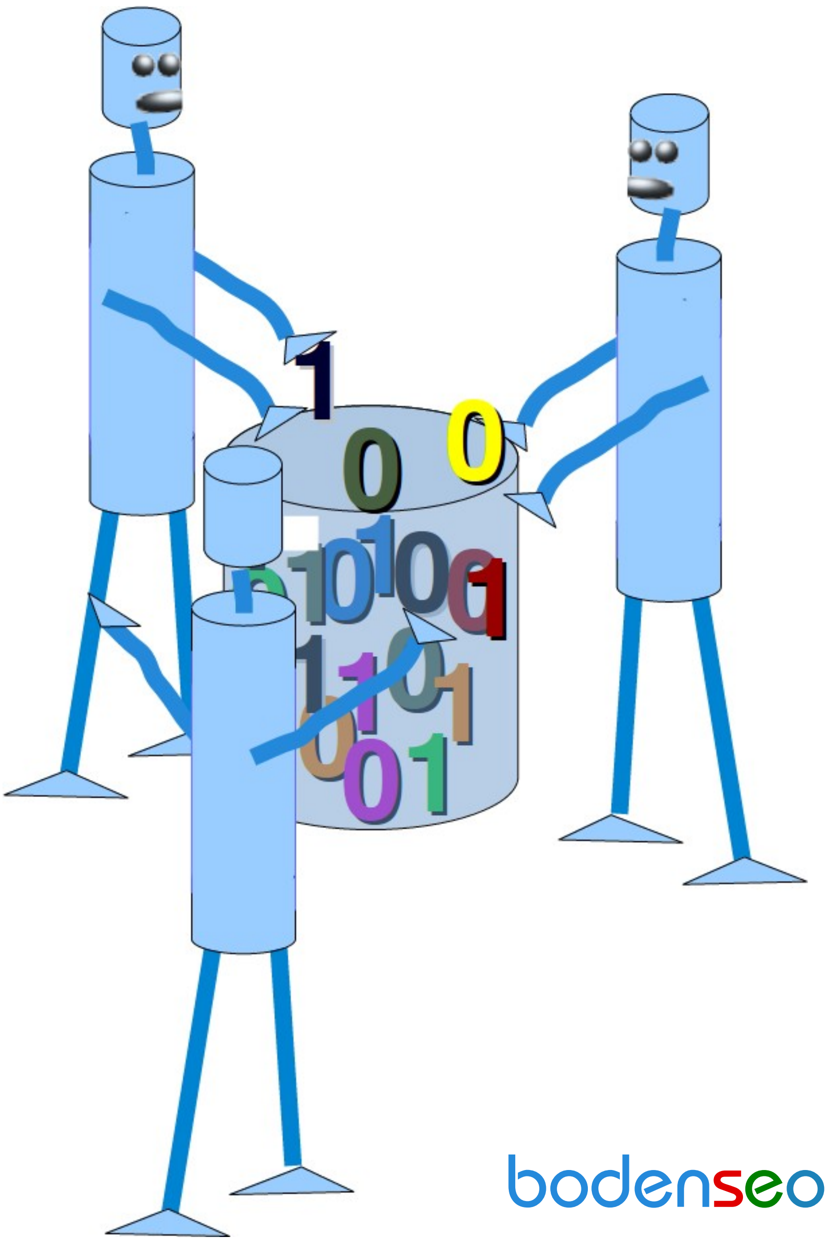
Data Analysis

Numpy, Matplotlib and Pandas

by
Bernd Klein



bodenseo

Python Course Data Analysis With Python by Bernd Klein

NUMERICAL PROGRAMMING WITH PYTHON

NUMERICAL PROGRAMMING DEFINITION

The term "Numerical Computing" - a.k.a. numerical computing or scientific computing - can be misleading. One can think about it as "having to do with numbers" as opposed to algorithms dealing with texts for example. If you think of Google and the way it provides links to websites for your search inquiries, you may think about the underlying algorithm as a text based one. Yet, the core of the Google search engine is numerical. To perform the PageRank algorithm Google executes the world's largest matrix computation.

Numerical Computing defines an area of computer science and mathematics dealing with algorithms for numerical approximations of problems from mathematical or numerical analysis, in other words: Algorithms solving problems involving continuous variables. Numerical analysis is used to solve science and engineering problems.

DATA SCIENCE AND DATA ANALYSIS

This tutorial can be used as an online course on Numerical Python as it is needed by Data Scientists and Data Analysts.

Data science is an interdisciplinary subject which includes for example statistics and computer science, especially programming and problem solving skills. Data Science includes everything which is necessary to create and prepare data, to manipulate, filter and clense data and to analyse data. Data can be both structured and unstructured. We could also say Data Science includes all the techniques needed to extract and gain information and insight from data.

Data Science is an umbrella term which incorporates data analysis, statistics, machine learning and other related scientific fields in order to understand and analyze data.

Another term occuring quite often in this context is "Big Data". Big Data is for sure one of the most often used buzzwords in the software-related marketing world. Marketing managers have found out that using this term can boost the sales of their products, regardless of the fact if they are really dealing with big data or not. The term is often used in fuzzy ways.

Big data is data which is too large and complex, so that it is hard for data-processing application software to deal with them. The problems include capturing and collecting data, data storage, search the data, visualization of the data, querying, and so on.

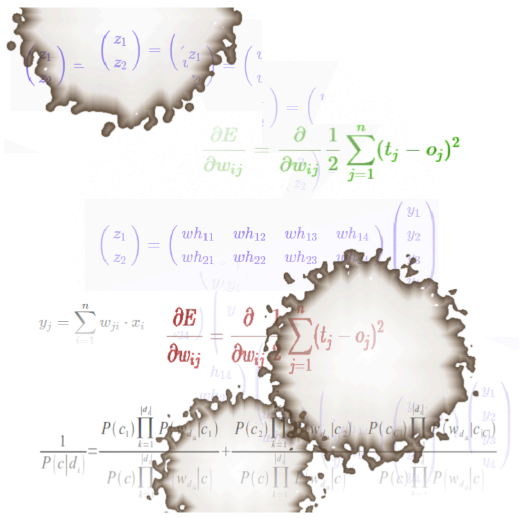
The following concepts are associated with big data:

- volume:
  the sheer amount of data, whether it will be giga-, tera-, peta- or exabytes
- velocity:
  the speed of arrival and processing of data
- veracity:

uncertainty or imprecision of data
- variety:
  the many sources and types of data both structured and unstructured

The big question is how useful Python is for these purposes. If we would only use Python without any special modules, this language could only poorly perform on the previously mentioned tasks. We will describe the necessary tools in the following chapter.

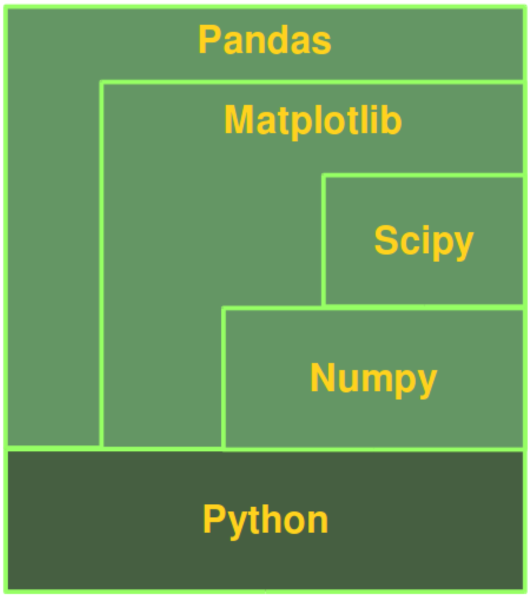
CONNECTIONS BETWEEN PYTHON, NUMPY, MATPLOTLIB, SCIPY AND PANDAS



Python is a general-purpose language and as such it can and it is widely used by system administrators for operating system administration, by web developpers as a tool to create dynamic websites and by linguists for natural language processing tasks. Being a truely general-purpose language, Python can of course - without using any special numerical modules - be used to solve numerical problems as well. So far so good, but the crux of the matter is the execution speed. Pure Python without any numerical modules couldn't be used for numerical tasks Matlab, R and other languages are designed for. If it comes to computational problem solving, it is of greatest importance to consider the performance of algorithms, both concerning speed and data usage.

If we use Python in combination with its modules NumPy, SciPy, Matplotlib and Pandas, it belongs to the top numerical programming languages. It is as efficient - if not even more efficient - than Matlab or R.

Numpy is a module which provides the basic data structures, implementing multi-dimensional arrays and matrices. Besides that the module supplies the necessary functionalities to create and manipulate these data structures. SciPy is based on top of Numpy, i.e. it uses the data structures provided by NumPy. It extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.



Matplotlib is a plotting library for the Python programming language and the numerically oriented modules like NumPy and SciPy.

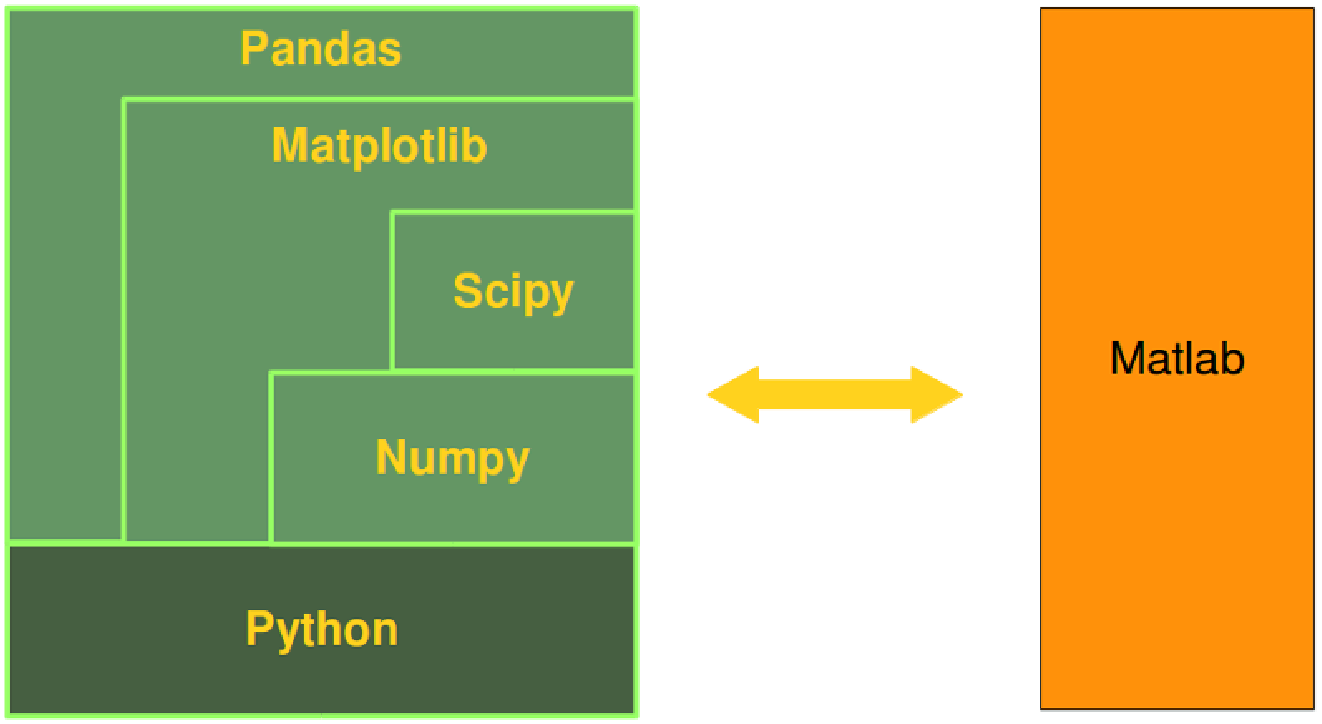
The youngest child in this family of modules is Pandas. Pandas is using all of the previously mentioned modules. It's build on top of them to provide a module for the Python language, which is also capable of data manipulation and analysis. The special focus of Pandas consists in offering data structures and operations for manipulating numerical tables and time series. The name is derived from the term "panel data". Pandas is well suited for working with tabular data as it is known from spread sheet programming like Excel.

PYTHON, AN ALTERNATIVE TO MATLAB

Python is becoming more and more the main programming language for data scientists. Yet, there are still many scientists and engineers in the scientific and engineering world that use R and MATLAB to solve their data analysis and data science problems. It's a question troubling lots of people, which language they should choose: The functionality of R was developed with statisticians in mind, whereas Python is a general-purpose language. Nevertheless, Python is also - in combination with its specialized modules, like Numpy, Scipy, Matplotlib, Pandas and so, - an ideal programming language for solving numerical problems. Furthermore, the community of Python is a lot larger and faster growing than the one from R.

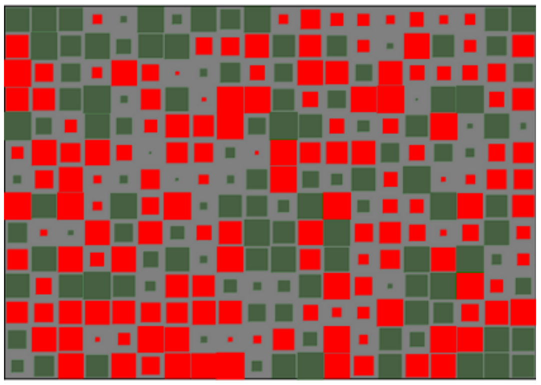
The principal disadvantage of MATLAB against Python are the costs. Python with NumPy, SciPy, Matplotlib and Pandas is completely free, whereas MATLAB can be very expensive. "Free" means both "free" as in "free beer" and "free" as in "freedom"! Even though MATLAB has a huge number of additional toolboxes available, Python has the advantage that it is a more modern and complete programming language. Python is continually becoming more powerful by a rapidly growing number of specialized modules.

Python in combination with Numpy, Scipy, Matplotlib and Pandas can be used as a complete replacement for MATLAB.


Pandas
Matplotlib
Scipy
Numpy
Python
Matlab

NUMPY TUTORIAL

INTRODUCTION



NumPy is a module for Python. The name is an acronym for "Numeric Python" or "Numerical Python". It is pronounced /ˈnʌmpaɪ/ (NUM-py) or less often /ˈnʌmpi (NUM-pee)). It is an extension module for Python, mostly written in C. This makes sure that the precompiled mathematical and numerical functions and functionalities of Numpy guarantee great execution speed.

Furthermore, NumPy enriches the programming language Python with powerful data structures, implementing multi-dimensional arrays and matrices. These data structures guarantee efficient calculations with matrices and arrays. The implementation is even aiming at huge matrices and arrays, better know under the heading of "big data". Besides that the module supplies a large library of high-level mathematical functions to operate on these matrices and arrays.

SciPy (Scientific Python) is often mentioned in the same breath with NumPy. SciPy needs Numpy, as it is based on the data structures of Numpy and furthermore its basic creation and manipulation functions. It extends the capabilities of NumPy with further useful functions for minimization, regression, Fourier-transformation and many others.

Both NumPy and SciPy are not part of a basic Python installation. They have to be installed after the Python installation. NumPy has to be installed before installing SciPy.

(Comment: The diagram of the image on the right side is the graphical visualisation of a matrix with 14 rows and 20 columns. It's a so-called Hinton diagram. The size of a square within this diagram corresponds to the size of the value of the depicted matrix. The colour determines, if the value is positive or negative. In our example: the colour red denotes negative values and the colour green denotes positive values.)

NumPy is based on two earlier Python modules dealing with arrays. One of these is Numeric. Numeric is like NumPy a Python module for high-performance, numeric computing, but it is obsolete nowadays. Another predecessor of NumPy is Numarray, which is a complete rewrite of Numeric but is deprecated as well. NumPy is a merger of those two, i.e. it is build on the code of Numeric and the features of Numarray.

COMPARISON BETWEEN CORE PYTHON AND NUMPY

When we say "Core Python", we mean Python without any special modules, i.e. especially without NumPy.

The advantages of Core Python:

- high-level number objects: integers, floating point
- containers: lists with cheap insertion and append methods, dictionaries with fast lookup

Advantages of using Numpy with Python:

- array oriented computing
- efficiently implemented multi-dimensional arrays
- designed for scientific computation

A SIMPLE NUMPY EXAMPLE

Before we can use NumPy we will have to import it. It has to be imported like any other module:

```
import numpy
```

But you will hardly ever see this. Numpy is usually renamed to np:

```
import numpy as np
```

Our first simple Numpy example deals with temperatures. Given is a list with values, e.g. temperatures in Celsius:

```
cvalues = [20.1, 20.8, 21.9, 22.5, 22.7, 22.3, 21.8, 21.2, 20.9, 2
0.1]
```

We will turn our list "cvalues" into a one-dimensional numpy array:

```
C = np.array(cvalues)
print(C)
```

```
[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```

Let's assume, we want to turn the values into degrees Fahrenheit. This is very easy to accomplish with a numpy array. The solution to our problem can be achieved by simple scalar multiplication:

```
print(C * 9 / 5 + 32)
```

```
[68.18 69.44 71.42 72.5  72.86 72.14 71.24 70.16 69.62 68.18]
```

The array C has not been changed by this expression:

```
print(C)
```

```
[20.1 20.8 21.9 22.5 22.7 22.3 21.8 21.2 20.9 20.1]
```

Compared to this, the solution for our Python list looks awkward:

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

```
[68.18, 69.44, 71.42, 72.5, 72.86, 72.14, 71.24000000000001, 70.1
6, 69.62, 68.18]
```

So far, we referred to C as an array. The internal type is "ndarray" or to be even more precise "C is an instance of the class numpy.ndarray":

```
type(C)
```

Output:
```
numpy.ndarray
```

In the following, we will use the terms "array" and "ndarray" in most cases synonymously.

GRAPHICAL REPRESENTATION OF THE VALUES

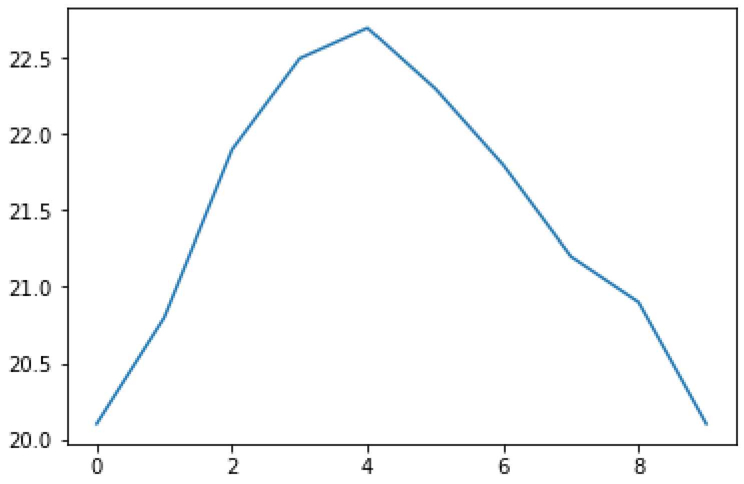
Even though we want to cover the module matplotlib not until a later chapter, we want to demonstrate how we can use this module to depict our temperature values. To do this, we us the package pyplot from matplotlib.

If you use the jupyter notebook, you might be well advised to include the following line of code to prevent an external window to pop up and to have your diagram included in the notebook:

```
%matplotlib inline
```

The code to generate a plot for our values looks like this:

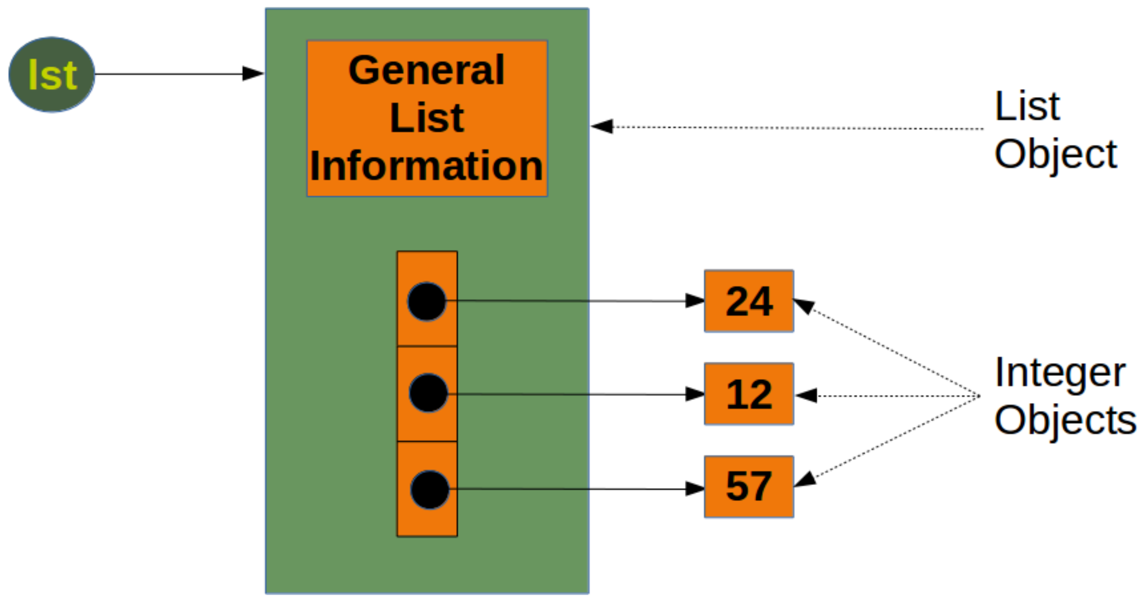
```
import matplotlib.pyplot as plt

plt.plot(C)
plt.show()
```



The function plot uses the values of the array C for the values of the ordinate, i.e. the y-axis. The indices of the array C are taken as values for the abscissa, i.e. the x-axis.

MEMORY CONSUMPTION: NDARRAY AND LIST

The main benefits of using numpy arrays should be smaller memory consumption and better runtime behaviour. We want to look at the memory usage of numpy arrays in this subchapter of our turorial and compare it to the memory consumption of Python lists.



To calculate the memory consumption of the list from the above picture, we will use the function getsizeof from the module sys.

```
from sys import getsizeof as size

lst = [24, 12, 57]

size_of_list_object = size(lst)   # only green box
size_of_elements = len(lst) * size(lst[0]) # 24, 12, 57

total_list_size = size_of_list_object + size_of_elements
print("Size without the size of the elements: ", size_of_list_obje
ct)
print("Size of all the elements: ", size_of_elements)
print("Total size of list, including elements: ", total_list_size)
```

```
Size without the size of the elements:  96
Size of all the elements:  84
Total size of list, including elements:  180
```

The size of a Python list consists of the general list information, the size needed for the references to the elements and the size of all the elements of the list. If we apply sys.getsizeof to a list, we get only the size without the size of the elements. In the previous example, we made the assumption that all the integer elements of our list have the same size. Of course, this is not valid in general, because memory consumption will be higher for larger integers.

We will check now, how the memory usage changes, if we add another integer element to the list. We also look at an empty list:

```
lst = [24, 12, 57, 42]

size_of_list_object = size(lst)   # only green box
size_of_elements = len(lst) * size(lst[0]) # 24, 12, 57, 42

total_list_size = size_of_list_object + size_of_elements
print("Size without the size of the elements: ", size_of_list_obje
ct)
print("Size of all the elements: ", size_of_elements)
print("Total size of list, including elements: ", total_list_size)

lst = []
print("Emtpy list size: ", size(lst))
```

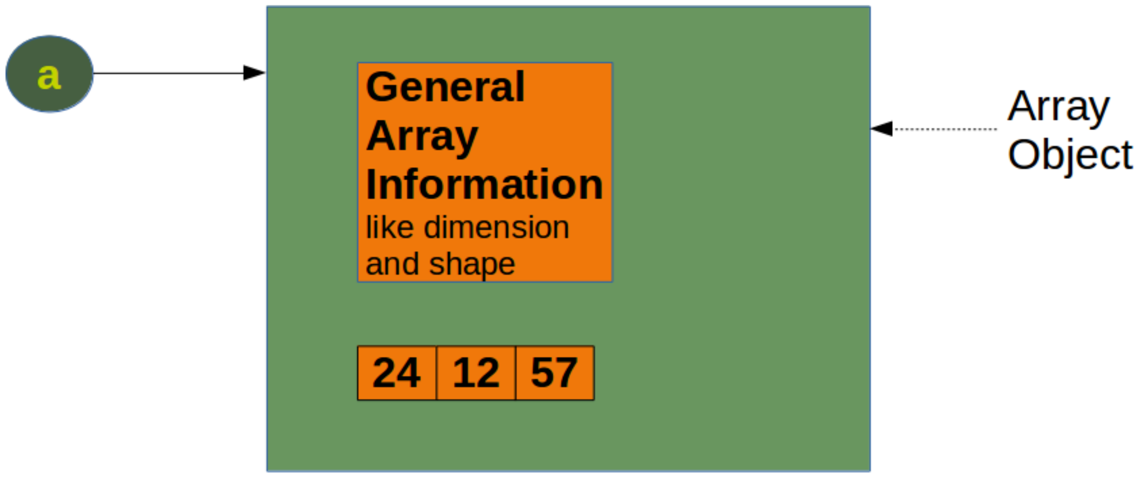
```
Size without the size of the elements:  104
Size of all the elements:  112
Total size of list, including elements:  216
Emtpy list size:  72
```

We can conclude from this that for every new element, we need another eight bytes for the reference to the new object. The new integer object itself consumes 28 bytes. The size of a list "lst" without the size of the elements can be calculated with:

64 + 8 * len(lst)

To get the complete size of an arbitrary list of integers, we have to add the sum of all the sizes of the integers.

We will examine now the memory consumption of a numpy.array. To this purpose, we will have a look at the implementation in the following picture:



We will create the numpy array of the previous diagram and calculate the memory usage:

```
a = np.array([24, 12, 57])
print(size(a))
```

```
120
```

We get the memory usage for the general array information by creating an empty array:

```
e = np.array([])
print(size(e))
```

```
96
```

We can see that the difference between the empty array "e" and the array "a" with three integers consists in 24 Bytes. This means that an arbitrary integer array of length "n" in numpy needs

96 + n * 8 Bytes

whereas a list of integers needs, as we have seen before

64 + 8 *len(lst)* + *len(lst)* 28

This is a minimum estimation, as Python integers can use more than 28 bytes.

When we define a Numpy array, numpy automatically chooses a fixed integer size. In our example "int64". We can determine the size of the integers, when we define an array. Needless to say, this changes the memory requirement:

```
a = np.array([24, 12, 57], np.int8)
print(size(a) - 96)

a = np.array([24, 12, 57], np.int16)
print(size(a) - 96)

a = np.array([24, 12, 57], np.int32)
print(size(a) - 96)

a = np.array([24, 12, 57], np.int64)
print(size(a) - 96)
```

```
3
6
12
24
```

TIME COMPARISON BETWEEN PYTHON LISTS AND NUMPY ARRAYS

One of the main advantages of NumPy is its advantage in time compared to standard Python. Let's look at the following functions:

```
import time
size_of_vec = 1000

def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = [X[i] + Y[i] for i in range(len(X)) ]
    return time.time() - t1

def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
```

Let's call these functions and see the time consumption:

```
t1 = pure_python_version()
t2 = numpy_version()

print(t1, t2)
print("Numpy is in this example " + str(t1/t2) + " faster!")
```

```
0.0010614395141601562 5.2928924560546875e-05
Numpy is in this example 20.054054054054053 faster!
```

It's an easier and above all better way to measure the times by using the timeit module. We will use the Timer class in the following script.

The constructor of a Timer object takes a statement to be timed, an additional statement used for setup, and a timer function. Both statements default to 'pass'.

The statements may contain newlines, as long as they don't contain multi-line string literals.

A Timer object has a timeit method. timeit is called with a parameter number:

`timeit(number=1000000)`

The main statement will be executed "number" times. This executes the setup statement once, and then returns the time it takes to execute the main statement a "number" of times. It returns the time in seconds.

