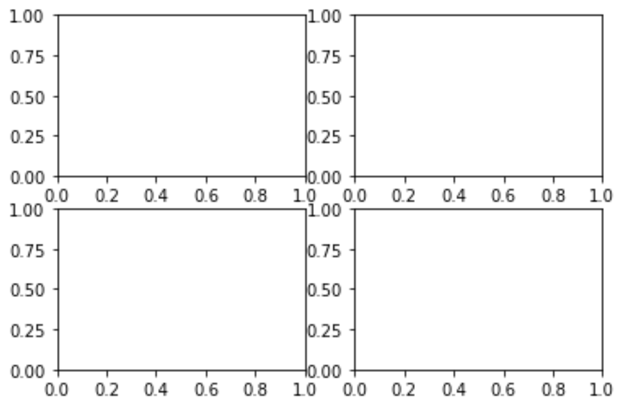
And finally we can export our image as an independent picture using the `fig` reference:

In [9]:
```
fig.savefig('My_first_plot.png')
```

3.2.3 Grids

Using the sort of syntax described above it is very easy to crate complex plots with multiple panels. The simplest solution is to specify a *grid* of plots when creating the figure using `plt.subplots()` . This provides a list of `Axes` objects, each corresponding to one element of the grid:

```
In [10]: fig, ax = plt.subplots(2,2)
```



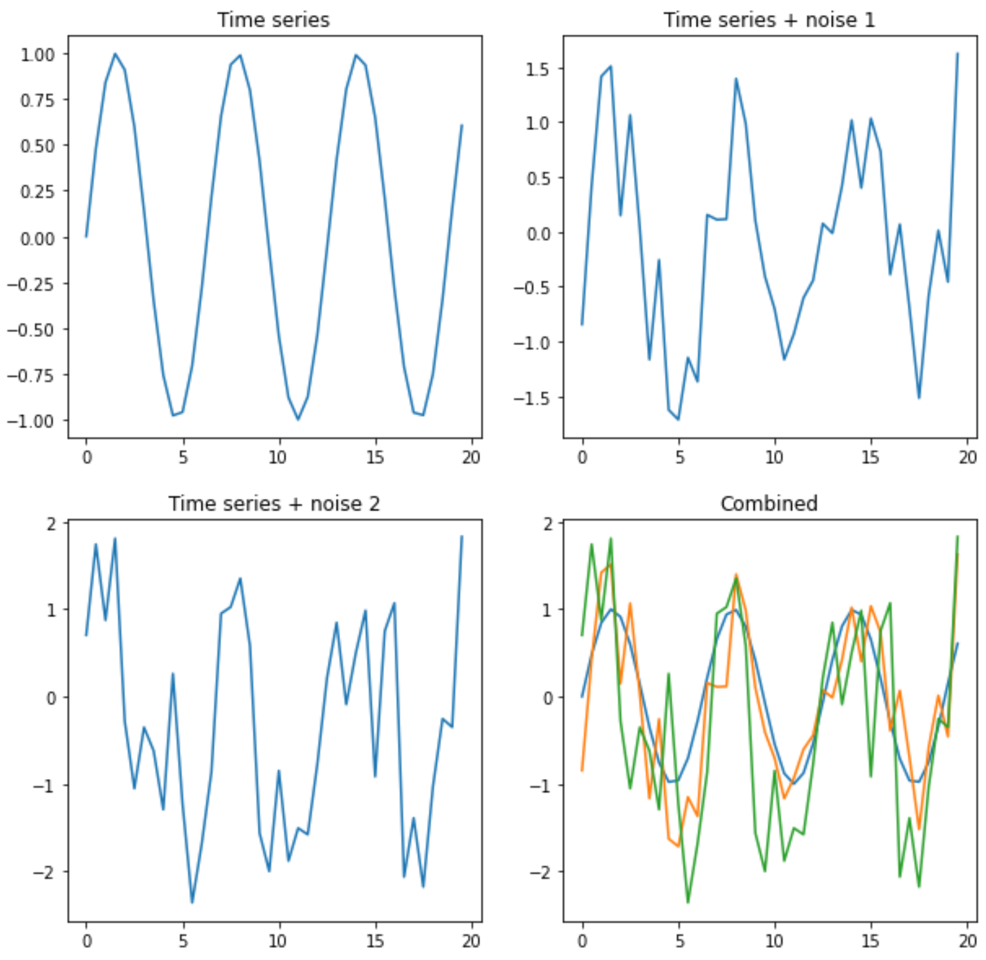
Here `ax` is now a 2D numpy array whose elements are `Axis` objects:

```
In [11]: type(ax)
Out[11]: numpy.ndarray
```

```
In [12]: ax.shape
Out[12]: (2, 2)
```

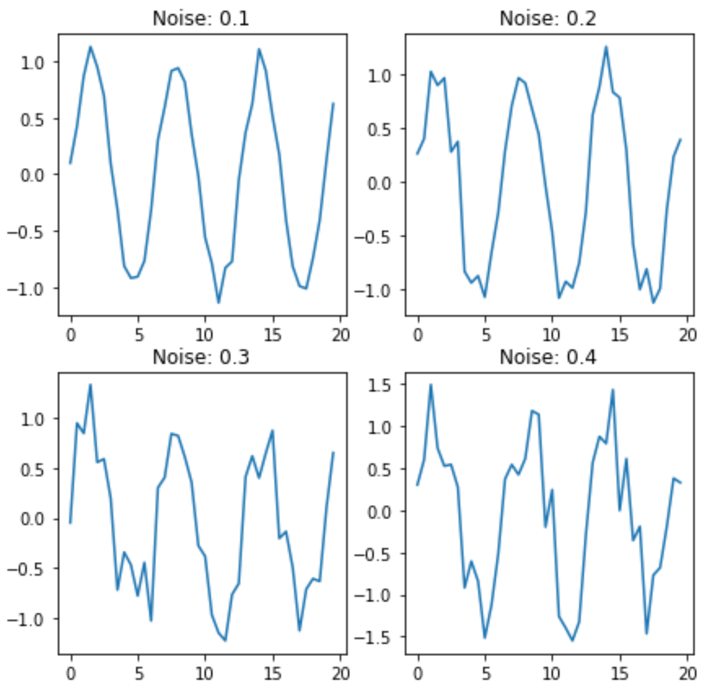
We access each element of the `ax` array like a regular list and use them to plot:

In [13]:
```
# we create additional data
time_series_noisy2 = time_series + np.random.normal(0,1,len(time_series))# c
reate figure with 2x2 subplots
time_series_noisy3 = time_series + np.random.normal(0,1.5,len(time_series))#
create figure with 2x2 subplots

# create the figure and axes
fig, ax = plt.subplots(2,2, figsize=(10,10))

# fill each subplot
ax[0,0].plot(time, time_series);
ax[0,1].plot(time, time_series_noisy);
ax[1,0].plot(time, time_series_noisy2);

# in the last plot, we combined all plots
ax[1,1].plot(time, time_series);
ax[1,1].plot(time, time_series_noisy);
ax[1,1].plot(time, time_series_noisy2);

# we can add titles to subplots
ax[0,0].set_title('Time series')
ax[0,1].set_title('Time series + noise 1')
ax[1,0].set_title('Time series + noise 2')
ax[1,1].set_title('Combined');
```



An alternative is to use `add_subplot` . Here we only create a figure, and progressively add new subplots in a pre-determined grid. This variant is useful when programmatically creating a figure, as it easily allows to create plots in a loop:

In [14]:
```
# create a figure
fig = plt.figure(figsize=(7,7))
for x in range(1,5):
    # add subplot and create an axis
    ax = fig.add_subplot(2,2,x)
    # plot the histogram in the axis
    ax.plot(time, time_series + np.random.normal(0,x/10, len(time)))
    # customize axis
    ax.set_title(f'Noise: {x/10}')
```



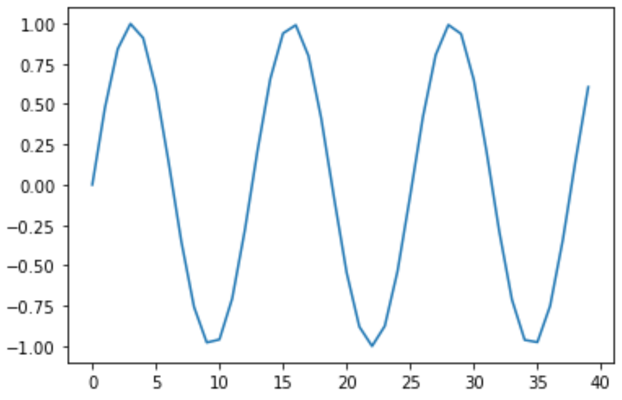
3.3 Plot types

There is an extensive choice of plot types available in Matplotlib. Here we limit the presentation to the three most common ones: line plot, histogram and image.

3.3.1 Line plot

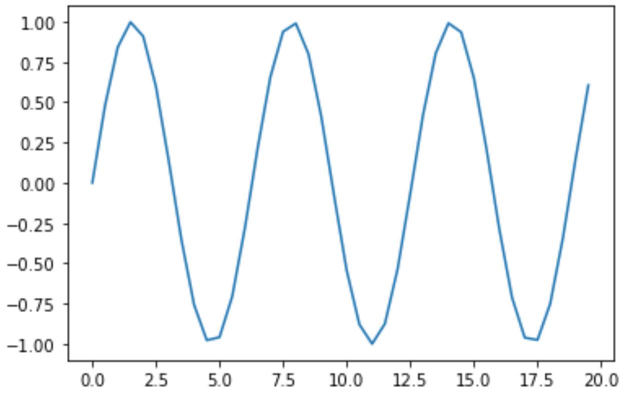
We have already seen line plots above, but we didn't customize the plot itself. A 1D array can simply be plotted by using:

In [15]:
```
plt.plot(time_series);
```



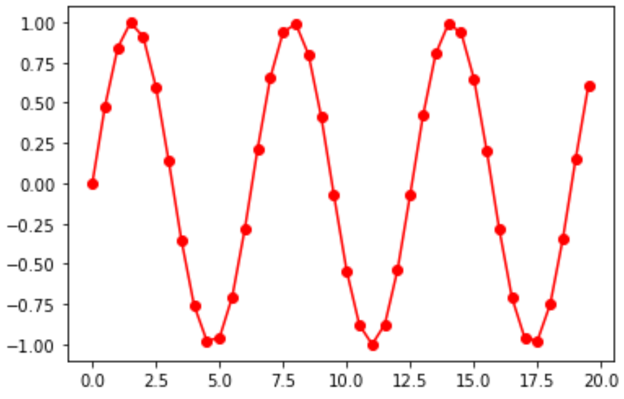
This generates by default a line plot where the x-axis simply uses the array index and the array itself is plotted as y-axis. We can explicitly specify the x-axis by passing first x-axis array, here the `time` array:

```
In [16]: plt.plot(time, time_series);
```



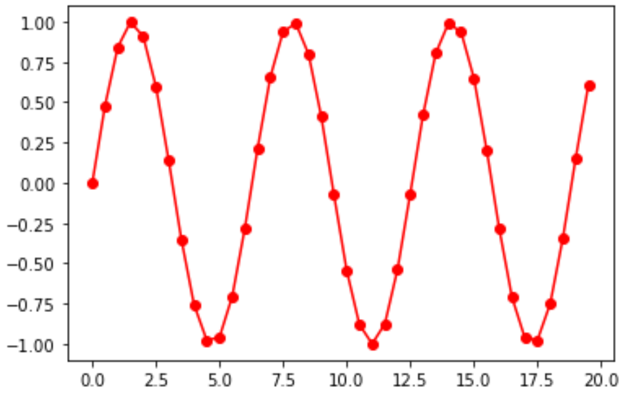
Each Matplotlib plot can be extensively customized. We only give here a few examples of what can be done. For example, we can change the plot color (for a list of named colors see here (https://matplotlib.org/3.1.0/gallery/color/named_colors.html)), and add markers (for a list of markers see here (https://matplotlib.org/3.1.1/api/markers_api.html)):

```
In [17]: plt.plot(time, time_series, color='red', marker='o');
```



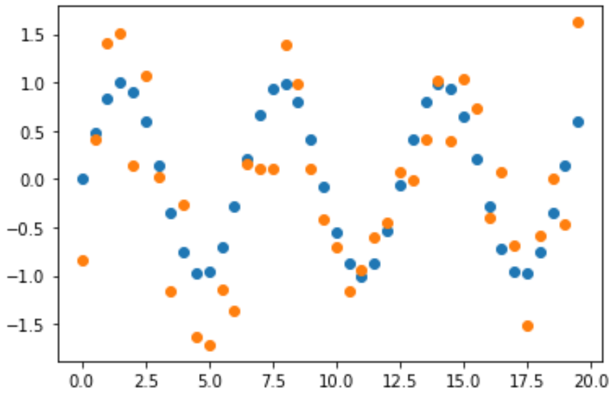
Conveniently, several of this styling options can be added in a short form. In this example we can specify that we want a line (`-`), markers (`o`) and the color red (`r`) using `-or` :

In [18]:
```
plt.plot(time, time_series, '-or');
```



Of course if the data are not representing a continuous signal but just a cloud of points, we can skip the line argument to obtain a scatter plot. You can also directly use the `plt.scatter()` function:

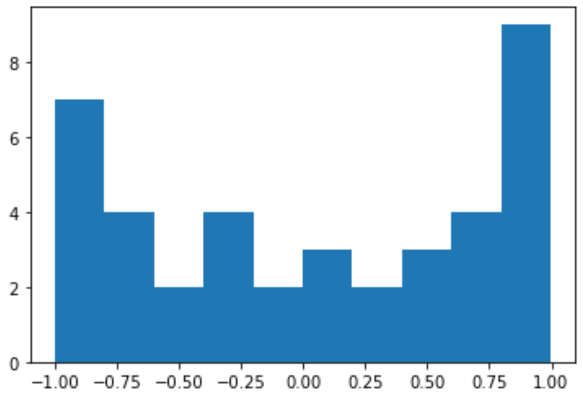
In [19]:
```
plt.plot(time, time_series, 'o');
plt.plot(time, time_series_noisy, 'o');
```



3.3.2 Histogram

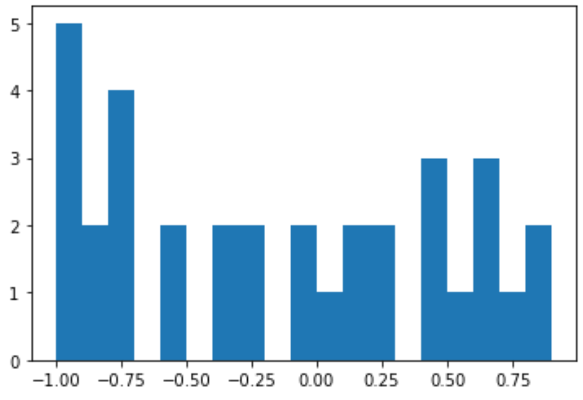
To get an idea of the contents of an array, it is very common to plot a histogram of it. This can be done with the `plt.hist()` function:

In [20]:
```
plt.hist(time_series);
```



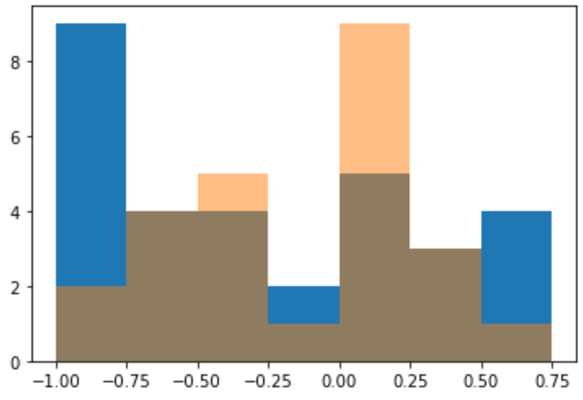
Matplotlib selects bins for you, but most of the time you'll want to change those. The simplest is just to specify all bins using `np.arange()` :

```
In [21]: plt.hist(time_series, bins = np.arange(-1,1,0.1));
```



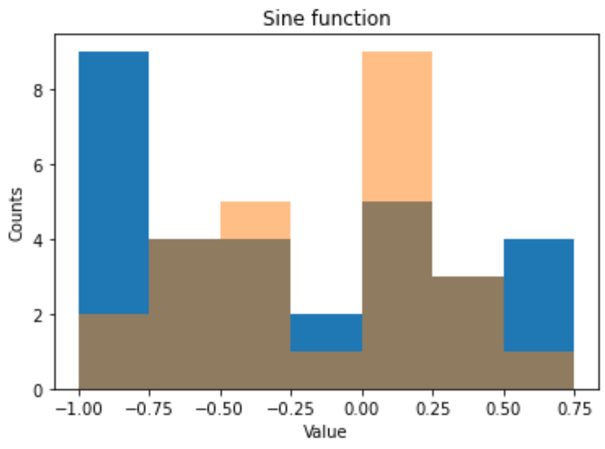
Just like for line plots, you can superpose histograms. However they will overlap, so you may want to fix the transparency of the additional layers with the `alpha` parameter:

```
In [22]: plt.hist(time_series, bins = np.arange(-1,1,0.25));
         plt.hist(time_series_noisy, bins = np.arange(-1,1,0.25), alpha = 0.5);
```



And also as demonstrated before you can adjust the settings of your figure, by creating figure and axis objects:

In [23]:
```
fig, ax = plt.subplots()
ax.hist(time_series, bins = np.arange(-1,1,0.25));
ax.hist(time_series_noisy, bins = np.arange(-1,1,0.25), alpha = 0.5);
ax.set_xlabel('Value')
ax.set_ylabel('Counts');
ax.set_title('Sine function');
```



3.3.4 Image plot

Finally, we often need to look at 2D arrays. These can of course be 2D functions but most of the time they are images. We can again create synthetic data with Numpy. First we create a two 2D grids that contain the x,y indices of each element:

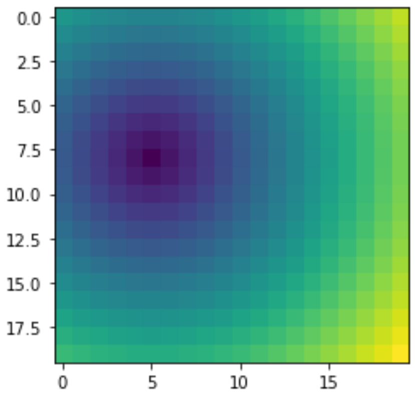
In [24]:
```
xindices, yindices = np.meshgrid(np.arange(20), np.arange(20))
```

Then we can crete an array that contains the euclidian distance from a given point $d = ((x-x_0)^2 + (y-y_0)^2)^{1/2}$

In [25]:
```
centerpoint = [5,8]
dist = ((xindices - centerpoint[0])**2 + (yindices - centerpoint[1])**2)**0.
5
```

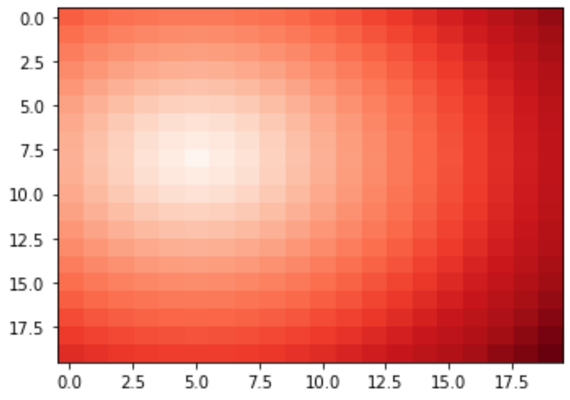
If we want to visualize this array, we can then use `plt.imshow()` :

In [26]:
```
plt.imshow(dist);
```



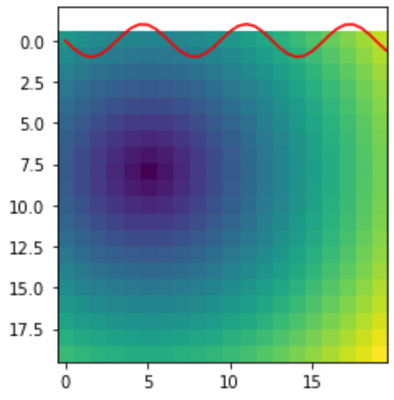
Like the other functions `plt.imshow()` has numerous options to adjust the image aspect. For example one can change the default colormap, or the aspect ratio of the image:

```
In [27]: plt.imshow(dist, cmap='Reds', aspect=0.7);
```



Finally, one can mix different types of plot. We can for example add our line plot from the beginning on top of the image:

```
In [28]: plt.imshow(dist)
         plt.plot(time, time_series, color = 'r')

Out[28]: [<matplotlib.lines.Line2D at 0x113ade100>]
```



4. Indexing, slicing

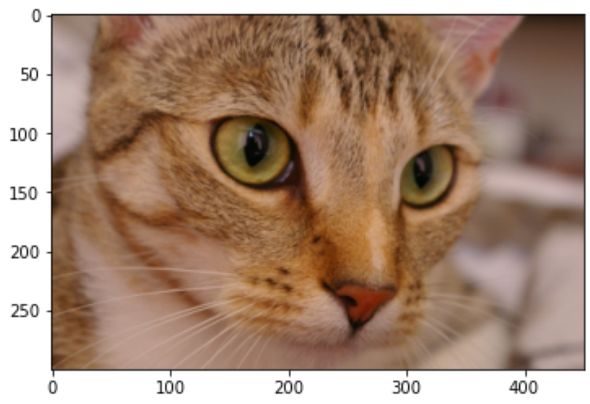
Each element of an array can be located by its position in each dimension. Numpy offers multiple ways to access single elements or groups of elements in very efficient ways. We will illustrate these concepts both with small simple matrices as well as a regular image, in order to illustrate them.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.gray();
import skimage
```

```
<Figure size 432x288 with 0 Axes>
```

We first load an image included in the scikit-image package:

In [2]:
```
image = skimage.data.chelsea()
plt.imshow(image);
```



We can check the dimensions of the image and see that it is an RGB image with 3 channels:

In [3]:
```
image.shape
```

Out[3]:
```
(300, 451, 3)
```

4.1 Accessing single values

We create a small 2D array to use as an example:

In [4]:
```
normal_array = np.random.normal(10, 2, (3,4))
normal_array
```

Out[4]:
```
array([[12.99205086,  7.7157832 , 14.66021898,  8.21412356],
       [ 9.19391119,  7.92142871, 13.31222213,  8.19957688],
       [11.08009573,  8.54243953, 12.71096417, 10.09637761]])
```

It is very easy to access an array's values. One can just pass an *index* for each dimensions. For example to recover the value on the last row and second column of the `normal_array` array we just write (remember counting starts at 0):

In [5]:
```
single_value = normal_array[2,1]
single_value
```
Out[5]:
```
8.542439525354693
```

What is returned in that case is a single number that we can re-use:

In [6]:
```
single_value += 10
single_value
```
Out[6]:
```
18.542439525354695
```

And that change doesn't affect the original value in the array:

In [7]:
```
normal_array
```
Out[7]:
```
array([[12.99205086,  7.7157832 , 14.66021898,  8.21412356],
       [ 9.19391119,  7.92142871, 13.31222213,  8.19957688],
       [11.08009573,  8.54243953, 12.71096417, 10.09637761]])
```

However we can also directly change the value in an array:

In [8]:
```
normal_array[2,1] = 23
```

In [9]:
```
normal_array
```
Out[9]:
```
array([[12.99205086,  7.7157832 , 14.66021898,  8.21412356],
       [ 9.19391119,  7.92142871, 13.31222213,  8.19957688],
       [11.08009573, 23.        , 12.71096417, 10.09637761]])
```

4.2 Accessing part of an array with indices: slicing

4.2.1 Selecting a range of elements

One can also select multiple elements in each dimension (e.g. multiple rows and columns in 2D) by using the `start:end:step` syntax. By default, if omitted, `start=0` , `end=last element and step=1` . For example to select the first **and** second rows of the first column, we can write:

In [10]:
```
normal_array[0:2,0]
```
Out[10]:
```
array([12.99205086,  9.19391119])
```

Note that the `end` element is **not** included. One can use the same notation for all dimensions:

In [11]:
```
normal_array[0:2,2:4]
```
Out[11]:
```
array([[14.66021898,  8.21412356],
       [13.31222213,  8.19957688]])
```

In [12]:
```
normal_array[1:,2:4]
```
Out[12]:
```
array([[13.31222213,  8.19957688],
       [12.71096417, 10.09637761]])
```

4.2.2 Selecting all elements

If we only specify `:`, it means we want to recover all elements in that dimension:

```
In [13]: normal_array[:,2:4]
```

```
Out[13]: array([[14.66021898,  8.21412356],
                [13.31222213,  8.19957688],
                [12.71096417, 10.09637761]])
```

Also in general, if you only specify the value for a single axis, this will take the first element of the first dimension:

```
In [14]: normal_array
```

```
Out[14]: array([[12.99205086,  7.7157832 , 14.66021898,  8.21412356],
                [ 9.19391119,  7.92142871, 13.31222213,  8.19957688],
                [11.08009573, 23.        , 12.71096417, 10.09637761]])
```

```
In [15]: normal_array[1]
```

```
Out[15]: array([ 9.19391119,  7.92142871, 13.31222213,  8.19957688])
```

Finally note that if you want to recover only one element along a dimension (single row, column etc), you can do that in two ways:

```
In [16]: normal_array[0,:]
```

```
Out[16]: array([12.99205086,  7.7157832 , 14.66021898,  8.21412356])
```

This returns a one-dimensional array containing a single row from the original array:

```
In [17]: normal_array[0,:].shape
```

```
Out[17]: (4,)
```

Instead, if you specify actual boundaries that still return only a single row:

```
In [18]: normal_array[0:1,:]
```

```
Out[18]: array([[12.99205086,  7.7157832 , 14.66021898,  8.21412356]])
```

```
In [19]: normal_array[0:1,:].shape
```

```
Out[19]: (1, 4)
```

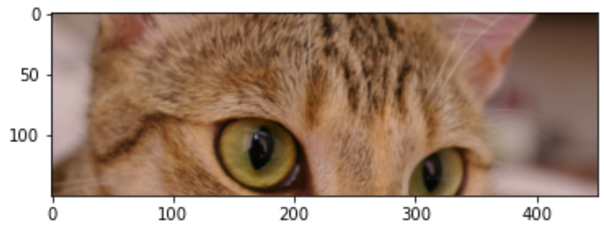
you recover a tow dimensional array where one of the dimensions has a size of 1.

4.2.3 Illustration on an image

We can for example only select half the rows of the image but all columns and channels:

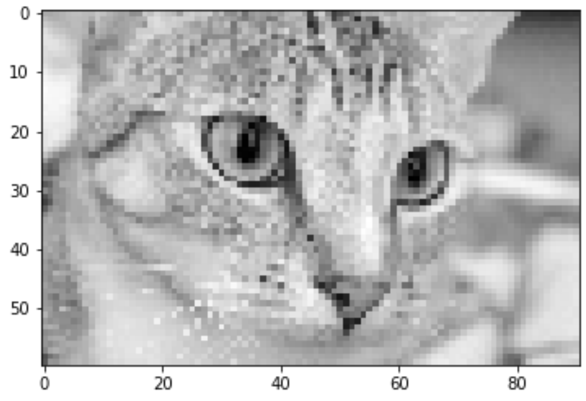
```
In [20]: image.shape
Out[20]: (300, 451, 3)
```

```
In [21]: sub_image = image[0:150,:,:]
         plt.imshow(sub_image);
```



Or we can take every fith column and row from a single channel, which returns a pixelated version of the original image:

```
In [22]: plt.imshow(image[::5,::5,0]);
```



4.3 Sub-arrays are not copies!

As often with Python when you create a new variable using a sub-array, that variable **is not independent** from the original variable:

```
In [23]: sub_array = normal_array[:,2:4]
```

```
In [24]: sub_array
Out[24]: array([[14.66021898,  8.21412356],
                [13.31222213,  8.19957688],
                [12.71096417, 10.09637761]])
```

```
In [25]: normal_array
Out[25]: array([[12.99205086,  7.7157832 , 14.66021898,  8.21412356],
                [ 9.19391119,  7.92142871, 13.31222213,  8.19957688],
                [11.08009573, 23.        , 12.71096417, 10.09637761]])
```

If for example we modify `normal_array`, this is going to be reflected in `sub_array` too:

```
In [26]: normal_array[0,2] = 100
In [27]: normal_array

Out[27]: array([[ 12.99205086,   7.7157832 , 100.        ,   8.21412356],
               [  9.19391119,   7.92142871,  13.31222213,   8.19957688],
               [ 11.08009573,  23.        ,  12.71096417,  10.09637761]])
```

```
In [28]: sub_array

Out[28]: array([[100.        ,   8.21412356],
               [ 13.31222213,   8.19957688],
               [ 12.71096417,  10.09637761]])
```

The converse is also true:

```
In [29]: sub_array[0,1] = 50
```

```
In [30]: sub_array

Out[30]: array([[100.        ,  50.        ],
               [ 13.31222213,   8.19957688],
               [ 12.71096417,  10.09637761]])
```

```
In [31]: normal_array

Out[31]: array([[ 12.99205086,   7.7157832 , 100.        ,  50.        ],
               [  9.19391119,   7.92142871,  13.31222213,   8.19957688],
               [ 11.08009573,  23.        ,  12.71096417,  10.09637761]])
```

If you want your sub-array to be an *independent* copy of the original, you have to use the `.copy()` method:

```
In [32]: sub_array_copy = normal_array[1:3,:].copy()
```

```
In [33]: sub_array_copy

Out[33]: array([[ 9.19391119,  7.92142871, 13.31222213,  8.19957688],
               [11.08009573, 23.        , 12.71096417, 10.09637761]])
```

```
In [34]: sub_array_copy[0,0] = 500
```

```
In [35]: sub_array_copy

Out[35]: array([[500.        ,   7.92142871,  13.31222213,   8.19957688],
               [ 11.08009573,  23.        ,  12.71096417,  10.09637761]])
```

```
In [36]: normal_array

Out[36]: array([[ 12.99205086,   7.7157832 , 100.        ,  50.        ],
               [  9.19391119,   7.92142871,  13.31222213,   8.19957688],
               [ 11.08009573,  23.        ,  12.71096417,  10.09637761]])
```

4.4. Accessing parts of an array with coordinates

In the above case, we are limited to select rectangular sub-regions of the array. But sometimes we want to recover a series of specific elements for example the elements (row=0, column=3) and (row=2, column=2). To achieve that we can simply index the array with a list containing row indices and another with columns indices:

In [37]:
```
row_indices = [0,2]
col_indices = [3,2]

normal_array[row_indices, col_indices]
```
Out[37]:
```
array([50.        , 12.71096417])
```

In [38]:
```
normal_array
```
Out[38]:
```
array([[ 12.99205086,   7.7157832 , 100.        ,  50.        ],
       [  9.19391119,   7.92142871,  13.31222213,   8.19957688],
       [ 11.08009573,  23.        ,  12.71096417,  10.09637761]])
```

In [39]:
```
selected_elements = normal_array[row_indices, col_indices]
```

In [40]:
```
selected_elements
```
Out[40]:
```
array([50.        , 12.71096417])
```

4.5 Logical indexing

The last way of extracting elements from an array is to use a boolean array of same shape. For example let's create a boolean array by comparing our original matrix to a threshold:

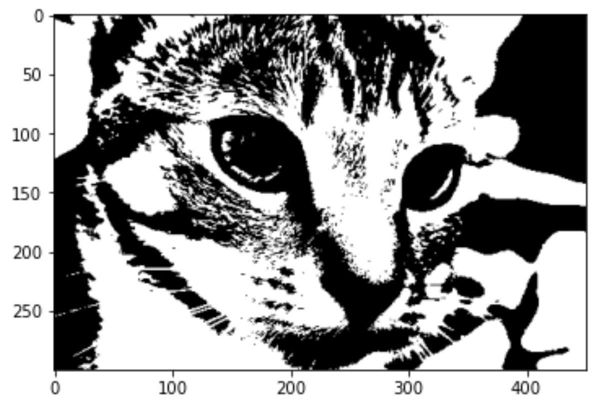
In [41]:
```
bool_array = normal_array > 40
bool_array
```
Out[41]:
```
array([[False, False,  True,  True],
       [False, False, False, False],
       [False, False, False, False]])
```

We see that we only have two elements which are above the threshold. Now we can use this logical array to *index* the original array. Imagine that the logical array is a mask with holes only in `True` positions and that we superpose it to the original array. Then we just take all the values visible in the holes:

In [42]:
```
normal_array[bool_array]
```
Out[42]:
```
array([100.,  50.])
```

Coming back to our real image, we can for example first create an image that contains a single channel and then find bright regions in it:

```
In [43]: single_channel = image[:,:,0]
         mask = single_channel > 150
         plt.imshow(mask);
```



And now we can recover all the pixels that are "selected" by this mask:

```
In [44]: single_channel[mask]

Out[44]: array([152, 152, 154, ..., 161, 161, 162], dtype=uint8)
```

4.6 Reshaping arrays

Often it is necessary to reshape arrays, i.e. keep elements unchanged but change their position. There are multiple functions that allow one to do this. The main one is of course `reshape` .

4.6.1 `reshape`

Given an array of MxN elements, one can reshape it with a shape OxP as long as $M*N = O*P$.

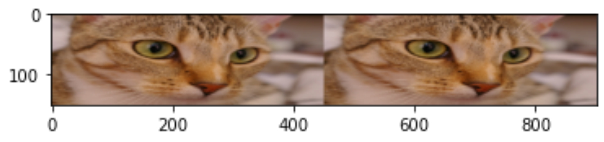
```
In [45]: reshaped = np.reshape(normal_array,(2,6))
         reshaped

Out[45]: array([[ 12.99205086,   7.7157832 , 100.        ,  50.        ,
                   9.19391119,   7.92142871],
                [ 13.31222213,   8.19957688,  11.08009573,  23.        ,
                  12.71096417,  10.09637761]])
```

```
In [46]: reshaped.shape

Out[46]: (2, 6)
```

```
In [47]: 300*451/150

Out[47]: 902.0
```

With the image as example, we can reshape the array from $300x451x3$ to $150x902x3$:

```
In [48]: plt.imshow(np.reshape(image, (150,902,3)))
Out[48]: <matplotlib.image.AxesImage at 0x11a925d60>
```



4.6.2 Flattening

It's also possible to simply flatten an array i.e. remove all dimensions to create a 1D array. This can be useful for example to create a histogram of a high-dimensional array.

```
In [49]: flattened = np.ravel(normal_array)
         flattened
Out[49]: array([ 12.99205086,   7.7157832 , 100.        ,  50.        ,
                  9.19391119,   7.92142871,  13.31222213,   8.19957688,
                 11.08009573,  23.        ,  12.71096417,  10.09637761])
```

```
In [50]: flattened.shape
Out[50]: (12,)
```

4.6.3 Dimension collapse

Another common way that leads to reshaping is projection. Let's consider again our `normal_array` :

```
In [51]: normal_array
Out[51]: array([[ 12.99205086,   7.7157832 , 100.        ,  50.        ],
               [  9.19391119,   7.92142871,  13.31222213,   8.19957688],
               [ 11.08009573,  23.        ,  12.71096417,  10.09637761]])
```

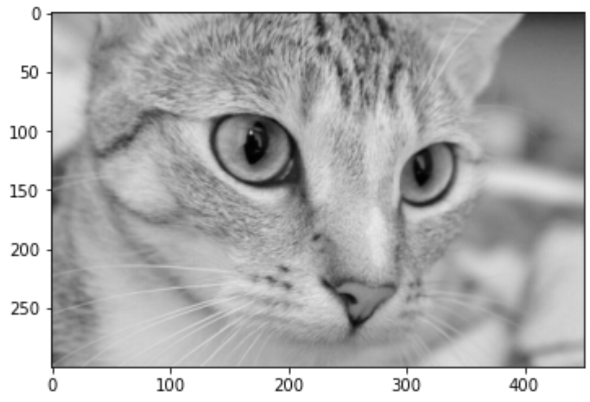
We can project all values along the first or second axis, to recover for each row/column the largest value:

```
In [52]: proj0 = np.max(normal_array, axis = 0)
         proj0
Out[52]: array([ 12.99205086,  23.        , 100.        ,  50.        ])
```

```
In [53]: proj0.shape
Out[53]: (4,)
```

We see that our projected array has lost a dimension, the one along which we performed the projection. With the image, we could project all channels along the third dimension:

```
In [54]: plt.imshow(image.max(axis=2));
```



4.6.4 Swaping dimensions

We can also simply exchange the position of dimensions. This can be achieved in different ways. For example we can `np.roll` dimensions, i.e. circularly shift dimensions. This conserves the relative oder of all axes:

```
In [55]: array3D = np.ones((4, 10, 20))
         array3D.shape

Out[55]: (4, 10, 20)
```

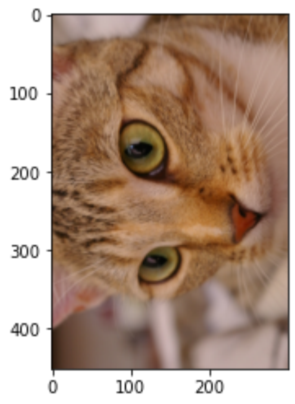
```
In [56]: array_rolled = np.rollaxis(array3D, axis=1, start=0)
         array_rolled.shape

Out[56]: (10, 4, 20)
```

Alternatively you can swap two axes. This doesn't preserver their relative positions:

```
In [57]: array_swapped = np.swapaxes(array3D, 0,2)
         array_swapped.shape

Out[57]: (20, 10, 4)
```

With the image, we can for example swap the two first axes:

```
In [58]: plt.imshow(np.swapaxes(image, 0, 1));
```



4.6.5 Change positions

Finally, we can also change the position of elements without changing the shape of the array. For example if we have an array with two columns, we can swap them:

```
In [59]: array2D = np.random.normal(0,1,(4,2))
         array2D
```

```
Out[59]: array([[ 1.69380702,  0.45317243],
                [ 0.97985485, -1.10186616],
                [ 2.16001609,  0.29160533],
                [-0.29204481, -0.80523649]])
```

```
In [60]: np.fliplr(array2D)
```

```
Out[60]: array([[ 0.45317243,  1.69380702],
                [-1.10186616,  0.97985485],
                [ 0.29160533,  2.16001609],
                [-0.80523649, -0.29204481]])
```

Similarly, if we have two rows:

```
In [61]: array2D = np.random.normal(0,1,(2,4))
         array2D
```

```
Out[61]: array([[-0.00285876,  0.76241924,  1.18546015, -0.13881594],
                [-1.42554951,  0.36561497,  0.73252833, -1.43307846]])
```

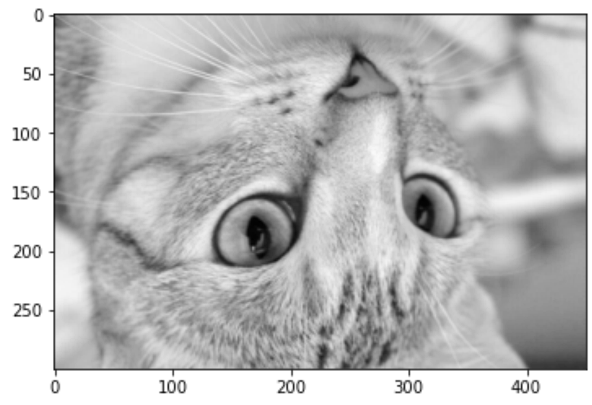
```
In [62]: np.flipud(array2D)
```

```
Out[62]: array([[-1.42554951,  0.36561497,  0.73252833, -1.43307846],
                [-0.00285876,  0.76241924,  1.18546015, -0.13881594]])
```

For more complex cases you can also use the more general `np.flip()` function.

With the image, flipping a dimension just mirrors the picture. To do that we select a single channel:

```
In [63]: plt.imshow(np.flipud(image[:,:,0]));
```



5. Combining arrays

We have already seen how to create arrays and how to modify their dimensions. One last operation we can do is to combine multiple arrays. There are two ways to do that: by assembling arrays of same dimensions (concatenation, stacking etc.) or by combining arrays of different dimensions using *broadcasting*. Like in the previous chapter, we illustrate with small arrays and a real image.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import skimage
plt.gray();
image = skimage.data.chelsea()
```

```
<Figure size 432x288 with 0 Axes>
```

5.1 Arrays of same dimensions

Let's start by creating a few two 2D arrays:

In [2]:
```
array1 = np.ones((10,5))
array2 = 2*np.ones((10,3))
array3 = 3*np.ones((10,5))
```

5.1.1 Concatenation

The first operation we can perform is concatenation, i.e. assembling the two 2D arrays into a larger 2D array. Of course we have to be careful with the size of each dimension. For example if we try to concatenate `array1` and `array2` along the first dimension, we get:

In [3]:
```
np.concatenate([array1, array2])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-580de54a6ac0> in <module>
----> 1 np.concatenate([array1, array2])

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must ma
tch exactly, but along dimension 1, the array at index 0 has size 5 and the a
rray at index 1 has size 3
```

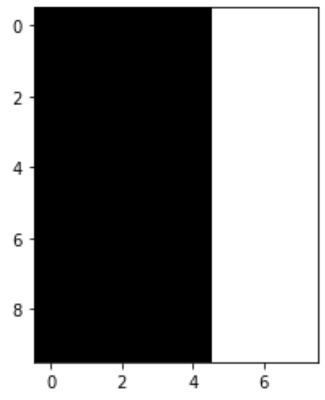
Both array have 10 lines, but one has 3 and the other 5 columns. We can therefore only concatenate them along the second dimensions:

In [4]:
```
array_conc = np.concatenate([array1, array2], axis = 1)
```

In [5]:
```
array_conc.shape
```

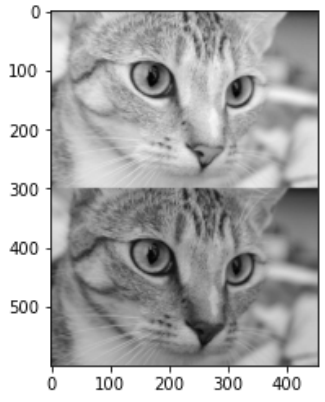
Out[5]:
```
(10, 8)
```

In [6]:
```
plt.imshow(array_conc, cmap = 'gray');
```

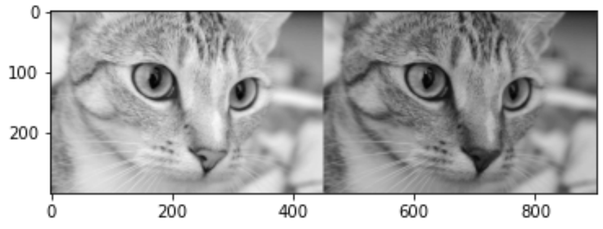


If we now use our example of real image, we can for example concatenate the two first channels of our RGB image:

In [7]:
```
plt.imshow(np.concatenate([image[:,:,0], image[:,:,1]]));
```



In [8]:
```
plt.imshow(np.concatenate([image[:,:,0], image[:,:,1]], axis=1));
```



5.1.2 Stacking

If we have several arrays with exact same sizes, we can also *stack* them, i.e. assemble them along a *new* dimension. For example we can create a 3D stack out of two 2D arrays:

In [9]:
```
array_stack = np.stack([array1, array3])
```

In [10]:
```
array_stack.shape
```

Out[10]:
```
(2, 10, 5)
```

We can select the dimension along which to stack, again by using the `axis` keyword. For example if we want our new dimensions to be the *third* axis we can write:

```
In [11]: array_stack = np.stack([array1, array3], axis = 2)
```

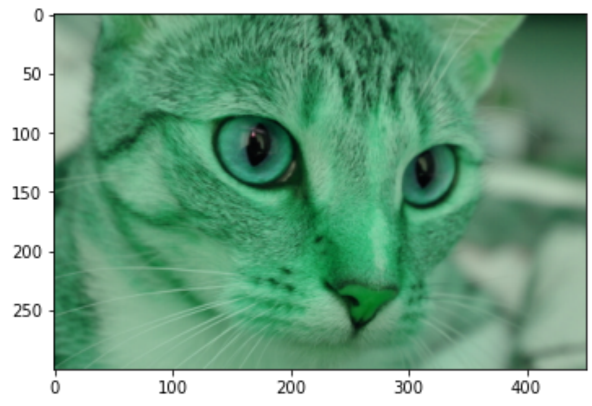
```
In [12]: array_stack.shape
```

```
Out[12]: (10, 5, 2)
```

With our real image, we can for example stack the different channels in a new order (note that one could do that easily with `np.swapaxis`):

```
In [13]: image_stack = np.stack([image[:,:,2], image[:,:,0], image[:,:,1]], axis=2)
```

```
In [14]: plt.imshow(image_stack);
```



As we placed the red channel, which has the highest intensity, at the position of the green one (second position) our image now is dominated by green tones.

5.2 Arrays of different dimensions

5.2.1 Broadcasting

Numpy has a powerful feature called **broadcasting**. This is the feature that for example allows you to write:

```
In [15]: 2 * array1
```

```
Out[15]: array([[2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.],
               [2., 2., 2., 2., 2.]])
```

Here we just combined a single number with an array and Numpy *re-used* or *broadcasted* the element with less dimensions (the number 2) across the entire `array1` . This does not only work with single numbers but also with arrays of different dimensions. Broadcasting can become very complex, so we limit ourselves here to a few common examples.