```
import numpy as np
from timeit import Timer

size_of_vec = 1000

X_list = range(size_of_vec)
Y_list = range(size_of_vec)
```

```
X = np.arange(size_of_vec)
Y = np.arange(size_of_vec)

def pure_python_version():
    Z = [X_list[i] + Y_list[i] for i in range(len(X_list)) ]

def numpy_version():
    Z = X + Y


#timer_obj = Timer("x = x + 1", "x = 0")
timer_obj1 = Timer("pure_python_version()",
                   "from __main__ import pure_python_version")
timer_obj2 = Timer("numpy_version()",
                   "from __main__ import numpy_version")

for i in range(3):
    t1 = timer_obj1.timeit(10)
    t2 = timer_obj2.timeit(10)
    print("time for pure Python version: ", t1)
    print("time for Numpy version: ", t2)
    print(f"Numpy was {t1 / t2:7.2f} times faster!")
```

```
time for pure Python version:  0.0021230499987723306
time for Numpy version:  0.0004346180066931993
Numpy was    4.88 times faster!
time for pure Python version:  0.003020321993972175
time for Numpy version:  0.00014882600225973874
Numpy was   20.29 times faster!
time for pure Python version:  0.002028984992648475
time for Numpy version:  0.0002098319964716211
Numpy was    9.67 times faster!
```

The repeat() method is a convenience to call timeit() multiple times and return a list of results:

```
print(timer_obj1.repeat(repeat=3, number=10))
print(timer_obj2.repeat(repeat=3, number=10))
```

```
[0.0030275019962573424, 0.0029995880031492564, 0.0022120869980426505]
[6.104000203777105e-05, 0.0001641790004214272, 1.904800592456013e-05]
```

In [ ]:

NUMPY TUTORIAL: CREATING ARRAYS

We have alreday seen in the previous chapter of our Numpy tutorial that we can create Numpy arrays from lists and tuples. We want to introduce now further functions for creating basic arrays.

There are functions provided by Numpy to create arrays with evenly spaced values within a given interval. One 'arange' uses a given distance and the other one 'linspace' needs the number of elements and creates the distance automatically.

CREATION OF ARRAYS WITH EVENLY SPACED VALUES

ARANGE

The syntax of arange:

```
arange([start,] stop[, step], [, dtype=None])
```

arange returns evenly spaced values within a given interval. The values are generated within the half-open interval '[start, stop)' If the function is used with integers, it is nearly equivalent to the Python built-in function range, but arange returns an ndarray rather than a list iterator as range does. If the 'start' parameter is not given, it will be set to 0. The end of the interval is determined by the parameter 'stop'. Usually, the interval will not include this value, except in some cases where 'step' is not an integer and floating point round-off affects the length of output ndarray. The spacing between two adjacent values of the output array is set with the optional parameter 'step'. The default value for 'step' is 1. If the parameter 'step' is given, the 'start' parameter cannot be optional, i.e. it has to be given as well. The type of the output array can be specified with the parameter 'dtype'. If it is not given, the type will be automatically inferred from the other input arguments.

```
import numpy as np

a = np.arange(1, 10)
print(a)

x = range(1, 10)
```

```
print(x)     # x is an iterator
print(list(x))

# further arange examples:
x = np.arange(10.4)
print(x)
x = np.arange(0.5, 10.4, 0.8)
print(x)
```

```
[1 2 3 4 5 6 7 8 9]
range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[ 0.5  1.3  2.1  2.9  3.7  4.5  5.3  6.1  6.9  7.7  8.5  9.3 10.1]
```

Be careful, if you use a float value for the `step` parameter, as you can see in the following example:

```
np.arange(12.04, 12.84, 0.08)
```

Output:
```
array([12.04, 12.12, 12.2 , 12.28, 12.36, 12.44, 12.52, 12.6
, 12.68,
       12.76, 12.84])
```

The help of `arange` has to say the following for the `stop` parameter: "End of interval. The interval does not include this value, except in some cases where `step` is not an integer and floating point round-off affects the length of `out`. This is what happened in our example.

The following usages of `arange` is a bit offbeat. Why should we use float values, if we want integers as result. Anyway, the result might be confusing. Before arange starts, it will round the start value, end value and the stepsize:

```
x = np.arange(0.5, 10.4, 0.8, int)
print(x)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12]
```

This result defies all logical explanations. A look at help also helps here: "When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `numpy.linspace` for these cases. Using `linspace` is not an easy workaround in some situations, because the number of values has to be known.

LINSPACE

The syntax of linspace:

```
linspace(start, stop, num=50, endpoint=True, retstep=False)
```

linspace returns an ndarray, consisting of 'num' equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop). If a closed or a half-open interval will be returned, depends on whether 'endpoint' is True or False. The parameter 'start' defines the start value of the sequence which will be created. 'stop' will the end value of the sequence, unless 'endpoint' is set to False. In the latter case, the resulting sequence will consist of all but the last of 'num + 1' evenly spaced samples. This means that 'stop' is excluded. Note that the step size changes when 'endpoint' is False. The number of samples to be generated can be set with 'num', which defaults to 50. If the optional parameter 'endpoint' is set to True (the default), 'stop' will be the last sample of the sequence. Otherwise, it is not included.

```python
import numpy as np

# 50 values between 1 and 10:
print(np.linspace(1, 10))
# 7 values between 1 and 10:
print(np.linspace(1, 10, 7))
# excluding the endpoint:
print(np.linspace(1, 10, 7, endpoint=False))
```

```
[ 1.          1.18367347  1.36734694  1.55102041  1.73469388  1.91
836735
  2.10204082  2.28571429  2.46938776  2.65306122  2.83673469  3.02
040816
  3.20408163  3.3877551   3.57142857  3.75510204  3.93877551  4.12
244898
  4.30612245  4.48979592  4.67346939  4.85714286  5.04081633  5.22
44898
  5.40816327  5.59183673  5.7755102   5.95918367  6.14285714  6.32
653061
  6.51020408  6.69387755  6.87755102  7.06122449  7.24489796  7.42
857143
  7.6122449   7.79591837  7.97959184  8.16326531  8.34693878  8.53
061224
  8.71428571  8.89795918  9.08163265  9.26530612  9.44897959  9.63
265306
  9.81632653 10.        ]
[ 1.   2.5  4.   5.5  7.   8.5 10. ]
[1.         2.28571429 3.57142857 4.85714286 6.14285714 7.42857143
 8.71428571]
```

We haven't discussed one interesting parameter so far. If the optional parameter 'retstep' is set, the function will also return the value of the spacing between adjacent values. So, the function will return a tuple ('samples', 'step'):

```
import numpy as np

samples, spacing = np.linspace(1, 10, retstep=True)
print(spacing)
samples, spacing = np.linspace(1, 10, 20, endpoint=True, retstep=T
rue)
print(spacing)
samples, spacing = np.linspace(1, 10, 20, endpoint=False, retste
p=True)
print(spacing)
```

```
0.1836734693877551
0.47368421052631576
0.45
```

ZERO-DIMENSIONAL ARRAYS IN NUMPY

It's possible to create multidimensional arrays in numpy. Scalars are zero dimensional. In the following example, we will create the scalar 42. Applying the ndim method to our scalar, we get the dimension of the array. We can also see that the type is a "numpy.ndarray" type.

```
import numpy as np
x = np.array(42)
print("x: ", x)
print("The type of x: ", type(x))
print("The dimension of x:", np.ndim(x))
```

```
x:  42
The type of x:  <class 'numpy.ndarray'>
The dimension of x: 0
```

ONE-DIMENSIONAL ARRAYS

We have already encountered a 1-dimenional array - better known to some as vectors - in our initial example. What we have not mentioned so far, but what you may have assumed, is the fact that numpy arrays are containers of items of the same type, e.g. only integers. The homogenous type of the array can be determined with the attribute "dtype", as we can learn from the following example:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])
V = np.array([3.4, 6.9, 99.8, 12.8])
```

```
print("F: ", F)
print("V: ", V)
print("Type of F: ", F.dtype)
print("Type of V: ", V.dtype)
print("Dimension of F: ", np.ndim(F))
print("Dimension of V: ", np.ndim(V))
```

```
F:  [ 1  1  2  3  5  8 13 21]
V:  [ 3.4  6.9 99.8 12.8]
Type of F:  int64
Type of V:  float64
Dimension of F:  1
Dimension of V:  1
```

TWO- AND MULTIDIMENSIONAL ARRAYS

Of course, arrays of NumPy are not limited to one dimension. They are of arbitrary dimension. We create them by passing nested lists (or tuples) to the array method of numpy.

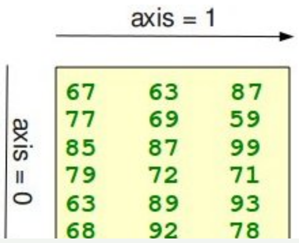
```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])
print(A)
print(A.ndim)
```

```
[[ 3.4  8.7  9.9]
 [ 1.1 -7.8 -0.7]
 [ 4.1 12.3  4.8]]
2
```

```
B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B)
print(B.ndim)
```

```
[[[111 112]
  [121 122]]

 [[211 212]
  [221 222]]

 [[311 312]
  [321 322]]]
3
```

SHAPE OF AN ARRAY



The function "shape" returns the shape of an array. The shape is a tuple of integers. These numbers denote the lengths of the corresponding array dimension. In other words: The "shape" of an array is a tuple with the number of elements per axis (dimension). In our example, the shape is equal to (6, 3), i.e. we have 6 lines and 3 columns.

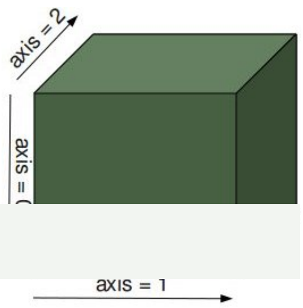
```
x = np.array([ [67, 63, 87],
               [77, 69, 59],
               [85, 87, 99],
               [79, 72, 71],
               [63, 89, 93],
               [68, 92, 78]])

print(np.shape(x))
```

```
(6, 3)
```

There is also an equivalent array property:

```
print(x.shape)
```

```
(6, 3)
```



The shape of an array tells us also something about the order in which the indices are processed, i.e. first rows, then columns and after that the further dimensions.

"shape" can also be used to change the shape of an array.

```
x.shape = (3, 6)
print(x)
```

```
[[67 63 87 77 69 59]
 [85 87 99 79 72 71]
 [63 89 93 68 92 78]]
```

```
x.shape = (2, 9)
print(x)
```

```
[[67 63 87 77 69 59 85 87 99]
 [79 72 71 63 89 93 68 92 78]]
```

You might have guessed by now that the new shape must correspond to the number of elements of the array, i.e. the total size of the new array must be the same as the old one. We will raise an exception, if this is not the case.

Let's look at some further examples.

The shape of a scalar is an empty tuple:

```
x = np.array(11)
print(np.shape(x))
```

```
()
```

```
B = np.array([ [[111, 112, 113], [121, 122, 123]],
               [[211, 212, 213], [221, 222, 223]],
               [[311, 312, 313], [321, 322, 323]],
               [[411, 412, 413], [421, 422, 423]] ])

print(B.shape)
```

```
(4, 2, 3)
```

INDEXING AND SLICING

Assigning to and accessing the elements of an array is similar to other sequential data types of Python, i.e. lists and tuples. We have also many options to indexing, which makes indexing in Numpy very powerful and similar to the indexing of lists and tuples.

Single indexing behaves the way, you will most probably expect it:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])
# print the first element of F
print(F[0])
# print the last element of F
print(F[-1])
```

```
1
21
```

Indexing multidimensional arrays:

```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])

print(A[1][0])
```

```
1.1
```

We accessed an element in the second row, i.e. the row with the index 1, and the first column (index 0). We accessed it the same way, we would have done with an element of a nested Python list.

You have to be aware of the fact, that way of accessing multi-dimensional arrays can be highly inefficient. The reason is that we create an intermediate array A[1] from which we access the element with the index 0. So it behaves similar to this:

```
tmp = A[1]
print(tmp)
print(tmp[0])
```

```
[ 1.1 -7.8 -0.7]
1.1
```

There is another way to access elements of multi-dimensional arrays in Numpy: We use only one pair of square brackets and all the indices are separated by commas:

```
print(A[1, 0])
```

```
1.1
```

We assume that you are familar with the slicing of lists and tuples. The syntax is the same in numpy for one-dimensional arrays, but it can be applied to multiple dimensions as well.

The general syntax for a one-dimensional array A looks like this:

`A[start:stop:step]`

We illustrate the operating principle of "slicing" with some examples. We start with the easiest case, i.e. the slicing of a one-dimensional array:

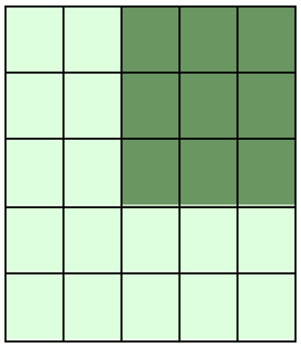
```
S = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(S[2:5])
print(S[:4])
print(S[6:])
print(S[:])
```

```
[2 3 4]
[0 1 2 3]
[6 7 8 9]
[0 1 2 3 4 5 6 7 8 9]
```

We will illustrate the multidimensional slicing in the following examples. The ranges for each dimension are separated by commas:

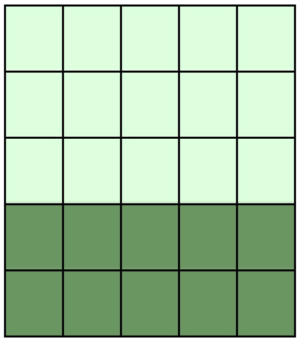
```
A = np.array([
[11, 12, 13, 14, 15],
[21, 22, 23, 24, 25],
[31, 32, 33, 34, 35],
[41, 42, 43, 44, 45],
[51, 52, 53, 54, 55]])

print(A[:3, 2:])
```

```
[[13 14 15]
 [23 24 25]
 [33 34 35]]
```



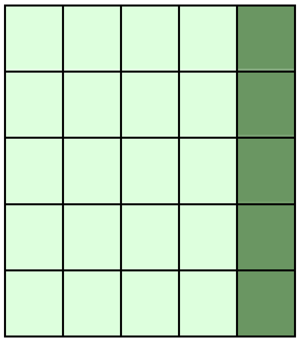
```
print(A[3:, :])
```

```
[[41 42 43 44 45]
 [51 52 53 54 55]]
```



```
print(A[:, 4:])
```

```
[[15]
 [25]
 [35]
 [45]
 [55]]
```



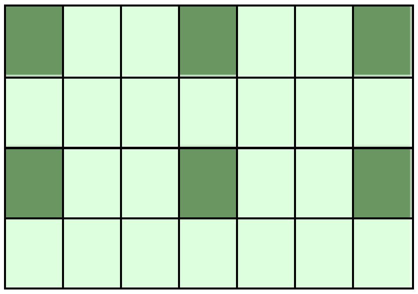
The following two examples use the third parameter "step". The reshape function is used to construct the two-dimensional array. We will explain reshape in the following subchapter:

```
X = np.arange(28).reshape(4, 7)
print(X)
```

```
[[ 0  1  2  3  4  5  6]
 [ 7  8  9 10 11 12 13]
 [14 15 16 17 18 19 20]
 [21 22 23 24 25 26 27]]
```

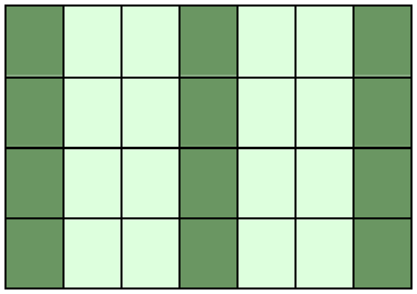
```
print(X[::2, ::3])
```

```
[[ 0  3  6]
 [14 17 20]]
```



```
print(X[::, ::3])
```

```
[[ 0  3  6]
 [ 7 10 13]
 [14 17 20]
 [21 24 27]]
```



If the number of objects in the selection tuple is less than the dimension N, then : is assumed for any subsequent dimensions:

```
        A = np.array(
    [ [ [45, 12, 4], [45, 13, 5], [46, 12, 6] ],
      [ [46, 14, 4], [45, 14, 5], [46, 11, 5] ],
      [ [47, 13, 2], [48, 15, 5], [52, 15, 1] ] ])

A[1:3, 0:2]  # equivalent to A[1:3, 0:2, :]
```

Output:

```
array([[[46, 14,  4],
        [45, 14,  5]],

       [[47, 13,  2],
        [48, 15,  5]]])
```

Attention: Whereas slicings on lists and tuples create new objects, a slicing operation on an array creates a view on the original array. So we get an another possibility to access the array, or better a part of the array. From this follows that if we modify a view, the original array will be modified as well.

```
A = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
S = A[2:6]
S[0] = 22
S[1] = 23
print(A)
```

```
[ 0  1 22 23  4  5  6  7  8  9]
```

Doing the similar thing with lists, we can see that we get a copy:

```
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
lst2 = lst[2:6]
lst2[0] = 22
lst2[1] = 23
print(lst)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If you want to check, if two array names share the same memory block, you can use the function np.may_share_memory.

```
np.may_share_memory(A, B)
```

To determine if two arrays A and B can share memory the memory-bounds of A and B are computed. The function returns True, if they overlap and False otherwise. The function may give false positives, i.e. if it returns True it just means that the arrays *may* be the same.

```
np.may_share_memory(A, S)
```

Output:
```
True
```

The following code shows a case, in which the use of may_share_memory is quite useful:

```
A = np.arange(12)
B = A.reshape(3, 4)
A[0] = 42
print(B)
```

```
[[42  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

We can see that A and B share the memory in some way. The array attribute "data" is an object pointer to the start of an array's data.

But we saw that if we change an element of one array the other one is changed as well. This fact is reflected by may_share_memory:

```
np.may_share_memory(A, B)
```

Output:
```
True
```

The result above is "false positive" example for may_share_memory in the sense that somebody may think that the arrays are the same, which is not the case.

CREATING ARRAYS WITH ONES, ZEROS AND EMPTY

There are two ways of initializing Arrays with Zeros or Ones. The method ones(t) takes a tuple t with the shape of the array and fills the array accordingly with ones. By default it will be filled with Ones of type float. If you need integer Ones, you have to set the optional parameter dtype to int:

```
import numpy as np

E = np.ones((2,3))
print(E)

F = np.ones((3,4),dtype=int)
print(F)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

What we have said about the method ones() is valid for the method zeros() analogously, as we can see in the following example:

```
Z = np.zeros((2,4))
print(Z)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

There is another interesting way to create an array with Ones or with Zeros, if it has to have the same shape as another existing array 'a'. Numpy supplies for this purpose the methods ones_like(a) and zeros_like(a).

```
x = np.array([2,5,18,14,4])
E = np.ones_like(x)
print(E)

Z = np.zeros_like(x)
print(Z)
```

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

There is also a way of creating an array with the `empty` function. It creates and returns a reference to a new array of given shape and type, without initializing the entries. Sometimes the entries are zeros, but you shouldn't be mislead. Usually, they are arbitrary values.

```
np.empty((2, 4))
```

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

COPYING ARRAYS

NUMPY.COPY()

`copy(obj, order='K')`

Return an array copy of the given object 'obj'.

Parameter	Meaning
obj	array_like input data.
order	The possible values are {'C', 'F', 'A', 'K'}. This parameter controls the memory layout of the copy. 'C' means C-order, 'F' means Fortran-order, 'A' means 'F' if the object 'obj' is Fortran contiguous, 'C' otherwise. 'K' means match the layout of 'obj' as closely as possible.

```
import numpy as np

x = np.array([[42,22,12],[44,53,66]], order='F')
y = x.copy()

x[0,0] = 1001
print(x)

print(y)
```

```
[[1001   22   12]
 [  44   53   66]]
[[42 22 12]
 [44 53 66]]
```

```
print(x.flags['C_CONTIGUOUS'])
print(y.flags['C_CONTIGUOUS'])
```

```
False
True
```

NDARRAY.COPY()

There is also a ndarray method 'copy', which can be directly applied to an array. It is similiar to the above function, but the default values for the order arguments are different.

```
a.copy(order='C')
```

Returns a copy of the array 'a'.

Parameter	Meaning
order	The same as with numpy.copy, but 'C' is the default value for order.

```
import numpy as np

x = np.array([[42,22,12],[44,53,66]], order='F')
y = x.copy()
x[0,0] = 1001
print(x)
print(y)

print(x.flags['C_CONTIGUOUS'])
print(y.flags['C_CONTIGUOUS'])
```

```
[[1001   22   12]
 [  44   53   66]]
[[42 22 12]
 [44 53 66]]
False
True
```

IDENTITY ARRAY

In linear algebra, the identity matrix, or unit matrix, of size n is the n × n square matrix with ones on the main diagonal and zeros elsewhere.

There are two ways in Numpy to create identity arrays:

- identy
- eye

THE IDENTITY FUNCTION

We can create identity arrays with the function identity:

`identity(n, dtype=None)`

The parameters:

Parameter	Meaning
n	An integer number defining the number of rows and columns of the output, i.e. 'n' x 'n'
dtype	An optional argument, defining the data-type of the output. The default is 'float'

The output of identity is an 'n' x 'n' array with its main diagonal set to one, and all other elements are 0.

```
import numpy as np

np.identity(4)
```

Output:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

```
np.identity(4, dtype=int) # equivalent to np.identity(3, int)
```

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

THE EYE FUNCTION

Another way to create identity arrays provides the function eye. This function creates also diagonal arrays consisting solely of ones.

It returns a 2-D array with ones on the diagonal and zeros elsewhere.

```
eye(N, M=None, k=0, dtype=float)
```

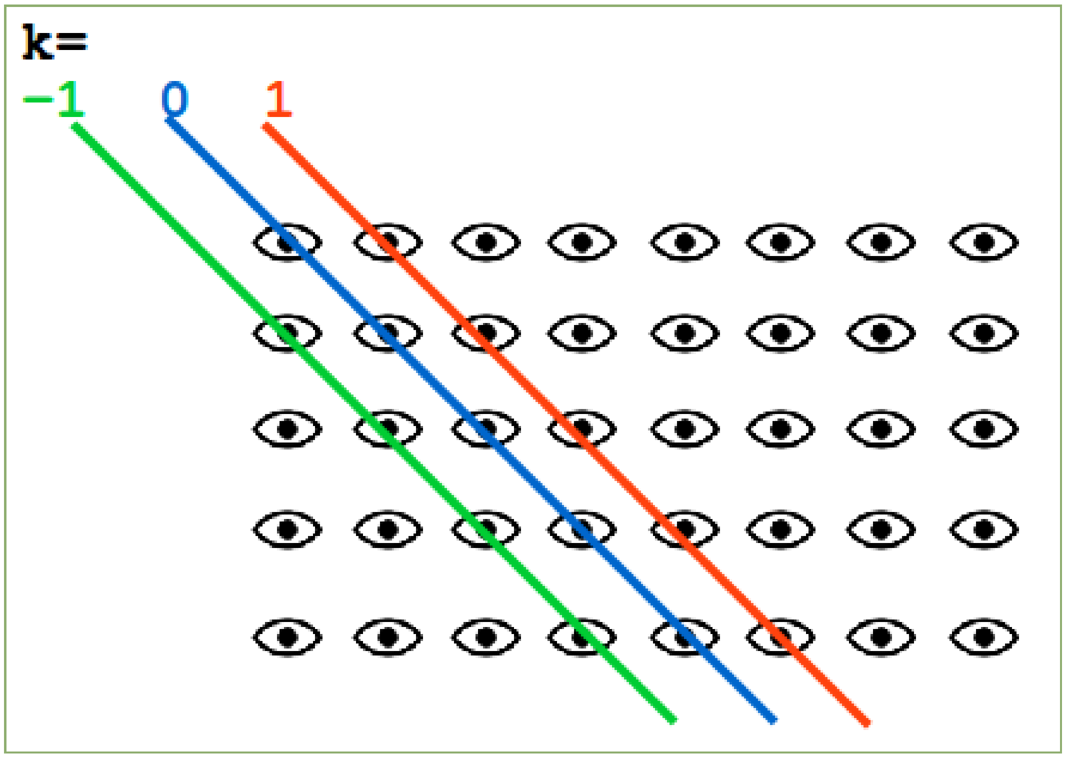
Parameter	Meaning
N	An integer number defining the rows of the output array.
M	An optional integer for setting the number of columns in the output. If it is None, it defaults to 'N'.
k	Defining the position of the diagonal. The default is 0. 0 refers to the main diagonal. A positive value refers to an upper diagonal, and a negative value to a lower diagonal.
dtype	Optional data-type of the returned array.

eye returns an ndarray of shape (N,M). All elements of this array are equal to zero, except for the 'k'-th diagonal, whose values are equal to one.

```
import numpy as np

np.eye(5, 8, k=1, dtype=int)
```

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

The principle of operation of the parameter 'd' of the eye function is illustrated in the following diagram:



EXERCISES:

1) Create an arbitrary one dimensional array called "v".

2) Create a new array which consists of the odd indices of previously created array "v".

3) Create a new array in backwards ordering from v.

4) What will be the output of the following code:

```
a = np.array([1, 2, 3, 4, 5])
```

```
        b = a[1:4]
        b[0] = 200
        print(a[1])
```

5) Create a two dimensional array called "m".

6) Create a new array from m, in which the elements of each row are in reverse order.

7) Another one, where the rows are in reverse order.

8) Create an array from m, where columns and rows are in reverse order.

9) Cut of the first and last row and the first and last column.

SOLUTIONS TO THE EXERCISES:

1)

```
        import numpy as np
        a = np.array([3,8,12,18,7,11,30])
```

2)

```
    odd_elements = a[1::2]
```

3) reverse_order = a[::-1]

4) The output will be 200, because slices are views in numpy and not copies.

5) m = np.array([[11, 12, 13, 14], [21, 22, 23, 24], [31, 32, 33, 34]])

6) m[::,::-1]

7) m[::-1]

8) m[::-1,::-1]

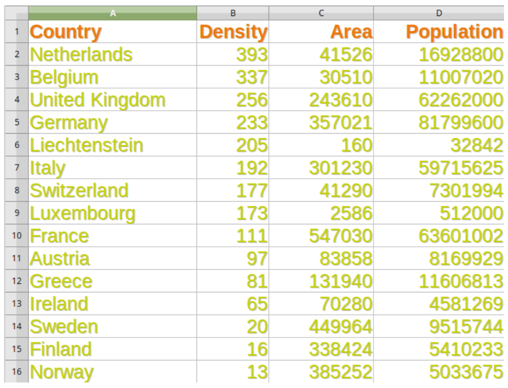
9) m[1:-1,1:-1]

DATA TYPE OBJECTS, DTYPE

DTYPE

The data type object 'dtype' is an instance of numpy.dtype class. It can be created with numpy.dtype.

So far, we have used in our examples of numpy arrays only fundamental numeric data types like 'int' and 'float'. These numpy arrays contained solely homogenous data types. dtype objects are construed by combinations of fundamental data types. With the aid of dtype we are capable to create "Structured Arrays", - also known as "Record Arrays". The structured arrays provide us with the ability to have different data types per column. It has similarity to the structure of excel or csv documents. This makes it possibe to define data like the one in the following table with dtype:



	A	B	C	D
1	Country	Density	Area	Population
2	Netherlands	393	41526	16928800
3	Belgium	337	30510	11007020
4	United Kingdom	256	243610	62262000
5	Germany	233	357021	81799600
6	Liechtenstein	205	160	32842
7	Italy	192	301230	59715625
8	Switzerland	177	41290	7301994
9	Luxembourg	173	2586	512000
10	France	111	547030	63601002
11	Austria	97	83858	8169929
12	Greece	81	131940	11606813
13	Ireland	65	70280	4581269
14	Sweden	20	449964	9515744
15	Finland	16	338424	5410233
16	Norway	13	385252	5033675

Country	Population Density	Area	Population
Netherlands	393	41526	16,928,800
Belgium	337	30510	11,007,020
United Kingdom	256	243610	62,262,000
Germany	233	357021	81,799,600
Liechtenstein	205	160	32,842
Italy	192	301230	59,715,625
Switzerland	177	41290	7,301,994
Luxembourg	173	2586	512,000
France	111	547030	63,601,002
Austria	97	83858	8,169,929

Country	Population Density	Area	Population
Greece	81	131940	11,606,813
Ireland	65	70280	4,581,269
Sweden	20	449964	9,515,744
Finland	16	338424	5,410,233
Norway	13	385252	5,033,675

Before we start with a complex data structure like the previous data, we want to introduce dtype in a very simple example. We define an int16 data type and call this type i16. (We have to admit, that this is not a nice name, but we use it only here!). The elements of the list 'lst' are turned into i16 types to create the two-dimensional array A.