The general rule is that in an operation with arrays of different dimensions, **missing dimensions** or **dimensions of size 1** get *repeated* to create two arrays of same size. Note that comparisons of dimension size start from the **last** dimensions. For example if we have a 1D array and a 2D array:

```
In [16]: array1D = np.arange(4)
         array1D

Out[16]: array([0, 1, 2, 3])
```

```
In [17]: array2D = np.ones((6,4))
         array2D

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

```
In [18]: array1D * array2D

Out[18]: array([[0., 1., 2., 3.],
                [0., 1., 2., 3.],
                [0., 1., 2., 3.],
                [0., 1., 2., 3.],
                [0., 1., 2., 3.],
                [0., 1., 2., 3.]])
```

Here `array1D` which has a *single line* got *broadcasted* over *each line* of the 2D array `array2D` . Note the the size of each dimension is important. If `array1D` had for example more columns, that broadcasting could not work:

```
In [19]: array1D = np.arange(3)
         array1D

Out[19]: array([0, 1, 2])
```

```
In [20]: array1D * array2D

         ---------------------------------------------------------------------------
         ValueError                                Traceback (most recent call last)
         <ipython-input-20-30434b67efb8> in <module>
         ----> 1 array1D * array2D

         ValueError: operands could not be broadcast together with shapes (3,) (6,4)
```

As mentioned above, dimension sizes comparison start from the last dimension, so for example if `array1D` had a length of 6, like the first dimension of `array2D` , broadcasting would fail:

```
In [21]: array1D = np.arange(6)
         array1D.shape

Out[21]: (6,)
```

```
In [22]: array2D.shape
Out[22]: (6, 4)
```

```
In [23]: array1D * array2D
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-30434b67efb8> in <module>
----> 1 array1D * array2D

ValueError: operands could not be broadcast together with shapes (6,) (6,4)
```

5.2.2 Higher dimensions

Broadcasting can be done in higher dimensional cases. Imagine for example that you have an RGB image with dimensions $NxMx3$. If you want to modify each channel independently, for example to rescale them, you can use broadcasting. We can use again our real image:

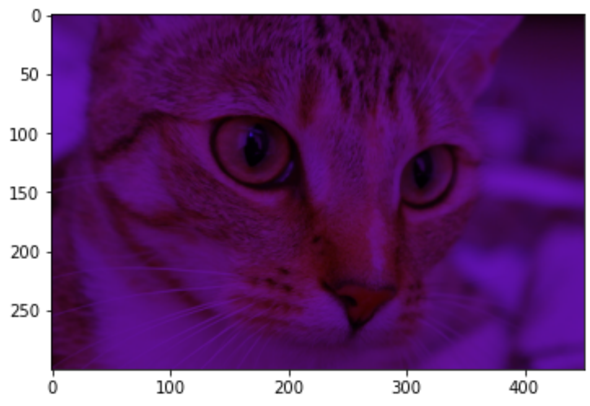
```
In [24]: image.shape
Out[24]: (300, 451, 3)
```

```
In [25]: scale_factor = np.array([0.5, 0.1, 1])
         scale_factor
Out[25]: array([0.5, 0.1, 1. ])
```

In [26]:
```
rescaled_image = scale_factor * image
rescaled_image
```

Out[26]:
```
array([[[ 71.5,  12. , 104. ],
        [ 71.5,  12. , 104. ],
        [ 70.5,  11.8, 102. ],
        ...,
        [ 22.5,   2.7,  13. ],
        [ 22.5,   2.7,  13. ],
        [ 22.5,   2.7,  13. ]],

       [[ 73. ,  12.3, 107. ],
        [ 72.5,  12.2, 106. ],
        [ 71.5,  12. , 104. ],
        ...,
        [ 23. ,   2.9,  13. ],
        [ 22.5,   2.9,  13. ],
        [ 23.5,   3. ,  14. ]],

       [[ 74. ,  12.6, 112. ],
        [ 73.5,  12.5, 111. ],
        [ 73. ,  12.2, 109. ],
        ...,
        [ 24. ,   2.8,  17. ],
        [ 24.5,   2.9,  18. ],
        [ 25. ,   3. ,  19. ]],

       ...,

       [[ 46. ,   5.8,  30. ],
        [ 52.5,   7.1,  43. ],
        [ 66. ,   9.8,  71. ],
        ...,
        [ 86. ,  14.5, 138. ],
        [ 86. ,  14.5, 138. ],
        [ 86. ,  14.5, 138. ]],

       [[ 64. ,   9.2,  60. ],
        [ 69.5,  10.3,  71. ],
        [ 67. ,   9.5,  64. ],
        ...,
        [ 83. ,  14.2, 132. ],
        [ 83. ,  14.2, 132. ],
        [ 83.5,  14.3, 133. ]],

       [[ 69.5,  10.3,  71. ],
        [ 63.5,   8.8,  57. ],
        [ 62.5,   8.6,  53. ],
        ...,
        [ 80.5,  13.7, 127. ],
        [ 80.5,  13.7, 127. ],
        [ 81. ,  13.8, 128. ]]])
```

```
In [27]: plt.imshow(rescaled_image.astype(int))
Out[27]: <matplotlib.image.AxesImage at 0x11eabbcd0>
```



Note that if we the image has the dimensions $3xNxM$ (RGB planes in the first dimension), we encounter the same problem as before: a mismatch in size for the **last** dimension:

```
In [28]: image2 = np.rollaxis(image, axis=2)
         image2.shape
Out[28]: (3, 300, 451)

In [29]: scale_factor.shape
Out[29]: (3,)

In [30]: scale_factor * image2

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-30-7a7267773c9f> in <module>
----> 1 scale_factor * image2

ValueError: operands could not be broadcast together with shapes (3,) (3,300,
451)
```

5.2.3 Adding axes

As seen above, if we have a mismatch in dimension size, the broadcasting mechanism doesn't work. To salvage such cases, we still have the possibility to *add* empty axes in an array to restore the matching of the non-empty dimension.

In the above example our arrays have the following shapes:

```
In [31]: image2.shape
Out[31]: (3, 300, 451)

In [32]: scale_factor.shape
Out[32]: (3,)
```

So we need to add two "empty" axes after the single dimension of `scale_factor` :

```
In [33]: scale_factor_corr = scale_factor[:, np.newaxis, np.newaxis]
```

```
In [34]: scale_factor_corr.shape
```

```
Out[34]: (3, 1, 1)
```

```
In [35]: image2_rescaled = scale_factor_corr * image2
```

6. Pandas Introduction

In the previous chapters, we have learned how to handle Numpy arrays that can be used to efficiently perform numerical calculations. Those arrays are however homogeneous structures i.e. they can only contain one type of data. Also, even if we have a single type of data, the different rows or columns of an array do not have labels, making it difficult to track what they contain. For such cases, we need a structure closer to a table as can be found in Excel, and these structures are implemented by the package Pandas.

But why can't we simply use Excel then? While Excel is practical to browse through data, it is very cumbersome to use to combine, re-arrange and thoroughly analyze data: code is hidden and difficult to share, there's no version control, it's difficult to automate tasks, the manual clicking around leads to mistakes etc.

In the next chapters, you will learn how to handle tabular data with Pandas, a Python package widely used in the scientific and data science areas. You will learn how to create and import tables, how to combine them, modify them, do statistical analysis on them and finally how to use them to easily create complex visualizations.

So that you see where this leads, we start with a short example of how Pandas can be used in a project. We look here at data provided openly by the Swiss National Science Foundation about grants attributed since 1975.

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
```

6.1 Importing data

Before anything, we need access to the data that can be found here (https://opendata.swiss/de/dataset/p3-export-projects-people-and-publications). We can either manually download them and then use the path to read the data or directly use the url. The latter has the advantage that if you have an evolving source of data, these will always be up to date:

In [2]:
```
# local import
projects = pd.read_csv('Data/P3_GrantExport.csv',sep = ';')

# import from url
#projects = pd.read_csv('http://p3.snf.ch/P3Export/P3_GrantExport.csv',sep =
';')
```

Then we can have a brief look at the table itself that Jupyter displays in a formated way and limit the view to the 5 first rows using `head()` :

```
In [3]: projects.head(5)
```

Out[3]:

	Project Number	Project Number String	Project Title	Project Title English	Responsible Applicant	Funding Instrument	Funding Instrument Hierarchy	
0	1	1000-000001	Schlussband (Bd. VI) der Jacob Burckhardt-Biog...	NaN	Kaegi Werner	Project funding (Div. I-III)	Project funding	
1	4	1000-000004	Batterie de tests à l'usage des enseignants po...	NaN	Massarenti Léonard	Project funding (Div. I-III)	Project funding	Psych Scier
2	5	1000-000005	Kritische Erstausgabe der 'Evidentiae contra D...	NaN	Kommission für das Corpus philosophorum medii ...	Project funding (Div. I-III)	Project funding	Komm philoso
3	6	1000-000006	Katalog der datierten Handschriften in der Sch...	NaN	Burckhardt Max	Project funding (Div. I-III)	Project funding	Hanc Alte Dr
4	7	1000-000007	Wissenschaftliche Mitarbeit am Thesaurus Lingu...	NaN	Schweiz. Thesauruskommission	Project funding (Div. I-III)	Project funding	Thesauru

6.2 Exploring data

Pandas offers a variety of tools to compile information about data, and that compilation can be done very efficiently without the need for loops, conditionals etc.

For example we can quickly count how many times each University appear in that table. We just use the `value_counts()` method for that:

```
In [4]: projects['University'].value_counts().head(10)
```

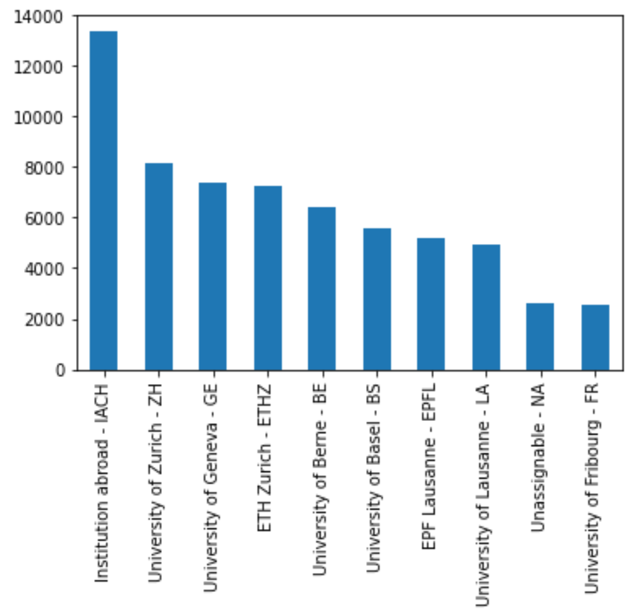
```
Out[4]: Institution abroad - IACH      13348
        University of Zurich - ZH       8170
        University of Geneva - GE       7385
        ETH Zurich - ETHZ               7278
        University of Berne - BE        6445
        University of Basel - BS        5560
        EPF Lausanne - EPFL             5174
        University of Lausanne - LA     4944
        Unassignable - NA               2642
        University of Fribourg - FR     2535
        Name: University, dtype: int64
```

Then we can very easily plot the resulting information, either using directly Pandas or a more advanced library like Seaborn, plotnine or Altair.

Here first with plain Pandas (using Matplotlib under the hood):

```
In [5]: projects['University'].value_counts().head(10).plot(kind='bar')
```

```
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x104df7040>
```



6.3 Handling different data types

Unlike Numpy arrays, Pandas can handle a variety of different data types in a dataframe. For example it is very efficient at dealing with dates. We see that our table contains e.g. a `Start Date`. We can turn this string into an actual date:

```
In [6]: projects['start'] = pd.to_datetime(projects['Start Date'])
        projects['year'] = projects.start.apply(lambda x: x.year)
```

```
In [7]: projects.loc[0].start
```

```
Out[7]: Timestamp('1975-01-10 00:00:00')
```

```
In [8]: projects.loc[0].year
```

```
Out[8]: 1975.0
```

6.4 Data wrangling, aggregation and statistics

Pandas is very efficient at wrangling and aggregating data, i.e. grouping several elements of a table to calculate statistics on them. For example we first need here to convert the `Approved Amount` to a numeric value. Certain rows contain text (e.g. "not applicable") and we force the conversion:

```
In [9]: projects['Approved Amount'] = pd.to_numeric(projects['Approved Amount'], err
        ors = 'coerce')
```

Then we want to extract the type of filed without subfields e.g. "Humanities" instead of "Humanities and Social Sciences;Theology & religion". For that we can create a custom function and apply it to an entire column:

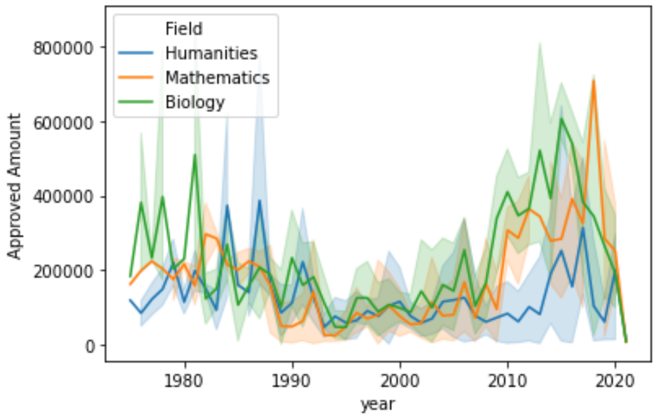
In [10]:
```
science_types = ['Humanities', 'Mathematics','Biology']
projects['Field'] = projects['Discipline Name Hierarchy'].apply(
    lambda el: next((y for y in [x for x in science_types if x in el] if y i
s not None),None) if not pd.isna(el) else el)
```

Then we group the data by discipline and year, and calculate the mean of each group:

In [11]:
```
aggregated = projects.groupby(['Institution Country', 'year','Field'], as_in
dex=False).mean()
```

Finally we can use Seaborn to plot the data by "Field" using just keywords to indicate what the axes and colours should mean (following some principles of the grammar of graphics):

In [12]:
```
sns.lineplot(data = aggregated, x = 'year', y='Approved Amount', hue='Field
');
```



Note that here, axis labelling, colorouring, legend, interval of confidence have been done automatically based on the content of the dataframe.

We see a drastic augmentation around 2010: let's have a closer look. We can here again group data by year and funding type and calculate the total funding:

In [13]:
```
grouped = projects.groupby(['year','Funding Instrument Hierarchy']).agg(
    total_sum=pd.NamedAgg(column='Approved Amount', aggfunc='sum')).reset_in
dex()
```

```
In [14]: grouped
```

```
Out[14]:
```

	year	Funding Instrument Hierarchy	total_sum
0	1975.0	Project funding	32124534.0
1	1975.0	Science communication	44600.0
2	1976.0	Programmes;National Research Programmes (NRPs)	268812.0
3	1976.0	Project funding	96620284.0
4	1976.0	Science communication	126939.0
...	...	...	...
378	2020.0	Programmes;r4d (Swiss Programme for Research o...	195910.0
379	2020.0	Project funding	193568294.0
380	2020.0	Project funding;Project funding (special)	19239681.0
381	2020.0	Science communication	3451740.0
382	2021.0	Science communication	55200.0

383 rows × 3 columns

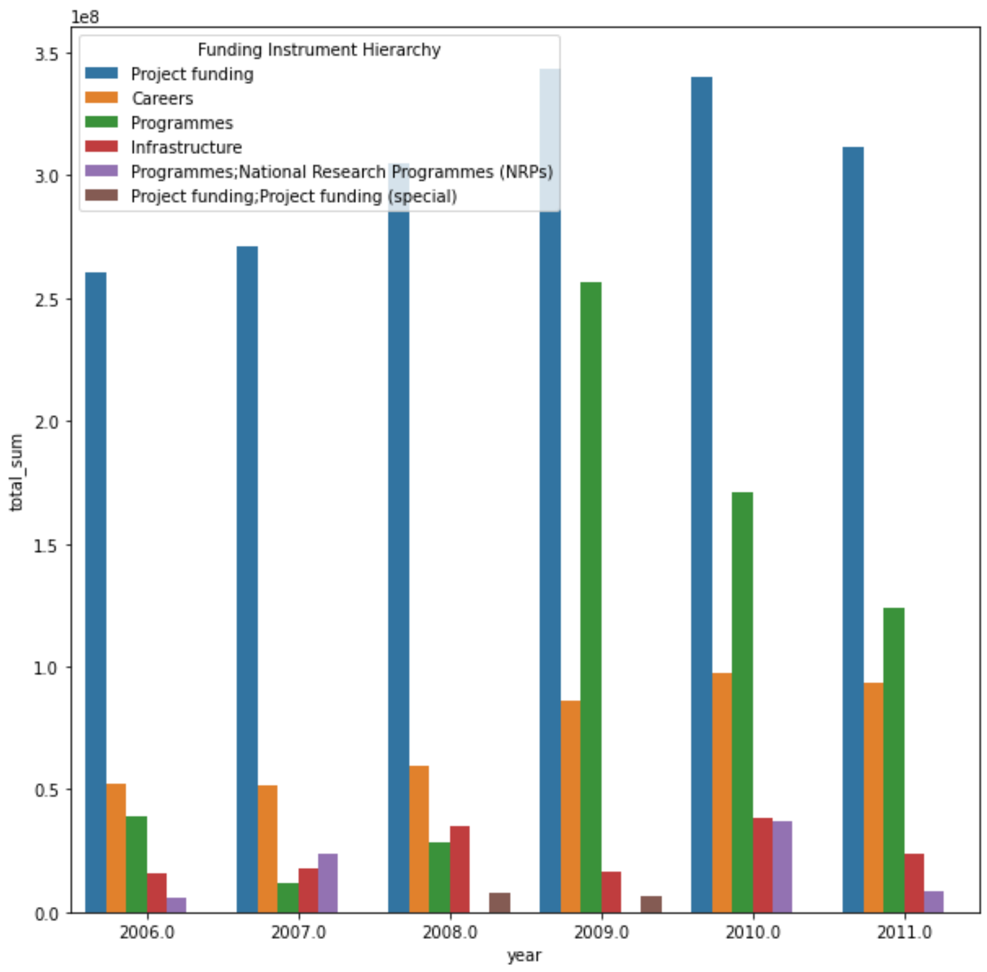
Now, for each year we keep only the 5 largest funding types to be able to plot them:

```
In [15]: group_sorted = grouped.groupby('year',as_index=False).apply(lambda x: (x.gro
         upby('Funding Instrument Hierarchy')
                                                       .sum()
                                                       .sort_values('total_sum', ascending=Fa
         lse))
                                                     .head(5)).reset_index()
```

Finally, we only keep year in the 2000's:

```
In [16]: instruments_by_year = group_sorted[(group_sorted.year > 2005) & (group_sorte
         d.year < 2012)]
```

```
In [17]: import matplotlib.pyplot as plt
         plt.figure(figsize=(10,10))
         sns.barplot(data=instruments_by_year,
                     x='year', y='total_sum', hue='Funding Instrument Hierarchy')

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x105e35670>
```



We see that the main change, is the sudden increase in funding for national research programs.

```
In [ ]:
```

7. Pandas objects

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Python has a series of data containers (list, dicts etc.) and Numpy offers multi-dimensional arrays, however none of these structures offers a simple way neither to handle tabular data, nor to easily do standard database operations. This is why Pandas exists: it offers a complete ecosystem of structures and functions dedicated to handle large tables with inhomogeneous contents.

In this first chapter, we are going to learn about the two main structures of Pandas: Series and Dataframes.

7.1 Series

7.1.1 Simple series

Series are a the Pandas version of 1-D Numpy arrays. We are rarely going to use them directly, but they often appear implicitly when handling data from the more general Dataframe structure. We therefore only give here basics.

To understand Series' specificities, let's create one. Usually Pandas structures (Series and Dataframes) are created from other simpler structures like Numpy arrays or dictionaries:

In [2]:
```
numpy_array = np.array([4,8,38,1,6])
```

The function `pd.Series()` allows us to convert objects into Series:

In [3]:
```
pd_series = pd.Series(numpy_array)
pd_series
```

Out[3]:
```
0     4
1     8
2    38
3     1
4     6
dtype: int64
```

The underlying structure can be recovered with the `.values` attribute:

In [4]:
```
pd_series.values
```

Out[4]:
```
array([ 4,  8, 38,  1,  6])
```

Otherwise, indexing works as for regular arrays:

In [5]:
```
pd_series[1]
```

Out[5]:
```
8
```

7.1.2 Indexing

On top of accessing values in a series by regular indexing, one can create custom indices for each element in the series:

```
In [6]: pd_series2 = pd.Series(numpy_array, index=['a', 'b', 'c', 'd','e'])
```

```
In [7]: pd_series2
```

```
Out[7]: a     4
        b     8
        c    38
        d     1
        e     6
        dtype: int64
```

Now a given element can be accessed either by using its regular index:

```
In [8]: pd_series2[1]
```

```
Out[8]: 8
```

or its chosen index:

```
In [9]: pd_series2['b']
```

```
Out[9]: 8
```

A more direct way to create specific indexes is to transform as dictionary into a Series:

```
In [10]: composer_birth = {'Mahler': 1860, 'Beethoven': 1770, 'Puccini': 1858, 'Shost
         akovich': 1906}
```

```
In [11]: pd_composer_birth = pd.Series(composer_birth)
         pd_composer_birth
```

```
Out[11]: Mahler         1860
         Beethoven      1770
         Puccini        1858
         Shostakovich   1906
         dtype: int64
```

```
In [12]: pd_composer_birth['Puccini']
```

```
Out[12]: 1858
```

7.2 Dataframes

In most cases, one has to deal with more than just one variable, e.g. one has the birth year and the death year of a list of composers. Also one might have different types of information, e.g. in addition to numerical variables (year) one might have string variables like the city of birth. The Pandas structure that allow one to deal with such complex data is called a Dataframe, which can somehow be seen as an aggregation of Series with a common index.

7.2.1 Creating a Dataframe

To see how to construct such a Dataframe, let's create some more information about composers:

In [13]:
```
composer_death = pd.Series({'Mahler': 1911, 'Beethoven': 1827, 'Puccini': 19
24, 'Shostakovich': 1975})
composer_city_birth = pd.Series({'Mahler': 'Kaliste', 'Beethoven': 'Bonn', '
Puccini': 'Lucques', 'Shostakovich': 'Saint-Petersburg'})
```

Now we can combine multiple series into a Dataframe by precising a variable name for each series. Note that all our series need to have the same indices (here the composers' name):

In [14]:
```
composers_df = pd.DataFrame({'birth': pd_composer_birth, 'death': composer_d
eath, 'city': composer_city_birth})
composers_df
```

Out[14]:
```
              birth  death              city
Mahler         1860   1911           Kaliste
Beethoven      1770   1827              Bonn
Puccini        1858   1924           Lucques
Shostakovich   1906   1975  Saint-Petersburg
```

A more common way of creating a Dataframe is to construct it directly from a dictionary of lists where each element of the dictionary turns into a column:

In [15]:
```
dict_of_list = {'birth': [1860, 1770, 1858, 1906], 'death':[1911, 1827, 192
4, 1975],
 'city':['Kaliste', 'Bonn', 'Lucques', 'Saint-Petersburg']}
```

In [16]:
```
pd.DataFrame(dict_of_list)
```

Out[16]:
```
   birth  death              city
0   1860   1911           Kaliste
1   1770   1827              Bonn
2   1858   1924           Lucques
3   1906   1975  Saint-Petersburg
```

However we now lost the composers name. We can enforce it by providing, as we did before for the Series, a list of indices:

```
In [17]: pd.DataFrame(dict_of_list, index=['Mahler', 'Beethoven', 'Puccini', 'Shostak
         ovich'])
```

Out[17]:

	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

7.2.2 Accessing values

There are multiple ways of accessing values or series of values in a Dataframe. Unlike in Series, a simple bracket gives access to a column and not an index, for example:

```
In [18]: composers_df['city']
```

```
Out[18]: Mahler                    Kaliste
         Beethoven                    Bonn
         Puccini                   Lucques
         Shostakovich     Saint-Petersburg
         Name: city, dtype: object
```

returns a Series. Alternatively one can also use the *attributes* synthax and access columns by using:

```
In [19]: composers_df.city
```

```
Out[19]: Mahler                    Kaliste
         Beethoven                    Bonn
         Puccini                   Lucques
         Shostakovich     Saint-Petersburg
         Name: city, dtype: object
```

The attributes synthax has some limitations, so in case something does not work as expected, revert to the brackets notation.

When specifiying multiple columns, a DataFrame is returned:

```
In [20]: composers_df[['city', 'birth']]
```

Out[20]:

	city	birth
Mahler	Kaliste	1860
Beethoven	Bonn	1770
Puccini	Lucques	1858
Shostakovich	Saint-Petersburg	1906

One of the important differences with a regular Numpy array is that here, regular indexing doesn't work:

```
In [21]: #composers_df[0,0]
```

Instead one has to use either the `.iloc[]` or the `.loc[]` method. `.iloc[]` can be used to recover the regular indexing:

In [22]:
```
composers_df.iloc[0,1]
```

Out[22]:
```
1911
```

While `.loc[]` allows one to recover elements by using the **explicit** index, on our case the composers name:

In [23]:
```
composers_df.loc['Mahler','death']
```

Out[23]:
```
1911
```

Remember that `loc` and ``iloc``` use brackets [] and not parenthesis ().

Numpy style indexing works here too

In [24]:
```
composers_df.iloc[1:3,:]
```

Out[24]:

	birth	death	city
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques

If you are working with a large table, it might be useful to sometimes have a list of all the columns. This is given by the `.keys()` attribute:

In [25]:
```
composers_df.keys()
```

Out[25]:
```
Index(['birth', 'death', 'city'], dtype='object')
```

7.2.3 Adding columns

It is very simple to add a column to a Dataframe. One can e.g. just create a column a give it a default value that we can change later:

In [26]:
```
composers_df['country'] = 'default'
```

In [27]:
```
composers_df
```

Out[27]:

	birth	death	city	country
Mahler	1860	1911	Kaliste	default
Beethoven	1770	1827	Bonn	default
Puccini	1858	1924	Lucques	default
Shostakovich	1906	1975	Saint-Petersburg	default

Or one can use an existing list:

In [28]:
```
country = ['Austria','Germany','Italy','Russia']
```

In [29]:
```
composers_df['country2'] = country
```

In [30]:
```
composers_df
```

Out[30]:

	birth	death	city	country	country2
Mahler	1860	1911	Kaliste	default	Austria
Beethoven	1770	1827	Bonn	default	Germany
Puccini	1858	1924	Lucques	default	Italy
Shostakovich	1906	1975	Saint-Petersburg	default	Russia

8. Importing/export, basic plotting

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

We have seen in the previous chapter what structures are offered by Pandas and how to create them. Another very common way of "creating" a Pandas Dataframe is by importing a table from another format like CSV or Excel.

8.1 Simple import

An Excel table containing the same information as we had in Chapter 1 (01-Pandas_structures.ipynb) is provided in composers.xlsx (composers.xlsx) and can be read with the `read_excel` function. There are many more readers for other types of data (csv, json, html etc.) but we focus here on Excel.

In [2]:
```
pd.read_excel('Data/composers.xlsx')
```

Out[2]:

	composer	birth	death	city
0	Mahler	1860	1911	Kaliste
1	Beethoven	1770	1827	Bonn
2	Puccini	1858	1924	Lucques
3	Shostakovich	1906	1975	Saint-Petersburg

The reader automatically recognized the heaers of the file. However it created a new index. If needed we can specify which column to use as header:

In [3]:
```
pd.read_excel('Data/composers.xlsx', index_col = 'composer')
```

Out[3]:

	birth	death	city
composer			
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

If we open the file in Excel, we see that it is composed of more than one sheet. Clearly, when not specifying anything, the reader only reads the first sheet. However we can specify a sheet:

In [4]:
```
specific_sheet = pd.read_excel('Data/composers.xlsx', index_col = 'composer
',sheet_name='Sheet2')
```

In [5]:
```
specific_sheet
```

Out[5]:

	birth	death	city
composer			
Mahler	1860.0	1911	Kaliste
Beethoven	1770.0	1827	Bonn
Puccini	1858.0	1924	Lucques
Shostakovich	1906.0	1975	Saint-Petersburg
Sibelius	10.0	unknown	unknown
Haydn	NaN	NaN	Röhrau

For each reader, there is a long list of options to specify how the file should be read. We can see all these options using the help (see below). Imagine that our tables contains a title and unnecessary rows: we can use the `skiprows` argument. Imagine you have dates in your table: you can use the `date_parser` argument to specify how to format them etc.

In [6]:
```
#use shift+tab within the parenthesis to see optional arguemnts
#pd.read_excel()
```

8.2 Handling unknown values

As you can see above, some information is missing. Some missing values are marked as "unknown" while other are NaN. NaN is the standard symbol for unknown/missing values and is understood by Pandas while "unknown" is just seen as text. This is impractical as now we have e.g. columns with a mix of numbers and text which will make later computations difficult. What we would like to do is to replace all "irrelevant" values with the standard NaN symbol that says "no information".

Let's first do a regular import:

In [7]:
```
import1 = pd.read_excel('Data/composers.xlsx', index_col = 'composer',sheet_
name='Sheet2')
import1
```

Out[7]:

	birth	death	city
composer			
Mahler	1860.0	1911	Kaliste
Beethoven	1770.0	1827	Bonn
Puccini	1858.0	1924	Lucques
Shostakovich	1906.0	1975	Saint-Petersburg
Sibelius	10.0	unknown	unknown
Haydn	NaN	NaN	Röhrau

If we look now at one column, we can see that columns have been imported in different ways. One column is an object, i.e. mixed types, the other contains floats:

In [8]:
```
import1.birth
```

Out[8]:
```
composer
Mahler         1860.0
Beethoven      1770.0
Puccini        1858.0
Shostakovich   1906.0
Sibelius         10.0
Haydn             NaN
Name: birth, dtype: float64
```

In [9]:
```
import1.death
```

Out[9]:
```
composer
Mahler             1911
Beethoven          1827
Puccini            1924
Shostakovich       1975
Sibelius        unknown
Haydn               NaN
Name: death, dtype: object
```

If we want to do calculations, for example getting summary information using `describe()` we have a problem: the `death` column is skipped because no calculation can be done with strings:

In [10]:
```
import1.describe()
```

Out[10]:

	birth
count	5.000000
mean	1480.800000
std	823.674207
min	10.000000
25%	1770.000000
50%	1858.000000
75%	1860.000000
max	1906.000000

Now we specify that 'unknown' should be a NaN value:

In [11]:
```
import2 = pd.read_excel('Data/composers.xlsx', index_col = 'composer',
                        sheet_name='Sheet2', na_values=['unknown'])
import2
```

Out[11]:

	birth	death	city
composer			
Mahler	1860.0	1911.0	Kaliste
Beethoven	1770.0	1827.0	Bonn
Puccini	1858.0	1924.0	Lucques
Shostakovich	1906.0	1975.0	Saint-Petersburg
Sibelius	10.0	NaN	NaN
Haydn	NaN	NaN	Röhrau

And now computations are again possible, as Pandas knows how to deal with NaNs:

```
In [12]: import2.describe()
```

Out[12]:

	birth	death
count	5.000000	4.000000
mean	1480.800000	1909.250000
std	823.674207	61.396933
min	10.000000	1827.000000
25%	1770.000000	1890.000000
50%	1858.000000	1917.500000
75%	1860.000000	1936.750000
max	1906.000000	1975.000000

Handling bad or missing values is a very important part of data science. Taking care of the most common occurrences at import is a good solution.

8.3 Column types

We see above that the birth column has been "classified" as a float. However we know that this is not the case, it's just an integer. Here again, we can specify the column type already at import time using the dtype option and a dictionary:

```
In [13]: import2 = pd.read_excel('Data/composers.xlsx', index_col = 'composer',sheet_
         name='Sheet1', na_values=['unknown'],
                                  dtype={'composer':np.str,'birth':np.int32,'death':np.
         int32,'city':np.str})
```

```
In [14]: import2.birth
```

```
Out[14]: composer
         Mahler          1860
         Beethoven       1770
         Puccini         1858
         Shostakovich    1906
         Name: birth, dtype: int32
```

8.4 Modifications after import

Of course we don't have to do all these adjustement at import time. We can also do a default import and check what has to be corrected afterward.

8.4.1 Create NaNs

If we missed some bad values at import we can just replace all those directly in the dataframe. We can achieve that by using the `replace()` method and specifying what should be replaced:

```
In [15]: import1
```

Out[15]:

	birth	death	city
composer			
Mahler	1860.0	1911	Kaliste
Beethoven	1770.0	1827	Bonn
Puccini	1858.0	1924	Lucques
Shostakovich	1906.0	1975	Saint-Petersburg
Sibelius	10.0	unknown	unknown
Haydn	NaN	NaN	Röhrau

```
In [16]: import_nans = import1.replace('unknown', np.nan)
         import_nans.birth
```

```
Out[16]: composer
         Mahler          1860.0
         Beethoven       1770.0
         Puccini         1858.0
         Shostakovich    1906.0
         Sibelius          10.0
         Haydn              NaN
         Name: birth, dtype: float64
```

Note that when we fix "bad" values, e.g. here the "unknown" text value with NaNs, Pandas automatically adjust the type of the column, allowing us for exampel to later do mathemtical operations.

```
In [17]: import1.death.dtype
```

```
Out[17]: dtype('O')
```

```
In [18]: import_nans.death.dtype
```

```
Out[18]: dtype('float64')
```

8.4.2 Changing the type

We can also change the type of a column on an existing Dataframe with the same command as in Numpy:

```
In [19]: import2.birth
```

```
Out[19]: composer
         Mahler          1860
         Beethoven       1770
         Puccini         1858
         Shostakovich    1906
         Name: birth, dtype: int32
```

```
In [20]: import2.birth.astype('float')
```

```
Out[20]: composer
         Mahler          1860.0
         Beethoven       1770.0
         Puccini         1858.0
         Shostakovich    1906.0
         Name: birth, dtype: float64
```

If we look again at import2:

```
In [21]: import2.birth
```

```
Out[21]: composer
         Mahler          1860
         Beethoven       1770
         Puccini         1858
         Shostakovich    1906
         Name: birth, dtype: int32
```

we see that we didn't actually change the type. Changes on a Dataframe are only effective if we reassign the column:

```
In [22]: import2.birth = import2.birth.astype('float')
```

```
In [23]: import2.birth
```

```
Out[23]: composer
         Mahler          1860.0
         Beethoven       1770.0
         Puccini         1858.0
         Shostakovich    1906.0
         Name: birth, dtype: float64
```

8.5 Export

You can easily export a Dataframe that you worked on. Most commonly you will export it in a common format like CSV:

```
In [24]: import2.to_csv('mydataframe.csv')
```

If you have a complex dataframe that e.g. contains lists, you can save it as a *pickle* object, a specific Python format that allows one to save complex data:

```
In [25]: import2.to_pickle('Data/my_dataframe.pkl')
```

You can reload this type of data via the pickle loading function of Pandas:

```
In [26]: import3 = pd.read_pickle('Data/my_dataframe.pkl')
```

In [27]:
```
import3
```

Out[27]:

	birth	death	city
composer			
Mahler	1860.0	1911	Kaliste
Beethoven	1770.0	1827	Bonn
Puccini	1858.0	1924	Lucques
Shostakovich	1906.0	1975	Saint-Petersburg

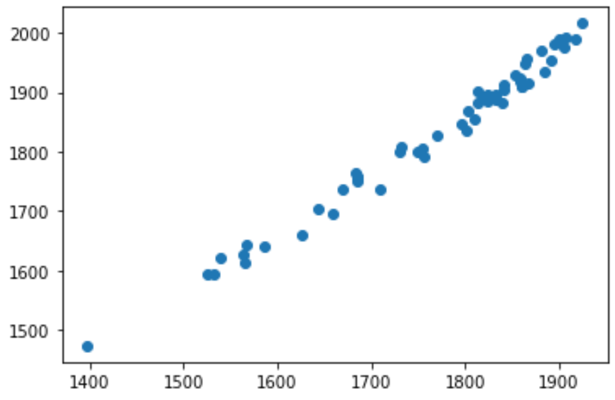
8.6 Plotting

We will learn more about plotting later, but let's see here some possibilities offered by Pandas. Pandas builds on top of Matplotlib but exploits the knowledge included in Dataframes to improve the default output. Let's see with a simple dataset.

In [28]:
```
composers = pd.read_excel('Data/composers.xlsx', sheet_name='Sheet5')
```

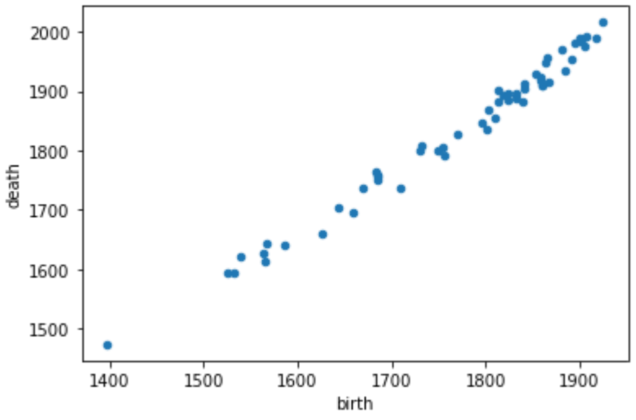
We can pass Series to Matplotlib which manages to understand them. Here's a default scatter plot:

In [29]:
```
plt.plot(composers.birth, composers.death, 'o')
plt.show()
```



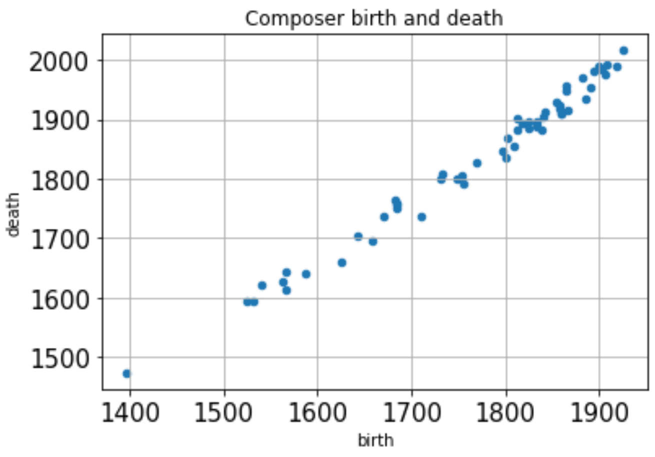
Now we look at the default Pandas output. Different types of plots are accessible when using the `data_frame.plot` function via the `kind` option. The variables to plot are column names passed as keywords instead of whole series like in Matplotlib:

In [30]:
```
composers.plot(x = 'birth', y = 'death', kind = 'scatter')
plt.show()
```



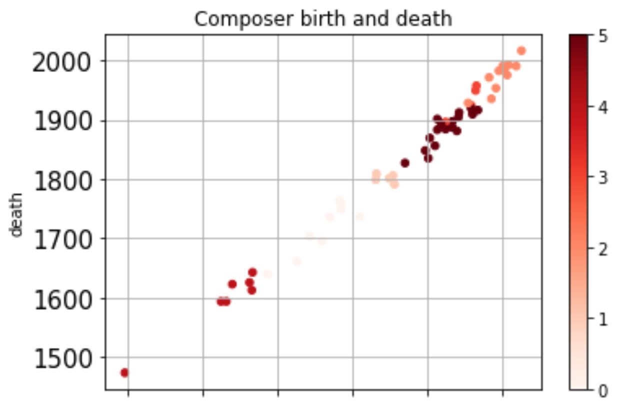
We see that the plot automatically gets axis labels. Another gain is that some obvious options like setting a title are directly accesible when creating the plot:

In [31]:
```
composers.plot(x = 'birth', y = 'death', kind = 'scatter',
                  title = 'Composer birth and death', grid = True, fontsize = 1
5)
plt.show()
```



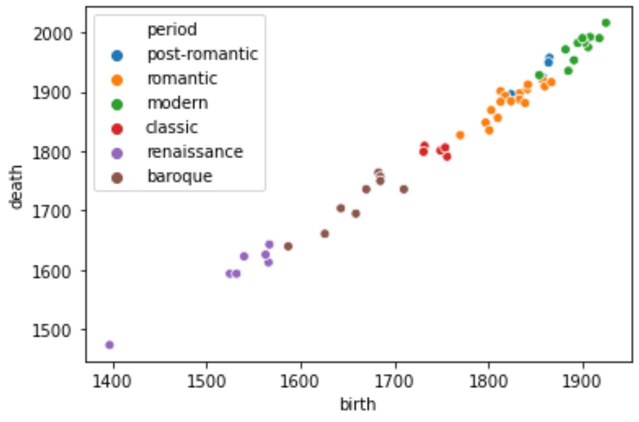
One can add even more information on the plot by using more arguments used in a similar way as a grammar of graphics. For example we can color the scatter plot by periods:

In [32]:
```
composers.plot(x = 'birth', y = 'death',kind = 'scatter',
               c = composers.period.astype('category').cat.codes, colormap =
'Reds', title = 'Composer birth and death', grid = True, fontsize = 15)
plt.show()
```



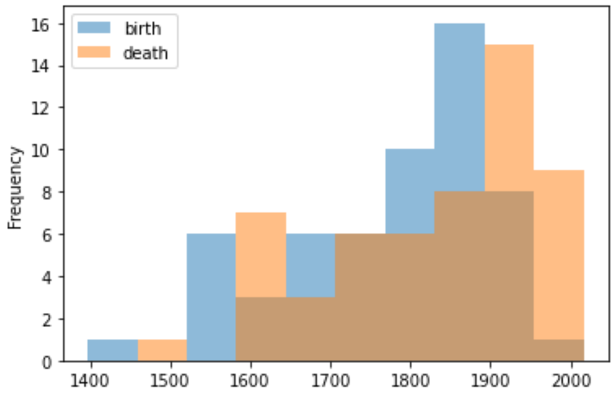
Here you see already a limitation of the plotting library. To color dots by the peiod category, we had to turn the latter into a series of numbers. We could then rename those to improve the plot, but it's better to use more specialized packages such as Seaborn which allow to realize this kind of plot easily:

In [33]:
```
sns.scatterplot(data = composers, x = 'birth', y = 'death', hue = 'period')
plt.show()
```



Some additional plotting options are available in the `plot()` module. For example histograms:

In [34]:
```
composers.plot.hist(alpha = 0.5)
plt.show()
```



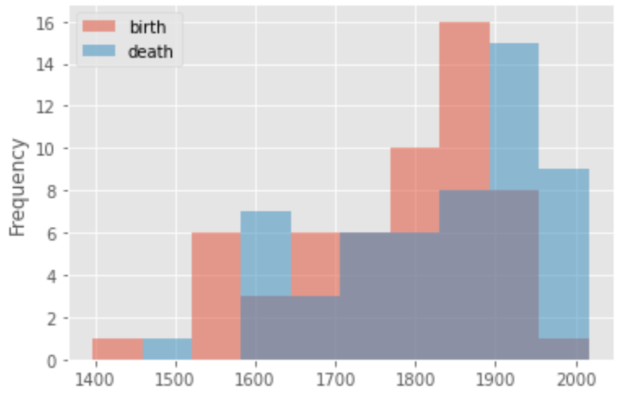
Here you see again the gain from using Pandas: without specifying anything, Pandas made a histogram of the two columns containing numbers, labelled the axis and even added a legend to the plot.

All these features are very nice and very helpful when exploring a dataset. When anaylzing data in depth and creating complex plots, Pandas's plotting might however be limiting and other options such as Seaborn or Plotnine can be used.

Finally, all plots can be "styled" down to the smallest detail, either by using Matplotlib options or by directly applying a style e.g.:

In [35]:
```
plt.style.use('ggplot')
```

In [36]:
```
composers.plot.hist(alpha = 0.5)
plt.show()
```



In [ ]:
```
```

9. Operations with Pandas objects

In [1]:
```
import pandas as pd
import numpy as np
```

One of the great advantages of using Pandas to handle tabular data is how simple it is to extract valuable information from them. Here we are going to see various types of operations that are available for this.

9.1 Matrix types of operations

The strength of Numpy is its natural way of handling matrix operations, and Pandas reuses a lot of these features. For example one can use simple mathematical operations to operate at the cell level:

In [2]:
```
compo_pd = pd.read_excel('Data/composers.xlsx')
compo_pd
```

Out[2]:

	composer	birth	death	city
0	Mahler	1860	1911	Kaliste
1	Beethoven	1770	1827	Bonn
2	Puccini	1858	1924	Lucques
3	Shostakovich	1906	1975	Saint-Petersburg

In [3]:
```
compo_pd['birth']*2
```

Out[3]:
```
0    3720
1    3540
2    3716
3    3812
Name: birth, dtype: int64
```

In [4]:
```
np.log(compo_pd['birth'])
```

Out[4]:
```
0    7.528332
1    7.478735
2    7.527256
3    7.552762
Name: birth, dtype: float64
```

Here we applied functions only to series. Indeed, since our Dataframe contains e.g. strings, no operation can be done on it:

In [5]:
```
#compo_pd+1
```

If however we have a homogenous Dataframe, this is possible:

In [6]:
```
compo_pd[['birth','death']]
```

Out[6]:

	birth	death
0	1860	1911
1	1770	1827
2	1858	1924
3	1906	1975

In [7]:
```
compo_pd[['birth','death']]*2
```

Out[7]:

	birth	death
0	3720	3822
1	3540	3654
2	3716	3848
3	3812	3950

9.2 Column operations

There are other types of functions whose purpose is to summarize the data. For example the mean or standard deviation. Pandas by default applies such functions column-wise and returns a series containing e.g. the mean of each column:

In [8]:
```
np.mean(compo_pd)
```

Out[8]:
```
birth    1848.50
death    1909.25
dtype: float64
```

Note that columns for which a mean does not make sense, like the city are discarded. A series of common functions like mean or standard deviation are directly implemented as methods and can be accessed in the alternative form:

In [9]:
```
compo_pd.describe()
```

Out[9]:

	birth	death
count	4.000000	4.000000
mean	1848.500000	1909.250000
std	56.836021	61.396933
min	1770.000000	1827.000000
25%	1836.000000	1890.000000
50%	1859.000000	1917.500000
75%	1871.500000	1936.750000
max	1906.000000	1975.000000

In [10]:
```
compo_pd.std()
```

Out[10]:
```
birth    56.836021
death    61.396933
dtype: float64
```

If you need the mean of only a single column you can of course chains operations:

```
In [11]: compo_pd.birth.mean()
Out[11]: 1848.5
```

9.3 Operations between Series

We can also do computations with multiple series as we would do with Numpy arrays:

```
In [12]: compo_pd['death']-compo_pd['birth']
Out[12]: 0    51
         1    57
         2    66
         3    69
         dtype: int64
```

We can even use the result of this computation to create a new column in our Dataframe:

```
In [13]: compo_pd
Out[13]:
```

	composer	birth	death	city
0	Mahler	1860	1911	Kaliste
1	Beethoven	1770	1827	Bonn
2	Puccini	1858	1924	Lucques
3	Shostakovich	1906	1975	Saint-Petersburg

```
In [14]: compo_pd['age'] = compo_pd['death']-compo_pd['birth']
```

```
In [15]: compo_pd
Out[15]:
```

	composer	birth	death	city	age
0	Mahler	1860	1911	Kaliste	51
1	Beethoven	1770	1827	Bonn	57
2	Puccini	1858	1924	Lucques	66
3	Shostakovich	1906	1975	Saint-Petersburg	69

9.4 Other functions

Sometimes one needs to apply to a column a very specific function that is not provided by default. In that case we can use one of the different `apply` methods of Pandas.