```
import numpy as np

i16 = np.dtype(np.int16)
print(i16)

lst = [ [3.4, 8.7, 9.9],
        [1.1, -7.8, -0.7],
        [4.1, 12.3, 4.8] ]

A = np.array(lst, dtype=i16)

print(A)
```

```
int16
[[ 3  8  9]
 [ 1 -7  0]
 [ 4 12  4]]
```

We introduced a new name for a basic data type in the previous example. This has nothing to do with the structured arrays, which we mentioned in the introduction of this chapter of our dtype tutorial.

STRUCTURED ARRAYS

ndarrays are homogeneous data objects, i.e. all elements of an array have to be of the same data type. The data type dytpe on the other hand allows as to define separate data types for each column.

Now we will take the first step towards implementing the table with European countries and the information on population, area and population density. We create a structured array with the 'density' column. The data type is defined as `np.dtype([('density', np.int)])`. We assign this data type to the variable 'dt' for the sake of convenience. We use this data type in the darray definition, in which we use the first three densities.

```
import numpy as np

dt = np.dtype([('density', np.int32)])

x = np.array([(393,), (337,), (256,)],
             dtype=dt)

print(x)

print("\nThe internal representation:")
print(repr(x))
```

```
[(393,) (337,) (256,)]

The internal representation:
array([(393,), (337,), (256,)],
      dtype=[('density', '<i4')])
```

We can access the content of the density column by indexing x with the key 'density'. It looks like accessing a dictionary in Python:

```
print(x['density'])
```

```
[393 337 256]
```

You may wonder that we have used 'np.int32' in our definition and the internal representation shows '<i4'. We can use in the dtype definition the type directly (e.g. np.int32) or we can use a string (e.g. 'i4'). So, we could have defined our dtype like this as well:

```
dt = np.dtype([('density', 'i4')])
x = np.array([(393,), (337,), (256,)],
             dtype=dt)
print(x)
```

```
[(393,) (337,) (256,)]
```

The 'i' means integer and the 4 means 4 bytes. What about the less-than sign in front of i4 in the result? We could have written '<i4' in our definition as well. We can prefix a type with the '<' and '>' sign. '<' means that the encoding will be little-endian and '>' means that the encoding will be big-endian. No prefix means that we get the native byte ordering. We demonstrate this in the following by defining a double-precision floating-point number in various orderings:

```
# little-endian ordering
dt = np.dtype('<d')
print(dt.name, dt.byteorder, dt.itemsize)

# big-endian ordering
dt = np.dtype('>d')
print(dt.name, dt.byteorder, dt.itemsize)

# native byte ordering
dt = np.dtype('d')
print(dt.name, dt.byteorder, dt.itemsize)
```

```
float64 = 8
float64 > 8
float64 = 8
```

The equal character '=' stands for 'native byte ordering', defined by the operating system. In our case this means 'little-endian', because we use a Linux computer.

Another thing in our density array might be confusing. We defined the array with a list containing one-tuples. So you may ask yourself, if it is possible to use tuples and lists interchangeably? This is not possible. The tuples are used to define the records - in our case consisting solely of a density - and the list is the 'container' for the records or in other words 'the lists are cursed upon'. The tuples define the atomic elements of the structure and the lists the dimensions.

Now we will add the country name, the area and the population number to our data type:

```
dt = np.dtype([('country', 'S20'), ('density', 'i4'), ('area', 'i
4'), ('population', 'i4')])
population_table = np.array([
    ('Netherlands', 393, 41526, 16928800),
    ('Belgium', 337, 30510, 11007020),
    ('United Kingdom', 256, 243610, 62262000),
    ('Germany', 233, 357021, 81799600),
    ('Liechtenstein', 205, 160, 32842),
    ('Italy', 192, 301230, 59715625),
    ('Switzerland', 177, 41290, 7301994),
    ('Luxembourg', 173, 2586, 512000),
    ('France', 111, 547030, 63601002),
    ('Austria', 97, 83858, 8169929),
    ('Greece', 81, 131940, 11606813),
    ('Ireland', 65, 70280, 4581269),
    ('Sweden', 20, 449964, 9515744),
    ('Finland', 16, 338424, 5410233),
    ('Norway', 13, 385252, 5033675)],
    dtype=dt)
print(population_table[:4])
```

```
[(b'Netherlands', 393,  41526, 16928800)
 (b'Belgium', 337,  30510, 11007020)
 (b'United Kingdom', 256, 243610, 62262000)
 (b'Germany', 233, 357021, 81799600)]
```

We can acces every column individually:

```
print(population_table['density'])
print(population_table['country'])
print(population_table['area'][2:5])
```

```
[393 337 256 233 205 192 177 173 111  97  81  65  20  16  13]
[b'Netherlands' b'Belgium' b'United Kingdom' b'Germany' b'Liechten
stein'
 b'Italy' b'Switzerland' b'Luxembourg' b'France' b'Austria' b'Gree
ce'
 b'Ireland' b'Sweden' b'Finland' b'Norway']
[243610 357021    160]
```

UNICODE STRINGS IN ARRAY

Some may have noticed that the strings in our previous array have been prefixed with a lower case "b". This means that we have created binary strings with the definition "('country', 'S20')". To get unicode strings we exchange this with the definition "('country', np.unicode, 20)". We will redefine our population table now:

```
dt = np.dtype([('country', np.unicode, 20),
               ('density', 'i4'),
               ('area', 'i4'),
               ('population', 'i4')])
population_table = np.array([
    ('Netherlands', 393, 41526, 16928800),
    ('Belgium', 337, 30510, 11007020),
    ('United Kingdom', 256, 243610, 62262000),
    ('Germany', 233, 357021, 81799600),
    ('Liechtenstein', 205, 160, 32842),
    ('Italy', 192, 301230, 59715625),
    ('Switzerland', 177, 41290, 7301994),
    ('Luxembourg', 173, 2586, 512000),
    ('France', 111, 547030, 63601002),
    ('Austria', 97, 83858, 8169929),
    ('Greece', 81, 131940, 11606813),
    ('Ireland', 65, 70280, 4581269),
    ('Sweden', 20, 449964, 9515744),
    ('Finland', 16, 338424, 5410233),
    ('Norway', 13, 385252, 5033675)],
    dtype=dt)
print(population_table[:4])
```

```
[('Netherlands', 393,  41526, 16928800) ('Belgium', 337,  30510, 1
1007020)
 ('United Kingdom', 256, 243610, 62262000)
 ('Germany', 233, 357021, 81799600)]
```

INPUT AND OUTPUT OF STRUCTURED ARRAYS

In most applications it will be necessary to save the data from a program into a file. We will write our previously created "darray" to a file with the command savetxt. You will find a detailled introduction into this topic in our chapter Reading and Writing Data Files

```
np.savetxt("population_table.csv",
           population_table,
           fmt="%s;%d;%d;%d",
           delimiter=";")
```

It is highly probable that you will need to read in the previously written file at a later date. This can be achieved with the function genfromtxt.

```
dt = np.dtype([('country', np.unicode, 20), ('density', 'i4'), ('a
rea', 'i4'), ('population', 'i4')])

x = np.genfromtxt("population_table.csv",
               dtype=dt,
               delimiter=";")

print(x)
```

```
[('Netherlands', 393,  41526, 16928800) ('Belgium', 337,  30510, 1
1007020)
 ('United Kingdom', 256, 243610, 62262000)
 ('Germany', 233, 357021, 81799600)
 ('Liechtenstein', 205,    160,    32842) ('Italy', 192, 301230, 5
9715625)
 ('Switzerland', 177,  41290,  7301994)
 ('Luxembourg', 173,   2586,   512000) ('France', 111, 547030, 636
01002)
 ('Austria',  97,  83858,  8169929) ('Greece',  81, 131940, 116068
13)
 ('Ireland',  65,  70280,  4581269) ('Sweden',  20, 449964,  95157
44)
 ('Finland',  16, 338424,  5410233) ('Norway',  13, 385252,  50336
75)]
```

There is also a function "loadtxt", but it is more difficult to use, because it returns the strings as binary strings!

To overcome this problem, we can use loadtxt with a converter function for the first column.

```
dt = np.dtype([('country', np.unicode, 20), ('density', 'i4'), ('a
rea', 'i4'), ('population', 'i4')])

x = np.loadtxt("population_table.csv",
               dtype=dt,
               converters={0: lambda x: x.decode('utf-8')},
               delimiter=";")

print(x)
```

```
[('Netherlands', 393,  41526, 16928800) ('Belgium', 337,  30510, 1
1007020)
 ('United Kingdom', 256, 243610, 62262000)
 ('Germany', 233, 357021, 81799600)
 ('Liechtenstein', 205,    160,    32842) ('Italy', 192, 301230, 5
9715625)
 ('Switzerland', 177,  41290,  7301994)
 ('Luxembourg', 173,   2586,   512000) ('France', 111, 547030, 636
01002)
 ('Austria',  97,  83858,  8169929) ('Greece',  81, 131940, 116068
13)
 ('Ireland',  65,  70280,  4581269) ('Sweden',  20, 449964,  95157
44)
 ('Finland',  16, 338424,  5410233) ('Norway',  13, 385252,  50336
75)]
```

EXERCISES:

Before you go on, you may take time to do some exercises to deepen the understanding of the previously learned stuff.

1. Exercise:
   Define a structured array with two columns. The first column contains the product ID, which can be defined as an int32. The second column shall contain the price for the product. How can you print out the column with the product IDs, the first row and the price for the third article of this structured array?

2. Exercise:
   Figure out a data type definition for time records with entries for hours, minutes and seconds.

SOLUTIONS:

Solution to the first exercise:

```
import numpy as np

mytype = [('productID', np.int32), ('price', np.float64)]

stock = np.array([(34765, 603.76),
                  (45765, 439.93),
                  (99661, 344.19),
                  (12129, 129.39)], dtype=mytype)

print(stock[1])
print(stock["productID"])
print(stock[2]["price"])
print(stock)
```

```
(45765,  439.93)
[34765 45765 99661 12129]
344.19
[(34765,  603.76) (45765,  439.93) (99661,  344.19) (12129,  129.3
9)]
```

Solution to the second exercise:

```
time_type = np.dtype( [('h', int), ('min', int),
('sec', int)])

times = np.array([(11, 38, 5),
                  (14, 56, 0),
                  (3, 9, 1)], dtype=time_type)
print(times)
print(times[0])
# reset the first time record:
times[0] = (11, 42, 17)
print(times[0])
```

```
[(11, 38, 5) (14, 56, 0) ( 3,  9, 1)]
(11, 38, 5)
(11, 42, 17)
```

A MORE COMPLEX EXAMPLE:

We will increase the complexity of our previous example by adding temperatures to the records.

```
time_type = np.dtype( np.dtype([('time', [('h', int), ('min', in
t), ('sec', int)]),
                                ('temperature', float)] ))

times = np.array( [((11, 42, 17), 20.8), ((13, 19, 3), 23.2) ], dt
ype=time_type)
print(times)
print(times['time'])
print(times['time']['h'])
print(times['temperature'])
```

```
[((11, 42, 17),  20.8) ((13, 19,  3),  23.2)]
[(11, 42, 17) (13, 19,  3)]
[11 13]
[ 20.8  23.2]
```

EXERCISE

This exercise should be closer to real life examples. Usually, we have to create or get the data for our structured array from some data base or file. We will use the list, which we have created in our chapter on file I/O "File Management". The list has been saved with the aid of pickle.dump in the file cities_and_times.pkl.

So the first task consists in unpickling our data:

```
import pickle
fh = open("cities_and_times.pkl", "br")
cities_and_times = pickle.load(fh)
print(cities_and_times[:30])
```

```
[('Amsterdam', 'Sun', (8, 52)), ('Anchorage', 'Sat', (23, 52)),
('Ankara', 'Sun', (10, 52)), ('Athens', 'Sun', (9, 52)), ('Atlant
a', 'Sun', (2, 52)), ('Auckland', 'Sun', (20, 52)), ('Barcelona',
'Sun', (8, 52)), ('Beirut', 'Sun', (9, 52)), ('Berlin', 'Sun',
(8, 52)), ('Boston', 'Sun', (2, 52)), ('Brasilia', 'Sun', (5, 5
2)), ('Brussels', 'Sun', (8, 52)), ('Bucharest', 'Sun', (9, 52)),
('Budapest', 'Sun', (8, 52)), ('Cairo', 'Sun', (9, 52)), ('Calgar
y', 'Sun', (1, 52)), ('Cape Town', 'Sun', (9, 52)), ('Casablanc
a', 'Sun', (7, 52)), ('Chicago', 'Sun', (1, 52)), ('Columbus', 'Su
n', (2, 52)), ('Copenhagen', 'Sun', (8, 52)), ('Dallas', 'Sun',
(1, 52)), ('Denver', 'Sun', (1, 52)), ('Detroit', 'Sun', (2, 5
2)), ('Dubai', 'Sun', (11, 52)), ('Dublin', 'Sun', (7, 52)), ('Edm
onton', 'Sun', (1, 52)), ('Frankfurt', 'Sun', (8, 52)), ('Halifa
x', 'Sun', (3, 52)), ('Helsinki', 'Sun', (9, 52))]
```

Turning our data into a structured array:

```
time_type = np.dtype([('city', 'U30'), ('day', 'U3'), ('time',
[('h', int), ('min', int)])])

times = np.array( cities_and_times , dtype=time_type)
print(times['time'])
print(times['city'])
x = times[27]
x[0]
```

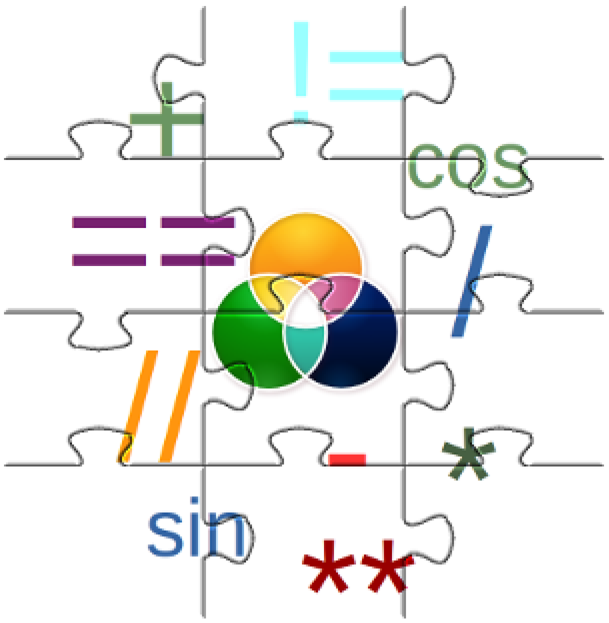
```
[( 8, 52) (23, 52) (10, 52) ( 9, 52) ( 2, 52) (20, 52) ( 8, 52) (
9, 52)
 ( 8, 52) ( 2, 52) ( 5, 52) ( 8, 52) ( 9, 52) ( 8, 52) ( 9, 52) (
1, 52)
 ( 9, 52) ( 7, 52) ( 1, 52) ( 2, 52) ( 8, 52) ( 1, 52) ( 1, 52) (
2, 52)
 (11, 52) ( 7, 52) ( 1, 52) ( 8, 52) ( 3, 52) ( 9, 52) ( 1, 52) (
2, 52)
 (10, 52) ( 9, 52) ( 9, 52) (13, 37) (10, 52) ( 0, 52) ( 7, 52) (
7, 52)
 ( 0, 52) ( 8, 52) (18, 52) ( 2, 52) ( 1, 52) ( 2, 52) (10, 52) (
1, 52)
 ( 2, 52) ( 8, 52) ( 2, 52) ( 8, 52) ( 2, 52) ( 0, 52) ( 8, 52) (
7, 52)
 (10, 52) ( 8, 52) ( 1, 52) ( 0, 52) ( 1, 52) ( 4, 52) ( 0, 52) (1
5, 52)
 (15, 52) ( 8, 52) (18, 52) ( 5, 52) (16, 52) ( 2, 52) ( 0, 52) (
8, 52)
 ( 8, 52) ( 2, 52) ( 1, 52) ( 8, 52)]
['Amsterdam' 'Anchorage' 'Ankara' 'Athens' 'Atlanta' 'Auckland' 'B
arcelona'
 'Beirut' 'Berlin' 'Boston' 'Brasilia' 'Brussels' 'Bucharest' 'Bud
apest'
 'Cairo' 'Calgary' 'Cape Town' 'Casablanca' 'Chicago' 'Columbus'
 'Copenhagen' 'Dallas' 'Denver' 'Detroit' 'Dubai' 'Dublin' 'Edmont
on'
 'Frankfurt' 'Halifax' 'Helsinki' 'Houston' 'Indianapolis' 'Istanb
ul'
 'Jerusalem' 'Johannesburg' 'Kathmandu' 'Kuwait City' 'Las Vegas'
'Lisbon'
 'London' 'Los Angeles' 'Madrid' 'Melbourne' 'Miami' 'Minneapolis'
 'Montreal' 'Moscow' 'New Orleans' 'New York' 'Oslo' 'Ottawa' 'Par
is'
 'Philadelphia' 'Phoenix' 'Prague' 'Reykjavik' 'Riyadh' 'Rome'
 'Salt Lake City' 'San Francisco' 'San Salvador' 'Santiago' 'Seatt
le'
 'Shanghai' 'Singapore' 'Stockholm' 'Sydney' 'São Paulo' 'Tokyo'
'Toronto'
 'Vancouver' 'Vienna' 'Warsaw' 'Washington DC' 'Winnipeg' 'Zuric
h']
```

Output:
```
'Frankfurt'
```

In [ ]:

```

```

NUMERICAL OPERATIONS ON NUMPY ARRAYS



We have seen lots of operators in our Python tutorial. Of course, we have also seen many cases of operator overloading, e.g. "+" for the addition of numerical values and the concatenation of strings.

```
42 + 5
```

```
"Python is one of the best " + "or
maybe the best programming language!"
```

We will learn in this introduction that the operator signs are overloaded in Numpy as well, so that they can be used in a "natural" way.

We can, for example, add a scalar to an ndarrays, i.e. the scalar will be added to every component. The same is possible for subtraction, division, multiplication and even for applying functions, like sine, cosine and so on, to an array.

It is also extremely easy to use all these operators on two arrays as well.

USING SCALARS

Let's start with adding scalars to arrays:

```
import numpy as np
lst = [2,3, 7.9, 3.3, 6.9, 0.11, 10.3, 12.9]
v = np.array(lst)
v = v + 2
```

```
print(v)
```

```
[  4.     5.     9.9    5.3    8.9    2.11  12.3   14.9 ]
```

Multiplication, Subtraction, Division and exponentiation are as easy as the previous addition:

```
print(v * 2.2)
```

```
[  4.4     6.6    17.38    7.26   15.18    0.242  22.66   28.38 ]
```

```
print(v - 1.38)
```

```
[  0.62   1.62   6.52   1.92   5.52  -1.27   8.92  11.52]
```

```
print(v ** 2)
print(v ** 1.5)
```

```
[  4.00000000e+00   9.00000000e+00   6.24100000e+01   1.08900000
e+01
   4.76100000e+01   1.21000000e-02   1.06090000e+02   1.66410000
e+02]
[  2.82842712e+00   5.19615242e+00   2.22044815e+01   5.99474770
e+00
   1.81248172e+01   3.64828727e-02   3.30564215e+01   4.63323753
e+01]
```

We started this example with a list lst, which we turned into the array v. Do you know how to perform the above operations on a list, i.e. multiply, add, subtract and exponentiate every element of the list with a scalar? We could use a for loop for this purpose. Let us do it for the addition without loss of generality. We will add the value 2 to every element of the list:

```
lst = [2,3, 7.9, 3.3, 6.9, 0.11, 10.3, 12.9]
res = []
for val in lst:
    res.append(val + 2)

print(res)
```

```
[4, 5, 9.9, 5.3, 8.9, 2.11, 12.3, 14.9]
```

Even though this solution works it is not the Pythonic way to do it. We will rather use a list comprehension for this purpose than the clumsy solution above. If you are not familar with this approach, you may consult our chapter on list comprehension in our Python course.

```
res = [ val + 2 for val in lst]
print(res)
```

```
[4, 5, 9.9, 5.3, 8.9, 2.11, 12.3, 14.9]
```

Even though we had already measured the time consumed by Numpy compared to "plane" Python, we will compare these two approaches as well:

```
v = np.random.randint(0, 100, 1000)

%timeit v + 1
```

```
1000000 loops, best of 3: 1.69 µs per loop
```

```
lst = list(v)

%timeit [ val + 2 for val in lst]
```

```
1000 loops, best of 3: 452 µs per loop
```

ARITHMETIC OPERATIONS WITH TWO ARRAYS

If we use another array instead of a scalar, the elements of both arrays will be component-wise combined:

```
import numpy as np

A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])
B = np.ones((3,3))

print("Adding to arrays: ")
print(A + B)

print("\nMultiplying two arrays: ")
print(A * (B + 1))
```

```
Adding to arrays:
[[ 12.  13.  14.]
 [ 22.  23.  24.]
 [ 32.  33.  34.]]

Multiplying two arrays:
[[ 22.  24.  26.]
 [ 42.  44.  46.]
 [ 62.  64.  66.]]
```

"A * B" in the previous example shouldn't be mistaken for matrix multiplication. The elements are solely component-wise multiplied.

MATRIX MULTIPLICATION:

For this purpose, we can use the dot product. Using the previous arrays, we can calculate the matrix multiplication:

```
np.dot(A, B)
```

Output:

```
array([[ 36.,  36.,  36.],
       [ 66.,  66.,  66.],
       [ 96.,  96.,  96.]])
```

DEFINITION OF THE DOT PRODUCT

The dot product is defined like this:

dot(a, b, out=None)

For 2-D arrays the dot product is equivalent to matrix multiplication. For 1-D arrays it is the same as the inner product of vectors (without complex conjugation). For N dimensions it is a sum product over the last axis of 'a' and the second-to-last of 'b'::

Parameter	Meaning
a	array or array like argument
b	array or array like argument
out	'out' is an optional parameter, which must have the exact kind of what would be returned, if it was not used. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

The function dot returns the dot product of 'a' and 'b'. If 'a' and 'b' are both scalars or both 1-D arrays then a scalar is returned; otherwise an array will returned.

It will raise a ValueError, if the shape of the last dimension of 'a' is not the same size as the shape of the second-to-last dimension of 'b', i.e. a.shape[-1] == b.shape[-2]

EXAMPLES OF USING THE DOT PRODUCT

We will begin with the cases in which both arguments are scalars or one-dimensional array:

```
print(np.dot(3, 4))
x = np.array([3])
y = np.array([4])
print(x.ndim)
print(np.dot(x, y))

x = np.array([3, -2])
y = np.array([-4, 1])
print(np.dot(x, y))
```

```
12
1
12
-14
```

Let's go to the two-dimensional use case:

```
A = np.array([ [1, 2, 3],
               [3, 2, 1] ])
B = np.array([ [2, 3, 4, -2],
               [1, -1, 2, 3],
               [1, 2, 3, 0] ])

# es muss gelten:
print(A.shape[-1] == B.shape[-2], A.shape[1])
print(np.dot(A, B))
```

```
(True, 3)
[[ 7  7 17  4]
 [ 9  9 19  0]]
```

We can learn from the previous example that the number of columns of the first two-dimension array have to be the same as the number of the lines of the second two-dimensional array.

THE DOT PRODUCT IN THE 3-DIMENSIONAL CASE

It's getting really vexing, if we use 3-dimensional arrays as the arguments of dot.

We will use two symmetrical three-dimensional arrays in the first example:

```
import numpy as np
X = np.array( [[[3, 1, 2],
                [4, 2, 2],
                [2, 4, 1]],

               [[3, 2, 2],
                [4, 4, 3],
                [4, 1, 1]],

               [[2, 2, 1],
                [3, 1, 3],
                [3, 2, 3]]])

Y = np.array( [[[2, 3, 1],
                [2, 2, 4],
                [3, 4, 4]],

               [[1, 4, 1],
                [4, 1, 2],
                [4, 1, 2]],

               [[1, 2, 3],
                [4, 1, 1],
                [3, 1, 4]]])


R = np.dot(X, Y)

print("The shapes:")
print(X.shape)
print(Y.shape)
print(R.shape)

print("\nThe Result R:")
print(R)
```

```
The shapes:
(3, 3, 3)
(3, 3, 3)
(3, 3, 3, 3)

The Result R:
[[[[14 19 15]
   [15 15  9]
   [13  9 18]]

  [[18 24 20]
   [20 20 12]
   [18 12 22]]

  [[15 18 22]
   [22 13 12]
   [21  9 14]]]


 [[[16 21 19]
   [19 16 11]
   [17 10 19]]

  [[25 32 32]
   [32 23 18]
   [29 15 28]]

  [[13 18 12]
   [12 18  8]
   [11 10 17]]]


 [[[11 14 14]
   [14 11  8]
   [13  7 12]]

  [[17 23 19]
   [19 16 11]
   [16 10 22]]

  [[19 25 23]
   [23 17 13]
   [20 11 23]]]]
```

To demonstrate how the dot product in the three-dimensional case works, we will use different arrays with non-symmetrical shapes in the following example:

```
import numpy as np
X = np.array(
    [[[3, 1, 2],
      [4, 2, 2]],

     [[-1, 0, 1],
      [1, -1, -2]],

     [[3, 2, 2],
      [4, 4, 3]],

     [[2, 2, 1],
      [3, 1, 3]]])

Y = np.array(
    [[[2, 3, 1, 2, 1],
      [2, 2, 2, 0, 0],
      [3, 4, 0, 1, -1]],

     [[1, 4, 3, 2, 2],
      [4, 1, 1, 4, -3],
      [4, 1, 0, 3, 0]]])


R = np.dot(X, Y)



print("X.shape: ", X.shape, "   X.ndim: ", X.ndim)
print("Y.shape: ", Y.shape, "   Y.ndim: ", Y.ndim)
print("R.shape: ",     R.shape, "R.ndim: ", R.ndim)


print("\nThe result array R:\n")
print(R)
```

```
('X.shape: ', (4, 2, 3), '   X.ndim: ', 3)
('Y.shape: ', (2, 3, 5), '   Y.ndim: ', 3)
('R.shape: ', (4, 2, 2, 5), 'R.ndim: ', 4)

The result array R:

[[[[ 14  19   5   8   1]
   [ 15  15  10  16   3]]

  [[ 18  24   8  10   2]
   [ 20  20  14  22   2]]]


 [[[  1   1  -1  -1  -2]
   [  3  -3  -3   1  -2]]

  [[ -6  -7  -1   0   3]
   [-11   1   2  -8   5]]]


 [[[ 16  21   7   8   1]
   [ 19  16  11  20   0]]

  [[ 25  32  12  11   1]
   [ 32  23  16  33  -4]]]


 [[[ 11  14   6   5   1]
   [ 14  11   8  15  -2]]

  [[ 17  23   5   9   0]
   [ 19  16  10  19   3]]]]
```

Let's have a look at the following sum products:

```
i = 0
for j in range(X.shape[1]):
    for k in range(Y.shape[0]):
        for m in range(Y.shape[2]):
            fmt = "    sum(X[{}, {}, :] * Y[{}, :, {}] :  {}"
            arguments = (i, j, k, m, sum(X[i, j, :] * Y[k, :, m]))
            print(fmt.format(*arguments))
    sum(X[0, 0, :] * Y[0, :, 0] :  14
    sum(X[0, 0, :] * Y[0, :, 1] :  19
    sum(X[0, 0, :] * Y[0, :, 2] :  5
    sum(X[0, 0, :] * Y[0, :, 3] :  8
    sum(X[0, 0, :] * Y[0, :, 4] :  1
    sum(X[0, 0, :] * Y[1, :, 0] :  15
    sum(X[0, 0, :] * Y[1, :, 1] :  15
    sum(X[0, 0, :] * Y[1, :, 2] :  10
    sum(X[0, 0, :] * Y[1, :, 3] :  16
    sum(X[0, 0, :] * Y[1, :, 4] :  3
    sum(X[0, 1, :] * Y[0, :, 0] :  18
    sum(X[0, 1, :] * Y[0, :, 1] :  24
    sum(X[0, 1, :] * Y[0, :, 2] :  8
    sum(X[0, 1, :] * Y[0, :, 3] :  10
    sum(X[0, 1, :] * Y[0, :, 4] :  2
    sum(X[0, 1, :] * Y[1, :, 0] :  20
    sum(X[0, 1, :] * Y[1, :, 1] :  20
    sum(X[0, 1, :] * Y[1, :, 2] :  14
    sum(X[0, 1, :] * Y[1, :, 3] :  22
    sum(X[0, 1, :] * Y[1, :, 4] :  2
```

Hopefully, you have noticed that we have created the elements of R[0], one ofter the other.