The simplest case is to apply a function to a column, or Series of a DataFrame. Let's say for example that we want to define the the age >60 as 'old' and <60 as 'young'. We can define the following general function:

```
In [16]: def define_age(x):
             if x>60:
                 return 'old'
             else:
                 return 'young'
```

```
In [17]: define_age(30)
```

```
Out[17]: 'young'
```

```
In [18]: define_age(70)
```

```
Out[18]: 'old'
```

We can now apply this function on an entire Series:

```
In [19]: compo_pd.age.apply(define_age)
```

```
Out[19]: 0    young
         1    young
         2      old
         3      old
         Name: age, dtype: object
```

```
In [20]: compo_pd.age.apply(lambda x: x**2)
```

```
Out[20]: 0    2601
         1    3249
         2    4356
         3    4761
         Name: age, dtype: int64
```

And again, if we want, we can directly use this output to create a new column:

```
In [21]: compo_pd['age_def'] = compo_pd.age.apply(define_age)
         compo_pd
```

Out[21]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

We can also apply a function to an entire DataFrame. For example we can ask how many composers have birth and death dates within the XIXth century:

```
In [22]: def nineteen_century_count(x):
             return np.sum((x>=1800)&(x<1900))
```

```
In [23]: compo_pd[['birth','death']].apply(nineteen_century_count)
```

```
Out[23]: birth    2
         death    1
         dtype: int64
```

The function is applied column-wise and returns a single number for each in the form of a series.

In [24]:
```
def nineteen_century_true(x):
    return (x>=1800)&(x<1900)
```

In [25]:
```
compo_pd[['birth','death']].apply(nineteen_century_true)
```

Out[25]:

	birth	death
0	True	False
1	False	True
2	True	False
3	False	False

Here the operation is again applied column-wise but the output is a Series.

There are more combinations of what can be the in- and output of the apply function and in what order (column- or row-wise) they are applied that cannot be covered here.

9.5 Logical indexing

Just like with Numpy, it is possible to subselect parts of a Dataframe using logical indexing. Let's have a look again at an example:

In [26]:
```
compo_pd
```

Out[26]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

If we use a logical comparison on a series, this yields a **logical Series**:

In [27]:
```
compo_pd['birth']
```

Out[27]:
```
0    1860
1    1770
2    1858
3    1906
Name: birth, dtype: int64
```

In [28]:
```
compo_pd['birth'] > 1859
```

Out[28]:
```
0     True
1    False
2    False
3     True
Name: birth, dtype: bool
```

Just like in Numpy we can use this logical Series as an index to select elements in the Dataframe:

In [29]:
```
log_indexer = compo_pd['birth'] > 1859
log_indexer
```

Out[29]:
```
0     True
1    False
2    False
3     True
Name: birth, dtype: bool
```

In [30]:
```
compo_pd
```

Out[30]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

In [31]:
```
~log_indexer
```

Out[31]:
```
0    False
1     True
2     True
3    False
Name: birth, dtype: bool
```

In [32]:
```
compo_pd[~log_indexer]
```

Out[32]:

	composer	birth	death	city	age	age_def
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old

We can also create more complex logical indexings:

In [33]:
```
(compo_pd['birth'] > 1859)&(compo_pd['age']>60)
```

Out[33]:
```
0    False
1    False
2    False
3     True
dtype: bool
```

In [34]:
```
compo_pd[(compo_pd['birth'] > 1859)&(compo_pd['age']>60)]
```

Out[34]:

	composer	birth	death	city	age	age_def
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

And we can create new arrays containing only these subselections:

In [35]:
```
compos_sub = compo_pd[compo_pd['birth'] > 1859]
```

In [36]:
```
compos_sub
```

Out[36]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

We can then modify the new array:

In [37]:
```
compos_sub.loc[0,'birth'] = 3000
```

```
/Users/gw18g940/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas
/core/indexing.py:966: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/s
table/user_guide/indexing.html#returning-a-view-versus-a-copy
  self.obj[item] = s
```

Note that we get this SettingWithCopyWarning warning. This is a very common problem hand has to do with how new arrays are created when making subselections. Simply stated, did we create an entirely new array or a "view" of the old one? This will be very case-dependent and to avoid this, if we want to create a new array we can just enforce it using the `copy()` method (for more information on the topic see for example this [explanation (https://www.dataquest.io/blog/settingwithcopywarning/)](https://www.dataquest.io/blog/settingwithcopywarning/):

In [38]:
```
compos_sub2 = compo_pd[compo_pd['birth'] > 1859].copy()
compos_sub2.loc[0,'birth'] = 3000
```

In [39]:
```
compos_sub2
```

Out[39]:

	composer	birth	death	city	age	age_def
0	Mahler	3000	1911	Kaliste	51	young
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

10. Combining information in Pandas

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Often information is comming from different sources and it is necessary to combine it into one object. We are going to see the different ways in which information contained within separate Dataframes can be combined in a meaningful way.

10.1 Concatenation

The simplest way we can combine two Dataframes is simply to "paste" them together:

In [2]:
```
composers1 = pd.read_excel('Data/composers.xlsx', index_col='composer',sheet
_name='Sheet1')
composers1
```

Out[2]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

In [3]:
```
composers2 = pd.read_excel('Data/composers.xlsx', index_col='composer',sheet
_name='Sheet3')
composers2
```

Out[3]:

composer	birth	death	city
Verdi	1813	1901	Roncole
Dvorak	1841	1904	Nelahozeves
Schumann	1810	1856	Zwickau
Stravinsky	1882	1971	Oranienbaum
Mahler	1860	1911	Kaliste

To be concatenated, Dataframes need to be provided as a list:

In [4]:
```
all_composers = pd.concat([composers1,composers2])
```

In [5]:
```
all_composers
```

Out[5]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg
Verdi	1813	1901	Roncole
Dvorak	1841	1904	Nelahozeves
Schumann	1810	1856	Zwickau
Stravinsky	1882	1971	Oranienbaum
Mahler	1860	1911	Kaliste

One potential problem is that two tables contain duplicated information:

In [6]:
```
all_composers.loc['Mahler']
```

Out[6]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Mahler	1860	1911	Kaliste

It is very easy to get rid of it using. `duplicated()` gives us a boolean series of duplications and we can just selected non-duplicated rows:

In [7]:
```
all_composers.duplicated()
```

Out[7]:
```
composer
Mahler          False
Beethoven       False
Puccini         False
Shostakovich    False
Verdi           False
Dvorak          False
Schumann        False
Stravinsky      False
Mahler           True
dtype: bool
```

In [8]:
```
all_composers[~all_composers.duplicated()]
```

Out[8]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg
Verdi	1813	1901	Roncole
Dvorak	1841	1904	Nelahozeves
Schumann	1810	1856	Zwickau
Stravinsky	1882	1971	Oranienbaum

10.2 Joining two tables

An other classical case is that of two list with similar index but containing different information, e.g.

In [9]:
```
composers1 = pd.read_excel('Data/composers.xlsx', index_col='composer',sheet
_name='Sheet1')
composers1
```

Out[9]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

In [10]:
```
composers2 = pd.read_excel('Data/composers.xlsx', index_col='composer',sheet
_name='Sheet4')
composers2
```

Out[10]:

composer	first name
Mahler	Gustav
Beethoven	Ludwig van
Puccini	Giacomo
Brahms	Johannes

If we we use again simple concatenation, this doesn't help us much. We just end up with a large matrix with lots of NaN's:

In [11]:
```
pd.concat([composers1, composers2])
```

Out[11]:

	birth	death	city	first name
composer				
Mahler	1860.0	1911.0	Kaliste	NaN
Beethoven	1770.0	1827.0	Bonn	NaN
Puccini	1858.0	1924.0	Lucques	NaN
Shostakovich	1906.0	1975.0	Saint-Petersburg	NaN
Mahler	NaN	NaN	NaN	Gustav
Beethoven	NaN	NaN	NaN	Ludwig van
Puccini	NaN	NaN	NaN	Giacomo
Brahms	NaN	NaN	NaN	Johannes

The better way of doing this is to **join** the tables. This is a classical database concept available in Pandas.

`join()` operates on two tables: the first one is the "left" table which uses `join()` as a method. The other table is the "right" one.

Let's try the default join settings:

In [12]:
```
composers1
```

Out[12]:

	birth	death	city
composer			
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

In [13]:
```
composers2
```

Out[13]:

	first name
composer	
Mahler	Gustav
Beethoven	Ludwig van
Puccini	Giacomo
Brahms	Johannes

In [14]:
```
composers1.join(composers2)
```

Out[14]:

composer	birth	death	city	first name
Mahler	1860	1911	Kaliste	Gustav
Beethoven	1770	1827	Bonn	Ludwig van
Puccini	1858	1924	Lucques	Giacomo
Shostakovich	1906	1975	Saint-Petersburg	NaN

We see that Pandas was smart enough to notice that the two tables had a index name and used it to combine the tables. We also see that one element from the second table (Brahms) is missing. The reason for this is the way indices not present in both tables are handled. There are four ways of doing this with two tables called here the "left" and "right" table.

10.2.1. Join left

Here "left" and "right" just represent two Dataframes that should be merged. They have a common index, but not necessarily the same items. For example here Shostakovich is missing in the second table, while Brahms is missing in the first one. When using the "right" join, we use the first Dataframe as basis and only use the indices that appear there.

In [15]:
```
composers1.join(composers2, how = 'left')
```

Out[15]:

composer	birth	death	city	first name
Mahler	1860	1911	Kaliste	Gustav
Beethoven	1770	1827	Bonn	Ludwig van
Puccini	1858	1924	Lucques	Giacomo
Shostakovich	1906	1975	Saint-Petersburg	NaN

Hence Brahms is left out.

10.2.2. Join right

We can do the the opposite and use the indices of the second Dataframe as basis:

In [16]:
```
composers1.join(composers2, how = 'right')
```

Out[16]:

composer	birth	death	city	first name
Mahler	1860.0	1911.0	Kaliste	Gustav
Beethoven	1770.0	1827.0	Bonn	Ludwig van
Puccini	1858.0	1924.0	Lucques	Giacomo
Brahms	NaN	NaN	NaN	Johannes

Here we have Brahms but not Shostakovich.

10.2.3. Inner, outer

Finally, we can just say that we want to recover eihter only the items that appaer in both Dataframes (inner, like in a Venn diagram) or all the items (outer).

```
In [17]: composers1.join(composers2, how = 'inner')
```

Out[17]:

composer	birth	death	city	first name
Mahler	1860	1911	Kaliste	Gustav
Beethoven	1770	1827	Bonn	Ludwig van
Puccini	1858	1924	Lucques	Giacomo

```
In [18]: composers1.join(composers2, how = 'outer')
```

Out[18]:

composer	birth	death	city	first name
Beethoven	1770.0	1827.0	Bonn	Ludwig van
Brahms	NaN	NaN	NaN	Johannes
Mahler	1860.0	1911.0	Kaliste	Gustav
Puccini	1858.0	1924.0	Lucques	Giacomo
Shostakovich	1906.0	1975.0	Saint-Petersburg	NaN

10.3.4 Joining on columns : merge

Above we have used `join` to join based on indices. However sometimes tables don't have the same indices but similar contents that we want to merge. For example let's imagine whe have the two Dataframes below:

```
In [19]: composers1 = pd.read_excel('Data/composers.xlsx', sheet_name='Sheet1')
         composers2 = pd.read_excel('Data/composers.xlsx', sheet_name='Sheet6')
```

```
In [20]: composers1
```

Out[20]:

	composer	birth	death	city
0	Mahler	1860	1911	Kaliste
1	Beethoven	1770	1827	Bonn
2	Puccini	1858	1924	Lucques
3	Shostakovich	1906	1975	Saint-Petersburg

In [21]:
```
composers2
```

Out[21]:

	last name	first name
0	Puccini	Giacomo
1	Beethoven	Ludwig van
2	Brahms	Johannes
3	Mahler	Gustav

The indices don't match and are not the composer name. In addition the columns containing the composer names have different labels. Here we can use `merge()` and specify which columns we want to use for merging, and what type of merging we need (inner, left etc.)

In [22]:
```
pd.merge(composers1, composers2, left_on='composer', right_on='last name')
```

Out[22]:

	composer	birth	death	city	last name	first name
0	Mahler	1860	1911	Kaliste	Mahler	Gustav
1	Beethoven	1770	1827	Bonn	Beethoven	Ludwig van
2	Puccini	1858	1924	Lucques	Puccini	Giacomo

Again we can use another variety of join than the default inner:

In [23]:
```
pd.merge(composers1, composers2, left_on='composer', right_on='last name',ho
w = 'outer')
```

Out[23]:

	composer	birth	death	city	last name	first name
0	Mahler	1860.0	1911.0	Kaliste	Mahler	Gustav
1	Beethoven	1770.0	1827.0	Bonn	Beethoven	Ludwig van
2	Puccini	1858.0	1924.0	Lucques	Puccini	Giacomo
3	Shostakovich	1906.0	1975.0	Saint-Petersburg	NaN	NaN
4	NaN	NaN	NaN	NaN	Brahms	Johannes

In [24]:
```
pd.merge(composers1, composers2, left_on='composer', right_on='last name',ho
w = 'right')
```

Out[24]:

	composer	birth	death	city	last name	first name
0	Mahler	1860.0	1911.0	Kaliste	Mahler	Gustav
1	Beethoven	1770.0	1827.0	Bonn	Beethoven	Ludwig van
2	Puccini	1858.0	1924.0	Lucques	Puccini	Giacomo
3	NaN	NaN	NaN	NaN	Brahms	Johannes

11. Splitting data

Often one has tables that mix regular variables (e.g. the size of cells in microscopy images) with categorical variables (e.g. the type of cell to which they belong). In that case, it is quite usual to split the data by categories or *groups* to do computations. Pandas allows to do this very easily.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

11.1 Grouping

Let's import some data and have a look at them:

In [2]:
```
composers = pd.read_excel('Data/composers.xlsx', sheet_name='Sheet5')
```

In [3]:
```
composers.head()
```

Out[3]:

	composer	birth	death	period	country
0	Mahler	1860	1911.0	post-romantic	Austria
1	Beethoven	1770	1827.0	romantic	Germany
2	Puccini	1858	1924.0	post-romantic	Italy
3	Shostakovich	1906	1975.0	modern	Russia
4	Verdi	1813	1901.0	romantic	Italy

We also add a column here to calculate the composers' age:

In [4]:
```
composers['age'] = composers.death - composers.birth
```

11.1.1 Single level

What if we want now to count how many composers we have in a certain category like the period or country? In classical computing we would maybe do a for loop to count occurrences. Pandas simplifies this with the `groupby()` function, which actually groups elements by a certain criteria, e.g. a categorical variable like the period:

In [5]:
```
composer_grouped = composers.groupby('period')
composer_grouped
```

Out[5]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x11d2fc850>
```

The output is a bit cryptic. What we actually have is a new object called a group which has a lot of handy properties. First let's see what the groups actually are. We can find all groups with `groups` :

In [6]:
```
composer_grouped.groups
```

Out[6]:
```
{'baroque': Int64Index([14, 16, 17, 20, 21, 28, 29, 30, 31, 47], dtype='int64
'),
 'classic': Int64Index([9, 10, 32, 40, 51], dtype='int64'),
 'modern': Int64Index([3, 7, 11, 12, 19, 25, 45, 46, 50, 53, 54, 55, 56], dty
pe='int64'),
 'post-romantic': Int64Index([0, 2, 8, 18, 49], dtype='int64'),
 'renaissance': Int64Index([13, 26, 27, 36, 37, 43, 44], dtype='int64'),
 'romantic': Int64Index([1, 4, 5, 6, 15, 22, 23, 24, 33, 34, 35, 38, 39, 41,
42, 48, 52], dtype='int64')}
```

We have a dictionary, where each *period* that appears in the Dataframe is a key and each key contains a list of dataframe *indices* of rows with those periods. We will rarely directly use those indices, as most operations on groups only use those "behind the scene".

For example we can use `describe()` on a group object, just like we did it before for a Dataframe:

In [7]:
```
composer_grouped.describe()#.loc['Austria','birth']
```

Out[7]:

	birth								death		
	count	mean	std	min	25%	50%	75%	max	count	mean	.
period											
baroque	10.0	1663.300000	36.009412	1587.0	1647.0	1676.5	1685.0	1710.0	10.0	1720.200000	.
classic	5.0	1744.400000	12.054045	1731.0	1732.0	1749.0	1754.0	1756.0	5.0	1801.200000	.
modern	13.0	1905.692308	28.595992	1854.0	1891.0	1902.0	1918.0	1971.0	11.0	1974.090909	.
post-romantic	5.0	1854.200000	17.123084	1824.0	1858.0	1860.0	1864.0	1865.0	5.0	1927.400000	.
renaissance	7.0	1527.142857	59.881629	1397.0	1528.5	1540.0	1564.5	1567.0	7.0	1595.285714	.
romantic	17.0	1824.823529	25.468695	1770.0	1810.0	1824.0	1841.0	1867.0	17.0	1883.588235	.

6 rows × 24 columns

We see here that the statistical analysis has been done for each group, the index of each row being the group name (or key in the dictionary). If we are interested in a specific group we can also easily recover it:

In [8]:
```
composer_grouped.get_group('classic')
```

Out[8]:

	composer	birth	death	period	country	age
9	Haydn	1732	1809.0	classic	Austria	77.0
10	Mozart	1756	1791.0	classic	Austria	35.0
32	Cimarosa	1749	1801.0	classic	Italy	52.0
40	Soler	1754	1806.0	classic	Spain	52.0
51	Dusek	1731	1799.0	classic	Czechia	68.0

We see that this returns a sub-group from the original table. Effectively it is almost equivalent to:

In [9]:
```
composers[composers.period == 'classic']
```

Out[9]:

	composer	birth	death	period	country	age
9	Haydn	1732	1809.0	classic	Austria	77.0
10	Mozart	1756	1791.0	classic	Austria	35.0
32	Cimarosa	1749	1801.0	classic	Italy	52.0
40	Soler	1754	1806.0	classic	Spain	52.0
51	Dusek	1731	1799.0	classic	Czechia	68.0

11.1.2 Multi-level

If one has multiple categorical variables, one can also do a grouping on several levels. For example here we want to classify composers both by period and country. For this we just give two column names to the `groupby()` function:

In [10]:
```
composer_grouped = composers.groupby(['period','country'])
composer_grouped.describe()
```

Out[10]:

		birth								death	
		count	mean	std	min	25%	50%	75%	max	count	m
period	country										
baroque	England	1.0	1659.000000	NaN	1659.0	1659.00	1659.0	1659.00	1659.0	1.0	16
	France	3.0	1650.666667	29.263174	1626.0	1634.50	1643.0	1663.00	1683.0	3.0	17
	Germany	2.0	1685.000000	0.000000	1685.0	1685.00	1685.0	1685.00	1685.0	2.0	17
	Italy	4.0	1663.000000	53.285395	1587.0	1649.25	1677.5	1691.25	1710.0	4.0	17
classic	Austria	2.0	1744.000000	16.970563	1732.0	1738.00	1744.0	1750.00	1756.0	2.0	18
	Czechia	1.0	1731.000000	NaN	1731.0	1731.00	1731.0	1731.00	1731.0	1.0	17
	Italy	1.0	1749.000000	NaN	1749.0	1749.00	1749.0	1749.00	1749.0	1.0	18
	Spain	1.0	1754.000000	NaN	1754.0	1754.00	1754.0	1754.00	1754.0	1.0	18
modern	Austria	1.0	1885.000000	NaN	1885.0	1885.00	1885.0	1885.00	1885.0	1.0	19
	Czechia	1.0	1854.000000	NaN	1854.0	1854.00	1854.0	1854.00	1854.0	1.0	19
	England	2.0	1936.500000	48.790368	1902.0	1919.25	1936.5	1953.75	1971.0	1.0	19
	France	2.0	1916.500000	12.020815	1908.0	1912.25	1916.5	1920.75	1925.0	2.0	20
	Germany	1.0	1895.000000	NaN	1895.0	1895.00	1895.0	1895.00	1895.0	1.0	19
	RUssia	1.0	1891.000000	NaN	1891.0	1891.00	1891.0	1891.00	1891.0	1.0	19
	Russia	2.0	1894.000000	16.970563	1882.0	1888.00	1894.0	1900.00	1906.0	2.0	19
	USA	3.0	1918.333333	18.502252	1900.0	1909.00	1918.0	1927.50	1937.0	2.0	19
post-romantic	Austria	2.0	1842.000000	25.455844	1824.0	1833.00	1842.0	1851.00	1860.0	2.0	19
	Finland	1.0	1865.000000	NaN	1865.0	1865.00	1865.0	1865.00	1865.0	1.0	19
	Germany	1.0	1864.000000	NaN	1864.0	1864.00	1864.0	1864.00	1864.0	1.0	19
	Italy	1.0	1858.000000	NaN	1858.0	1858.00	1858.0	1858.00	1858.0	1.0	19
renaissance	Belgium	2.0	1464.500000	95.459415	1397.0	1430.75	1464.5	1498.25	1532.0	2.0	15
	England	2.0	1551.500000	16.263456	1540.0	1545.75	1551.5	1557.25	1563.0	2.0	16
	Italy	3.0	1552.666667	23.965253	1525.0	1545.50	1566.0	1566.50	1567.0	3.0	16
romantic	Czechia	2.0	1832.500000	12.020815	1824.0	1828.25	1832.5	1836.75	1841.0	2.0	18
	France	3.0	1821.000000	19.672316	1803.0	1810.50	1818.0	1830.00	1842.0	3.0	18
	Germany	4.0	1806.500000	26.388129	1770.0	1800.00	1811.5	1818.00	1833.0	4.0	18
	Italy	4.0	1817.250000	28.004464	1797.0	1800.00	1807.0	1824.25	1858.0	4.0	18
	Russia	2.0	1836.000000	4.242641	1833.0	1834.50	1836.0	1837.50	1839.0	2.0	18
	Spain	2.0	1863.500000	4.949747	1860.0	1861.75	1863.5	1865.25	1867.0	2.0	19

29 rows × 24 columns

In [11]:
```
composer_grouped.get_group(('baroque','Germany'))
```

Out[11]:

	composer	birth	death	period	country	age
14	Haendel	1685	1759.0	baroque	Germany	74.0
47	Bach	1685	1750.0	baroque	Germany	65.0

11.2 Operations on groups

The main advantage of this Group object is that it allows us to do very quickly both computations and plotting without having to loop through different categories. Indeed Pandas makes all the work for us: it applies functions on each group and then reassembles the results into a Dataframe (or Series depending on the operation).

For example we can apply most functions we used for Dataframes (mean, sum etc.) on groups as well and Pandas seamlessly does the work for us:

In [12]:
```
composer_grouped.mean()
```

Out[12]:

		birth	death	age
period	**country**			
baroque	**England**	1659.000000	1695.000000	36.000000
	France	1650.666667	1709.666667	59.000000
	Germany	1685.000000	1754.500000	69.500000
	Italy	1663.000000	1717.250000	54.250000
classic	**Austria**	1744.000000	1800.000000	56.000000
	Czechia	1731.000000	1799.000000	68.000000
	Italy	1749.000000	1801.000000	52.000000
	Spain	1754.000000	1806.000000	52.000000
modern	**Austria**	1885.000000	1935.000000	50.000000
	Czechia	1854.000000	1928.000000	74.000000
	England	1936.500000	1983.000000	81.000000
	France	1916.500000	2004.000000	87.500000
	Germany	1895.000000	1982.000000	87.000000
	RUssia	1891.000000	1953.000000	62.000000
	Russia	1894.000000	1973.000000	79.000000
	USA	1918.333333	1990.000000	81.000000
post-romantic	**Austria**	1842.000000	1903.500000	61.500000
	Finland	1865.000000	1957.000000	92.000000
	Germany	1864.000000	1949.000000	85.000000
	Italy	1858.000000	1924.000000	66.000000
renaissance	**Belgium**	1464.500000	1534.000000	69.500000
	England	1551.500000	1624.500000	73.000000
	Italy	1552.666667	1616.666667	64.000000
romantic	**Czechia**	1832.500000	1894.000000	61.500000
	France	1821.000000	1891.333333	70.333333
	Germany	1806.500000	1865.750000	59.250000
	Italy	1817.250000	1875.750000	58.500000
	Russia	1836.000000	1884.000000	48.000000
	Spain	1863.500000	1912.500000	49.000000

In [13]:
```
composer_grouped.count()
```

Out[13]:
```
                              composer  birth  death  age
period          country
baroque         England              1      1      1    1
                France               3      3      3    3
                Germany              2      2      2    2
                Italy                4      4      4    4
classic         Austria              2      2      2    2
                Czechia              1      1      1    1
                Italy                1      1      1    1
                Spain                1      1      1    1
modern          Austria              1      1      1    1
                Czechia              1      1      1    1
                England              2      2      1    1
                France               2      2      2    2
                Germany              1      1      1    1
                RUssia               1      1      1    1
                Russia               2      2      2    2
                USA                  3      3      2    2
post-romantic   Austria              2      2      2    2
                Finland              1      1      1    1
                Germany              1      1      1    1
                Italy                1      1      1    1
renaissance     Belgium              2      2      2    2
                England              2      2      2    2
                Italy                3      3      3    3
romantic        Czechia              2      2      2    2
                France               3      3      3    3
                Germany              4      4      4    4
                Italy                4      4      4    4
                Russia               2      2      2    2
                Spain                2      2      2    2
```

We can also design specific functions (again, like in the case of Dataframes) and apply them on groups:

In [14]:
```
def mult(myseries):
    return myseries.max() * 3
```

In [15]:
```
composer_grouped.apply(mult)
```

Out[15]:

		composer	birth	death	period
period	**country**				
baroque	**England**	PurcellPurcellPurcell	4977	5085.0	baroquebaroquebaroque
	France	RameauRameauRameau	5049	5292.0	baroquebaroquebaroque
	Germany	HaendelHaendelHaendel	5055	5277.0	baroquebaroquebaroque
	Italy	ScarlattiScarlattiScarlatti	5130	5271.0	baroquebaroquebaroque
classic	**Austria**	MozartMozartMozart	5268	5427.0	classicclassicclassic
	Czechia	DusekDusekDusek	5193	5397.0	classicclassicclassic
	Italy	CimarosaCimarosaCimarosa	5247	5403.0	classicclassicclassic
	Spain	SolerSolerSoler	5262	5418.0	classicclassicclassic
modern	**Austria**	BergBergBerg	5655	5805.0	modernmodernmodern
	Czechia	JanacekJanacekJanacek	5562	5784.0	modernmodernmodern
	England	WaltonWaltonWalton	5913	5949.0	modernmodernmodern
	France	MessiaenMessiaenMessiaen	5775	6048.0	modernmodernmodern
	Germany	OrffOrffOrff	5685	5946.0	modernmodernmodern
	RUssia	ProkofievProkofievProkofiev	5673	5859.0	modernmodernmodern
	Russia	StravinskyStravinskyStravinsky	5718	5925.0	modernmodernmodern
	USA	GlassGlassGlass	5811	5970.0	modernmodernmodern
post-romantic	**Austria**	MahlerMahlerMahler	5580	5733.0	post-romanticpost-romanticpost-romantic
	Finland	SibeliusSibeliusSibelius	5595	5871.0	post-romanticpost-romanticpost-romantic
	Germany	StraussStraussStrauss	5592	5847.0	post-romanticpost-romanticpost-romantic
	Italy	PucciniPucciniPuccini	5574	5772.0	post-romanticpost-romanticpost-romantic
renaissance	**Belgium**	LassusLassusLassus	4596	4782.0	renaissancerenaissancerenaissance
	England	DowlandDowlandDowland	4689	4878.0	renaissancerenaissancerenaissance
	Italy	PalestrinaPalestrinaPalestrina	4701	4929.0	renaissancerenaissancerenaissance
romantic	**Czechia**	SmetanaSmetanaSmetana	5523	5712.0	romanticromanticromantic
	France	MassenetMassenetMassenet	5526	5736.0	romanticromanticromantic
	Germany	WagnerWagnerWagner	5499	5691.0	romanticromanticromantic
	Italy	VerdiVerdiVerdi	5574	5757.0	romanticromanticromantic
	Russia	MussorsgskyMussorsgskyMussorsgsky	5517	5661.0	romanticromanticromantic
	Spain	GranadosGranadosGranados	5601	5748.0	romanticromanticromantic

11.3 Reshaping dataframes

As we see above, grouping operations can create more or less complex dataframes by adding one or multiple indexing levels. There are multiple ways to "reshape" such dataframes in order to make thm usable e.g. for plotting. Typically, plotting software based on a grammar of graphics expect a simple 2D dataframe where each line is an observation with several properties.

11.3.1 re-indexing, unstacking

One of the most common "reshaping" is to reset the index. In its simplest form, it will create a new dataframe, where each row corresponds to one observation. For example in the case of a dataframe with multi-indices, it will re-cast these indices as columns:

In [16]:
```
composer_grouped = composers.groupby(['period','country']).mean()
composer_grouped.head(10)
```

Out[16]:

		birth	death	age
period	**country**			
baroque	**England**	1659.000000	1695.000000	36.00
	France	1650.666667	1709.666667	59.00
	Germany	1685.000000	1754.500000	69.50
	Italy	1663.000000	1717.250000	54.25
classic	**Austria**	1744.000000	1800.000000	56.00
	Czechia	1731.000000	1799.000000	68.00
	Italy	1749.000000	1801.000000	52.00
	Spain	1754.000000	1806.000000	52.00
modern	**Austria**	1885.000000	1935.000000	50.00
	Czechia	1854.000000	1928.000000	74.00

In [17]:
```
composer_grouped.reset_index().head(5)
```

Out[17]:

	period	country	birth	death	age
0	baroque	England	1659.000000	1695.000000	36.00
1	baroque	France	1650.666667	1709.666667	59.00
2	baroque	Germany	1685.000000	1754.500000	69.50
3	baroque	Italy	1663.000000	1717.250000	54.25
4	classic	Austria	1744.000000	1800.000000	56.00

One can of course be more specific and reset only specific indices e.g. by level:

In [18]:
```
composer_grouped.reset_index(level=1).head(5)
```

Out[18]:

	country	birth	death	age
period				
baroque	England	1659.000000	1695.000000	36.00
baroque	France	1650.666667	1709.666667	59.00
baroque	Germany	1685.000000	1754.500000	69.50
baroque	Italy	1663.000000	1717.250000	54.25
classic	Austria	1744.000000	1800.000000	56.00

11.3.2 unstacking

Another way to move indices to columns is to *unstack* a dataframe, in other words pivot some indices to columns:

In [19]:
```
composer_grouped.unstack()
```

Out[19]:

	birth									F
country	**Austria**	**Belgium**	**Czechia**	**England**	**Finland**	**France**	**Germany**	**Italy**	**RUssia**	
period										
baroque	NaN	NaN	NaN	1659.0	NaN	1650.666667	1685.0	1663.000000	NaN	
classic	1744.0	NaN	1731.0	NaN	NaN	NaN	NaN	1749.000000	NaN	
modern	1885.0	NaN	1854.0	1936.5	NaN	1916.500000	1895.0	NaN	1891.0	
post-romantic	1842.0	NaN	NaN	NaN	1865.0	NaN	1864.0	1858.000000	NaN	
renaissance	NaN	1464.5	NaN	1551.5	NaN	NaN	NaN	1552.666667	NaN	
romantic	NaN	NaN	1832.5	NaN	NaN	1821.000000	1806.5	1817.250000	NaN	

6 rows × 36 columns

This creates a multi-level column indexing.

11.3.3 Wide to long: melt

A very common operation when handling tables is to switch from wide to long format and vice versa. In our composer example, let's for example imagine that you want both `birth` and `death` dates to be grouped in a single column called `dates` . But you still need to know if that data is a birth or date, so you need a new column that indicates that. To achieve that, we need to specify `id_vars` a list of columns to be used as *identifiers* e.g. the composer name, and `value_vars` , a list of columns that should become rows:

In [20]:
```
composers.head(5)
```

Out[20]:

	composer	birth	death	period	country	age
0	Mahler	1860	1911.0	post-romantic	Austria	51.0
1	Beethoven	1770	1827.0	romantic	Germany	57.0
2	Puccini	1858	1924.0	post-romantic	Italy	66.0
3	Shostakovich	1906	1975.0	modern	Russia	69.0
4	Verdi	1813	1901.0	romantic	Italy	88.0

In [21]:
```
pd.melt(composers, id_vars=['composer'], value_vars=['birth', 'death'])
```

Out[21]:

	composer	variable	value
0	Mahler	birth	1860.0
1	Beethoven	birth	1770.0
2	Puccini	birth	1858.0
3	Shostakovich	birth	1906.0
4	Verdi	birth	1813.0
...	...	...	...
109	Smetana	death	1884.0
110	Janacek	death	1928.0
111	Copland	death	1990.0
112	Bernstein	death	1990.0
113	Glass	death	NaN

114 rows × 3 columns

We can keep more of the original columns as *identifiers* and also specify names for the *variable* and *value* columns:

In [22]:
```
melted = pd.melt(composers, id_vars=['composer','period','age','country'], v
alue_vars=['birth', 'death'],
                 var_name = 'date_type', value_name='dates')
melted
```

Out[22]:

	composer	period	age	country	date_type	dates
0	Mahler	post-romantic	51.0	Austria	birth	1860.0
1	Beethoven	romantic	57.0	Germany	birth	1770.0
2	Puccini	post-romantic	66.0	Italy	birth	1858.0
3	Shostakovich	modern	69.0	Russia	birth	1906.0
4	Verdi	romantic	88.0	Italy	birth	1813.0
...	...	...	...	...	...	...
109	Smetana	romantic	60.0	Czechia	death	1884.0
110	Janacek	modern	74.0	Czechia	death	1928.0
111	Copland	modern	90.0	USA	death	1990.0
112	Bernstein	modern	72.0	USA	death	1990.0
113	Glass	modern	NaN	USA	death	NaN

114 rows × 6 columns

11.4 Plotting

We have seen above that we can create groups and apply functions to them to get some summary of them as new dataframes or series that could then also be reshaped. The final result of these operations is then ideally suited to be plotted in a very efficient way.

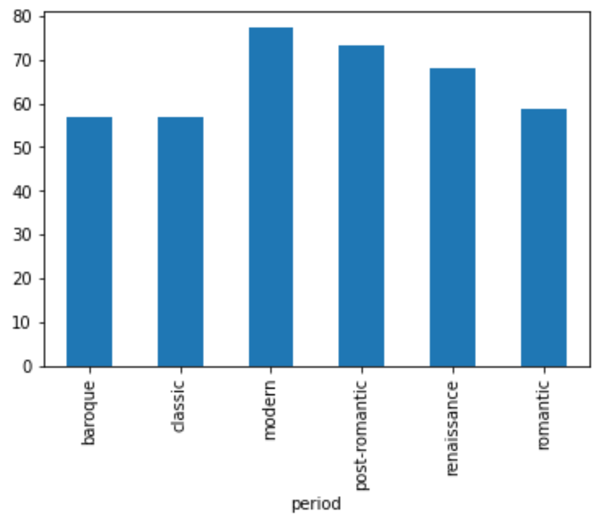
Here's a simple example: we group composers by periods and then calculate the mean age, resulting in a series where periods are indices:

```
In [23]: composers.groupby('period')['age'].mean()
```

```
Out[23]: period
         baroque          56.900000
         classic          56.800000
         modern           77.181818
         post-romantic    73.200000
         renaissance      68.142857
         romantic         58.764706
         Name: age, dtype: float64
```

We can just add one more operation to that line to create a bar plot illustrating this:

```
In [24]: composers.groupby('period')['age'].mean().plot(kind = 'bar');
```



The built-in plotting capabilities of Pandas automatically used the indices to label the bars, and also used the series name as a general label.

Using more advanced libraries, we can go further than that and use multiple columns to create complex plots. This will be shown in the next chapter.

12. A complete example

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import seaborn as sns
```

We have seen now most of the basic features of Pandas including importing data, combining dataframes, aggregating information and plotting it. In this chapter, we are going to re-use these concepts with the real data seen in the introduction chapter (06-DA_Pandas_introduction.ipynb). We are also going to explore some more advanced plotting libraries that exploit to the maximum dataframe structures.

12.1 Importing data

We are importing here two tables provided openly by the Swiss National Science Foundation. One contains a list of all *projects* to which funds have been allocated since 1975. The other table contains a list of all *people* to which funds have been awarded during the same period:

In [7]:
```
# local import
projects = pd.read_csv('Data/P3_GrantExport.csv',sep = ';')
persons = pd.read_csv('Data/P3_PersonExport.csv',sep = ';')

# import from url
#projects = pd.read_csv('http://p3.snf.ch/P3Export/P3_GrantExport.csv',sep =
';')
#persons = pd.read_csv('http://p3.snf.ch/P3Export/P3_PersonExport.csv',sep =
';')
```

We can have a brief look at both tables:

In [8]:
```
projects.head(5)
```

Out[8]:

	Project Number	Project Number String	Project Title	Project Title English	Responsible Applicant	Funding Instrument	Funding Instrument Hierarchy	
0	1	1000-000001	Schlussband (Bd. VI) der Jacob Burckhardt-Biog...	NaN	Kaegi Werner	Project funding (Div. I-III)	Project funding	
1	4	1000-000004	Batterie de tests à l'usage des enseignants po...	NaN	Massarenti Léonard	Project funding (Div. I-III)	Project funding	Psych Scier
2	5	1000-000005	Kritische Erstausgabe der 'Evidentiae contra D...	NaN	Kommission für das Corpus philosophorum medii ...	Project funding (Div. I-III)	Project funding	Komm philoso
3	6	1000-000006	Katalog der datierten Handschriften in der Sch...	NaN	Burckhardt Max	Project funding (Div. I-III)	Project funding	Hanc Alte Dr
4	7	1000-000007	Wissenschaftliche Mitarbeit am Thesaurus Lingu...	NaN	Schweiz. Thesauruskommission	Project funding (Div. I-III)	Project funding	Thesauru

In [9]:
```
persons.head(5)
```

Out[9]:

	Last Name	First Name	Gender	Institute Name	Institute Place	Person ID SNSF	OCRID	Projects as responsible Applicant	Projects as Applicant	Projects as Partner	Projec a Practi Partn
0	a Marca	Davide	male	NaN	NaN	53856	NaN	NaN	NaN	NaN	Na
1	a Marca	Andrea	male	NaN	NaN	132628	NaN	67368	NaN	NaN	Na
2	A. Jafari	Golnaz	female	Universität Luzern	Luzern	747886	NaN	191432	NaN	NaN	Na
3	Aaberg	Johan	male	NaN	NaN	575257	NaN	NaN	NaN	NaN	Na
4	Aahman	Josefin	female	NaN	NaN	629557	NaN	NaN	NaN	NaN	Na

We see that the `persons` table gives information such as the role of a person in various projects (applicant, employee etc.), her/his gender etc. The *project* table on the other side gives information such as the period of a grant, how much money was awarded etc.

What if we now wish to know for example:

- How much money is awarded on average depending on gender?
- How long does it typically take for a researcher to go from employee to applicant status on a grant?

We need a way to *link* the two tables, i.e. create a large table where *each row* corresponds to a single *observation* containing information from the two tables such as: applicant, gender, awarded funds, dates etc. We will now go through all necessary steps to achieve that goal.

12.2 Merging tables

If each row of the persons table contained a single observation with a single person and a single project (the same person would appear of course multiple times), we could just *join* the two tables based e.g. on the project ID. Unfortunately, in the persons table, each line corresponds to a *single researcher* with all projects IDs lumped together in a list. For example:

```
In [12]: persons.iloc[10041]
```

```
Out[12]: Last Name
         Bodenmann
         First Name
         Guy
         Gender
         male
         Institute Name                      Lehrstuhl für Klinische Psychologie Kind
         er/Jug...
         Institute Place
         Zürich
         Person ID SNSF
         47670
         OCRID                                                              0000-0003-
         0964-6409
         Projects as responsible Applicant    46820;56660;62901;109547;115948;128960;1
         29627;...
         Projects as Applicant                                             112141;1220
         90;166348
         Projects as Partner
         NaN
         Projects as Practice Partner
         NaN
         Projects as Employee
         62901
         Projects as Contact Person
         NaN
         Name: 10041, dtype: object
```

```
In [13]: persons.iloc[10041]['Projects as responsible Applicant']
```

```
Out[13]: '46820;56660;62901;109547;115948;128960;129627;129699;133004;146775;147634;17
         3270'
```

Therefore the first thing we need to do is to split those strings into actual lists. We can do that by using classic Python string splitting. We simply `apply` that function to the relevant columns. We need to take care of rows containing NaNs on which we cannot use `split()`. We create two series, one for applicants, one for employees:

```
In [14]: projID_a = persons['Projects as responsible Applicant'].apply(lambda x: x.sp
         lit(';') if not pd.isna(x) else np.nan)
         projID_e = persons['Projects as Employee'].apply(lambda x: x.split(';') if n
         ot pd.isna(x) else np.nan)
```

```
In [15]: projID_a
```

```
Out[15]: 0                                         NaN
         1                                     [67368]
         2                                    [191432]
         3                                         NaN
         4                                         NaN
                                  ...
         110811    [52821, 143769, 147153, 165510, 183584]
         110812                                    NaN
         110813                                    NaN
         110814                                    NaN
         110815                                    NaN
         Name: Projects as responsible Applicant, Length: 110816, dtype: object
```

```
In [17]: projID_a[10041]
```

```
Out[17]: ['46820',
          '56660',
          '62901',
          '109547',
          '115948',
          '128960',
          '129627',
          '129699',
          '133004',
          '146775',
          '147634',
          '173270']
```

Now, to avoid problems later we'll only keep rows that are not NaNs. We first add the two series to the dataframe and then remove NaNs:

```
In [18]: pd.isna(projID_a)
```

```
Out[18]: 0          True
         1         False
         2         False
         3          True
         4          True
                   ...
         110811    False
         110812     True
         110813     True
         110814     True
         110815     True
         Name: Projects as responsible Applicant, Length: 110816, dtype: bool
```

```
In [19]: applicants = persons.copy()
         applicants['projID'] = projID_a
         applicants = applicants[~pd.isna(projID_a)]

         employees = persons.copy()
         employees['projID'] = projID_e
         employees = employees[~pd.isna(projID_e)]
```

Now we want each of these projects to become a single line in the dataframe. Here we use a function that we haven't used before called `explode` which turns every element in a list into a row (a good illustration of the variety of available functions in Pandas):

```
In [20]: applicants = applicants.explode('projID')
         employees = employees.explode('projID')
```

```
In [21]: applicants.head(5)
```

Out[21]:

	Last Name	First Name	Gender	Institute Name	Institute Place	Person ID SNSF	OCRID	Projects as responsible Applicant	Projects as Applicant	Projects as Partner
1	a Marca	Andrea	male	NaN	NaN	132628	NaN	67368	NaN	NaN
2	A. Jafari	Golnaz	female	Universität Luzern	Luzern	747886	NaN	191432	NaN	NaN
7	Aapro	Matti S.	male	Clinique de Genolier F.M.H. Oncologie-Hématolo...	Genolier	3268	NaN	8532;9513	8155	NaN
7	Aapro	Matti S.	male	Clinique de Genolier F.M.H. Oncologie-Hématolo...	Genolier	3268	NaN	8532;9513	8155	NaN
11	Aas	Gregor	male	Lehrstuhl für Pflanzenphysiologie Universität ...	Bayreuth	36412	NaN	52037	NaN	NaN

So now we have one large table, where each row corresponds to a *single* applicant and a *single* project. We can finally do our merging operation where we combined information on persons and projects. We will do two such operations: one for applicants using the `projID_a` column for merging and one using the `projID_e` column. We have one last problem to fix:

```
In [22]: applicants.loc[1].projID
Out[22]: '67368'
```

```
In [23]: projects.loc[1]['Project Number']
Out[23]: 4
```

We need the project ID in the persons table to be a *number* and not a *string*. We can try to convert but get an error:

```
In [24]: applicants.projID = applicants.projID.astype(int)
         employees.projID = employees.projID.astype(int)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-fca9460da04e> in <module>
----> 1 applicants.projID = applicants.projID.astype(int)
      2 employees.projID = employees.projID.astype(int)

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/generic.
py in astype(self, dtype, copy, errors)
   5696         else:
   5697             # else, only a single dtype is given
-> 5698             new_data = self._data.astype(dtype=dtype, copy=copy, erro
rs=errors)
   5699             return self._constructor(new_data).__finalize__(self)
   5700

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/internal
s/managers.py in astype(self, dtype, copy, errors)
    580
    581     def astype(self, dtype, copy: bool = False, errors: str = "rais
e"):
--> 582         return self.apply("astype", dtype=dtype, copy=copy, errors=er
rors)
    583
    584     def convert(self, **kwargs):

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/internal
s/managers.py in apply(self, f, filter, **kwargs)
    440                 applied = b.apply(f, **kwargs)
    441             else:
--> 442                 applied = getattr(b, f)(**kwargs)
    443             result_blocks = _extend_blocks(applied, result_blocks)
    444

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/internal
s/blocks.py in astype(self, dtype, copy, errors)
    623             vals1d = values.ravel()
    624             try:
--> 625                 values = astype_nansafe(vals1d, dtype, copy=True)
    626             except (ValueError, TypeError):
    627                 # e.g. astype_nansafe can fail on object-dtype of str
ings

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/dtypes/c
ast.py in astype_nansafe(arr, dtype, copy, skipna)
    872         # work around NumPy brokenness, #1987
    873         if np.issubdtype(dtype.type, np.integer):
--> 874             return lib.astype_intsafe(arr.ravel(), dtype).reshape(ar
r.shape)
    875
    876         # if we have a datetime/timedelta array of objects

pandas/_libs/lib.pyx in pandas._libs.lib.astype_intsafe()

ValueError: invalid literal for int() with base 10: ''
```

It looks like we have a row that doesn't conform to expectation and only contains ''. Let's try to figure out what happened. First we find the location with the issue:

In [25]:
```
applicants[applicants.projID=='']
```

Out[25]:

	Last Name	First Name	Gender	Institute Name	Institute Place	Person ID SNSF	OCRID	Projects as responsible Applicant	Projects as Applicant	Proje Part
50947	Kleinewefers	Henner	male	Séminaire de politique économique, d'économie ...	Fribourg	10661	NaN	8;	NaN	N
62384	Massarenti	Léonard	male	Faculté de Psychologie et des Sciences de l'Ed...	Genève 4	11138	NaN	4;	NaN	N

Then we look in the original table:

In [26]:
```
persons.loc[50947]
```

Out[26]:
```
Last Name                                                                    Kle
inewefers
First Name
Henner
Gender
male
Institute Name                     Séminaire de politique économique, d'éco
nomie ...
Institute Place
Fribourg
Person ID SNSF
10661
OCRID
NaN
Projects as responsible Applicant
8;
Projects as Applicant
NaN
Projects as Partner
NaN
Projects as Practice Partner
NaN
Projects as Employee
NaN
Projects as Contact Person
NaN
Name: 50947, dtype: object
```

Unfortunately, as is often the case, we have a misformatting in the original table. The project as applicant entry has a single number but still contains the `;` sign. Therefore when we split the text, we end up with `['8','']`. Can we fix this? We can for example filter the table and remove rows where `projID` has length 0:

In [30]:
```
applicants = applicants[applicants.projID.apply(lambda x: len(x) > 0)]
employees = employees[employees.projID.apply(lambda x: len(x) > 0)]
```

Now we can convert the `projID` column to integer:

```
In [31]: applicants.projID = applicants.projID.astype(int)
         employees.projID = employees.projID.astype(int)
```

Finally we can use `merge` to combine both tables. We will combine the projects (on 'Project Number') and persons table (on 'projID_a' and 'projID_e'):

```
In [32]: merged_appl = pd.merge(applicants, projects, left_on='projID', right_on='Pro
         ject Number')
         merged_empl = pd.merge(employees, projects, left_on='projID', right_on='Proj
         ect Number')
```

```
In [33]: applicants.head(5)
```

Out[33]:

	Last Name	First Name	Gender	Institute Name	Institute Place	Person ID SNSF	OCRID	Projects as responsible Applicant	Projects as Applicant	Projects as Partner
1	a Marca	Andrea	male	NaN	NaN	132628	NaN	67368	NaN	NaN
2	A. Jafari	Golnaz	female	Universität Luzern	Luzern	747886	NaN	191432	NaN	NaN
7	Aapro	Matti S.	male	Clinique de Genolier F.M.H. Oncologie-Hématolo...	Genolier	3268	NaN	8532;9513	8155	NaN
7	Aapro	Matti S.	male	Clinique de Genolier F.M.H. Oncologie-Hématolo...	Genolier	3268	NaN	8532;9513	8155	NaN
11	Aas	Gregor	male	Lehrstuhl für Pflanzenphysiologie Universität ...	Bayreuth	36412	NaN	52037	NaN	NaN

12.3 Reformatting columns: time

We now have in those tables information on both scientists and projects. Among other things we now when each project of each scientist has started via the `Start Date` column:

```
In [34]: merged_empl['Start Date']
```

```
Out[34]: 0         01.04.1993
         1         01.04.1993
         2         01.04.1993
         3         01.04.1993
         4         01.04.1993
                      ...
         127126    01.04.1990
         127127    01.04.1991
         127128    01.11.1998
         127129    01.11.1992
         127130    01.10.2008
         Name: Start Date, Length: 127131, dtype: object
```

If we want to do computations with dates (e.g. measuring time spans) we have to change the type of the column. Currently it is indeed just a string. We could parse that string, but Pandas already offers tools to handle dates. For example we can use `pd.to_datetime` to transform the string into a Python `datetime` format. Let's create a new `date` column:

```
In [35]: merged_empl['date'] = pd.to_datetime(merged_empl['Start Date'])
         merged_appl['date'] = pd.to_datetime(merged_appl['Start Date'])
```

```
In [36]: merged_empl.iloc[0]['date']
Out[36]: Timestamp('1993-01-04 00:00:00')
```

```
In [37]: merged_empl.iloc[0]['date'].year
Out[37]: 1993
```

Let's add a year column to our dataframe:

```
In [38]: merged_empl['year'] = merged_empl.date.apply(lambda x: x.year)
         merged_appl['year'] = merged_appl.date.apply(lambda x: x.year)
```

12.4 Completing information

As we did in the introduction, we want to be able to broadly classify projects into three categories. We therefore search for a specific string ('Humanities', 'Mathematics','Biology') within the 'Discipline Name Hierarchy' column to create a new column called 'Field'^:

```
In [39]: science_types = ['Humanities', 'Mathematics','Biology']
         merged_appl['Field'] = merged_appl['Discipline Name Hierarchy'].apply(
             lambda el: next((y for y in [x for x in science_types if x in el] if y i
         s not None),None) if not pd.isna(el) else el)
```

We will use the amounts awarded in our analysis. Let's look at that column:

```
In [40]: merged_appl['Approved Amount']
Out[40]: 0                     20120.00
         1         data not included in P3
         2                    211427.00
         3                    174021.00
         4                      8865.00
                             ...
         74650                150524.00
         74651                346000.00
         74652                262960.00
         74653                449517.00
         74654               1433628.00
         Name: Approved Amount, Length: 74655, dtype: object
```

Problem: we have rows that are not numerical. Let's coerce that column to numerical:

```
In [41]: merged_appl['Approved Amount'] = pd.to_numeric(merged_appl['Approved Amount
         '], errors='coerce')
```

```
In [42]: merged_appl['Approved Amount']
```

```
Out[42]: 0          20120.0
         1              NaN
         2         211427.0
         3         174021.0
         4           8865.0
                     ...
         74650     150524.0
         74651     346000.0
         74652     262960.0
         74653     449517.0
         74654    1433628.0
         Name: Approved Amount, Length: 74655, dtype: float64
```

12.5 Data anaylsis

We are finally done tidying up our tables so that we can do proper data analysis. We can *aggregate* data to answer some questions.

12.5.1 Amounts by gender

Let's see for example what is the average amount awarded every year, split by gender. We keep only the 'Project funding' category to avoid obscuring the results with large funds awarded for specific projects (PNR etc):

```
In [44]: merged_projects = merged_appl[merged_appl['Funding Instrument Hierarchy'] ==
         'Project funding']
```

```
In [45]: grouped_gender = merged_projects.groupby(['Gender','year'])['Approved Amount
         '].mean().reset_index()
         grouped_gender
```

Out[45]:

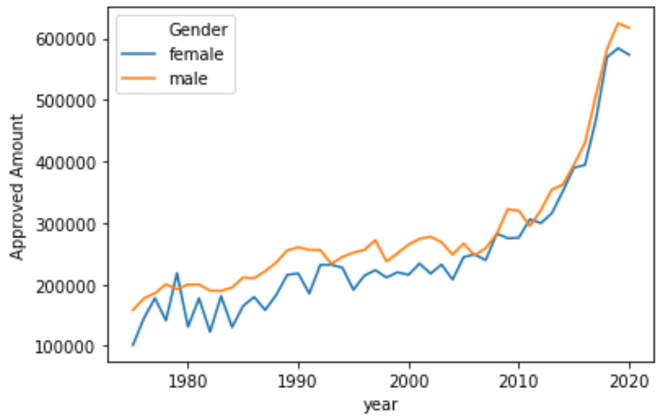
	Gender	year	Approved Amount
0	female	1975.0	101433.200000
1	female	1976.0	145017.750000
2	female	1977.0	177826.157895
3	female	1978.0	141489.857143
4	female	1979.0	218496.904762
...	...	...	...
87	male	2016.0	429717.055907
88	male	2017.0	507521.397098
89	male	2018.0	582461.020513
90	male	2019.0	624826.387985
91	male	2020.0	617256.523404

92 rows × 3 columns

To generate a plot, we use here Seaborn which uses some elements of a grammar of graphics. For example we can assign variables to each "aspect" of our plot. Here x and y axis are year and amount while color ('hue') is the gender. In one line, we can generate a plot that compiles all the information:

```
In [46]: sns.lineplot(data = grouped_gender, x='year', y='Approved Amount', hue='Gend
         er')
```

```
Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x122c5d0d0>
```



There seems to be a small but systematic difference in the average amount awarded.

We can now use a plotting library that is essentially a Python port of ggplot to add even more complexity to this plot. For example, let's split the data also by Field:

```
In [47]: import plotnine as p9
```

```
In [48]: grouped_gender_field = merged_projects.groupby(['Gender','year','Field'])['A
         pproved Amount'].mean().reset_index()
```

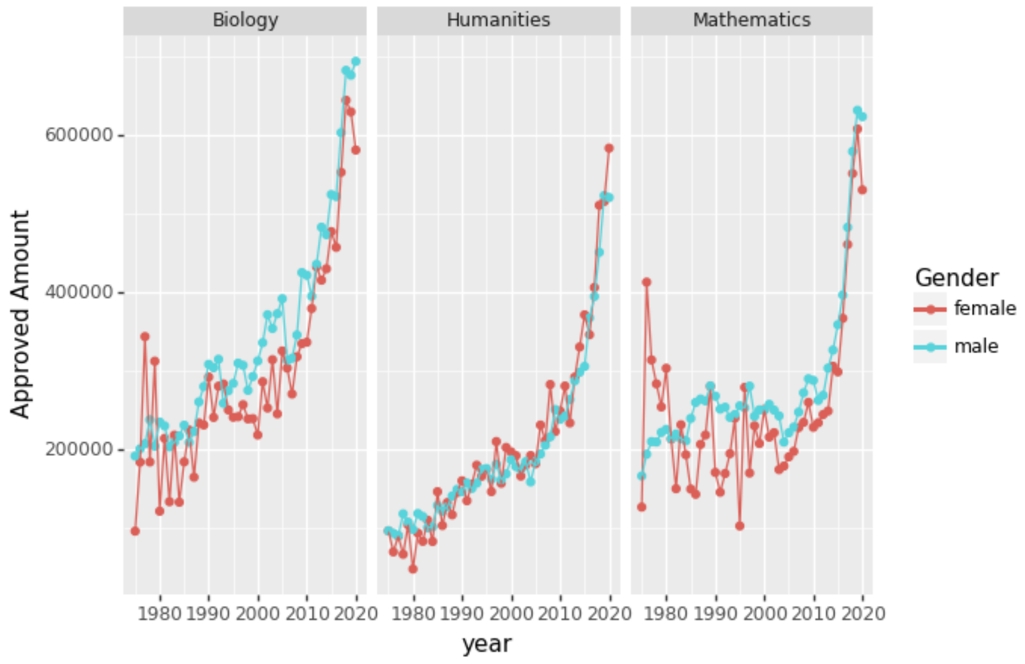
```
In [49]: grouped_gender_field
```

Out[49]:

	Gender	year	Field	Approved Amount
0	female	1975.0	Biology	95049.000000
1	female	1975.0	Humanities	95451.666667
2	female	1975.0	Mathematics	125762.000000
3	female	1976.0	Biology	183154.200000
4	female	1976.0	Humanities	68590.750000
...	...	...	...	...
271	male	2019.0	Humanities	523397.013072
272	male	2019.0	Mathematics	632188.796040
273	male	2020.0	Biology	694705.243590
274	male	2020.0	Humanities	520925.507246
275	male	2020.0	Mathematics	624141.068182

276 rows × 4 columns

```
In [50]: (p9.ggplot(grouped_gender_field, p9.aes('year', 'Approved Amount', color='Ge
         nder'))
          + p9.geom_point()
          + p9.geom_line()
          + p9.facet_wrap('~Field'))
```



```
Out[50]: <ggplot: (305412337)>
```

12.5.2 From employee to applicant

One of the questions we wanted to answer above was how much time goes by between the first time a scientist is mentioned as "employee" on an application and the first time he applies as main applicant. We have therefore to:

1. Find all rows corresponding to a specific scientist
2. Find the earliest date of project

For (1) we can use `groupby` and use the `Person ID SNSF` ID which is a unique ID assigned to each researcher. Once this *aggregation* is done, we can summarize each group by looking for the "minimal" date:

```
In [51]: first_empl = merged_empl.groupby('Person ID SNSF').date.min().reset_index()
         first_appl = merged_appl.groupby('Person ID SNSF').date.min().reset_index()
```

We have now two dataframes indexed by the `Person ID`:

```
In [52]: first_empl.head(5)
```

Out[52]:

	Person ID SNSF	date
0	1611	1990-01-10
1	1659	1988-01-11
2	1661	1978-01-07
3	1694	1978-01-06
4	1712	1982-01-04

Now we can again merge the two series to be able to compare applicant/employee start dates for single people:

```
In [53]: merge_first = pd.merge(first_appl, first_empl, on = 'Person ID SNSF', suffix
         es=('_appl', '_empl'))
```

```
In [54]: merge_first
```

Out[54]:

	Person ID SNSF	date_appl	date_empl
0	1659	1975-01-10	1988-01-11
1	1661	1978-01-07	1978-01-07
2	1694	1985-01-01	1978-01-06
3	1712	1982-01-04	1982-01-04
4	1726	1985-01-03	1985-01-03
...	...	...	...
10336	748652	2019-01-12	2019-01-12
10337	748760	2020-01-03	2020-01-03
10338	749430	2020-01-04	2020-01-04
10339	749991	2020-01-03	2020-01-03
10340	750593	2020-01-01	2020-01-01

10341 rows × 3 columns

Finally we merge with the full table, based on the index to recover the other paramters:

```
In [55]: full_table = pd.merge(merge_first, merged_appl,on = 'Person ID SNSF')
```

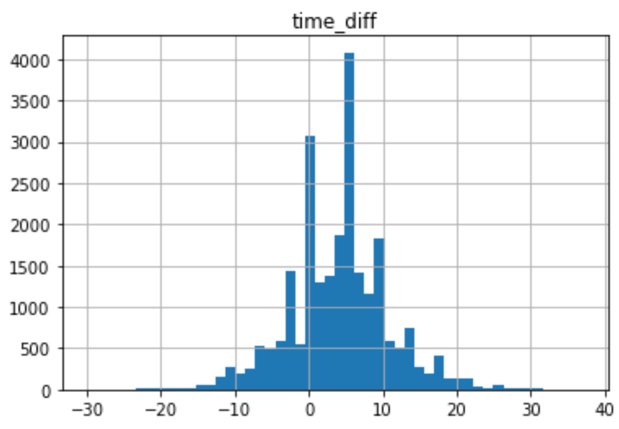
Finally we can add a column to that dataframe as a "difference in dates":

```
In [56]: full_table['time_diff'] = full_table.date_appl-full_table.date_empl
```

```
In [57]: full_table.time_diff = full_table.time_diff.apply(lambda x: x.days/365)
```

```
In [58]: full_table.hist(column='time_diff',bins = 50)
```

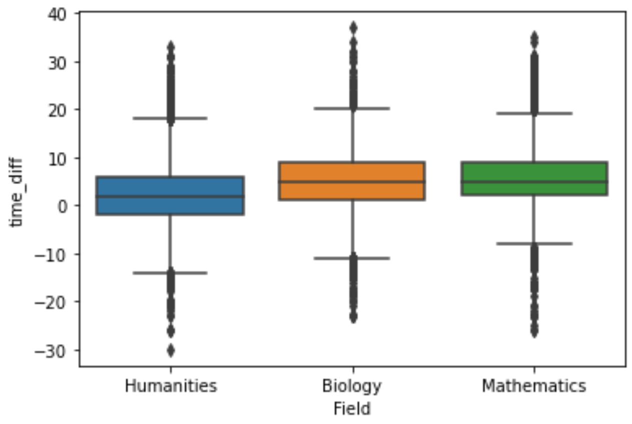
```
Out[58]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x12ba24970>]],
               dtype=object)
```



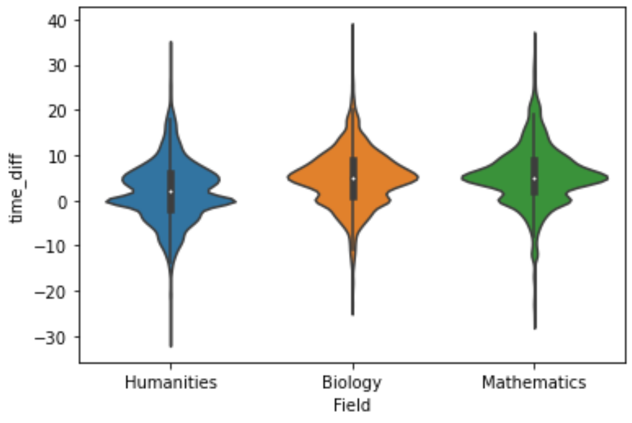
We see that we have one strong peak at $\Delta T == 0$ which corresponds to people who were paid for the first time through an SNSF grant when they applied themselves. The remaining cases have a peak around $\Delta T == 5$ which typically corresponds to the case where a PhD student was payed on a grant and then applied for a postdoc grant ~4-5 years later.

We can go further and ask how dependent this waiting time is on the Field of research. Obviously Humanities are structured very differently

```
In [60]: sns.boxplot(data=full_table, y='time_diff', x='Field');
```



```
In [61]: sns.violinplot(data=full_table, y='time_diff', x='Field', );
```



In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Exercice Numpy

1. Array creation

- Create a 1D array with values from 0 to 10 and in steps of 0.1. Check the shape of the array:

In [ ]:
```
```

- Create an array of normally distributed numbers with mean $\mu = 0$ and standard deviation $\sigma = 0.5$. It should have 20 rows and as many columns as there are elements in `xarray`. Call it `normal_array`:

In [ ]:
```
```

- Check the type of `normal_array`:

In [ ]:
```
```

2. Array mathematics

- Using `xarray` as x-variable, create a new array `yarray` as y-variable using the function $y = 10 * cos(x) * e^{-0.1x}$:

In [ ]:
```
```

- Create `array_abs` by taking the absolute value of `array_mul`:

In [ ]:
```
```

- Create a boolan array (logical array) where all positions > 0.3 in `array_abs` are `True` and the others `False`

In [ ]:
```
```

- Create a standard deviation projection along the second dimension (columns) of `array_abs`. Check that the dimensions are the ones you expected. Also are the values around the value you expect?

```
In [ ]:
```

3. Plotting

- Use a line plot to plot `yarray` vs `xarray` :

```
In [ ]:
```

- Try to change the color of the plot to red and to have markers on top of the line as squares:

```
In [ ]:
```

- Plot the `normal_array` as an imagage and change the colormap to 'gray':

```
In [ ]:
```

- Assemble the two above plots in a figure with one row and two columns grid:

```
In [ ]:
```

4. Indexing

- Create new arrays where you select every second element from xarray and yarray. Plot them on top of `xarray` and `yarray` .

```
In [ ]:
```

- Select all values of `yarray` that are larger than 0. Plot those on top of the regular `xarray` and `yarray` plot.

```
In [ ]:
```

- Flip the order of `xarray` use it to plot `yarray` :

```
In [ ]:
```

5. Combining arrays

- Create an array filled with ones with the same shape as `normal_array` . Concatenate it to `normal_array` along the first dimensions and plot the result:

```
In [ ]:
```

- `yarray` represents a signal. Each line of `normal_array` represents a possible random noise for that signal. Using broadcasting, try to create an array of noisy versions of `yarray` using `normal_array`. Finally, plot it:

```
In [ ]:
```

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Exercice Numpy

1. Array creation

- Create a 1D array with values from 0 to 10 and in steps of 0.1. Check the shape of the array:

In [145]:
```
xarray = np.arange(0,10,0.1)
xarray.shape
```

Out[145]:
```
(100,)
```

1.2. Create an array of normally distributed numbers with mean $\mu = 0$ and standard deviation $\sigma = 0.5$. It should have 20 rows and as many columns as there are elements in `xarray` . Call it `normal_array` :

In [146]:
```
normal_array = np.random.normal(0,0.5,(20, xarray.shape[0]))
```

- Check the type of `normal_array` :

In [147]:
```
normal_array.dtype
```

Out[147]:
```
dtype('float64')
```

2. Array mathematics

- Using `xarray` as x-variable, create a new array `yarray` as y-variable using the function $y = 10 * cos(x) * e^{-0.1x}$:

In [148]:
```
yarray = 5*np.cos(xarray)*np.exp(-0.1*xarray)
```

- 2.2 Create `array_abs` by taking the absolute value of `array_mul` :

In [149]:
```
array_abs = np.abs(yarray)
```

- 2.2 Create a boolan array (logical array) where all positions > 0.3 in `array_abs` are `True` and the others `False`

In [165]:
```
array_bool = array_abs > 0.3
```

- 2.3 Create a standard deviation projection along the second dimension (columns) of `array_abs` . Check that the dimensions are the ones you expected. Also are the values around the value you expect?