```
print(R[0])
```

```
[[[14 19  5  8  1]
  [15 15 10 16  3]]

 [[18 24  8 10  2]
  [20 20 14 22  2]]]
```

This means that we could have created the array R by applying the sum products in the way above. To "prove" this, we will create an array R2 by using the sum product, which is equal to R in the following example:

```
R2 = np.zeros(R.shape, dtype=np.int)

for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        for k in range(Y.shape[0]):
            for m in range(Y.shape[2]):
                R2[i, j, k, m] = sum(X[i, j, :] * Y[k, :, m])


print( np.array_equal(R, R2) )
```

```
True
```

MATRICES VS. TWO-DIMENSIONAL ARRAYS

Some may have taken two-dimensional arrays of Numpy as matrices. This is principially all right, because they behave in most aspects like our mathematical idea of a matrix. We even saw that we can perform matrix multiplication on them. Yet, there is a subtle difference. There are "real" matrices in Numpy. They are a subset of the two-dimensional arrays. We can turn a two-dimensional array into a matrix by applying the "mat" function. The main difference shows, if you multiply two two-dimensional arrays or two matrices. We get real matrix multiplication by multiplying two matrices, but the two-dimensional arrays will be only multiplied component-wise:

```
import numpy as np

A = np.array([ [1, 2, 3], [2, 2, 2], [3, 3, 3] ])
B = np.array([ [3, 2, 1], [1, 2, 3], [-1, -2, -3] ])

R = A * B
print(R)
```

```
[[ 3  4  3]
 [ 2  4  6]
 [-3 -6 -9]]
```

```
MA = np.mat(A)
MB = np.mat(B)

R = MA * MB
print(R)
```

```
[[ 2  0 -2]
 [ 6  4  2]
 [ 9  6  3]]
```

COMPARISON OPERATORS

We are used to comparison operators from Python, when we apply them to integers, floats or strings for example. We use them to test for True or False. If we compare two arrays, we don't get a simple True or False as a return value. The comparisons are performed elementswise. This means that we get a Boolean array as a return value:

```
import numpy as np

A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])
B = np.array([ [11, 102, 13], [201, 22, 203], [31, 32, 303] ])

A == B
```

Output:
```
array([[ True, False,  True],
       [False,  True, False],
       [ True,  True, False]], dtype=bool)
```

It is possible to compare complete arrays for equality as well. We use array_equal for this purpose. array_equal returns True if two arrays have the same shape and elements, otherwise False will be returned.

```
print(np.array_equal(A, B))
print(np.array_equal(A, A))
```

```
False
True
```

LOGICAL OPERATORS

We can also apply the logical 'or' and the logical 'and' to arrays elementwise. We can use the functions 'logical_or' and 'logical_and' to this purpose.

```
a = np.array([ [True, True], [False, False]])
b = np.array([ [True, False], [True, False]])

print(np.logical_or(a, b))
print(np.logical_and(a, b))
```

```
[[ True  True]
 [ True False]]
[[ True False]
 [False False]]
```

APPLYING OPERATORS ON ARRAYS WITH DIFFERENT SHAPES

So far we have covered two different cases with basic operators like "+" or "*":

- an operator applied to an array and a scalar
- an operator applied to two arrays of the same shape

We will see in the following that we can also apply operators on arrays, if they have different shapes. Yet, it works only under certain conditions.

BROADCASTING

Numpy provides a powerful mechanism, called Broadcasting, which allows to perform arithmetic operations on arrays of different shapes. This means that we have a smaller array and a larger array, and we transform or apply the smaller array multiple times to perform some operation on the larger array. In other words: Under certain conditions, the smaller array is "broadcasted" in a way that it has the same shape as the larger array.

With the aid of broadcasting we can avoid loops in our Python program. The looping occurs implicitly in the Numpy implementations, i.e. in C. We also avoid creating unnecessary copies of our data.

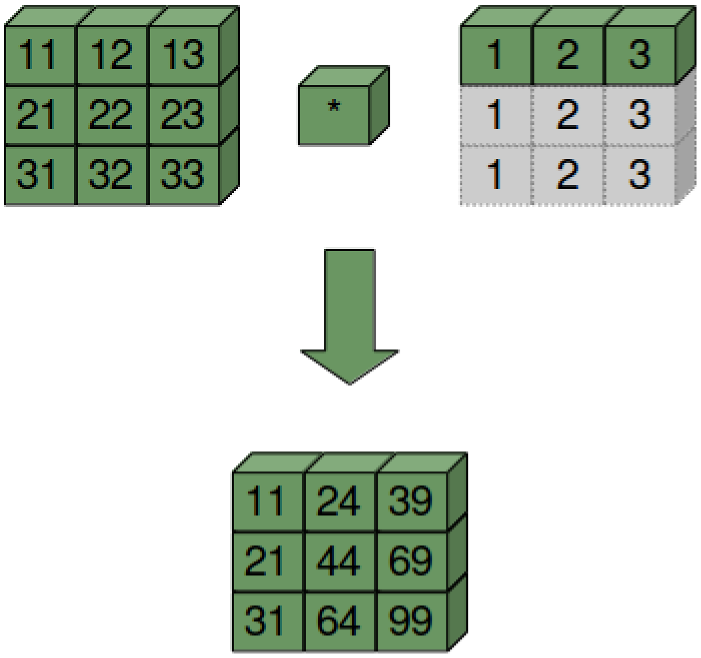
We demonstrate the operating principle of broadcasting in three simple and descriptive examples.

FIRST EXAMPLE OF BROADCASTING:

```
import numpy as np

A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])
B = np.array([1, 2, 3])

print("Multiplication with broadcasting: ")
print(A * B)
print("... and now addition with broadcasting: ")
print(A + B)
```

```
Multiplication with broadcasting:
[[11 24 39]
 [21 44 69]
 [31 64 99]]
... and now addition with broadcasting:
[[12 14 16]
 [22 24 26]
 [32 34 36]]
```

The following diagram illustrates the way of working of broadcasting:



B is treated as if it were construed like this:

```
B = np.array([[1, 2, 3],] * 3)
print(B)
```

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

SECOND EXAMPLE:

For this example, we need to know how to turn a row vector into a column vector:

```
B = np.array([1, 2, 3])
B[:, np.newaxis]
```

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

Now we are capable of doing the multipliplication using broadcasting:

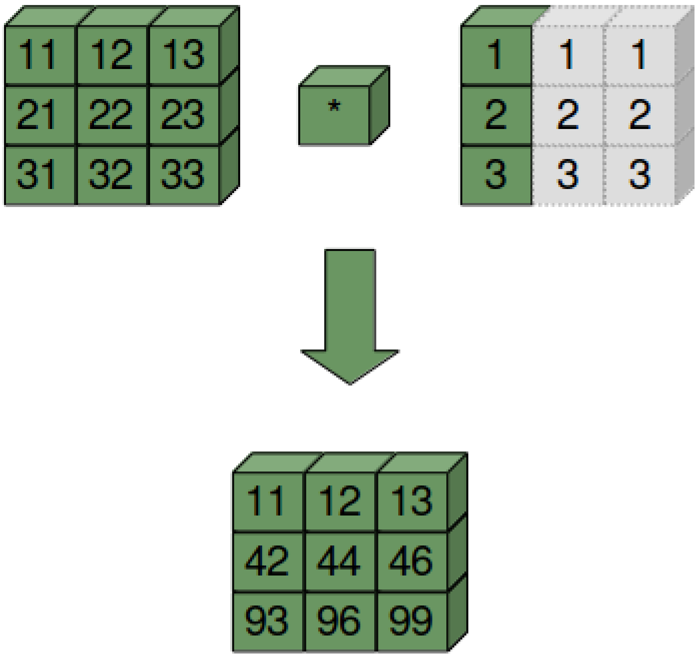
```
A * B[:, np.newaxis]
```

Output:
```
array([[11, 12, 13],
       [42, 44, 46],
       [93, 96, 99]])
```

B is treated as if it were construed like this:

```
np.array([[1, 2, 3],] * 3).transpose()
```

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



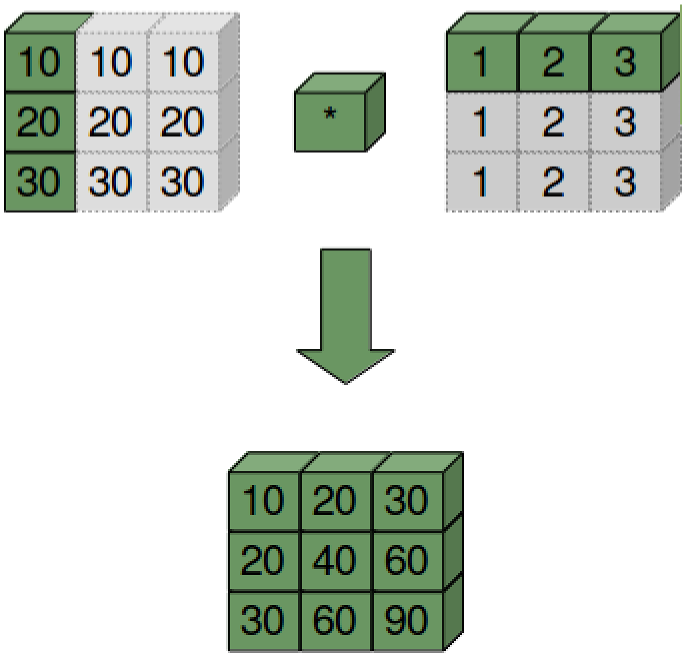
THIRD EXAMPLE:

```
A = np.array([10, 20, 30])
B = np.array([1, 2, 3])
A[:, np.newaxis]
```

Output:
```
array([[10],
       [20],
       [30]])
```

```
A[:, np.newaxis] * B
```

Output:
```
array([[10, 20, 30],
       [20, 40, 60],
       [30, 60, 90]])
```



ANOTHER WAY TO DO IT

Doing it without broadcasting:

```
import numpy as np

A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])

B = np.array([1, 2, 3])

B = B[np.newaxis, :]
B = np.concatenate((B, B, B))

print("Multiplication: ")
print(A * B)
print("... and now addition again: ")
print(A + B)
```

```
Multiplication:
[[11 24 39]
 [21 44 69]
 [31 64 99]]
... and now addition again:
[[12 14 16]
 [22 24 26]
 [32 34 36]]
```

Using 'tile':

```
import numpy as np

A = np.array([ [11, 12, 13], [21, 22, 23], [31, 32, 33] ])

B = np.tile(np.array([1, 2, 3]), (3, 1))

print(B)

print("Multiplication: ")
print(A * B)
print("... and now addition again: ")
print(A + B)
```

```
[[1 2 3]
 [1 2 3]
 [1 2 3]]
Multiplication:
[[11 24 39]
 [21 44 69]
 [31 64 99]]
... and now addition again:
[[12 14 16]
 [22 24 26]
 [32 34 36]]
```

DISTANCE MATRIX

In mathematics, computer science and especially graph theory, a distance matrix is a matrix or a two-dimensional array, which contains the distances between the elements of a set, pairwise taken. The size of this two-dimensional array in n x n, if the set consists of n elements.

A practical example of a distance matrix is a distance matrix between geographic locations, in our example Eurpean cities:

```
cities = ["Barcelona", "Berlin", "Brussels", "Bucharest",
          "Budapest", "Copenhagen", "Dublin", "Hamburg", "Istanbu
l",
          "Kiev", "London", "Madrid", "Milan", "Moscow", "Munich",
          "Paris", "Prague", "Rome", "Saint Petersburg",
          "Stockholm", "Vienna", "Warsaw"]

dist2barcelona = [0,  1498, 1063, 1968,
                  1498, 1758, 1469, 1472, 2230,
                  2391, 1138, 505, 725, 3007, 1055,
                  833, 1354, 857, 2813,
                  2277, 1347, 1862]

dists =  np.array(dist2barcelona[:12])
print(dists)
print(np.abs(dists - dists[:, np.newaxis]))
```

```
[   0 1498 1063 1968 1498 1758 1469 1472 2230 2391 1138  505]
[[   0 1498 1063 1968 1498 1758 1469 1472 2230 2391 1138  505]
 [1498    0  435  470    0  260   29   26  732  893  360  993]
 [1063  435    0  905  435  695  406  409 1167 1328   75  558]
 [1968  470  905    0  470  210  499  496  262  423  830 1463]
 [1498    0  435  470    0  260   29   26  732  893  360  993]
 [1758  260  695  210  260    0  289  286  472  633  620 1253]
 [1469   29  406  499   29  289    0    3  761  922  331  964]
 [1472   26  409  496   26  286    3    0  758  919  334  967]
 [2230  732 1167  262  732  472  761  758    0  161 1092 1725]
 [2391  893 1328  423  893  633  922  919  161    0 1253 1886]
 [1138  360   75  830  360  620  331  334 1092 1253    0  633]
 [ 505  993  558 1463  993 1253  964  967 1725 1886  633    0]]
```

3-DIMENSIONAL BROADCASTING

```
A = np.array([ [[3, 4, 7], [5, 0, -1] , [2, 1, 5]],
      [[1, 0, -1], [8, 2, 4], [5, 2, 1]],
      [[2, 1, 3], [1, 9, 4], [5, -2, 4]]])

B = np.array([ [[3, 4, 7], [1, 0, -1], [1, 2, 3]] ])
```

```
B * A
```

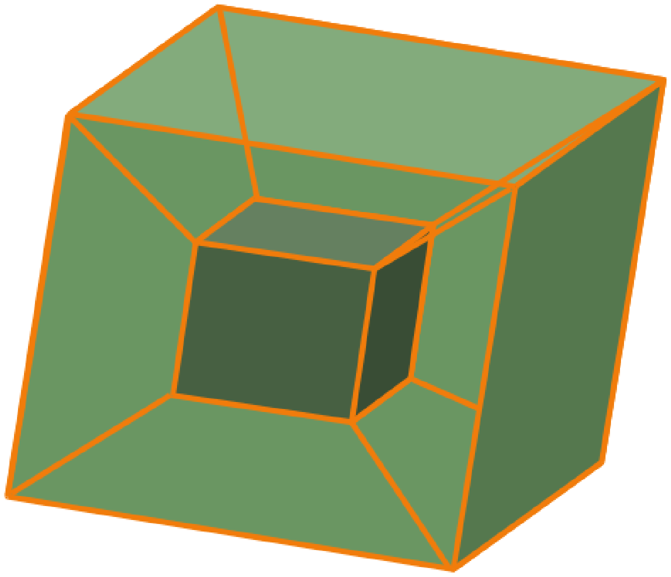
Output:
```
array([[[ 9, 16, 49],
        [ 5,  0,  1],
        [ 2,  2, 15]],

       [[ 3,  0, -7],
        [ 8,  0, -4],
        [ 5,  4,  3]],

       [[ 6,  4, 21],
        [ 1,  0, -4],
        [ 5, -4, 12]]])
```

We will use the following transformations in our chapter on Images Manipulation and Processing:

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

B = B[np.newaxis, :]
print(B.shape)
B = np.concatenate((B, B, B)).transpose()
print(B.shape)
B = B[:, np.newaxis]
print(B.shape)
print(B)

print(A * B)
```

```
(1, 3)
(3, 3)
(3, 1, 3)
[[[1 1 1]]

 [[2 2 2]]

 [[3 3 3]]]
[[[ 3  4  7]
  [ 5  0 -1]
  [ 2  1  5]]

 [[ 2  0 -2]
  [16  4  8]
  [10  4  2]]

 [[ 6  3  9]
  [ 3 27 12]
  [15 -6 12]]]
```

NUMPY ARRAYS: CONCATENATING, FLATTENING AND ADDING DIMENSIONS



So far, we have learned in our tutorial how to create arrays and how to apply numerical operations on numpy arrays. If we program with numpy, we will come sooner or later to the point, where we will need functions to manipulate the shape or dimension of arrays. We wil also learn how to concatenate arrays. Furthermore, we will demonstrate the possibilities to add dimensions to existing arrays and how to stack multiple arrays. We will end this chapter by showing an easy way to construct new arrays by repeating existing arrays.

The picture shows a tesseract. A tesseract is a hypercube in $\Re^4$. The tesseract is to the cube as the cube is to the square: the surface of the cube consists of six square sides, whereas the hypersurface of the tesseract consists of eight cubical cells.

FLATTEN AND RESHAPE ARRAYS

There are two methods to flatten a multidimensional array:

- flatten()
- ravel()

FLATTEN

flatten is a ndarry method with an optional keyword parameter "order". order can have the values "C", "F" and "A". The default of order is "C". "C" means to flatten C style in row-major ordering, i.e. the rightmost index "changes the fastest" or in other words: In row-major order, the row index varies the slowest, and the column index the quickest, so that a[0,1] follows [0,0].
"F" stands for Fortran column-major ordering. "A" means preserve the the C/Fortran ordering.

```
import numpy as np

A = np.array([[[ 0,  1],
               [ 2,  3],
               [ 4,  5],
               [ 6,  7]],
              [[ 8,  9],
               [10, 11],
               [12, 13],
               [14, 15]],
              [[16, 17],
               [18, 19],
               [20, 21],
               [22, 23]]])

Flattened_X = A.flatten()
print(Flattened_X)

print(A.flatten(order="C"))
print(A.flatten(order="F"))
print(A.flatten(order="A"))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 2
1 22 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 2
1 22 23]
[ 0  8 16  2 10 18  4 12 20  6 14 22  1  9 17  3 11 19  5 13 21  
7 15 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 2
1 22 23]
```

RAVEL

The order of the elements in the array returned by ravel() is normally "C-style".

```
ravel(a, order='C')
```

ravel returns a flattened one-dimensional array. A copy is made only if needed.

The optional keyword parameter "order" can be 'C','F', 'A', or 'K'

'C': C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. "C" is the default!

'F': Fortran-like index order with the first index changing fastest, and the last index changing slowest.

'A': Fortran-like index order if the array "a" is Fortran *contiguous* in memory, C-like order otherwise.

'K': read the elements in the order they occur in memory, except for reversing the data when strides are negative.

```
print(A.ravel())

print(A.ravel(order="A"))

print(A.ravel(order="F"))

print(A.ravel(order="A"))

print(A.ravel(order="K"))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 2
1 22 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 2
1 22 23]
[ 0  8 16  2 10 18  4 12 20  6 14 22  1  9 17  3 11 19  5 13 21  
7 15 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 2
1 22 23]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 2
1 22 23]
```

RESHAPE

The method reshape() gives a new shape to an array without changing its data, i.e. it returns a new array with a new shape.

```
reshape(a, newshape, order='C')
```

Parameter	Meaning
a	array_like, Array to be reshaped.
newshape	int or tuple of ints
order	'C', 'F', 'A', like in flatten or ravel

```
X = np.array(range(24))
Y = X.reshape((3,4,2))
Y
```

Output:
```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        [ 6,  7]],

       [[ 8,  9],
        [10, 11],
        [12, 13],
        [14, 15]],

       [[16, 17],
        [18, 19],
        [20, 21],
        [22, 23]]])
```

CONCATENATING ARRAYS

In the following example we concatenate three one-dimensional arrays to one array. The elements of the second array are appended to the first array. After this the elements of the third array are appended:

```
x = np.array([11,22])
y = np.array([18,7,6])
z = np.array([1,3,5])
c = np.concatenate((x,y,z))
print(c)
```

```
[11 22 18  7  6  1  3  5]
```

If we are concatenating multidimensional arrays, we can concatenate the arrays according to axis. Arrays must have the same shape to be concatenated with concatenate(). In the case of multidimensional arrays, we can arrange them according to the axis. The default value is axis = 0:

```
x = np.array(range(24))
x = x.reshape((3,4,2))
y = np.array(range(100,124))
y = y.reshape((3,4,2))
z = np.concatenate((x,y))
print(z)
```

```
[[[  0   1]
  [  2   3]
  [  4   5]
  [  6   7]]

 [[  8   9]
  [ 10  11]
  [ 12  13]
  [ 14  15]]

 [[ 16  17]
  [ 18  19]
  [ 20  21]
  [ 22  23]]

 [[100 101]
  [102 103]
  [104 105]
  [106 107]]

 [[108 109]
  [110 111]
  [112 113]
  [114 115]]

 [[116 117]
  [118 119]
  [120 121]
  [122 123]]]
```

We do the same concatenation now with axis=1:

```
z = np.concatenate((x,y),axis = 1)
print(z)
```

```
[[[  0   1]
  [  2   3]
  [  4   5]
  [  6   7]
  [100 101]
  [102 103]
  [104 105]
  [106 107]]

 [[  8   9]
  [ 10  11]
  [ 12  13]
  [ 14  15]
  [108 109]
  [110 111]
  [112 113]
  [114 115]]

 [[ 16  17]
  [ 18  19]
  [ 20  21]
  [ 22  23]
  [116 117]
  [118 119]
  [120 121]
  [122 123]]]
```

ADDING NEW DIMENSIONS

New dimensions can be added to an array by using slicing and np.newaxis. We illustrate this technique with an example:

```
x = np.array([2,5,18,14,4])
y = x[:, np.newaxis]
print(y)
```

```
[[ 2]
 [ 5]
 [18]
 [14]
 [ 4]]
```

VECTOR STACKING

```
A = np.array([3, 4, 5])
B = np.array([1,9,0])

print(np.row_stack((A, B)))

print(np.column_stack((A, B)))
np.shape(A)
```

```
[[3 4 5]
 [1 9 0]]
[[3 1]
 [4 9]
 [5 0]]
```

Output:
```
(3,)
```

```
A = np.array([[3, 4, 5],
              [1, 9, 0],
              [4, 6, 8]])
np.column_stack((A, A, A))
```

Output:
```
array([[3, 4, 5, 3, 4, 5, 3, 4, 5],
       [1, 9, 0, 1, 9, 0, 1, 9, 0],
       [4, 6, 8, 4, 6, 8, 4, 6, 8]])
```

```
np.column_stack((A[0], A[0], A[0]))
```

Output:
```
array([[3, 3, 3],
       [4, 4, 4],
       [5, 5, 5]])
```

```
np.dstack((A, A, A))
```

Output:
```
array([[[3, 3, 3],
        [4, 4, 4],
        [5, 5, 5]],

       [[1, 1, 1],
        [9, 9, 9],
        [0, 0, 0]],

       [[4, 4, 4],
        [6, 6, 6],
        [8, 8, 8]]])
```

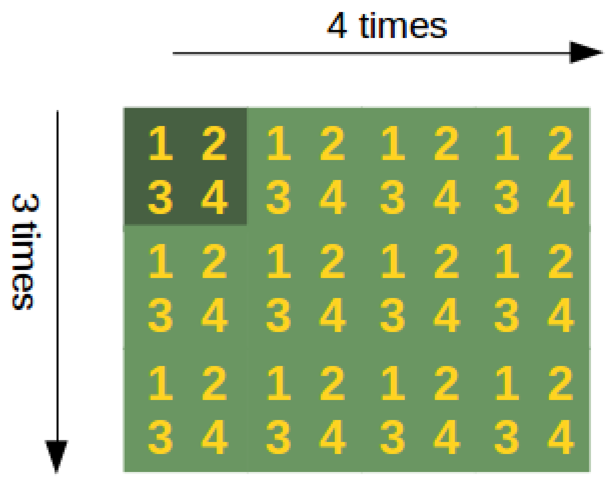
REPEATING PATTERNS, THE "TILE" METHOD

Sometimes, you want to or have to create a new matrix by repeating an existing matrix multiple times to create a new matrix with a different shape or even dimension. You may have for example a one-dimensional array `array([ 3.4])` and you want to turn it into an array `array([ 3.4,  3.4,  3.4,  3.4, 3.4])`

In another usecase you may have a two-dimensional array like `np.array([ [1, 2], [3, 4]])`, which you intend to use as a building block to construe the array with the shape (6, 8):

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

The idea of construction is depicted in the following diagram:



If this reminds you of tiling a bathroom or a kitchen, you are on the right track: The function which Numpy provides for this task is called "tile".

The formal syntax of tile looks like this:

```
tile(A, reps)
```

An array is constructed by repeating A the number of times given by reps.

'reps' is usually a tuple (or list) which defines the number of repetitions along the corresponding axis / directions. if we set reps to (3, 4) for example, A will be repeated 3 times for the "rows" and 4 times in the direction of the columna. We demonstrate this in the following example:

```
import numpy as np
x = np.array([ [1, 2], [3, 4]])
np.tile(x, (3,4))
```

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

```
import numpy as np

x = np.array([ 3.4])

y = np.tile(x, (5,))

print(y)
```

```
[ 3.4  3.4  3.4  3.4  3.4]
```

In the previous tile example, we could have written `y = np.tile(x, 5)` as well.

If we stick to writing reps in the tuple or list form, or consider `reps = 5` as an abbreviation for `reps = (5,)`, the following is true:

If 'reps' has length n, the dimension of the resulting array will be the maximum of n and A.ndim.

If 'A.ndim < n, 'A' is promoted to be n-dimensional by prepending new axes. So a shape (5,) array is promoted to (1, 5) for 2-D replication, or shape (1, 1, 5) for 3-D replication. If this is not the desired behavior, promote 'A' to n-dimensions manually before calling this function.

If 'A.ndim > d', 'reps' is promoted to 'A'.ndim by pre-pending 1's to it.

Thus for an array 'A' of shape (2, 3, 4, 5), a 'reps' of (2, 2) is treated as (1, 1, 2, 2).

Further examples:

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(np.tile(x, 2))
```

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

```
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(np.tile(x, (2, 1)))
```

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

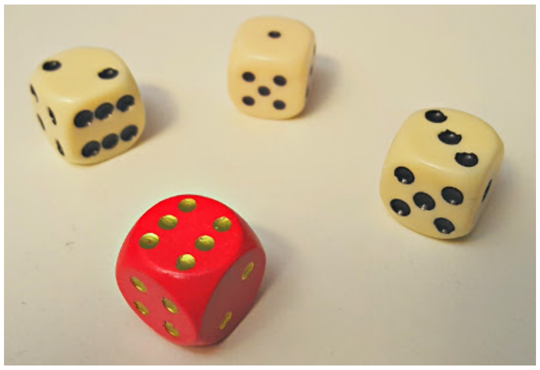
```
import numpy as np
x = np.array([[1, 2], [3, 4]])
print(np.tile(x, (2, 2)))
```

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

PYTHON, RANDOM NUMBERS AND PROBABILITY

INTRODUCTION

"Every American should have above average income, and my Administration is going to see they get it."



This saying is attributed to Bill Clinton on umpteen websites. Usually, there is no context given, so it is not clear, if he might have meant it as a "joke". Whatever his intentions might have been, we quoted him to show a "real" life example of statistics. Statistics and probability calculation is all around us in real-life situations. We have to cope with it whenever we have to make a decision from various options. Can we go for a hike in the afternoon or will it rain? The weather forecast tells us, that the probability of precipitation will be 30 %. So what now? Will we go for a hike?
Another situation: Every week you play the lottery and dream of a far away island. What is the likelihood of winning the Jackpot so that you will never have to work again and live in "paradise"? Now imagine that you right on this island of your dreams. Most probably not because you won the jackpot, but rather because you booked your time as an all-inclusive holiday package. You are on holiday on a paradisal island far from home. Suddenly, you meet your neighbor, spoiling in a jiffy all dreams. Against all odds?
Uncertainty is all araound us, yet only few people understand the basics of probability theory.

The programming language Python and even the numerical modules Numpy and Scipy will not help us in understanding the everyday problems mentioned above, but Python and Numpy provide us with powerful functionalities to calculate problems from statistics and probability theory.

RANDOM NUMBERS WITH PYTHON

THE RANDOM AND THE "SECRETS" MODULES

There is an explicit warning in the documentation of the random module:

Warning:
Note that the pseudo-random generators in the random module should NOT be used for security purposes. Use secrets on Python 3.6+ and os.urandom() on Python 3.5 and earlier.

The default pseudo-random number generator of the random module was designed with the focus on modelling and simulation, not on security. So, you shouldn't generate sensitive information such as passwords, secure tokens, session keys and similar things by using random. When we say that you shouldn't use the random module, we mean the basic functionalities "randint", "random", "choise" , and the likes. There is one exception as you will learn in the next paragraph: SystemRandom

The SystemRandom class offers a suitable way to overcome this security problem. The methods of this class use an alternate random number generator, which uses tools provided by the operating system (such as /dev/urandom on Unix or CryptGenRandom on Windows.

As there has been great concern that Python developers might inadvertently make serious security errors, - even though the warning is included in the documentaiton, - Python 3.6 comes with a new module "secrets" with a CSPRNG (Cryptographically Strong Pseudo Random Number Generator).

Let's start with creating random float numbers with the random function of the random module. Please remember that it shouldn't be used to generate sensitive information:

```
import random
random_number = random.random()
print(random_number)
```

```
0.34330263184538523
```

We will show an alternative and secure approach in the following example, in which we will use the class SystemRandom of the random module. It will use a different random number generator. It uses sources which are provided by the operating system. This will be /dev/urandom on Unix and CryptGenRandom on windows. The random method of the SystemRandom class generates a float number in the range from 0.0 (included) to 1.0 (not included):

```
from random import SystemRandom
crypto = SystemRandom()
print(crypto.random())
```

```
0.8875057137654113
```

GENERATE A LIST OF RANDOM NUMBERS

Quite often you will need more than one random number. We can create a list of random numbers by repeatedly calling random().

```
import random

def random_list(n, secure=True):
    random_floats = []
    if secure:
        crypto = random.SystemRandom()
        random_float = crypto.random
    else:
        random_float = random.random
    for _ in range(n):
        random_floats.append(random_float())
    return random_floats

print(random_list(10, secure=False))
```

```
[0.9702685982962019, 0.5095131905323179, 0.9324278634720967, 0.975
0405405778308, 0.9750927470224396, 0.2383439553695087, 0.035916944
33088444, 0.9203791901577599, 0.07793301506800698, 0.4691524576406
6404]
```

The "simple" random function of the random module is a lot faster as we can see in the following:

```
%%timeit
random_list(100)
```

```
10000 loops, best of 3: 158 µs per loop
```

```
%%timeit
random_list(100, secure=False)
```

```
100000 loops, best of 3: 8.64 µs per loop
```

```
crypto = random.SystemRandom()
[crypto.random() for _ in range(10)]
```

Output:
```
[0.5832874631978111,
 0.7494815897496974,
 0.6982338101218046,
 0.5164288598133177,
 0.15423895558995826,
 0.9447842390510461,
 0.08095707071826808,
 0.5407159221282145,
 0.6124979567571185,
 0.15764744205801628]
```

Alternatively, you can use a list comprehension to create a list of random float numbers:

```
%%timeit
[crypto.random() for _ in range(100)]
```

```
10000 loops, best of 3: 157 µs per loop
```

The fastest and most efficient way will be using the random package of the numpy module:

```
import numpy as np

np.random.random(10)
```

Output:
```
array([ 0.0422172 ,  0.98285327,  0.40386413,  0.34629582,
0.25666744,
        0.69242112,  0.9231164 ,  0.47445382,  0.63654389,
0.06781786])
```

```
%%timeit
np.random.random(100)
```

```
The slowest run took 16.56 times longer than the fastest. This cou
ld mean that an intermediate result is being cached.
100000 loops, best of 3: 2.1 µs per loop
```

Warning:
The random package of the Numpy module apparently - even though it doesn't say so in the documentation - is completely deterministic, using also the Mersenne twister sequence!

RANDOM NUMBERS SATISFYING SUM-TO-ONE CONDITION

It's very easy to create a list of random numbers satisfying the condition that they sum up to one. This way, we turn them into values, which could be used as probalities. We can use any of the methods explained above to normalize a list of random values. All we have to do is divide every value by the sum of the values. The easiest way will be using numpy again of course:

```
import numpy as np

list_of_random_floats = np.random.random(100)
sum_of_values = list_of_random_floats.sum()
print(sum_of_values)
normalized_values = list_of_random_floats / sum_of_values
print(normalized_values.sum())
```

```
52.3509839137
1.0
```

GENERATING RANDOM STRINGS OR PASSWORDS WITH PYTHON

We assume that you don't use and don't like weak passwords like "123456", "password", "qwerty" and the likes. Believe it or not, these passwords are always ranking to 10. So you looking for a safe password? You want to create passwords with Python? But don't use some of the functions ranking top 10 in the search results, because you may use a functions using the random function of the random module.

We will define a strong random password generator, which uses the SystemRandom class. This class uses, as we have alreay mentioned, a cryptographically strong pseudo random number generator:

```
from random import SystemRandom
sr = SystemRandom() # create an instance of the SystemRandom class


def generate_password(length,
                      valid_chars=None):
    """ generate_password(length, check_char) -> password
        length: the length of the created password
        check_char: a Boolean function used to check the validity
of a char
    """
    if valid_chars==None:
        valid_chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        valid_chars += valid_chars.lower() + "0123456789"

    password = ""
    counter = 0
    while counter < length:
        rnum = sr.randint(0, 128)
        char = chr(rnum)
        if char in valid_chars:
            password += chr(rnum)
            counter += 1
    return password

print("Automatically generated password by Python: " + generate_pa
ssword(15))
```

```
Automatically generated password by Python: 1l6Zki280gfUqMD
```

RANDOM INTEGER NUMBERS

Everybody is familar with creating random integer numbers without computers. If you roll a die, you create a random number between 1 and 6. In terms of probability theory, we would call "the rolling of the die" an experiment with a result from the set of possible outcomes {1, 2, 3, 4, 5, 6}. It is also called the sample space of the experiment.

How can we simulate the rolling of a die in Python? We don't need Numpy for this aim. "Pure" Python and its random module is enough.

```
import random

outcome = random.randint(1,6)
print(outcome)
```

```
4
```

Let's roll our virtual die 10 times:

```
import random

[ random.randint(1, 6) for _ in range(10) ]
```

Output:
```
[2, 1, 5, 5, 6, 5, 4, 4, 1, 1]
```

We can accomplish this easier with the NumPy package random:

```
import numpy as np

outcome = np.random.randint(1, 7, size=10)
print(outcome)
```

```
[6 6 6 1 3 6 2 5 3 3]
```

You may have noticed, that we used 7 instead of 6 as the second parameter. randint from numpy.random uses a "half-open" interval unlike randint from the Python random module, which uses a closed interval!

The formal definition:

numpy.random.randint(low, high=None, size=None)

This function returns random integers from 'low' (inclusive) to 'high' (exclusive). In other words: randint returns random integers from the "discrete uniform" distribution in the "half-open" interval ['low', 'high'). If 'high' is None or not given in the call, the results will range from [0, 'low'). The parameter 'size' defines the shape of the output. If 'size' is None, a single int will be the output. Otherwise the result will be an array. The parameter 'size' defines the shape of this array. So size should be a tuple. If size is defined as an integer n, this is considered to be the tuple (n,).

The following examples will clarify the behavior of the parameters:

```
import numpy as np

print(np.random.randint(1, 7))
print(np.random.randint(1, 7, size=1))
print(np.random.randint(1, 7, size=10))
print(np.random.randint(1, 7, size=(10,))) # the same as the previ
ous one
print(np.random.randint(1, 7, size=(5, 4)))
```

```
5
[3]
[1 4 3 5 5 1 5 4 5 6]
[2 1 4 3 2 1 6 5 3 3]
[[4 1 3 1]
 [6 4 5 6]
 [2 5 5 1]
 [4 3 2 3]
 [6 2 6 5]]
```

Simulating the rolling of a die is usually not a security-relevant issue, but if you want to create cryptographically strong pseudo random numbers you should use the SystemRandom class again:

```
import random

crypto = random.SystemRandom()

[ crypto.randint(1, 6) for _ in range(10) ]
```

Output:
```
[2, 1, 6, 4, 5, 6, 2, 5, 2, 1]
```

We have learned how to simulate the rolling of a die with Python. We assumed that our die is fair, i.e. the probability for each face is equal to 1/6. How can we simulate throwing a crooked or loaded die? The randint methods of both modules are not suitable for this purpose. We will write some functions in the following text to solve this problem.

First we want to have a look at other useful functions of the random module.

RANDOM CHOICES WITH PYTHON

"Having a choice" or "having choices" in real life is better than not having a choice. Even though some people might complain, if they have too much of a choice. Life means making decisions. There are simple choices like "Do I want a boiled egg?", "Soft or Hard boiled?" "Do I go to the cinema, theater or museum? Other choices may have further reaching consequences like choosing the right job, study or what is the best programming language to learn.

Let's do it with Python. The random module contains the right function for this purpose.This function can be used to choose a random element from a non-empty sequence.

This means that we are capable of picking a random character from a string or a random element from a list or a tuple, as we can see in the following examples. You want to have a city trip within Europe and you can't decide where to go? Let Python help you:

```
from random import choice

possible_destinations = ["Berlin", "Hamburg", "Munich",
                         "Amsterdam", "London", "Paris",
                         "Zurich", "Heidelberg", "Strasbourg",
                         "Augsburg", "Milan", "Rome"]

print(choice(possible_destinations))
```

```
Strasbourg
```

The choice function of the random package of the numpy module is more convenient, because it provides further possibilities. The default call, i.e. no further parameters are used, behaves like choice of the random module:

```
from numpy.random import choice

print(choice(possible_destinations))
```

```
Augsburg
```

With the help of the parameter "size" we can create a numpy.ndarray with choice values:

```
x1 = choice(possible_destinations, size=3)
print(x1)
```

```
x2 = choice(possible_destinations, size=(3, 4))
print(x2)
```

```
['London' 'Augsburg' 'London']
[['Strasbourg' 'London' 'Rome' 'Berlin']
 ['Berlin' 'Paris' 'Munich' 'Augsburg']
 ['Heidelberg' 'Paris' 'Berlin' 'Rome']]
```

You might have noticed that the city names can have multiple occurrences. We can prevent this by setting the optional parameter "replace" to "False":

```
print(choice(possible_destinations, size=(3, 4), replace=False))
```

```
[['Heidelberg' 'London' 'Milan' 'Munich']
 ['Hamburg' 'Augsburg' 'Paris' 'Rome']
 ['Berlin' 'Strasbourg' 'Zurich' 'Amsterdam']]
```

Setting the "size" parameter to a non None value, leads us to the sample function.

RANDOM SAMPLES WITH PYTHON

A sample can be understood as a representative part from a larger group, usually called a "population".

The module numpy.random contains a function random_sample, which returns random floats in the half open interval [0.0, 1.0). The results are from the "continuous uniform" distribution over the stated interval. This function takes just one parameter "size", which defines the output shape. If we set size to (3, 4) e.g., we will get an array with the shape (3, 4) filled with random elements:

```
import numpy as np

x = np.random.random_sample((3, 4))
print(x)
```

```
[[ 0.99824096  0.30837203  0.85396161  0.84814744]
 [ 0.45516418  0.64925709  0.19576679  0.8124502 ]
 [ 0.45498107  0.20100427  0.42826199  0.57355053]]
```

If we call random_sample with an integer, we get a one-dimensional array. An integer has the same effect as if we use a one-tuple as an argument:

```
x = np.random.random_sample(7)
print(x)

y = np.random.random_sample((7,))
print(y)
```

```
[ 0.07729483  0.07947532  0.27405822  0.34425005  0.2968612   0.27
234156
  0.41580785]
[ 0.19791769  0.64537929  0.02809775  0.2947372   0.5873195   0.55
059448
  0.98943354]
```

You can also generate arrays with values from an arbitrary interval [a, b), where a has to be less than b. It can be done like this:

`(b - a) * random_sample() + a`

Example:

```
a = -3.4
b = 5.9

A = (b - a) * np.random.random_sample((3, 4)) + a

print(A)
```

```
[[ 5.87026891 -0.13166798  5.56074144  3.48789786]
 [-2.2764547   4.84050253  0.71734827 -0.7357672 ]
 [ 5.8468095   4.56323308  0.05313938 -1.99266987]]
```

The standard module random of Python has a more general function "sample", which produces samples from a population. The population can be a sequence or a set.

The syntax of sample:

`sample(population, k)`

The function creates a list, which contains "k" elements from the "population". The result list contains no multiple occurrences, if the the population contains no multiple occurrences.

If you want to choose a sample within a range of integers, you can - or better you should - use range as the argument for the population.

In the following example we produce six numbers out of the range from 1 to 49 (inclusive). This corresponds to a drawing of the German lottery:

```
import random

print(random.sample(range(1, 50), 6))
```

```
[27, 36, 29, 7, 18, 45]
```

TRUE RANDOM NUMBERS

Have you ever played a game of dice and asked yourself, if something is wrong with the die? You rolled the die for so many times and you still haven't got a certain value like 6 for example.

You may also have asked yourself, if the random modules of Python can create "real" or "true" random numbers, which are e.g. equivalent to an ideal die. The truth is that most random numbers used in computer programs are pseudo-random. The numbers are generated in a predictable way, because the algorithm is deterministic. Pseudo-random numbers are good enough for many purposes, but it may not be "true" random rolling dice or lottery drawings.

The website RANDOM.ORG claims to offer true random numbers. They use the randomness which comes from atmospheric noise. The numbers created this way are for many purposes better than the pseudo-random number algorithms typically used in computer programs.

WEIGHTED PROBABILITIES

INTRODUCTION

In the previous chapter of our tutorial, we introduced the random module. We got to know the functions 'choice' and 'sample'. We used 'choice' to choose a random element from a non-empty sequence and 'sample' to chooses k unique random elements from a population sequence or set. The probality for all elements is evenly distributed, i.e. each element has of the sequences or sets have the same probability to be chosen. This is exactly what we want, if we simulate the rolling of dice. But what about loaded dice? Loaded dice are designed to favor some results over others for whatever reasons. In our previous chapter we had a look at the following examples:

```
from random import choice, sample

print(choice("abcdefghij"))

professions = ["scientist", "philoso
pher", "engineer", "priest"]
print(choice(professions))

print(choice(("beginner", "intermediate", "advanced")))

# rolling one die
x = choice(range(1, 7))
print("The dice shows: " + str(x))

# rolling two dice:
dice = sample(range(1, 7), 2)
print("The two dice show: " + str(dice))
```

```
c
philosopher
advanced
The dice shows: 4
The two dice show: [6, 1]
```

Like we said before, the chances for the elements of the sequence to be chosen are evenly distributed. So the chances for getting a 'scientist' as a return value of the call choice(professions) is 1/4. This is out of touch with reality. There are surely more scientists and engineers in the world than there are priests and philosophers. Just like with the loaded die, we have again the need of a weighted choice.

We will devise a function "weighted_choice", which returns a random element from a sequence like random.choice, but the elements of the sequence will be weighted.

WEIGHTED RANDOM CHOICES

We will define now the weighted choice function. Let's assume that we have three weights, e.g. 1/5, 1/2, 3/10. We can build the cumulative sum of the weights with np.cumsum(weights).

```
import numpy as np

weights = [0.2, 0.5, 0.3]
cum_weights = [0] + list(np.cumsum(weights))
print(cum_weights)
```

```
[0, 0.20000000000000001, 0.69999999999999996, 1.0]
```

If we create a random number x between 0 and 1 by using random.random(), the probability for x to lie within the interval [0, cum_weights[0]) is equal to 1/5. The probability for x to lie within the interval [cum_weights[0], cum_weights[1]) is equal to 1/2 and finally, the probability for x to lie within the interval [cum_weights[1], cum_weights[2]) is 3/10.

Now you are able to understand the basic idea of how weighted_choice operates:

```
import numpy as np

import random
from random import random

def weighted_choice(objects, weights):
    """ returns randomly an element from the sequence of 'object
s',
        the likelihood of the objects is weighted according
        to the sequence of 'weights', i.e. percentages."""

    weights = np.array(weights, dtype=np.float64)
    sum_of_weights = weights.sum()
    # standardization:
    np.multiply(weights, 1 / sum_of_weights, weights)
    weights = weights.cumsum()
    x = random()
    for i in range(len(weights)):
        if x < weights[i]:
            return objects[i]
```

Example:

We can use the function weighted_choice for the following task:

Suppose, we have a "loaded" die with P(6)=3/12 and P(1)=1/12. The probability for the outcomes of all the other possibilities is equally likely, i.e. P(2) = P(3) = P(4) = P(5) = p.

We can calculate p with

1 - P(1) - P(6) = 4 x p

that means

p = 1 / 6

How can we simulate this die with our weighted_choice function?

We call weighted_choice with 'faces_of_die' and the 'weights' list. Each call correspondents to a throw of the loaded die.

We can show that if we throw the die a large number of times, for example 10,000 times, we get roughly the probability values of the weights:

```
from collections import Counter

faces_of_die = [1, 2, 3, 4, 5, 6]
weights = [1/12, 1/6, 1/6, 1/6, 1/6, 3/12]

outcomes = []
n = 10000
for _ in range(n):
    outcomes.append(weighted_choice(faces_of_die, weights))

c = Counter(outcomes)
for key in c:
    c[key] = c[key] / n

print(sorted(c.values()), sum(c.values()))
```

```
[0.0832, 0.1601, 0.1614, 0.1665, 0.1694, 0.2594] 1.0
```

We can also use list of strings with our 'weighted_choice' function.

We define a list of cities and a list with their corresponding populations. The probability of a city to be chosen should be according to their size:

```
cities = ["Frankfurt",
          "Stuttgart",
          "Freiburg",
          "München",
          "Zürich",
          "Hamburg"]
populations = [736000, 628000, 228000, 1450000, 409241, 1841179]
total = sum(populations)
weights = [ round(pop / total, 2) for pop in populations]
print(weights)
for i in range(10):
    print(weighted_choice(cities, populations))
```

```
[0.14, 0.12, 0.04, 0.27, 0.08, 0.35]
Frankfurt
Hamburg
Stuttgart
Hamburg
Hamburg
Zürich
München
Stuttgart
München
Frankfurt
```

WEIGHTED RANDOM CHOICE WITH NUMPY

To produce a weighted choice of an array like object, we can also use the choice function of the numpy.random package. Actually, you should use functions from well-established module like 'NumPy' instead of reinventing the wheel by writing your own code. In addition the 'choice' function from NumPy can do even more. It generates a random sample from a given 1-D array or array like object like a list, tuple and so on. The function can be called with four parameters:

```
choice(a, size=None, replace=True, p=None)
```

Parameter	Meaning
a	a 1-dimensional array-like object or an int. If it is an array-like object, the function will return a random sample from the elements. If it is an int, it behaves as if we called it with np.arange(a)
size	This is an optional parameter defining the output shape. If the given shape is, e.g., `(m, n, k)` , then `m * n * k` samples are drawn. The Default is None, in which case a single value will be returned.
replace	An optional boolean parameter. It is used to define whether the output sample will be with or without replacements.
p	An optional 1-dimensional array-like object, which contains the probabilities associated with each entry in a. If it is not given the sample assumes a uniform distribution over all entries in a.

We will base our first exercise on the popularity of programming language as stated by the "Tiobe index"[1]:

```
from numpy.random import choice

professions = ["scientist",
               "philosopher",
               "engineer",
               "priest",
               "programmer"]

probabilities = [0.2, 0.05, 0.3, 0.15, 0.3]

choice(professions, p=probabilities)
```

Output:
```
'programmer'
```

Let us use the function choice to create a sample from our professions. To get two professions chosen, we set the `size` parameter to the shape `(2, )` . In this case multiple occurances are possible. The top ten programming languages in August 2019 were:

Programming Language	Percentage in August 2019	Change to August 2018
Java	16.028%	-0.85%
C	15.154%	+0.19%
Python	10.020%	+3.03%

Programming Language	Percentage in August 2019	Change to August 2018		
C++	6.057%	-1.41%		
C#	3.842%	+0.30%		
Visual	Basic	.NET	3.695%	-1.07%
JavaScript	2.258%	-0.15%		
PHP	2.075%	-0.85%		
Objective-C	1.690%	+0.33%		
SQL	1.625%	-0.69%		
Ruby	1.316%	+0.13%		

```python
programming_languages = ["Java", "C", "Python", "C++"]
weights = np.array([16, 15.2, 10, 6.1])

# normalization
weights /= sum(weights)
print(weights)

for i in range(10):
    print(choice(programming_languages, p=weights))
```

```
[0.33826638 0.32135307 0.21141649 0.12896406]
Java
C++
Python
Python
C
C++
C
C
Python
C
```

WEIGHTED SAMPLE

In the previous chapter on random numbers and probability, we introduced the function 'sample' of the module 'random' to randomly extract a population or sample from a group of objects liks lists or tuples. Every object had the same likelikhood to be drawn, i.e. to be part of the sample.

In real life situation there will be of course situation in which every or some objects will have different probabilities. We will start again by defining a function on our own. This function will use the previously defined 'weighted_choice' function.

```
def weighted_sample(population, weights, k):
    """
    This function draws a random sample (without repeats)
    of length k      from the sequence 'population' according
    to the list of weights
    """
    sample = set()
    population = list(population)
    weights = list(weights)
    while len(sample) < k:
        choice = weighted_choice(population, weights)
        sample.add(choice)
        index = population.index(choice)
        weights.pop(index)
        population.remove(choice)
        weights = [ x / sum(weights) for x in weights]
    return list(sample)
```

Example using the sample function:

Let's assume we have eight candies, coloured "red", "green", "blue", "yellow", "black", "white", "pink", and "orange". Our friend Peter will have the "weighted" preference 1/24, 1/6, 1/6, 1/12, 1/12, 1/24, 1/8, 7/24 for thes colours. He is allowed to take 3 candies:

```
candies = ["red", "green", "blue", "yellow", "black", "white", "pi
nk", "orange"]
weights = [ 1/24, 1/6, 1/6, 1/12, 1/12, 1/24, 1/8, 7/24]
for i in range(10):
    print(weighted_sample(candies, weights, 3))
```

```
['green', 'orange', 'pink']
['green', 'orange', 'black']
['yellow', 'black', 'pink']
['red', 'green', 'yellow']
['yellow', 'black', 'pink']
['green', 'orange', 'yellow']
['red', 'blue', 'pink']
['white', 'yellow', 'pink']
['green', 'blue', 'pink']
['orange', 'blue', 'black']
```

Let's approximate the likelihood for an orange candy to be included in the sample:

```
n = 100000
orange_counter = 0
for i in range(n):
    if "orange" in weighted_sample(candies, weights, 3):
        orange_counter += 1

print(orange_counter / n)
```

```
0.71015
```

It was completely unnecessary to write this function, because we can use the choice function of NumPy for this purpose as well. All we have to do is assign the shape '(2,)' to the optional parameter 'size'. Let us redo the previous example by substituting weighted_sampe with a call of np.random.choice:

```
n = 100000
orange_counter = 0
for i in range(n):
    if "orange" in np.random.choice(candies,
                                    p=weights,
                                    size=(3,),
                                    replace=False):
        orange_counter += 1

print(orange_counter / n)
```

```
0.71276
```

In addition, the function 'np.random.choice' gives us the possibility to allow repetitions, as we can see in the following example:

```
countries = ["Germany", "Switzerland",
                "Austria", "Netherlands",
                "Belgium", "Poland",
                "France", "Ireland"]
weights = np.array([83019200, 8555541, 8869537,
                    17338900, 11480534, 38413000,
                    67022000, 4857000])
weights = weights / sum(weights)

for i in range(4):
    print(np.random.choice(countries,
                           p=weights,
                           size=(3,),
                           replace=True))
```

```
['Germany' 'Belgium' 'France']
['Poland' 'Poland' 'Poland']
['Germany' 'Austria' 'Poland']
['France' 'France' 'France']
```

CARTESIAN CHOICE

The function cartesian_choice is named after the Cartesian product from set theory

CARTESIAN PRODUCT

The Cartesian product is an operation which returns a set from multiple sets. The result set from the Cartesian product is called a "product set" or simply the "product".

For two sets A and B, the Cartesian product A × B is the set of all ordered pairs (a, b) where a ∈ A and b ∈ B:

$A \times B = \{ (a, b) \mid a \in A \text{ and } b \in B \}$

If we have n sets A_1, A_2, ... A_n, we can build the Cartesian product correspondingly:

$A_1 \times A_2 \times ... \times A_n = \{ (a_1, a_2, ... a_n) \mid a_1 \in A_1, a_2 \in A_2, ... a_n \in A_n]$

The Cartesian product of n sets is sometimes called an n-fold Cartesian product.

CARTESIAN CHOICE: CARTESIAN_CHOICE

We will write now a function cartesian_choice, which takes an arbitrary number of iterables as arguments and returns a list, which consists of random choices from each iterator in the respective order.

Mathematically, we can see the result of the function cartesian_choice as an element of the Cartesian product of the iterables which have been passed as arguments.

```python
import numpy as np

def cartesian_choice(*iterables):
    """
    A list with random choices from each iterable of iterables
    is being created in respective order.

    The result list can be seen as an element of the
    Cartesian product of the iterables
    """
    res = []
    for population in iterables:
        res.append(np.random.choice(population))
    return res


cartesian_choice(["The", "A"],
                 ["red", "green", "blue", "yellow", "grey"],
                 ["car", "house", "fish", "light"],
                 ["smells", "dreams", "blinks", "shines"])
```

Output:
```
['The', 'green', 'house', 'shines']
```

We define now a weighted version of the previously defined function:

```python
import numpy as np

def weighted_cartesian_choice(*iterables):
    """
    An arbitrary number of tuple or lists,
    each consisting of population and weights.
    weighted_cartesian_choice returns a list
    with a chocie from each population
    """
    res = []
    for population, weights in iterables:
        # normalize weight:
        weights = np.array(weights) / sum(weights)
        lst = np.random.choice(population, p=weights)
        res.append(lst)
    return res

determiners = (["The", "A", "Each", "Every", "No"],
               [0.3, 0.3, 0.1, 0.1, 0.2])
colours = (["red", "green", "blue", "yellow", "grey"],
           [0.1, 0.3, 0.3, 0.2, 0.2])
nouns = (["water", "elephant", "fish", "light", "programming langu
age"],
         [0.3, 0.2, 0.1, 0.1, 0.3])
nouns2 = (["of happiness", "of chocolate", "of wisdom", "of challe
nges", "of air"],
         [0.5, 0.2, 0.1, 0.1, 0.1])
verb_phrases = (["smells", "dreams", "thinks", "is made of"],
         [0.4, 0.3, 0.2, 0.1])

print("It may or may not be true:")
for i in range(10):
    res = weighted_cartesian_choice(determiners,
                                    colours,
                                    nouns,
                                    verb_phrases,
                                    nouns2)
    print(" ".join(res) + ".")
```

```
It may or may not be true:
A grey light smells of happiness.
The green water smells of happiness.
A blue water smells of air.
The green elephant dreams of happiness.
A yellow fish smells of happiness.
The grey elephant dreams of air.
No grey programming language dreams of air.
Every blue elephant smells of happiness.
A blue light smells of air.
The grey programming language dreams of happiness.
```

We check in the following version, if the "probabilities" are all right:

```
import random

def weighted_cartesian_choice(*iterables):
    """
    A list with weighted random choices from each iterable of iter
ables
```

```
    is being created in respective order
    """
    res = []
    for population, weight in iterables:
        lst = weighted_choice(population, weight)
        res.append(lst)
    return res

determiners = (["The", "A", "Each", "Every", "No"],
               [0.3, 0.3, 0.1, 0.1, 0.2])
colours = (["red", "green", "blue", "yellow", "grey"],
           [0.1, 0.3, 0.3, 0.2, 0.2])
nouns = (["water", "elephant", "fish", "light", "programming langu
age"],
         [0.3, 0.2, 0.1, 0.1, 0.3])
nouns2 = (["of happiness", "of chocolate", "of wisdom", "of challe
nges", "of air"],
         [0.5, 0.2, 0.1, 0.1, 0.1])
verb_phrases = (["smells", "dreams", "thinks", "is made of"],
         [0.4, 0.3, 0.2, 0.1])

print("It may or may not be true:")
sentences = []
for i in range(10000):
    res = weighted_cartesian_choice(determiners,
                                    colours,
                                    nouns,
                                    verb_phrases,
                                    nouns2)
    sentences.append(" ".join(res) + ".")

words = ["smells", "dreams", "thinks", "is made of"]
from collections import Counter
c = Counter()
for sentence in sentences:
    for word in words:
        if word in sentence:
            c[word] += 1

wsum = sum(c.values())
for key in c:
    print(key, c[key] / wsum)
```

```
It may or may not be true:
thinks 0.2015
is made of 0.1039
smells 0.4016
dreams 0.293
```

RANDOM SEED

A random seed, - also called "seed state", or just "seed" - is a number used to initialize a pseudorandom number generator. When we called random.random() we expected and got a random number between 0 and 1. random.random() calculates a new random number by using the previously produced random number. What about the first time we use random in our program? Yes, there is no previously created random number. If a random number generator is called for the first time, it will have to create a first "random" number.

If we seed a pseudo-random number generator, we provide a first "previous" value. A seed value corresponds to a sequence of generated values for a given random number generator. If you use the same seed value again, you get and you can rely on getting the same sequence of numbers again.

The seed number itself doesn't need to be randomly chosen so that the algorithm creates values which follow a probability distribution in a pseudorandom manner. Yet, the seed matters in terms of security. If you know the seed, you could for example generate the secret encryption key which is based on this seed.

Random seeds are in many programming languages generated from the state of the computer system, which is in lots of cases the system time.

This is true for Python as well. Help on random.seed says that if you call the function with None or no argument it will seed "from current time or from an operating system specific randomness source if available."