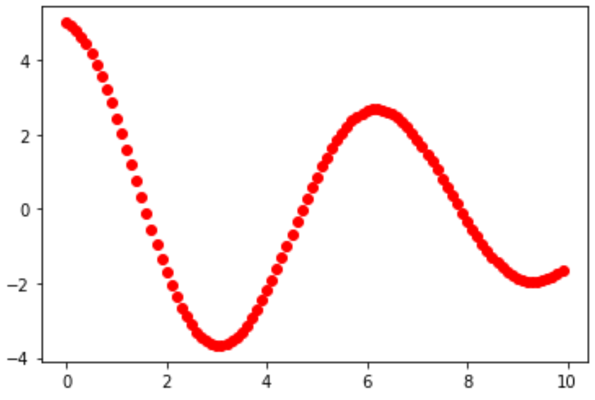
```
In [167]: array_min = normal_array.std(axis = 1)
          array_min.shape
Out[167]: (20,)
```

```
In [168]: array_min
Out[168]: array([0.54167658, 0.51651789, 0.4832876 , 0.54537271, 0.50834276,
                 0.47623427, 0.44677832, 0.47841273, 0.50255308, 0.50656681,
                 0.47822978, 0.52051232, 0.55511136, 0.46977863, 0.57914545,
                 0.47393849, 0.52705922, 0.43786828, 0.55795931, 0.45476456])
```

3. Plotting

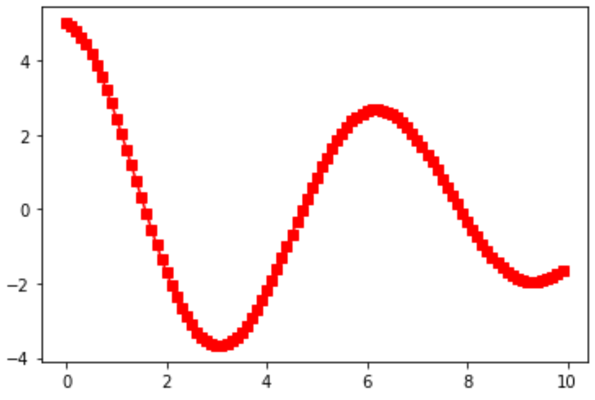
- Use a line plot to plot `yarray` vs `xarray` :

```
In [172]: plt.plot(xarray, yarray,'ro')
Out[172]: [<matplotlib.lines.Line2D at 0x11fb2b9d0>]
```



- Try to change the color of the plot to red and to have markers on top of the line as squares:

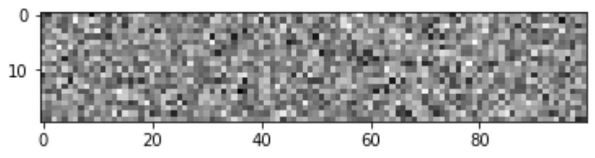
```
In [174]: plt.plot(xarray, yarray, '-sr')
Out[174]: [<matplotlib.lines.Line2D at 0x11f806070>]
```



- Plot the `normal_array` as an imagage and change the colormap to 'gray':

```
In [175]: plt.imshow(normal_array, cmap = 'gray')
```

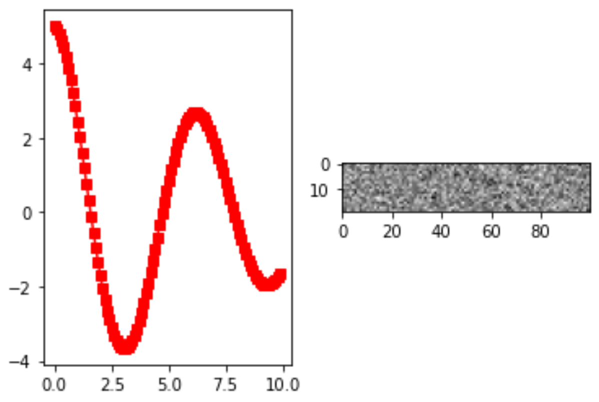
```
Out[175]: <matplotlib.image.AxesImage at 0x11fd9dfd0>
```



- Assemble the two above plots in a figure with one row and two columns grid:

```
In [176]: fig, ax = plt.subplots(1,2)
          ax[0].plot(xarray, yarray, '-sr')
          ax[1].imshow(normal_array, cmap = 'gray')
```

```
Out[176]: <matplotlib.image.AxesImage at 0x11fd9a340>
```

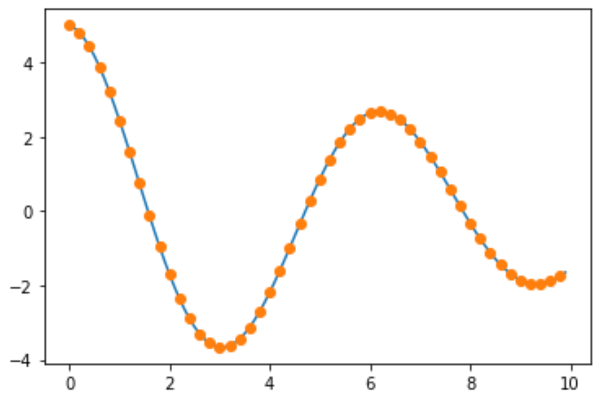


4. Indexing

- Create new arrays where you select every second element from xarray and yarray. Plot them on top of `xarray` and `yarray`.

In [179]:
```
xarray2 = xarray[::2]
yarray2 = yarray[::2]

plt.plot(xarray, yarray)
plt.plot(xarray2, yarray2,'o')
```

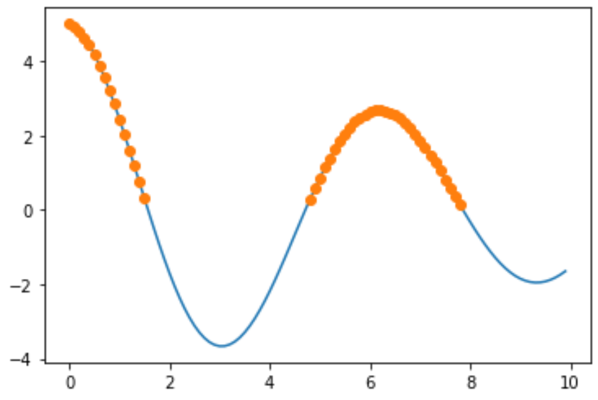
Out[179]:
```
[<matplotlib.lines.Line2D at 0x12045fbb0>]
```



- Select all values of `yarray` that are larger than 0. Plot those on top of the regular `xarray` and `yarray` plot.

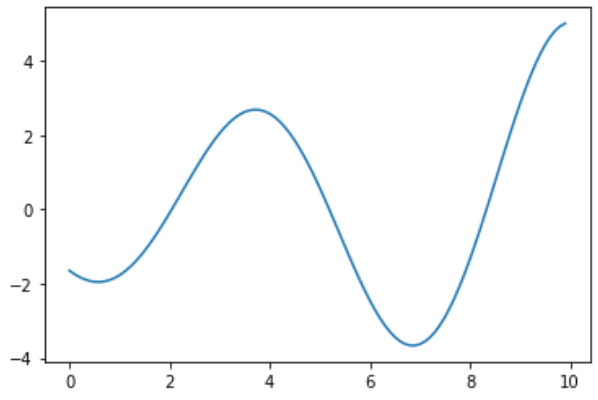
In [181]:
```
plt.plot(xarray, yarray)
plt.plot(xarray[yarray>0], yarray[yarray>0],'o')
```

Out[181]:
```
[<matplotlib.lines.Line2D at 0x1205f0880>]
```



- Flip the order of `xarray` use it to plot `yarray` :

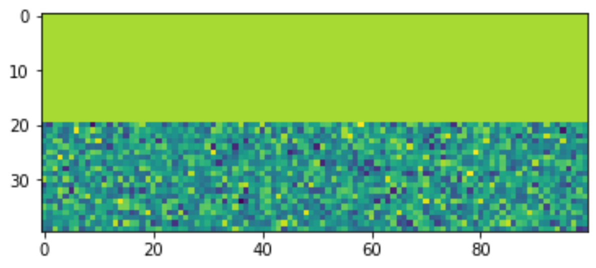
```
In [185]: flipped_array = np.flipud(xarray)
          plt.plot(flipped_array, yarray)

Out[185]: [<matplotlib.lines.Line2D at 0x120848dc0>]
```



5. Combining arrays

- Create an array filled with ones with the same shape as `normal_array` . Concatenate it to `normal_array` along the first dimensions and plot the result:

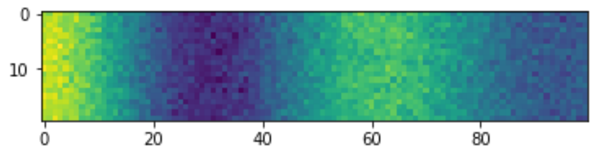
```
In [189]: ones_array = np.ones(normal_array.shape)
          concatenated = np.concatenate([ones_array, normal_array])

          plt.imshow(concatenated);
```



- `yarray` represents a signal. Each line of `normal_array` represents a possible random noise for that signal. Using broadcasting, try to create an array of noisy versions of `yarray` using `normal_array` . Finally, plot it:

The last dimensions of both arrays are matching. We can therefore simply added the two arrays, and `yarray` will simply be "replicated" as many times as needed:

```
In [194]: yarray_noise = yarray + normal_array

In [196]: plt.imshow(yarray_noise)

Out[196]: <matplotlib.image.AxesImage at 0x11b249b80>
```



In [21]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Exercise Pandas

For these exercices we are using a dataset (https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data/kernels) provided by Airbnb for a Kaggle competition. It describes its offer for New York City in 2019, including types of appartments, price, location etc.

1. Create a dataframe

Create a dataframe of a few lines with objects and their poperties (e.g fruits, their weight and colour). Calculate the mean of your Dataframe.

2. Import

- Import the table called `AB_NYC_2019.csv` as a dataframe. It is located in the Datasets folder. Have a look at the beginning of the table (head).
- Create a histogram of prices

3. Operations

Create a new column in the dataframe by multiplying the "price" and "availability_365" columns to get an estimate of the maximum yearly income.

3b. Subselection and plotting

Create a new Dataframe by first subselecting yearly incomes between 1 and 100'000. Then make a scatter plot of yearly income versus number of reviews

4. Combine

We provide below and additional table that contains the number of inhabitants of each of New York's boroughs ("neighbourhood_group" in the table). Use `merge` to add this population information to each element in the original dataframe.

5. Groups

- Using `groupby` calculate the average price for each type of room (room_type) in each neighbourhood_group. What is the average price for an entire home in Brooklyn ?
- Unstack the multi-level Dataframe into a regular Dataframe with `unstack()` and create a bar plot with the resulting table

6. Advanced plotting

Using Seaborn, create a scatter plot where x and y positions are longitude and lattitude, the color reflects price and the shape of the marker the borough (neighbourhood_group). Can you recognize parts of new york ? Does the map make sense ?

In [8]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Exercise

For these exercices we are using a dataset (https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data/kernels) provided by Airbnb for a Kaggle competition. It describes its offer for New York City in 2019, including types of appartments, price, location etc.

1. Create a dataframe

Create a dataframe of a few lines with objects and their poperties (e.g fruits, their weight and colour). Calculate the mean of your Dataframe.

In [5]:
```
fruits = pd.DataFrame({'fruits':['strawberry', 'orange','melon'], 'weight
':[20, 200, 1000],'weight2':[20, 200, 1000], 'color': ['red','orange','yello
w']})
```

In [6]:
```
fruits.describe()
```

Out[6]:

	weight	weight2
count	3.000000	3.000000
mean	406.666667	406.666667
std	521.664004	521.664004
min	20.000000	20.000000
25%	110.000000	110.000000
50%	200.000000	200.000000
75%	600.000000	600.000000
max	1000.000000	1000.000000

In [5]:
```
fruits.mean()
```

Out[5]:
```
weight    406.666667
dtype: float64
```

2. Import

- Import the table called `AB_NYC_2019.csv` as a dataframe. It is located in the Datasets folder. Have a look at the beginning of the table (head).
- Create a histogram of prices

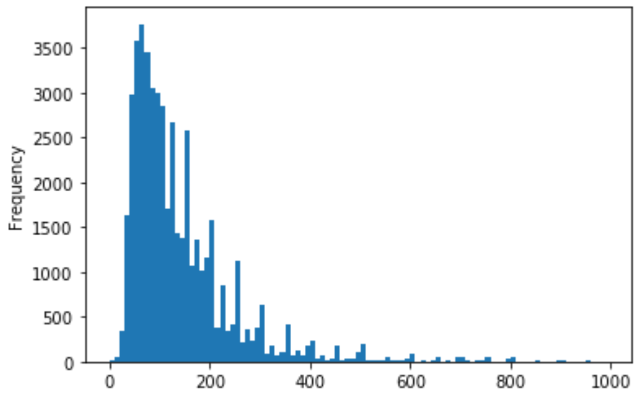
In [11]:
```
airbnb = pd.read_csv('Data/AB_NYC_2019.csv')
```

In [13]:
```
airbnb.head()
```

Out[13]:

	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude	longitude
0	2539	Clean & quiet apt home by the park	2787	John	Brooklyn	Kensington	40.64749	-73.97237
1	2595	Skylit Midtown Castle	2845	Jennifer	Manhattan	Midtown	40.75362	-73.98377
2	3647	THE VILLAGE OF HARLEM....NEW YORK !	4632	Elisabeth	Manhattan	Harlem	40.80902	-73.94190
3	3831	Cozy Entire Floor of Brownstone	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514	-73.95976
4	5022	Entire Apt: Spacious Studio/Loft by central park	7192	Laura	Manhattan	East Harlem	40.79851	-73.94399

In [17]:
```
airbnb['price'].plot(kind = 'hist', bins = range(0,1000,10));
```



3. Operations

Create a new column in the dataframe by multiplying the "price" and "availability_365" columns to get an estimate of the maximum yearly income.

In [18]:
```
airbnb['yearly_income'] = airbnb['price']*airbnb['availability_365']
```

In [19]:
```
airbnb['yearly_income']
```

Out[19]:
```
0        54385
1        79875
2        54750
3        17266
4            0
         ...  
48890      630
48891     1440
48892     3105
48893      110
48894     2070
Name: yearly_income, Length: 48895, dtype: int64
```

3b. Subselection and plotting

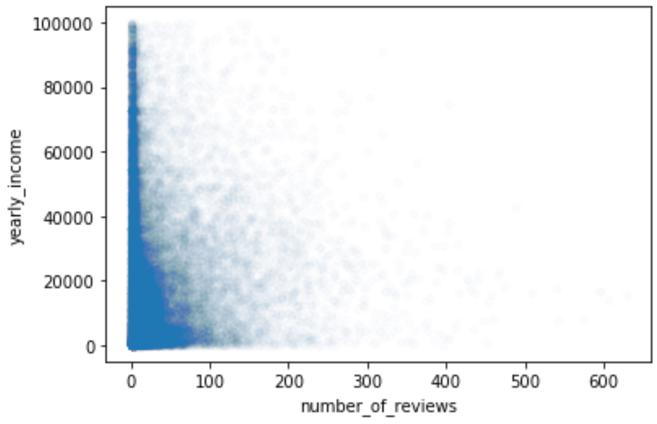
Create a new Dataframe by first subselecting yearly incomes between 1 and 100'000 and then by suppressing cases with 0 reviews. Then make a scatter plot of yearly income versus number of reviews

In [20]:
```
(airbnb.yearly_income>1)&(airbnb.yearly_income<100000)
```

Out[20]:
```
0         True
1         True
2         True
3         True
4        False
         ...
48890     True
48891     True
48892     True
48893     True
48894     True
Name: yearly_income, Length: 48895, dtype: bool
```

In [21]:
```
sub_airbnb = airbnb[(airbnb.yearly_income>1)&(airbnb.yearly_income<100000)].
copy()
```

In [22]:
```
sub_airbnb.plot(x = 'number_of_reviews', y = 'yearly_income', kind = 'scatte
r', alpha = 0.01)
plt.show()
```



4. Combine

We provide below and additional table that contains the number of inhabitants of each of New York's boroughs ("neighbourhood_group" in the table). Use `merge` to add this population information to each element in the original dataframe.

In [23]:
```
boroughs = pd.read_excel('Data/ny_boroughs.xlsx')
```

In [24]:
```
boroughs
```

Out[24]:

	borough	population
0	Brooklyn	2648771
1	Manhattan	1664727
2	Queens	2358582
3	Staten Island	479458
4	Bronx	1471160

In [25]:
```
airbnb
```

Out[25]:

	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude
0	2539	Clean & quiet apt home by the park	2787	John	Brooklyn	Kensington	40.64749
1	2595	Skylit Midtown Castle	2845	Jennifer	Manhattan	Midtown	40.75362
2	3647	THE VILLAGE OF HARLEM....NEW YORK !	4632	Elisabeth	Manhattan	Harlem	40.80902
3	3831	Cozy Entire Floor of Brownstone	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514
4	5022	Entire Apt: Spacious Studio/Loft by central park	7192	Laura	Manhattan	East Harlem	40.79851
...	...	...	...	...	...	...	...
48890	36484665	Charming one bedroom - newly renovated rowhouse	8232441	Sabrina	Brooklyn	Bedford-Stuyvesant	40.67853
48891	36485057	Affordable room in Bushwick/East Williamsburg	6570630	Marisol	Brooklyn	Bushwick	40.70184
48892	36485431	Sunny Studio at Historical Neighborhood	23492952	Ilgar & Aysel	Manhattan	Harlem	40.81475
48893	36485609	43rd St. Time Square-cozy single bed	30985759	Taz	Manhattan	Hell's Kitchen	40.75751
48894	36487245	Trendy duplex in the very heart of Hell's Kitchen	68119814	Christophe	Manhattan	Hell's Kitchen	40.76404

48895 rows × 17 columns

In [26]:
```
merged = pd.merge(airbnb, boroughs, left_on = 'neighbourhood_group', right_o
n='borough')
```

In [27]:
```
merged.head()
```

Out[27]:

	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude	longitude
0	2539	Clean & quiet apt home by the park	2787	John	Brooklyn	Kensington	40.64749	-73.97237
1	3831	Cozy Entire Floor of Brownstone	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514	-73.95976
2	5121	BlissArtsSpace!	7356	Garon	Brooklyn	Bedford-Stuyvesant	40.68688	-73.95596
3	5803	Lovely Room 1, Garden, Best Area, Legal rental	9744	Laurie	Brooklyn	South Slope	40.66829	-73.98779
4	6848	Only 2 stops to Manhattan studio	15991	Allen & Irina	Brooklyn	Williamsburg	40.70837	-73.95352

5. Groups

- Using `groupby` calculate the average price for each type of room (room_type) in each neighbourhood_group. What is the average price for an entire home in Brooklyn ?
- Unstack the multi-level Dataframe into a regular Dataframe with `unstack()` and create a bar plot with the resulting table

In [28]:
```
airbnb.groupby(['neighbourhood_group','room_type']).mean()
```

Out[28]:

		id	host_id	latitude	longitude	price	minimu
neighbourhood_group	**room_type**						
Bronx	Entire home/apt	2.269787e+07	1.037373e+08	40.848013	-73.880363	127.506596	
	Private room	2.235896e+07	1.060786e+08	40.849158	-73.886172	66.788344	
	Shared room	2.705442e+07	1.123450e+08	40.840873	-73.893407	59.800000	
Brooklyn	Entire home/apt	1.730117e+07	4.861704e+07	40.685211	-73.955603	178.327545	
	Private room	1.894125e+07	6.242636e+07	40.685513	-73.947150	76.500099	
	Shared room	2.358634e+07	1.040423e+08	40.669307	-73.948156	50.527845	
Manhattan	Entire home/apt	1.866860e+07	6.557697e+07	40.758266	-73.978402	249.239109	1
	Private room	1.880759e+07	6.982314e+07	40.776002	-73.968506	116.776622	
	Shared room	2.115615e+07	9.666720e+07	40.770035	-73.971700	88.977083	
Queens	Entire home/apt	2.112772e+07	8.713280e+07	40.728993	-73.874459	147.050573	
	Private room	2.197231e+07	1.008169e+08	40.732940	-73.871716	71.762456	
	Shared room	2.469434e+07	1.123200e+08	40.734411	-73.872973	69.020202	
Staten Island	Entire home/apt	2.170833e+07	9.618779e+07	40.605728	-74.109460	173.846591	
	Private room	2.106201e+07	1.017539e+08	40.614450	-74.103089	62.292553	
	Shared room	3.061484e+07	7.713866e+07	40.609894	-74.091077	57.444444	

In [29]:
```
summary = airbnb.groupby(['neighbourhood_group','room_type']).mean().price
```

In [30]:
```
summary
```

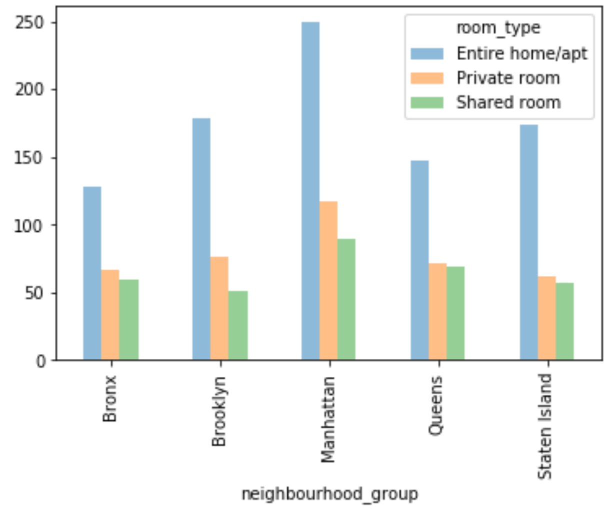
Out[30]:
```
neighbourhood_group  room_type
Bronx                Entire home/apt    127.506596
                     Private room        66.788344
                     Shared room         59.800000
Brooklyn             Entire home/apt    178.327545
                     Private room        76.500099
                     Shared room         50.527845
Manhattan            Entire home/apt    249.239109
                     Private room       116.776622
                     Shared room         88.977083
Queens               Entire home/apt    147.050573
                     Private room        71.762456
                     Shared room         69.020202
Staten Island        Entire home/apt    173.846591
                     Private room        62.292553
                     Shared room         57.444444
Name: price, dtype: float64
```

```
In [31]: summary[('Brooklyn','Entire home/apt')]
Out[31]: 178.32754472225128
```

```
In [32]: summary.unstack()
Out[32]:
```

room_type	Entire home/apt	Private room	Shared room
neighbourhood_group			
Bronx	127.506596	66.788344	59.800000
Brooklyn	178.327545	76.500099	50.527845
Manhattan	249.239109	116.776622	88.977083
Queens	147.050573	71.762456	69.020202
Staten Island	173.846591	62.292553	57.444444

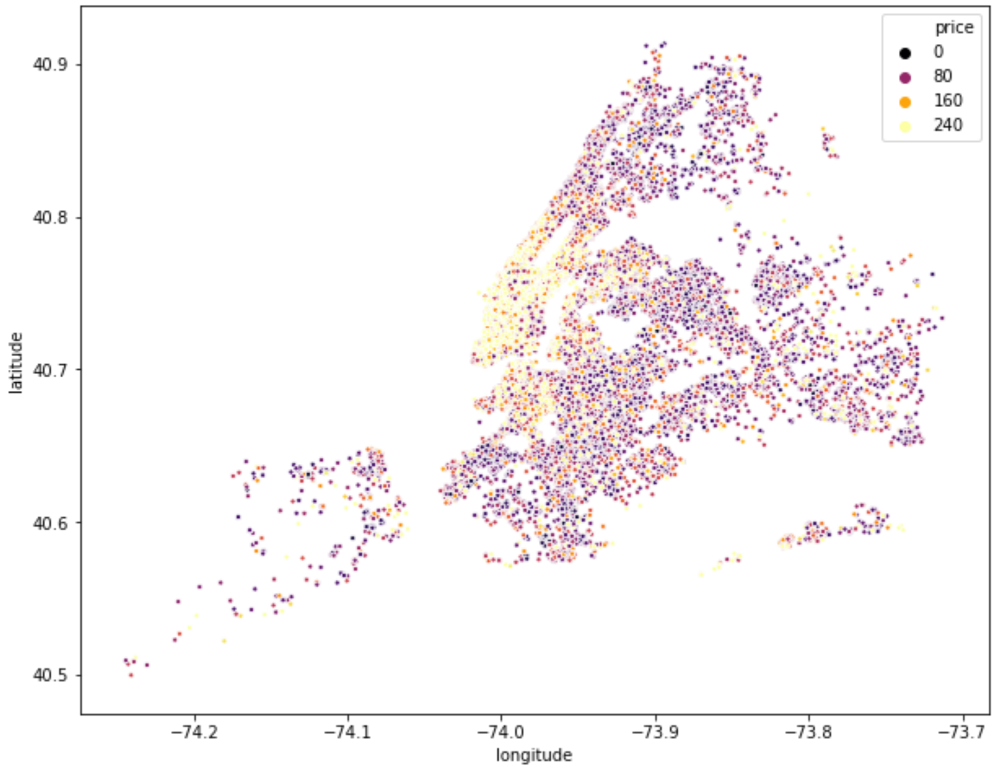
```
In [33]: summary.unstack().plot(kind = 'bar', alpha = 0.5)
         plt.show()
```



6. Advanced plotting

Using Seaborn, create a scatter plot where x and y positions are longitude and lattitude, the color reflects price and the shape of the marker the borough (neighbourhood_group). Can you recognize parts of new york ? Does the map make sense ?

```
In [32]: fig, ax = plt.subplots(figsize=(10,8))
         g = sns.scatterplot(data = airbnb, y = 'latitude', x = 'longitude', hue = 'p
         rice',
                             hue_norm=(0,200), s=10, palette='inferno')
```



```
In [ ]:
```