```
import random

help(random.seed)
```

```
Help on method seed in module random:

seed(a=None, version=2) method of random.Random instance
    Initialize internal state from hashable object.

    None or no argument seeds from current time or from an operati
ng
    system specific randomness source if available.

    For version 2 (the default), all of the bits are used if *a* i
s a str,
    bytes, or bytearray.  For version 1, the hash() of *a* is use
d instead.

    If *a* is an int, all bits are used.
```

The seed functions allows you to get a determined sequence of random numbers. You can repeat this sequence, whenever you need it again, e.g. for debugging purposes.

```python
import random

random.seed(42)

for _ in range(10):
    print(random.randint(1, 10), end=", ")

print("\nLet's create the same random numbers again:")
random.seed(42)
for _ in range(10):
    print(random.randint(1, 10), end=", ")
```

```
2, 1, 5, 4, 4, 3, 2, 9, 2, 10,
Let's create the same random numbers again:
2, 1, 5, 4, 4, 3, 2, 9, 2, 10,
```

RANDOM NUMBERS IN PYTHON WITH GAUSSIAN AND NORMALVARIATE DISTRIBUTION

We want to create now 1000 random numbers between 130 and 230 that have a gaussian distribution with the mean value mu set to 550 and the standard deviation sigma is set to 30.

```
from random import gauss

n = 1000

values = []
frequencies = {}

while len(values) < n:
    value = gauss(180, 30)
    if 130 < value < 230:
        frequencies[int(value)] = frequencies.get(int(value), 0)
+ 1
        values.append(value)

print(values[:10])
```

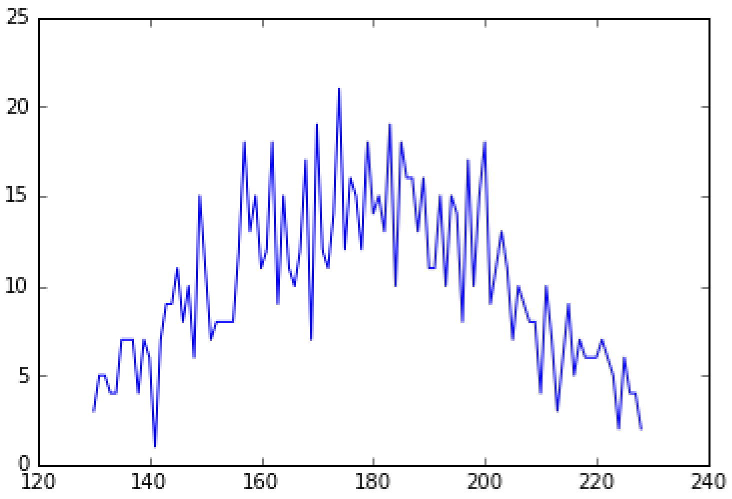
```
[173.49123947564414, 183.47654360102564, 186.96893210720162, 214.9
0676059797428, 199.69909520396007, 183.31521532331496, 157.8503519
2965537, 149.56012897536849, 187.39026585633607, 219.3324248161214
3]
```

The following program plots the random values, which we have created before. We haven't covered matplotlib so far, so it's not necessary to understand the code:

```
%matplotlib inline

import matplotlib.pyplot as plt

freq = list(frequencies.items())
freq.sort()


plt.plot(*list(zip(*freq)))
```

Output:

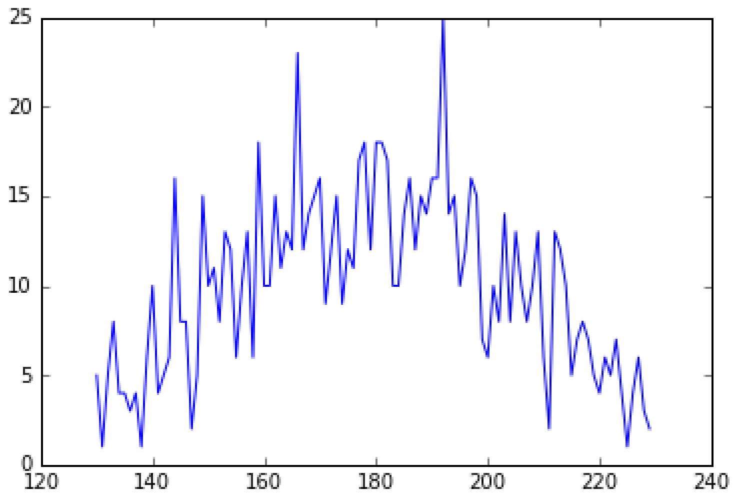
```
[<matplotlib.lines.Line2D at 0x7f282554a828>]
```



We do the same now with normvariate instead of gauss:

```
from random import normalvariate

n = 1000

values = []
frequencies = {}

while len(values) < n:
    value = normalvariate(180, 30)
    if 130 < value < 230:
        frequencies[int(value)] = frequencies.get(int(value), 0)
+ 1
        values.append(value)

freq = list(frequencies.items())
freq.sort()


plt.plot(*list(zip(*freq)))
```

Output:
```
[<matplotlib.lines.Line2D at 0x7f2824fef1d0>]
```



EXERCISE WITH ZEROS AND ONES

It might be a good idea to write the following function as an exercise yourself. The function should be called with a parameter p, which is a probabilty value between 0 and 1. The function returns a 1 with a probability of p, i.e. ones in p percent and zeros in (1 - p) percent of the calls:

```
import random

def random_ones_and_zeros(p):
    """ p: probability 0 <= p <= 1
        returns a 1 with the probability p
    """
    x = random.random()
    if x < p:
        return 1
    else:
        return 0
```

Let's test our little function:

```
n = 1000000
sum(random_ones_and_zeros(0.8) for i in range(n)) / n
```

Output:
```
0.800609
```

It might be a great idea to implement a task like this with a generator. If you are not familar with the way of working of a Python generator, we recommend to consult our chapter on generators and iterators of our Python tutorial.

```
import random

def random_ones_and_zeros(p):
    while True:
        x = random.random()
        yield 1 if x < p else 0

def firstn(generator, n):
        for i in range(n):
                yield next(generator)
```

```
n = 1000000
sum(x for x in firstn(random_ones_and_zeros(0.8), n)) / n
```

Output:
```
0.799762
```

Our generator random_ones_and_zeros can be seen as a sender, which emits ones and zeros with a probability of p and (1-p) respectively.

We will write now another generator, which is receiving this bitstream. The task of this new generator is to read the incoming bitstream and yield another bitstream with ones and zeros with a probability of 0.5 without knowing or using the probability p. It should work for an arbitrary probability value p.[2]

```
def ebitter(bitstream):
    while True:
        bit1 = next(bitstre
am)
        bit2 = next(bitstre
am)
        if bit1 + bit2 ==
1:
            bit3 = next(bitstream)
            if bit2 + bit3 == 1:
                yield 1
            else:
                yield 0
```

```
def ebitter2(bitstream):
    bit1 = next(bitstream)
    bit2 = next(bitstream)
    bit3 = next(bitstream)
    while True:
        if bit1 + bit2 == 1:
            if bit2 + bit3 == 1:
                yield 1
            else:
                yield 0
        bit1, bit2, bit3 = bit2, bit3, next(bitstream)
```

```
n = 1000000
sum(x for x in firstn(ebitter(random_ones_and_zeros(0.8)), n)) / n
```

Output:
```
0.49975
```

```
n = 1000000
sum(x for x in firstn(ebitter2(random_ones_and_zeros(0.8)), n)) /
n
```

Output:
```
0.500011
```

Underlying theory:

Our first generator emits a bitstream B_0, B_1, B_2,...

We check now an arbitrary pair of consecutive Bits B_i, B_{i+1}, ...

Such a pair can have the values 01, 10, 00 or 11. The probability P(01) = (p-1) x p and probability P(10) = p x (p-1), so that the combined probabilty that the two consecutive bits are either 01 or 10 (or the sum of the two bits is 1) is 2 x (p-1) x p

Now we look at another bit B_{i+2}. What is the probability that both

$B_i + B_{i+1} = 1$

and

$B_{i+1} + B_{i+2} = 1$?

The possible outcomes satisfying these conditions and their corresponding probabilities can be found in the following table:

Probability	**B_i**	**B_{i+1}**	**B_{i+2}**

p^2 x (1-p)	0	1	0
p x $(1 - p)^2$	1	0	1

We will denote the outcome sum(B_i, B_{i+1})=1 asX_1 and correspondingly the outcome sum(B_{i+1}, B_{i+2})=1 as X_2

So, the joint probability P(X_1, X_2) = p^2 x (1-p) + p x $(1 - p)^2$ which can be rearranged to p x (1-p)

The conditional probability of X_2 given X_1:

P(X_2 | X_1) = P(X_1, X_2) / P(X_2)

P(X_2 | X_1) = p x (1-p) / 2 x p x (1-p) = 1 / 2

SYNTHETICAL SALES FIGURES

In this subchapter we want to create a data file with sales figures. Imagine that we have a chain of shops in various European and Canadian cities: Frankfurt, Munich, Berlin, Zurich, Hamburg, London, Toronto, Strasbourg, Luxembourg, Amsterdam, Rotterdam, The Hague

We start with an array 'sales' of sales figures for the year 1997:

```
import numpy as np

sales = np.array([1245.89, 2220.00, 1635.77, 1936.25, 1002.03, 209
9.13,  723.99, 990.37, 541.44, 1765.00, 1802.84, 1999.00])
```

The aim is to create a comma separated list like the ones you get from Excel. The file should contain the sales figures, we don't know, for all the shops, we don't have, spanning the year from 1997 to 2016.

We will add random values to our sales figures year after year. For this purpose we construct an array with growthrates. The growthrates can vary between a minimal percent value (min_percent) and maximum percent value (max_percent):

```
min_percent = 0.98  # corresponds to -1.5 %
max_percent = 1.06   # 6 %
growthrates = (max_percent - min_percent) * np.random.random_sampl
```

```
e(12) + min_percent
print(growthrates)
```

```
[ 1.03476561  1.00885095  1.00614899  1.05164581  1.0307091   0.98
822763
  0.99366872  1.05810125  1.04798573  1.02784796  1.05035899  1.02
262023]
```

To get the new sales figures after a year, we multiply the sales array "sales" with the array "growthrates":

```
sales * growthrates
```

Output:

```
array([ 1289.20412146,  2239.649113  ,  1645.82833205,  203
6.24919608,
        1032.80143634,  2074.41825668,   719.40621318,  104
7.91173209,
         567.42139317,  1814.15165757,  1893.62920988,  204
4.21783979])
```

To get a more sustainable sales development, we change the growthrates only every four years.

This is our complete program, which saves the data in a file called sales_figures.csv:

```
import numpy as np
fh = open("sales_figures.csv", "w")

fh.write("Year, Frankfurt, Munich, Berlin, Zurich, Hamburg, Londo
n, Toronto, Strasbourg, Luxembourg, Amsterdam, Rotterdam, The Hagu
e\n")
sales = np.array([1245.89, 2220.00, 1635.77, 1936.25, 1002.03, 209
9.13,  723.99, 990.37, 541.44, 1765.00, 1802.84, 1999.00])

for year in range(1997, 2016):
    line = str(year) + ", " + ", ".join(map(str, sales))
    fh.write(line + "\n")
    if year % 4 == 0:
         min_percent = 0.98  # corresponds to -1.5 %
         max_percent = 1.06   # 6 %
         growthrates = (max_percent - min_percent) * np.random.ran
dom_sample(12) + min_percent
         #growthrates = 1 + (np.random.rand(12) * max_percent - ne
gative_max) / 100
    sales = np.around(sales * growthrates, 2)
fh.close()
```

The result is in the file sales_figures.csv.

We will use this file in our chapter on reading and writing in Numpy.

EXERCISES

1. Let's do some more die rolling. Prove empirically - by writing a simulation program - that the probability for the combined events "an even number is rolled" (E) and "A number greater than 2 is rolled" is 1/3.
2. The file ["universities_uk.txt"](universities_uk.txt) contains a list of universities in the United Kingdom by enrollment from 2013-2014 (data from ([Wikepedia](https://en.wikipedia.org/wiki/List_of_universities_in_the_United_Kingdom_by_enrollment#cite_note-1)). Write a function which returns a tuple (universities, enrollments, total_number_of_students) with - universities: list of University names - enrollments: corresponding list with enrollments - total_number_of_students: over all universities Now you can enroll a 100,000 fictional students with a likelihood corresponding to the real enrollments.
3.

Let me take you back in time and space in our next exercise. We will travel back into ancient Pythonia (Πηθωνια). It was the time when king Pysseus ruled as the benevolent dictator for live. It was the time when Pysseus sent out his messengers throughout the world to announce that the time has come for his princes Anacondos (Ανακονδος), Cobrion (Κομπριον), Boatos (Μποατος) and Addokles (Ανδοκλης) to merry. So, they organized the toughest programming contests amongst the fair and brave amazons, better known as Pythonistas of Pythonia. Finally, only eleven amazons were left to choose from:

1) The ethereal Airla (Αιρλα) 2) Barbara (Βαρβάρα), the one from a foreign country. 3) Eos (Ηως), looking divine in dawn 4) The sweet Glykeria (Γλυκερία) 5) The gracefull Hanna (Αννα) 6) Helen (Ελενη), the light in the dark 7) The good angel Agathangelos (Αγαθάγγελος) 8) the violet tinted cloud Iokaste (Ιοκάστη) 9) Medousa (Μέδουσα), the guardian 10) the self-controlled Sofronia (Σωφρονία) 11) Andromeda (Ανδρομεδα), the one who thinks man or a warrior.

On the day they arrived the chances to be drawn in the lottery are the same for every amazon, but Pysseus wants the lottery to be postponed to some day in the future. The probability changes every day: It will be lowered by 1/13 for the first seven amazones and it will be increased by 1/12 for the last four amazones.

How long will the king have to wait until he can be more than 90 percent sure that his princes Anacondos, Cobrion, Boatos and Addokles will be married to Iokaste, Medousa, Sofronia and Andromeda?

</li>

</ol>

SOLUTIONS TO OUR EXERCISES

1.

```
from random import randint

outcomes = [ randint(1, 6) for _ in range(10000)]

even_pips = [ x for x in outcomes if x % 2 == 0]
greater_two = [ x for x in outcomes if x > 2]

combined = [ x for x in outcomes if x % 2 == 0 and x > 2]

print(len(even_pips) / len(outcomes))
print(len(greater_two) / len(outcomes))
print(len(combined) / len(outcomes))
```

```
0.5061
0.6719
0.3402
```

- At first we will write the function "process_datafile" to process our data file:

```
def process_datafile(filename):
    """ process_datafile -> (universities,
                             enrollments,
                             total_number_of_students)
        universities: list of University names
        enrollments: corresponding list with enrollments
        total_number_of_students: over all universities
    """

    universities = []
    enrollments = []
    with open(filename) as fh:
        total_number_of_students = 0
        fh.readline() # get rid of descriptive first line
        for line in fh:
             line = line.strip()
             *praefix, undergraduates, postgraduates, total = lin
e.rsplit()
             university = praefix[1:]
             total = int(total.replace(",", ""))
             enrollments.append(total)
             universities.append(" ".join(university))
             total_number_of_students += total
    return (universities, enrollments, total_number_of_students)
```

Let's start our function and check the results:

```
universities, enrollments, total_students = process_datafile("univ
ersities_uk.txt")

for i in range(14):
    print(universities[i], end=": ")
    print(enrollments[i])
print("Total number of students onrolled in the UK: ", total_stude
nts)
```

```
Open University in England: 123490
University of Manchester: 37925
University of Nottingham: 33270
Sheffield Hallam University: 33100
University of Birmingham: 32335
Manchester Metropolitan University: 32160
University of Leeds: 30975
Cardiff University: 30180
University of South Wales: 29195
University College London: 28430
King's College London: 27645
University of Edinburgh: 27625
Northumbria University: 27565
University of Glasgow: 27390
Total number of students onrolled in the UK:  2299380
```

We want to enroll now a virtual student randomly to one of the universities. To get a weighted list suitable for our weighted_choice function, we have to normalize the values in the list enrollments:

```
normalized_enrollments = [ students / total_students for students 
in enrollments]

# enrolling a virtual student:
print(weighted_choice(universities, normalized_enrollments))
```

```
University of Dundee
```

We have been asked by the exercise to "enroll" 100,000 fictional students. This can be easily accomplished with a loop:

```
from collections import Counter

outcomes = []
n = 100000
for i in range(n):
    outcomes.append(weighted_choice(universities, normalized_enrol
lments))

c = Counter(outcomes)

print(c.most_common(20))
```

```
[('Open University in England', 5529), ('University of Mancheste
r', 1574), ('University of Nottingham', 1427), ('University of Bir
mingham', 1424), ('Sheffield Hallam University', 1410), ('Manchest
er Metropolitan University', 1408), ('Cardiff University', 1334),
('University of Leeds', 1312), ('University of South Wales', 126
4), ('University of Plymouth', 1218), ('University College Londo
n', 1209), ('Coventry University', 1209), ('University of the Wes
t of England', 1197), ('University of Edinburgh', 1196), ("King's
College London", 1183), ('University of Glasgow', 1181), ('Univers
ity of Central Lancashire', 1176), ('Nottingham Trent Universit
y', 1174), ('University of Sheffield', 1160), ('Northumbria Univer
sity', 1154)]
```

</li>

•

The bunch of amazons is implemented as a list, while we choose a set for Pysseusses favorites. The weights at the beginning are 1/11 for all, i.e. 1/len(amazons).

Every loop cycle corresponds to a new day. Every time we start a new loop cycle, we will draw "n" samples of Pythonistas to calculate the ratio of the number of times the sample is equal to the king's favorites divided by the number of times the sample doesn't match the king's idea of daughter-in-laws. This corresponds to the probability "prob". We stop the first time, the probability is equal or larger than 0.9.

We can use both the function "weighted_same" and "weighted_sample_alternative" to do the drawing.

```
import time

amazons = ["Airla", "Barbara", "Eos",
           "Glykeria", "Hanna", "Helen",
           "Agathangelos", "Iokaste",
           "Medousa", "Sofronia",
           "Andromeda"]

weights = [ 1/len(amazons) for _ in range(len(amazons)) ]

Pytheusses_favorites = {"Iokaste", "Medousa",
                        "Sofronia", "Andromeda"}
n = 1000
counter = 0

prob = 1 / 330
days = 0
factor1 = 1 / 13
factor2 = 1 / 12

start = time.clock()
while prob < 0.9:
    for i in range(n):
        the_chosen_ones = weighted_sample_alternative(amazons, wei
ghts, 4)
        if set(the_chosen_ones) == Pytheusses_favorites:
            counter += 1
    prob = counter / n
    counter = 0
    weights[:7] = [ p - p*factor1 for p in weights[:7] ]
    weights[7:] = [ p + p*factor2 for p in weights[7:] ]
    weights = [ x / sum(weights) for x in weights]
    days += 1
print(time.clock() - start)

print("Number of days, he has to wait: ", days)
```

```
2.870792999999999
Number of days, he has to wait:  33
```

Teh value for the number of days differs, if n is not large enough.

The following is a solutions without round-off errors. We will use Fraction from the module fractions.

```
import time
from fractions import Fraction


amazons = ["Airla", "Barbara", "Eos",
           "Glykeria", "Hanna", "Helen",
           "Agathangelos", "Iokaste",
           "Medousa", "Sofronia",
           "Andromeda"]

weights = [ Fraction(1, 11) for _ in range(len(amazons)) ]

Pytheusses_favorites = {"Iokaste", "Medousa",
                        "Sofronia", "Andromeda"}
n = 1000
counter = 0

prob = Fraction(1, 330)
days = 0
factor1 = Fraction(1, 13)
factor2 = Fraction(1, 12)

start = time.clock()
while prob < 0.9:
    #print(prob)
    for i in range(n):
        the_chosen_ones = weighted_sample_alternative(amazons, wei
ghts, 4)
        if set(the_chosen_ones) == Pytheusses_favorites:
            counter += 1
    prob = Fraction(counter, n)
    counter = 0
    weights[:7] = [ p - p*factor1 for p in weights[:7] ]
    weights[7:] = [ p + p*factor2 for p in weights[7:] ]
    weights = [ x / sum(weights) for x in weights]
    days += 1
print(time.clock() - start)

print("Number of days, he has to wait: ", days)
```

```
35.920345
Number of days, he has to wait:  33
```

We can see that the solution with fractions is beautiful but very slow. Whereas the greater precision doesn't play a role in our case.

So far, we haven't used the power of Numpy. We will do this in the next implementation of our problem:

```
import time
import numpy as np

amazons = ["Airla", "Barbara", "Eos",
           "Glykeria", "Hanna", "Helen",
           "Agathangelos", "Iokaste",
           "Medousa", "Sofronia",
           "Andromeda"]

weights = np.full(11, 1/len(amazons))


Pytheusses_favorites = {"Iokaste", "Medousa",
                        "Sofronia", "Andromeda"}


n = 1000
counter = 0

prob = 1 / 330
days = 0
factor1 = 1 / 13
factor2 = 1 / 12

start = time.clock()
while prob < 0.9:
    for i in range(n):
        the_chosen_ones = weighted_sample_alternative(amazons, wei
ghts, 4)
        if set(the_chosen_ones) == Pytheusses_favorites:
            counter += 1
    prob = counter / n
    counter = 0
    weights[:7] = weights[:7] - weights[:7] * factor1
    weights[7:] = weights[7:] + weights[7:] * factor2
    weights = weights / np.sum(weights)
    #print(weights)
    days += 1
print(time.clock() - start)

print("Number of days, he has to wait: ", days)
```

```
4.930090000000007
Number of days, he has to wait:  33
```

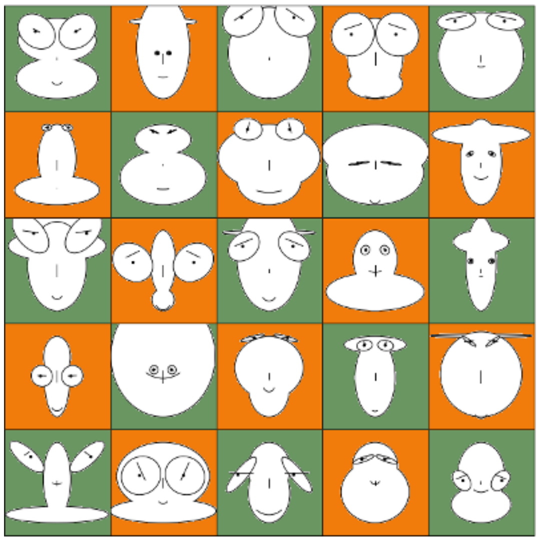
FOOTNOTES:

[1] The TIOBE index or The TIOBE Programming Community index is - according to the website "an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the best programming language or the language in which most lines of code have been written."

[2] I am thankful to Dr. Hanno Baehr who introduced me to the problem of "Random extraction" when participating in a Python training course in Nuremberg in January 2014. Hanno outlined some bits of the theoretical framework. During a night session in a pub called "Zeit & Raum" (english: "Time & Space") I implemented a corresponding Python program to back the theoretical solution empirically.

SYNTHETICAL TEST DATA WITH PYTHON

DEFINITION OF SYNTHETICAL DATA



There is hardly any engineer or scientist who doesn't understand the need for synthetical data, also called synthetic data. But some may have asked themselves what do we understand by synthetical test data? There are lots of situtations, where a scientist or an engineer needs learn or test data, but it is hard or impossible to get real data, i.e. a sample from a population obtained by measurement. The task or challenge of creating synthetical data consists in producing data which resembles or comes quite close to the intended "real life" data. Python is an ideal language for easily producing such data, because it has powerful numerical and linguistic functionalities.

Synthetic data are also necessary to satisfy specific needs or certain conditions that may not be found in the "real life" data. Another use case of synthetical data is to protect privacy of the data needed.

In our previous chapter "Python, Numpy and Probability", we have written some functions, which we will need in the following:

- find_interval
- weighted_choice
- cartesian_choice
- weighted_cartesian_choice
- weighted_sample

You should be familiar with the way of working of these functions.

We saved the functions in a module with the name bk_random.

DEFINITION OF THE SCOPE OF SYNTHETIC DATA CREATION

We want to provide solutions to the following task:

We have n finite sets containing data of various types:

$D_1, D_2, \ldots D_n$

The sets D_i are the data sets from which we want to deduce our synthetical data.

In the actual implementation, the sets will be tuples or lists for practical reasons.

The process of creating synthetic data can be defined by two functions "synthesizer" and "synthesize". Usually, the word synthesizer is used for a computerized electronic device which produces sound. Our synthesizer produces strings or alternatively tuples with data, as we will see later.

The function synthesizer creates the function synthesize:

synthesize = synthesizer($(D_1, D_2, \ldots D_n)$)

The function synthesize, - which may also be a generator like in our implementation, - takes no arguments and the result of a function call sythesize() will be

- a list or a tuple $t = (d_1, d_2, \ldots d_n)$ where d_i is drawn at random from D_i
- or a string which contains the elements $str(d_1), str(d_2), \ldots str(d_n)$ where d_i is also drawn at random from D_i

Let us start with a simple example. We have a list of firstnames and a list of surnames. We want to hire employees for an institute or company. Of course, it will be a lot easier in our synthetical Python environment to find and hire specialsts than in real life. The function "cartesian_choice" from the bk_random module and the concatenation of the randomly drawn firstnames and surnames is all it takes.

```python
import bk_random

firstnames = ["John", "Eve", "Jane", "Paul",
              "Frank", "Laura", "Robert",
              "Kathrin", "Roger", "Simone",
              "Bernard", "Sarah", "Yvonne"]
surnames = ["Singer", "Miles", "Moore",
            "Looper", "Rampman", "Chopman",
            "Smiley", "Bychan", "Smith",
            "Baker", "Miller", "Cook"]

number_of_specialists = 15

employees = set()
while len(employees) < number_of_specialists:
    employee = bk_random.cartesian_choice(firstnames, surnames)
    employees.add(" ".join(employee))

print(employees)
```

```
{'Laura Smith', 'Yvonne Miles', 'Sarah Cook', 'Jane Smith', 'Paul
Moore', 'Jane Miles', 'Jane Looper', 'Frank Singer', 'Frank Mile
s', 'Jane Cook', 'Frank Chopman', 'Laura Cook', 'Yvonne Bychan',
'Eve Miles', 'Simone Cook'}
```

This was easy enough, but we want to do it now in a more structured way, using the synthesizer approach we mentioned before. The code for the case in which the parameter "weights" is not None is still missing in the following implementation:

```
import bk_random

firstnames = ["John", "Eve", "Jane", "Paul",
              "Frank", "Laura", "Robert",
              "Kathrin", "Roger", "Simone",
              "Bernard", "Sarah", "Yvonne"]
surnames = ["Singer", "Miles", "Moore",
            "Looper", "Rampman", "Chopman",
            "Smiley", "Bychan", "Smith",
            "Baker", "Miller", "Cook"]

def synthesizer( data, weights=None, format_func=None, repeats=Tru
e):
    """
    data is a tuple or list of lists or tuples containing the
    data
    weights is a list or tuple of lists or tuples with the
    corresponding weights of the data lists or tuples
    format_func is a reference to a function which defines
    how a random result of the creator function will be formated.
    If None, "creator" will return the list "res".
    If repeats is set to True, the results of helper will not be u
nique
    """

    def synthesize():
        if not repeats:
            memory = set()
        while True:
            res = bk_random.cartesian_choice(*data)
            if not repeats:
                sres = str(res)
                while sres in memory:
                    res = bk_random.cartesian_choice(*data)
                    sres = str(res)
                memory.add(sres)
            if format_func:
                yield format_func(res)
            else:
                yield res
    return synthesize

recruit_employee = synthesizer( (firstnames, surnames),
                                 format_func=lambda x: " ".join(x),
                                 repeats=False)

employee = recruit_employee()
for _ in range(15):
    print(next(employee))
```

```
Sarah Baker
Frank Smiley
Simone Smiley
Frank Bychan
Sarah Moore
Simone Chopman
Frank Chopman
Eve Rampman
Bernard Miller
Simone Bychan
Jane Singer
Roger Smith
John Baker
Robert Cook
Kathrin Cook
```

Every name, i.e first name and last name, had the same likehood to be drawn in the previous example. This is not very realistic, because we will expect in countries like the US or England names like Smith and Miller to occur more often than names like Rampman or Bychan. We will extend our synthesizer function with additional code for the "weighted" case, i.e. weights is not None. If weights are given, we will have to use the function weighted_cartesian_choice from the bk_random module. If "weights" is set to None, we will have to call the function cartesian_choice. We put this decision into a different subfunction of synthesizer to keep the function synthesize clearer.

We do not want to fiddle around with probabilites between 0 and 1 in defining the weights, so we take the detour with integer, which we normalize afterwards.

```
from bk_random import cartesian_choice, weighted_cartesian_choice
```

```
weighted_firstnames = [ ("John", 80), ("Eve", 70), ("Jane", 2),
                        ("Paul", 8), ("Frank", 20), ("Laura", 6),
                        ("Robert", 17), ("Zoe", 3), ("Roger", 8),
                        ("Edgar", 4), ("Susanne", 11), ("Dorothe
e", 22),
                        ("Tim", 17), ("Donald", 12), ("Igor", 15),
                        ("Simone", 9), ("Bernard", 8), ("Sarah",
7),
                        ("Yvonne", 11), ("Bill", 12), ("Bernd", 1
0)]

weighted_surnames = [('Singer', 2), ('Miles', 2), ('Moore', 5),
                     ('Strongman', 5), ('Romero', 3), ("Yiang",
4),
                     ('Looper', 1), ('Rampman', 1), ('Chopman',
1),
                     ('Smiley', 1), ('Bychan', 1), ('Smith', 15
0),
                     ('Baker', 144), ('Miller', 87), ('Cook', 5),
                     ('Joyce', 1), ('Bush', 5), ('Shorter', 6),
                     ('Wagner', 10), ('Sundigos', 10), ('Firenz
e', 8),
                     ('Puttner', 20), ('Faulkner', 10), ('Bowma
n', 11),
                     ('Klein', 1), ('Jungster', 14), ("Warner", 1
4),
                     ('Tiller', 9), ('Wogner', 10), ('Blumentha
l', 16)]


firstnames, weights = zip(*weighted_firstnames)
wsum = sum(weights)
weights_firstnames = [ x / wsum for x in weights]

surnames, weights = zip(*weighted_surnames)
wsum = sum(weights)
weights_surnames = [ x / wsum for x in weights]

weights = (weights_firstnames, weights_surnames)


def synthesizer( data, weights=None, format_func=None, repeats=Tru
e):
    """
    "data" is a tuple or list of lists or tuples containing the
    data.

    "weights" is a list or tuple of lists or tuples with the
    corresponding weights of the data lists or tuples.

    "format_func" is a reference to a function which defines
    how a random result of the creator function will be formated.
    If None,the generator "synthesize" will yield the list "res".

    If "repeats" is set to True, the output values yielded by
    "synthesize" will not be unique.
    """

    def choice(data, weights):
        if weights:
            return weighted_cartesian_choice(*zip(data, weights))
        else:
            return cartesian_choice(*data)

    def synthesize():
        if not repeats:
            memory = set()
        while True:
            res = choice(data, weights)
            if not repeats:
                sres = str(res)
                while sres in memory:
                    res = choice(data, weights)
                    sres = str(res)
                memory.add(sres)
            if format_func:
                yield format_func(res)
            else:
                yield res
    return synthesize



recruit_employee = synthesizer( (firstnames, surnames),
                                weights = weights,
                                format_func=lambda x: " ".join(x),
                                repeats=False)

employee = recruit_employee()
```

```
for _ in range(12):
    print(next(employee))
```

```
Frank Baker
Frank Smith
Eve Smith
Dorothee Baker
John Smith
Bill Bush
John Sundigos
Laura Blumenthal
Zoe Smith
Igor Baker
Bill Miller
Eve Baker
```

WINE EXAMPLE

Let's imagine that you have to describe a dozen wines. Most probably a nice imagination for many, but I have to admit that it is not for me. The main reason is that I am not a wine drinker!

We can write a little Python program, which will use our synthesize function to create automatically "sophisticated criticisms" like this one:

This wine is light-bodied with a conveniently juicy bouquet leading to a lingering flamboyant finish!

Try to find some adverbs, like "seamlessly", "assertively", and some adjectives, like "fruity" and "refined", to describe the aroma.

If you have defined your lists, you can use the synthesize function.

Here is our solution, in case you don't want to do it on your own:

```
import bk_random

body = ['light-bodied', 'medium-bodied', 'full-bodied']

adverbs = ['appropriately', 'assertively', 'authoritatively',
           'compellingly', 'completely', 'continually',
           'conveniently', 'credibly', 'distinctively',
           'dramatically', 'dynamically', 'efficiently',
           'energistically', 'enthusiastically', 'fungibly',
           'globally', 'holisticly', 'interactively',
           'intrinsically', 'monotonectally', 'objectively',
           'phosfluorescently', 'proactively', 'professionally',
           'progressively', 'quickly', 'rapidiously',
           'seamlessly', 'synergistically', 'uniquely']

noun = ['aroma', 'bouquet', 'flavour']

aromas = ['angular', 'bright', 'lingering', 'butterscotch',
          'buttery', 'chocolate', 'complex', 'earth', 'flabby',
          'flamboyant', 'fleshy', 'flowers', 'food friendly',
          'fruits', 'grass', 'herbs', 'jammy', 'juicy', 'mocha',
          'oaked', 'refined', 'structured', 'tight', 'toast',
          'toasty', 'tobacco', 'unctuous', 'unoaked', 'vanilla',
          'velvetly']

example = """This wine is light-bodied with a completely buttery
bouquet leading to a lingering fruity  finish!"""

def describe(data):
    body, adv, adj, noun, adj2 = data
    format_str = "This wine is %s with a %s %s %s\nleading to"
    format_str += " a lingering %s finish!"
    return format_str % (body, adv, adj, noun, adj2)

t = bk_random.cartesian_choice(body, adverbs, aromas, noun, aroma
s)

data = (body, adverbs, aromas, noun, aromas)
synthesize = synthesizer( data, weights=None, format_func=describ
e, repeats=True)
criticism = synthesize()

for i in range(1, 13):
    print("{0:d}. wine:".format(i))
    print(next(criticism))
    print()
```

```
1. wine:
This wine is light-bodied with a progressively earth bouquet
leading to a lingering complex finish!

2. wine:
This wine is medium-bodied with a energistically unctuous bouquet
leading to a lingering vanilla finish!

3. wine:
This wine is medium-bodied with a synergistically flamboyant flavo
ur
leading to a lingering unoaked finish!

4. wine:
This wine is light-bodied with a uniquely toasty flavour
leading to a lingering juicy finish!

5. wine:
This wine is full-bodied with a holisticly flowers flavour
leading to a lingering tobacco finish!

6. wine:
This wine is full-bodied with a energistically toasty flavour
leading to a lingering chocolate finish!

7. wine:
This wine is full-bodied with a proactively tobacco bouquet
leading to a lingering velvetly finish!

8. wine:
This wine is full-bodied with a authoritatively mocha aroma
leading to a lingering juicy finish!

9. wine:
This wine is light-bodied with a dynamically vanilla flavour
leading to a lingering juicy finish!

10. wine:
This wine is medium-bodied with a dynamically structured flavour
leading to a lingering complex finish!

11. wine:
This wine is full-bodied with a distinctively fruits flavour
leading to a lingering complex finish!

12. wine:
```

```
This wine is medium-bodied with a conveniently tight aroma
leading to a lingering chocolate finish!
```

EXERCISE: INTERNATIONAL DISASTER OPERATION

It would be gorgeous, if the problem described in this exercise, would be purely synthetic, i.e. there would be no further catastophes in the world. Completely unrealistic, but a nice daydream. So, the task of this exercise is to provide synthetical test data for an international disaster operation. The countries taking part in this mission might be e.g. France, Switzerland, Germany, Canada, The Netherlands, The United States, Austria, Belgium and Luxembourg.

We want to create a file with random entries of aides. Each line should consist of:

UniqueIdentifier, FirstName, LastName, Country, Field

For example:

```
001, Jean-Paul,  Rennier, France, Medical Aid
002, Nathan, Bloomfield, Canada, Security Aid
003, Michael, Mayer, Germany, Social Worker
```

For practical reasons, we will reduce the countries to France, Italy, Switzerland and Germany in the following example implementation:

```
from bk_random import cartesian_choice, weighted_cartesian_choice

countries = ["France", "Switzerland", "Germany"]

w_firstnames = { "France" : [ ("Marie", 10), ("Thomas", 10),
                            ("Camille", 10), ("Nicolas", 9),
                            ("Léa", 10), ("Julien", 9),
                            ("Manon", 9), ("Quentin", 9),
                            ("Chloé", 8), ("Maxime", 9),
                            ("Laura", 7), ("Alexandre", 6),
                            ("Clementine", 2), ("Grégory", 2),
                            ("Sandra", 1), ("Philippe", 1)],
               "Switzerland": [ ("Sarah", 10), ("Hans", 10),
                            ("Laura", 9), ("Peter", 8),
                            ("Mélissa", 9), ("Walter", 7),
                            ("Océane", 7), ("Daniel", 7),
                            ("Noémie", 6), ("Reto", 7),
                            ("Laura", 7), ("Bruno", 6),
                            ("Eva", 2), ("Urli", 4),
                            ("Sandra", 1), ("Marcel", 1)],
               "Germany": [ ("Ursula", 10), ("Peter", 10),
                            ("Monika", 9), ("Michael", 8),
                            ("Brigitte", 9), ("Thomas", 7),
                            ("Stefanie", 7), ("Andreas", 7),
                            ("Maria", 6), ("Wolfgang", 7),
                            ("Gabriele", 7), ("Manfred", 6),
                            ("Nicole", 2), ("Matthias", 4),
                            ("Christine", 1), ("Dirk", 1)],
               "Italy" : [ ("Francesco", 20), ("Alessandro", 19),
                            ("Mattia", 19), ("Lorenzo", 18),
                            ("Leonardo", 16), ("Andrea", 15),
                            ("Gabriele", 14), ("Matteo", 14),
                            ("Tommaso", 12), ("Riccardo", 11),
                            ("Sofia", 20), ("Aurora", 18),
                            ("Giulia", 16), ("Giorgia", 15),
                            ("Alice", 14), ("Martina", 1
3)]}


w_surnames = { "France" : [ ("Matin", 10), ("Bernard", 10),
                          ("Camille", 10), ("Nicolas", 9),
                          ("Dubois", 10), ("Petit", 9),
                            ("Durand", 8), ("Leroy", 8),
                            ("Fournier", 7), ("Lambert", 6),
                            ("Mercier", 5), ("Rousseau", 4),
                            ("Mathieu", 2), ("Fontaine", 2),
                            ("Muller", 1), ("Robin", 1)],
               "Switzerland": [ ("Müller", 10), ("Meier", 10),
                            ("Schmid", 9), ("Keller", 8),
                            ("Weber", 9), ("Huber", 7),
                            ("Schneider", 7), ("Meyer", 7),
                            ("Steiner", 6), ("Fischer", 7),
                            ("Gerber", 7), ("Brunner", 6),
                            ("Baumann", 2), ("Frei", 4),
                            ("Zimmermann", 1), ("Moser", 1)],
               "Germany": [ ("Müller", 10), ("Schmidt", 10),
                            ("Schneider", 9), ("Fischer", 8),
                            ("Weber", 9), ("Meyer", 7),
                            ("Wagner", 7), ("Becker", 7),
                            ("Schulz", 6), ("Hoffmann", 7),
                            ("Schäfer", 7), ("Koch", 6),
                            ("Bauer", 2), ("Richter", 4),
                            ("Klein", 2), ("Schröder", 1)],
               "Italy" : [ ("Rossi", 20), ("Russo", 19),
                            ("Ferrari", 19), ("Esposito", 18),
                            ("Bianchi", 16), ("Romano", 15),
                            ("Colombo", 14), ("Ricci", 14),
                            ("Marino", 12), ("Grecco", 11),
                            ("Bruno", 10), ("Gallo", 12),
                            ("Conti", 16), ("De Luca", 15),
                            ("Costa", 14), ("Giordano", 13),
                            ("Mancini", 14), ("Rizzo", 13),
                            ("Lombardi", 11), ("Moretto", 9)]}

# separate names and weights
synthesize = {}
identifier = 1
for country in w_firstnames:
    firstnames, weights = zip(*w_firstnames[country])
    wsum = sum(weights)
    weights_firstnames = [ x / wsum for x in weights]
    w_firstnames[country] = [firstnames, weights_firstnames]

    surnames, weights = zip(*w_surnames[country])
    wsum = sum(weights)
    weights_surnames = [ x / wsum for x in weights]
    w_surnames[country] = [surnames, weights_firstnames]

    synthesize[country] = synthesizer( (firstnames, surnames),
                                       (weights_firstnames,
                                        weights_surnames),
                                 format_func=lambda x: " ".joi
n(x),
                                 repeats=False)
nation_prob = [("Germany", 0.3),
               ("France", 0.4),
               ("Switzerland", 0.2),
               ("Italy", 0.1)]

profession_prob = [("Medical Aid", 0.3),
                   ("Social Worker", 0.6),
                   ("Security Aid", 0.1)]

helpers = []
```

```python
for _ in range(200):
    country = weighted_cartesian_choice(zip(*nation_prob))
    profession = weighted_cartesian_choice(zip(*profession_prob))
    country, profession = country[0], profession[0]
    s = synthesize[country]()
    uid = "{id:05d}".format(id=identifier)
    helpers.append((uid, country, next(s), profession ))
    identifier += 1

print(helpers)
```

```
[('00001', 'Germany', 'Brigitte Wagner', 'Social Worker'), ('0000
2', 'France', 'Chloé Muller', 'Medical Aid'), ('00003', 'Switzerla
nd', 'Laura Steiner', 'Medical Aid'), ('00004', 'France', 'Laura M
atin', 'Medical Aid'), ('00005', 'France', 'Léa Fontaine', 'Socia
l Worker'), ('00006', 'Switzerland', 'Océane Meyer', 'Social Worke
r'), ('00007', 'France', 'Léa Fournier', 'Social Worker'), ('0000
8', 'France', 'Marie Matin', 'Social Worker'), ('00009', 'Franc
e', 'Laura Durand', 'Security Aid'), ('00010', 'France', 'Maxime D
ubois', 'Social Worker'), ('00011', 'France', 'Nicolas Mercier',
'Social Worker'), ('00012', 'Italy', 'Mattia Gallo', 'Medical Ai
d'), ('00013', 'France', 'Quentin Leroy', 'Social Worker'), ('0001
4', 'Germany', 'Wolfgang Koch', 'Medical Aid'), ('00015', 'Franc
e', 'Manon Matin', 'Social Worker'), ('00016', 'Switzerland', 'Mél
issa Schneider', 'Social Worker'), ('00017', 'Germany', 'Thomas Ko
ch', 'Social Worker'), ('00018', 'Germany', 'Wolfgang Schäfer', 'M
edical Aid'), ('00019', 'Germany', 'Peter Schäfer', 'Security Ai
d'), ('00020', 'Italy', 'Alice Costa', 'Medical Aid'), ('00021',
'Switzerland', 'Océane Steiner', 'Social Worker'), ('00022', 'Fran
ce', 'Manon Durand', 'Medical Aid'), ('00023', 'Switzerland', 'Dan
iel Meier', 'Social Worker'), ('00024', 'France', 'Laura Fournie
r', 'Social Worker'), ('00025', 'Switzerland', 'Daniel Schneide
r', 'Security Aid'), ('00026', 'Germany', 'Maria Weber', 'Social W
orker'), ('00027', 'Switzerland', 'Sarah Weber', 'Medical Aid'),
('00028', 'Germany', 'Wolfgang Weber', 'Social Worker'), ('0002
9', 'Germany', 'Michael Fischer', 'Social Worker'), ('00030', 'Ger
many', 'Stefanie Hoffmann', 'Social Worker'), ('00031', 'France',
'Laura Mercier', 'Social Worker'), ('00032', 'France', 'Nicolas Le
roy', 'Social Worker'), ('00033', 'Germany', 'Peter Becker', 'Soci
al Worker'), ('00034', 'France', 'Maxime Petit', 'Social Worke
r'), ('00035', 'France', 'Maxime Matin', 'Security Aid'), ('0003
6', 'Germany', 'Stefanie Becker', 'Medical Aid'), ('00037', 'Franc
e', 'Laura Petit', 'Social Worker'), ('00038', 'Switzerland', 'Han
s Fischer', 'Social Worker'), ('00039', 'France', 'Nicolas Lero
y', 'Medical Aid'), ('00040', 'France', 'Léa Matin', 'Social Worke
r'), ('00041', 'Switzerland', 'Bruno Fischer', 'Social Worker'),
('00042', 'France', 'Julien Dubois', 'Medical Aid'), ('00043', 'Fr
ance', 'Alexandre Petit', 'Social Worker'), ('00044', 'France', 'C
amille Camille', 'Social Worker'), ('00045', 'France', 'Camille Ro
usseau', 'Medical Aid'), ('00046', 'France', 'Julien Lambert', 'So
cial Worker'), ('00047', 'France', 'Léa Dubois', 'Social Worke
r'), ('00048', 'Italy', 'Lorenzo Mancini', 'Security Aid'), ('0004
9', 'Germany', 'Ursula Hoffmann', 'Social Worker'), ('00050', 'Ger
many', 'Brigitte Meyer', 'Medical Aid'), ('00051', 'France', 'Sand
ra Lambert', 'Social Worker'), ('00052', 'Italy', 'Alice Rizzo',
'Medical Aid'), ('00053', 'France', 'Chloé Nicolas', 'Social Worke
r'), ('00054', 'Germany', 'Gabriele Schröder', 'Social Worker'),
('00055', 'France', 'Thomas Durand', 'Medical Aid'), ('00056', 'Fr
ance', 'Léa Dubois', 'Medical Aid'), ('00057', 'France', 'Maxime M
ercier', 'Social Worker'), ('00058', 'Germany', 'Peter Schmidt',
'Social Worker'), ('00059', 'France', 'Quentin Durand', 'Social Wo
rker'), ('00060', 'France', 'Camille Petit', 'Social Worker'), ('0
0061', 'Switzerland', 'Laura Schmid', 'Medical Aid'), ('00062', 'I
taly', 'Gabriele Lombardi', 'Social Worker'), ('00063', 'Switzerla
nd', 'Peter Meier', 'Medical Aid'), ('00064', 'Switzerland', 'Ret
o Huber', 'Medical Aid'), ('00065', 'Italy', 'Matteo Mancini', 'Me
dical Aid'), ('00066', 'France', 'Marie Petit', 'Social Worker'),
('00067', 'Germany', 'Manfred Hoffmann', 'Medical Aid'), ('0006
8', 'Germany', 'Brigitte Schmidt', 'Medical Aid'), ('00069', 'Fran
ce', 'Manon Matin', 'Medical Aid'), ('00070', 'France', 'Nicolas P
etit', 'Social Worker'), ('00071', 'France', 'Léa Petit', 'Social
Worker'), ('00072', 'Germany', 'Monika Schulz', 'Social Worker'),
('00073', 'Italy', 'Mattia Rizzo', 'Social Worker'), ('00074', 'It
aly', 'Sofia Colombo', 'Social Worker'), ('00075', 'Germany', 'Mic
hael Schäfer', 'Medical Aid'), ('00076', 'Germany', 'Matthias Hoff
mann', 'Social Worker'), ('00077', 'Germany', 'Wolfgang Schneide
r', 'Social Worker'), ('00078', 'France', 'Julien Dubois', 'Socia
l Worker'), ('00079', 'Germany', 'Peter Fischer', 'Social Worke
r'), ('00080', 'France', 'Julien Leroy', 'Social Worker'), ('0008
1', 'France', 'Julien Bernard', 'Social Worker'), ('00082', 'Germa
ny', 'Michael Schmidt', 'Social Worker'), ('00083', 'France', 'Man
on Bernard', 'Social Worker'), ('00084', 'Switzerland', 'Hans Hube
r', 'Security Aid'), ('00085', 'Germany', 'Monika Schneider', 'Med
ical Aid'), ('00086', 'Switzerland', 'Noémie Müller', 'Security Ai
d'), ('00087', 'Switzerland', 'Sarah Gerber', 'Medical Aid'), ('00
088', 'Germany', 'Thomas Müller', 'Medical Aid'), ('00089', 'Switz
erland', 'Sarah Weber', 'Medical Aid'), ('00090', 'France', 'Laur
a Petit', 'Medical Aid'), ('00091', 'Switzerland', 'Sarah Gerbe
r', 'Medical Aid'), ('00092', 'Switzerland', 'Reto Schmid', 'Medic
al Aid'), ('00093', 'Germany', 'Monika Schneider', 'Medical Ai
d'), ('00094', 'France', 'Quentin Matin', 'Social Worker'), ('0009
5', 'Italy', 'Aurora Colombo', 'Social Worker'), ('00096', 'German
y', 'Ursula Meyer', 'Social Worker'), ('00097', 'Germany', 'Manfre
d Weber', 'Social Worker'), ('00098', 'Italy', 'Giulia Ferrari',
'Medical Aid'), ('00099', 'France', 'Thomas Muller', 'Social Worke
r'), ('00100', 'Switzerland', 'Daniel Schneider', 'Medical Aid'),
('00101', 'France', 'Maxime Camille', 'Medical Aid'), ('00102', 'F
rance', 'Laura Petit', 'Social Worker'), ('00103', 'Germany', 'Man
fred Schmidt', 'Medical Aid'), ('00104', 'Italy', 'Martina Lombard
i', 'Social Worker'), ('00105', 'Switzerland', 'Sarah Baumann', 'M
edical Aid'), ('00106', 'Switzerland', 'Bruno Gerber', 'Security A
id'), ('00107', 'Switzerland', 'Laura Müller', 'Social Worker'),
('00108', 'Germany', 'Andreas Weber', 'Social Worker'), ('00109',
'Switzerland', 'Hans Fischer', 'Social Worker'), ('00110', 'Switze
rland', 'Daniel Meyer', 'Social Worker'), ('00111', 'France', 'Jul
ien Rousseau', 'Security Aid'), ('00112', 'Switzerland', 'Reto Sch
mid', 'Social Worker'), ('00113', 'Switzerland', 'Urli Schneide
r', 'Social Worker'), ('00114', 'France', 'Grégory Rousseau', 'Med
ical Aid'), ('00115', 'France', 'Marie Durand', 'Social Worker'),
('00116', 'France', 'Léa Durand', 'Social Worker'), ('00117', 'Fra
nce', 'Camille Matin', 'Medical Aid'), ('00118', 'Germany', 'Wolfg
ang Schneider', 'Social Worker'), ('00119', 'France', 'Julien Mati
n', 'Social Worker'), ('00120', 'France', 'Marie Leroy', 'Social W
orker'), ('00121', 'Switzerland', 'Mélissa Brunner', 'Security Ai
d'), ('00122', 'Germany', 'Ursula Schneider', 'Social Worker'),
('00123', 'France', 'Camille Mercier', 'Social Worker'), ('0012
4', 'France', 'Julien Camille', 'Social Worker'), ('00125', 'Switz
erland', 'Laura Schmid', 'Medical Aid'), ('00126', 'France', 'Cami
lle Durand', 'Social Worker'), ('00127', 'France', 'Marie Camill
e', 'Medical Aid'), ('00128', 'Germany', 'Monika Wagner', 'Social
Worker'), ('00129', 'Italy', 'Giorgia Esposito', 'Security Aid'),
('00130', 'France', 'Clementine Mercier', 'Social Worker'), ('0013
1', 'France', 'Marie Matin', 'Social Worker'), ('00132', 'Switzerl
and', 'Noémie Brunner', 'Medical Aid'), ('00133', 'France', 'Nicol
as Leroy', 'Security Aid'), ('00134', 'France', 'Camille Camill
e', 'Social Worker'), ('00135', 'Germany', 'Wolfgang Fischer', 'Me
dical Aid'), ('00136', 'Germany', 'Brigitte Müller', 'Medical Ai
d'), ('00137', 'Germany', 'Peter Schneider', 'Social Worker'), ('0
0138', 'Switzerland', 'Laura Schneider', 'Medical Aid'), ('0013
9', 'France', 'Chloé Rousseau', 'Social Worker'), ('00140', 'Ital
y', 'Alice De Luca', 'Medical Aid'), ('00141', 'France', 'Thomas B
ernard', 'Social Worker'), ('00142', 'Italy', 'Francesco Grecco',
'Medical Aid'), ('00143', 'Switzerland', 'Peter Frei', 'Medical Ai
d'), ('00144', 'France', 'Philippe Mercier', 'Security Aid'), ('00
145', 'Germany', 'Monika Meyer', 'Social Worker'), ('00146', 'Fran
ce', 'Alexandre Lambert', 'Medical Aid'), ('00147', 'Switzerlan
d', 'Sarah Brunner', 'Security Aid'), ('00148', 'Germany', 'Wolfga
ng Schneider', 'Social Worker'), ('00149', 'Germany', 'Manfred Mül
ler', 'Social Worker'), ('00150', 'France', 'Léa Dubois', 'Medica
l Aid'), ('00151', 'Switzerland', 'Reto Schmid', 'Medical Aid'),
('00152', 'France', 'Manon Lambert', 'Social Worker'), ('00153',
'France', 'Chloé Fournier', 'Social Worker'), ('00154', 'France',
'Grégory Bernard', 'Social Worker'), ('00155', 'Italy', 'Martina B
runo', 'Social Worker'), ('00156', 'France', 'Marie Nicolas', 'Soc
ial Worker'), ('00157', 'Italy', 'Giorgia Romano', 'Social Worke
r'), ('00158', 'France', 'Thomas Mercier', 'Security Aid'), ('0015
9', 'Germany', 'Manfred Richter', 'Social Worker'), ('00160', 'Ger
many', 'Wolfgang Schäfer', 'Social Worker'), ('00161', 'Germany',
'Peter Müller', 'Security Aid'), ('00162', 'Switzerland', 'Océane
```

```
Meyer', 'Social Worker'), ('00163', 'Germany', 'Monika Schneide
r', 'Social Worker'), ('00164', 'France', 'Chloé Dubois', 'Social
Worker'), ('00165', 'Germany', 'Peter Fischer', 'Social Worker'),
('00166', 'Germany', 'Christine Müller', 'Social Worker'), ('0016
7', 'Switzerland', 'Walter Steiner', 'Security Aid'), ('00168', 'G
ermany', 'Dirk Bauer', 'Medical Aid'), ('00169', 'Germany', 'Matth
ias Schmidt', 'Social Worker'), ('00170', 'Germany', 'Andreas Schn
eider', 'Medical Aid'), ('00171', 'Italy', 'Gabriele Grecco', 'Med
ical Aid'), ('00172', 'France', 'Léa Matin', 'Security Aid'), ('00
173', 'France', 'Nicolas Dubois', 'Social Worker'), ('00174', 'Swi
tzerland', 'Bruno Fischer', 'Social Worker'), ('00175', 'France',
'Camille Matin', 'Social Worker'), ('00176', 'Switzerland', 'Mélis
sa Zimmermann', 'Social Worker'), ('00177', 'Germany', 'Stefanie B
ecker', 'Medical Aid'), ('00178', 'France', 'Maxime Leroy', 'Socia
l Worker'), ('00179', 'Germany', 'Michael Fischer', 'Security Ai
d'), ('00180', 'Germany', 'Stefanie Schmidt', 'Medical Aid'), ('00
181', 'Germany', 'Peter Schneider', 'Social Worker'), ('00182', 'S
witzerland', 'Laura Huber', 'Social Worker'), ('00183', 'France',
'Marie Fournier', 'Medical Aid'), ('00184', 'Italy', 'Leonardo Mor
etto', 'Social Worker'), ('00185', 'Germany', 'Peter Meyer', 'Soci
al Worker'), ('00186', 'France', 'Alexandre Durand', 'Social Worke
r'), ('00187', 'Switzerland', 'Walter Müller', 'Social Worker'),
('00188', 'France', 'Chloé Leroy', 'Medical Aid'), ('00189', 'Swit
zerland', 'Walter Weber', 'Social Worker'), ('00190', 'Switzerlan
d', 'Sarah Steiner', 'Social Worker'), ('00191', 'Germany', 'Wolfg
ang Fischer', 'Social Worker'), ('00192', 'Germany', 'Matthias Bec
ker', 'Security Aid'), ('00193', 'Germany', 'Ursula Schäfer', 'Soc
ial Worker'), ('00194', 'Switzerland', 'Océane Keller', 'Security
Aid'), ('00195', 'Germany', 'Brigitte Richter', 'Medical Aid'),
('00196', 'Germany', 'Ursula Müller', 'Medical Aid'), ('00197', 'I
taly', 'Tommaso Rizzo', 'Social Worker'), ('00198', 'Switzerlan
d', 'Marcel Fischer', 'Social Worker'), ('00199', 'France', 'Léa P
etit', 'Medical Aid'), ('00200', 'France', 'Nicolas Camille', 'Sec
urity Aid')]
```

```python
with open("disaster_mission.txt", "w") as fh:
    fh.write("Reference number,Country,Name,Function\n")
    for el in helpers:
        fh.write(",".join(el) + "\n")
```

In [ ]:

```python

```

NUMPY: BOOLEAN INDEXING

```
import numpy as np

A = np.array([4, 7, 3, 4, 2, 8])

print(A == 4)
```

```
[ True False False  True False Fals
e]
```

Every element of the Array A is tested, if it is equal to 4. The results of these tests are the Boolean elements of the result array.

Of course, it is also possible to check on "<", "<=", ">" and ">=".

```
print(A < 5)
```

```
[ True False  True  True  True False]
```

It works also for higher dimensions:

```
B = np.array([[42,56,89,65],
              [99,88,42,12],
              [55,42,17,18]])

print(B>=42)
```

```
[[ True  True  True  True]
 [ True  True  True False]
 [ True  True False False]]
```

It is a convenient way to threshold images.

```
import numpy as np

A = np.array([
[12, 13, 14, 12, 16, 14, 11, 10,  9],
[11, 14, 12, 15, 15, 16, 10, 12, 11],
[10, 12, 12, 15, 14, 16, 10, 12, 12],
[ 9, 11, 16, 15, 14, 16, 15, 12, 10],
[12, 11, 16, 14, 10, 12, 16, 12, 13],
[10, 15, 16, 14, 14, 14, 16, 15, 12],
[13, 17, 14, 10, 14, 11, 14, 15, 10],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 19, 12, 14, 11, 12, 14, 18, 10],
[14, 22, 17, 19, 16, 17, 18, 17, 13],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 16, 12, 14, 11, 12, 14, 18, 11],
[10, 19, 12, 14, 11, 12, 14, 18, 10],
[14, 22, 12, 14, 11, 12, 14, 17, 13],
[10, 16, 12, 14, 11, 12, 14, 18, 11]])

B = A < 15
B.astype(np.int)
```

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

If you have a close look at the previous output, you will see, that it the upper case 'A' is hidden in the array B.

FANCY INDEXING

We will index an array C in the following example by using a Boolean mask. It is called fancy indexing, if arrays are indexed by using boolean or integer arrays (masks). The result will be a copy and not a view.

In our next example, we will use the Boolean mask of one array to select the corresponding elements of another array. The new array R contains all the elements of C where the corresponding value of (A<=5) is True.

```
C = np.array([123,188,190,99,77,88,100])
A = np.array([4,7,2,8,6,9,5])

R = C[A<=5]
print(R)
```

```
[123 190 100]
```

INDEXING WITH AN INTEGER ARRAY

In the following example, we will index with an integer array:

```
C[[0, 2, 3, 1, 4, 1]]
```

Output:

```
array([123, 190,  99, 188,  77, 188])
```

Indices can appear in every order and multiple times!

EXERCISES

Extract from the array np.array([3,4,6,10,24,89,45,43,46,99,100]) with Boolean masking all the number

- which are not divisible by 3
- which are divisible by 5
- which are divisible by 3 and 5
- which are divisible by 3 and set them to 42

SOLUTIONS

```
import numpy as np
A = np.array([3,4,6,10,24,89,45,43,46,99,100])

div3 = A[A%3!=0]
print("Elements of A not divisible by 3:")
print(div3)


div5 = A[A%5==0]
print("Elements of A divisible by 5:")
print(div5)

print("Elements of A, which are divisible by 3 and 5:")
print(A[(A%3==0) & (A%5==0)])
print("------------------")

#

A[A%3==0] = 42
print("""New values of A after setting the elements of A,
which are divisible by 3, to 42:""")
print(A)
```

```
Elements of A not divisible by 3:
[  4  10  89  43  46 100]
Elements of A divisible by 5:
[ 10  45 100]
Elements of A, which are divisible by 3 and 5:
[45]
------------------
New values of A after setting the elements of A,
which are divisible by 3, to 42:
[ 42   4  42  10  42  89  42  43  46  42 100]
```

NONZERO AND WHERE

There is an ndarray method called nonzero and a numpy method with this name. The two functions are equivalent.

For an ndarray a both numpy.nonzero(a) and a.nonzero() return the indices of the elements of a that are non-zero. The indices are returned as a tuple of arrays, one for each dimension of 'a'. The corresponding non-zero values can be obtained with:

```
a[numpy.nonzero(a)]
```

```
import numpy as np

a = np.array([[0, 2, 3, 0, 1],
              [1, 0, 0, 7, 0],
              [5, 0, 0, 1, 0]])

print(a.nonzero())
```

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

If you want to group the indices by element, you can use transpose:

`transpose(nonzero(a))`

A two-dimensional array is returned. Every row corresponds to a non-zero element.

```
np.transpose(a.nonzero())
```

Output:
```
array([[0, 1],
       [0, 2],
       [0, 4],
       [1, 0],
       [1, 3],
       [2, 0],
       [2, 3]])
```

The corresponding non-zero values can be retrieved with:

```
 a[a.nonzero()]
```

Output:
```
array([2, 3, 1, 1, 7, 5, 1])
```

The function 'nonzero' can be used to obtain the indices of an array, where a condition is True. In the following script, we create the Boolean array B >= 42:

```
B = np.array([[42,56,89,65],
              [99,88,42,12],
              [55,42,17,18]])

print(B >= 42)
```

```
[[ True  True  True  True]
 [ True  True  True False]
 [ True  True False False]]
```

np.nonzero(B >= 42) yields the indices of the B where the condition is true:

EXERCISE

Calculate the prime numbers between 0 and 100 by using a Boolean array.

Solution:

```
import numpy as np

is_prime = np.ones((100,), dtype=bool)

# Cross out 0 and 1 which are not primes:
is_prime[:2] = 0

# cross out its higher multiples (sieve of Eratosthenes):
nmax = int(np.sqrt(len(is_prime)))
for i in range(2, nmax):
    is_prime[2*i::i] = False

print(np.nonzero(is_prime))
```

```
(array([ 2,  3,  5,  7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 4
7, 53, 59,
       61, 67, 71, 73, 79, 83, 89, 97]),)
```

FLATNONZERO AND COUNT_NONZERO

similar functions:

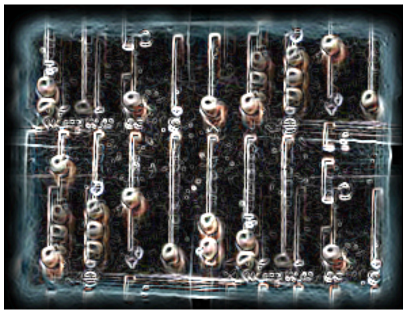
- flatnonzero :

```
  Return indices that are non-zero in the flattened versio
n of the input
  array.
```

- count_nonzero :

```
  Counts the number of non-zero elements in the input arra
y.
```

MATRIX MULTIPLICAION, DOT AND CROSS PRODUCT



Distorted Abacus In the previous chapter of our introduction in NumPy we have demonstrated how to create and change Arrays. In this chapter we want to show, how we can perform in Python with the module NumPy all the basic Matrix Arithmetics like

- Matrix addition
- Matrix subtraction
- Matrix multiplication
- Scalar product
- Cross product
- and lots of other operations on matrices

The arithemtic standard Operators

- +
- -
- *
- /

- %

are applied on the elements, this means that the arrays have to have the same size.

```
import numpy as np

x = np.array([1, 5, 2])
y = np.array([7, 4, 1])
x + y
```

Output:
```
array([8, 9, 3])
```

```
x * y
```

Output:
```
array([ 7, 20,  2])
```

In [ ]:
```
x - y
```

```
x / y
```

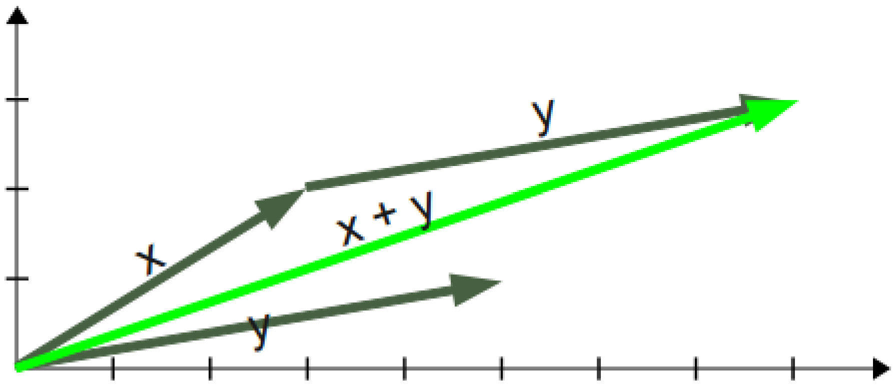
Output:
```
array([0.14285714, 1.25      , 2.        ])
```

```
x % y
```

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

VECTOR ADDITION AND SUBTRACTION

Many people know vector addition and subtraction from physics, to be exact from the parallelogram of forces. It is a method for solving (or visualizing) the results of applying two forces to an object.

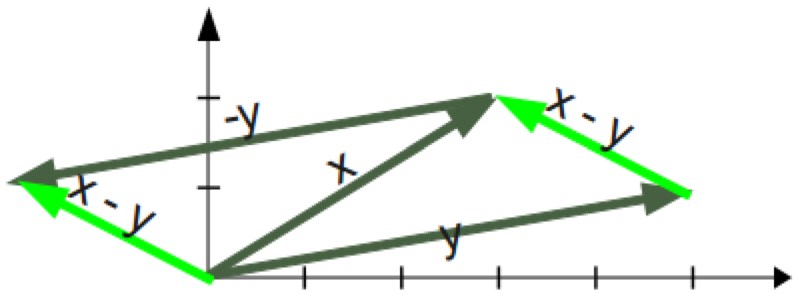


The addition of two vectors, in our example (see picture) x and y, may be represented graphically by placing the start of the arrow y at the tip of the arrow x, and then drawing an arrow from the start (tail) of x to the tip (head) of y. The new arrow drawn represents the vector x + y

```
x = np.array([3, 2])
y = np.array([5, 1])
z = x + y
z
```

Output:
```
array([8, 3])
```

Subtracting a vector is the same as adding its negative. So, the difference of the vectors x and y is equal to the sum of x and -y: x - y = x + (-y) Subtraction of two vectors can be geometrically defined as follows: to subtract y from x, we place the end points of x and y at the same point, and then draw an arrow from the tip of y to the tip of x. That arrow represents the vector x - y, see picture on the right side.



Mathematically, we subtract the corresponding components of vector y from the vector x.

SCALAR PRODUCT / DOT PRODUCT

In mathematics, the dot product is an algebraic operation that takes two coordinate vectors of equal size and returns a single number. The result is calculated by multiplying corresponding entries and adding up those products. The name "dot product" stems from the fact that the centered dot "·" is often used to designate this operation. The name "scalar product" focusses on the scalar nature of the result. of the result.

Definition of the scalar product:

$$\vec{a} \cdot \vec{b} = |\vec{a}|\,|\vec{b}| \cos\measuredangle(\vec{a},\ \vec{b})$$

We can see from the definition of the scalar product that it can be used to calculate the cosine of the angle between two vectors.

Calculation of the scalar product:

$$\vec{a} \cdot \vec{b} = a_1b_1 + a_2b_2 + a_3c_3$$

Finally, we want to demonstrate how to calculate the scalar product in Python:

```
x = np.array([1, 2, 3])
y = np.array([-7, 8, 9])
np.dot(x, y)
```

Output:
```
36
```

```
dot = np.dot(x ,y)
x_modulus = np.sqrt((x*x).sum())
y_modulus = np.sqrt((y*y).sum())
cos_angle = dot / x_modulus / y_modulus # cosine of angle between
x and y
angle = np.arccos(cos_angle)
angle
```

Output:
```
0.808233789010825
```

```
angle * 360 / 2 / np.pi # angle in degrees
```

Output:
```
46.308384970187326
```

In [ ]:

```
36
>>> dot = np.dot(x,y)
>>> x_modulus = np.sqrt((x*x).sum())
>>> y_modulus = np.sqrt((y*y).sum())
>>> cos_angle = dot / x_modulus / y_modulus # cosine of angle betw
```

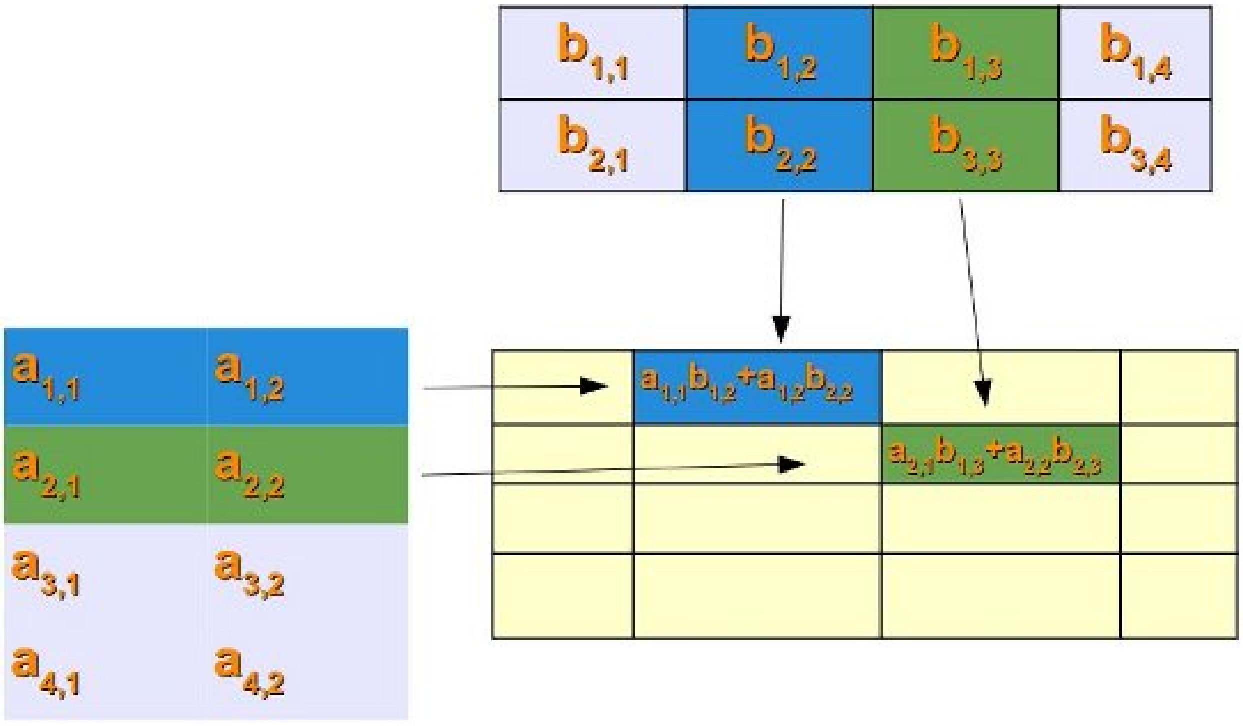
```
een x and y
>>> angle = np.arccos(cos_angle)
>>> angle
0.80823378901082499
>>> angle * 360 / 2 / np.pi # angle in degrees
```

MATRIX PRODUCT

The matrix product of two matrices can be calculated if the number of columns of the left matrix is equal to the number of rows of the second or right matrix.
The product of a (l x m)-matrix A = $(a_{ij})_{i=1...l,\ j=1..m}$ and an (m x n)-matrix B = $(b_{ij})_{i=1...m,\ j=1..n}$ is a matrix C = $(c_{ij})_{i=1...l,\ j=1..n}$, which is calculated like this:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

The following picture illustrates it further:



If we want to perform matrix multiplication with two numpy arrays (ndarray), we have to use the dot product:

```
x = np.array( ((2,3), (3, 5)) )
```

```
y = np.matrix( ((1,2), (5, -1)) )
print(np.dot(x,y))
```

```
[[17  1]
 [28  1]]
```

SIMPLE PRACTICAL APPLICATION FOR MATRIX MULTIPLICATION

In the following practical example, we come to talk about the sweet things of life.
Let's assume there are four people, and we call them Lucas, Mia, Leon and Hannah. Each of them has bought chocolates out of a choice of three. The brand are A, B and C, not very marketable, we have to admit. Lucas bought 100 g of brand A, 175 g of brand B and 210 of C. Mia choose 90 g of A, 160 g of B and 150 g of C. Leon bought 200 g of A, 50 of B and 100 g of C. Hannah apparently didn't like brand B, because she hadn't bought any of those. But she she seems to be a real fan of brand C, because she bought 310 g of them. Furthermore she bought 120 g of A.

So, what's the price in Euro of these chocolates: A costs 2.98 per 100 g, B costs 3.90 and C only 1.99 Euro.

If we have to calculate how much each of them had to pay, we can use Python, NumPy and Matrix multiplication:

```
import numpy as np
NumPersons = np.array([[100, 175, 210],
                       [90, 160, 150],
                       [200, 50, 100],
                       [120, 0, 310]])
Price_per_100_g = np.array([2.98, 3.90, 1.99])
Price_in_Cent = np.dot(NumPersons,Price_per_100_g)
Price_in_Euro = Price_in_Cent / 100
Price_in_Euro
```

Output:
```
array([13.984, 11.907,  9.9  ,  9.745])
```

CROSS PRODUCT

Let's stop consuming delicious chocolates and come back to a more mathematical and less high-calorie topic, i.e. the cross product.

The cross product or vector product is a binary operation on two vectors in three-dimensional space. The result is a vector which is perpendicular to the vectors being multiplied and normal to the plane containing them.

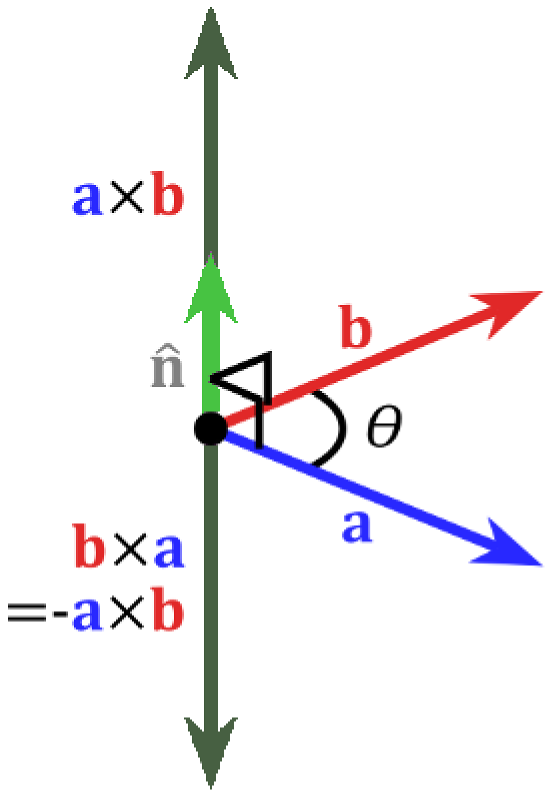
The cross product of two vectors a and b is denoted by a × b.

It's defined as:

$$\vec{a} \times \vec{b} = |\vec{a}|\ |\vec{b}|\ \sin\measuredangle(\vec{a}, \vec{b})\ |\vec{n}|$$

where n is a unit vector perpendicular to the plane containing a and b in the direction given by the right-hand rule.

If either of the vectors being multiplied is zero or the vectors are parallel then their cross product is zero. More generally, the magnitude of the product equals the area of a parallelogram with the vectors as sides. If the vectors are perpendicular the parallelogram is a rectangle and the magnitude of the product is the product of their lengths.



```
x = np.array([0, 0, 1])
y = np.array([0, 1, 0])

np.cross(x, y)
```

Output:
```
array([-1,  0,  0])
```

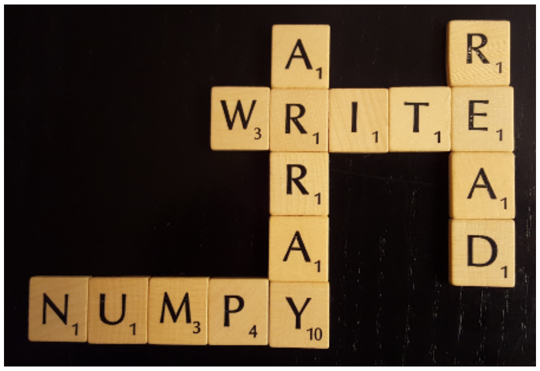
```
np.cross(y, x)
```

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

READING AND WRITING DATA FILES

There are lots of ways for reading from file and writing to data files in numpy. We will discuss the different ways and corresponding functions in this chapter:

- savetxt
- loadtxt
- tofile
- fromfile
- save
- load
- genfromtxt



SAVING TEXTFILES WITH SAVETXT

The first two functions we will cover are savetxt and loadtxt.

In the following simple example, we define an array x and save it as a textfile with savetxt:

```
import numpy as np

x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]], np.int32)

np.savetxt("test.txt", x)
```

The file "test.txt" is a textfile and its content looks like this:

```
bernd@andromeda:~/Dropbox/notebooks/numpy$ more test.txt
1.000000000000000000e+00 2.000000000000000000e+00 3.000000000000000
000e+00
4.000000000000000000e+00 5.000000000000000000e+00 6.000000000000000
000e+00
7.000000000000000000e+00 8.000000000000000000e+00 9.000000000000000
000e+00
```

Attention: The above output has been created on the Linux command prompt!

It's also possible to print the array in a special format, like for example with three decimal places or as integers, which are preceded with leading blanks, if the number of digits is less than 4 digits. For this purpose we assign a format string to the third parameter 'fmt'. We saw in our first example that the default delimeter is a blank. We can change this behaviour by assigning a string to the parameter "delimiter". In most cases this string will consist solely of a single character but it can be a sequence of character, like a smiley " :-) " as well:

```
np.savetxt("test2.txt", x, fmt="%2.3f", delimiter=",")
np.savetxt("test3.txt", x, fmt="%04d", delimiter=" :-) ")
```

The newly created files look like this:

```
bernd@andromeda:~/Dropbox/notebooks/numpy$ more test2.txt
1.000,2.000,3.000
4.000,5.000,6.000
7.000,8.000,9.000
bernd@andromeda:~/Dropbox/notebooks/numpy$ more test3.txt
0001 :-) 0002 :-) 0003
0004 :-) 0005 :-) 0006
0007 :-) 0008 :-) 0009
```

The complete syntax of savetxt looks like this:

```
savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', heade
r='', footer='', comments='# ')
```

Parameter	Meaning
X	array_like Data to be saved to a text file.
fmt	str or sequence of strs, optional A single format (%10.5f), a sequence of formats, or a multi-format string, e.g. 'Iteration %d -- %10.5f', in which case 'delimiter' is ignored. For complex 'X', the legal options for 'fmt' are: a) a single specifier, "fmt='%.4e'", resulting in numbers formatted like "' (%s+%sj)' % (fmt, fmt)" b) a full string specifying every real and imaginary part, e.g. "' %.4e %+.4j %.4e %+.4j %.4e %+.4j'" for 3 columns c) a list of specifiers, one per column - in this case, the real and imaginary part must have separate specifiers, e.g. "['%.3e + %.3ej', '(%.15e%+.15ej)']" for 2 columns
delimiter	A string used for separating the columns.
newline	A string (e.g. "\n", "\r\n" or ",\n") which will end a line instead of the default line ending
header	A String that will be written at the beginning of the file.
footer	A String that will be written at the end of the file.
comments	A String that will be prepended to the 'header' and 'footer' strings, to mark them as comments. The hash tag '#' is

Parameter	Meaning
	used as the default.

LOADING TEXTFILES WITH LOADTXT

We will read in now the file "test.txt", which we have written in our previous subchapter:

```
y = np.loadtxt("test.txt")
print(y)
```

```
[[ 1.   2.   3.]
 [ 4.   5.   6.]
 [ 7.   8.   9.]]
```

```
y = np.loadtxt("test2.txt", delimiter=",")
print(y)
```

```
[[ 1.   2.   3.]
 [ 4.   5.   6.]
 [ 7.   8.   9.]]
```

Nothing new, if we read in our text, in which we used a smiley to separator:

```
y = np.loadtxt("test3.txt", delimiter=" :-) ")
print(y)
```

```
[[ 1.   2.   3.]
 [ 4.   5.   6.]
 [ 7.   8.   9.]]
```

It's also possible to choose the columns by index:

```
y = np.loadtxt("test3.txt", delimiter=" :-) ", usecols=(0,2))
print(y)
```

```
[[ 1.   3.]
 [ 4.   6.]
 [ 7.   9.]]
```

We will read in our next example the file "times_and_temperatures.txt", which we have created in our chapter on Generators of our Python tutorial. Every line contains a time in the format "hh::mm::ss" and random temperatures between 10.0 and 25.0 degrees. We have to convert the time string into float numbers. The time will be in minutes with seconds in the hundred. We define first a function which converts "hh::mm::ss" into minutes:

```
def time2float_minutes(time):
    if type(time) == bytes:
        time = time.decode()
    t = time.split(":")
    minutes = float(t[0])*60 + float(t[1]) + float(t[2]) * 0.05 /
3
    return minutes

for t in ["06:00:10", "06:27:45", "12:59:59"]:
    print(time2float_minutes(t))
```

```
360.1666666666667
387.75
779.9833333333333
```

You might have noticed that we check the type of time for binary. The reason for this is the use of our function "time2float_minutes in loadtxt in the following example. The keyword parameter converters contains a dictionary which can hold a function for a column (the key of the column corresponds to the key of the dictionary) to convert the string data of this column into a float. The string data is a byte string. That is why we had to transfer it into a a unicode string in our function:

```
y = np.loadtxt("times_and_temperatures.txt",
               converters={ 0: time2float_minutes})
print(y)
```

```
[[  360.      20.1]
 [  361.5     16.1]
 [  363.      16.9]
 ...,
 [ 1375.5     22.5]
 [ 1377.      11.1]
 [ 1378.5     15.2]]
```

```
# delimiter = ";" , # i.e. use ";" as delimiter instead of whitesp
ace
```

TOFILE

tofile is a function to write the content of an array to a file both in binary, which is the default, and text format.

`A.tofile(fid, sep="", format="%s")`

The data of the A ndarry is always written in 'C' order, regardless of the order of A.

The data file written by this method can be reloaded with the function `fromfile()`.

Parameter	Meaning
fid	can be either an open file object, or a string containing a filename.
sep	The string 'sep' defines the separator between array items for text output. If it is empty (''), a binary file is written, equivalent to file.write(a.tostring()).
format	Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using 'format' % item.

Remark:

Information on endianness and precision is lost. Therefore it may not be a good idea to use the function to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

```
dt = np.dtype([('time', [('min', int), ('sec', int)]),
               ('temp', float)])
x = np.zeros((1,), dtype=dt)
x['time']['min'] = 10
x['temp'] = 98.25
print(x)

fh = open("test6.txt", "bw")
x.tofile(fh)
```

```
[((10, 0), 98.25)]
```

FROMFILE

`fromfile` to read in data, which has been written with the `tofile` function. It's possible to read binary data, if the data type is known. It's also possible to parse simply formatted text files. The data from the file is turned into an array.

The general syntax looks like this:

```
numpy.fromfile(file, dtype=float, count=-1, sep='')
```

Parameter	Meaning
file	'file' can be either a file object or the name of the file to read.
dtype	defines the data type of the array, which will be constructed from the file data. For binary files, it is used to determine the size and byte-order of the items in the file.
count	defines the number of items, which will be read. -1 means all items will be read.
sep	The string 'sep' defines the separator between the items, if the file is a text file. If it is empty (''), the file will be treated as a binary file. A space (" ") in a separator matches zero or more whitespace characters. A separator consisting solely of spaces has to match at least one whitespace.

```
fh = open("test4.txt", "rb")

np.fromfile(fh, dtype=dt)
```

Output:
```
array([((4294967296, 12884901890), 1.0609978957e-313),
       ((30064771078, 38654705672), 2.33419537056e-313),
       ((55834574860, 64424509454), 3.60739284543e-313),
       ((81604378642, 90194313236), 4.8805903203e-313),
       ((107374182424, 115964117018), 6.1537877952e-313),
       ((133143986206, 141733920800), 7.42698527006e-313),
       ((158913789988, 167503724582), 8.70018274493e-313),
       ((184683593770, 193273528364), 9.9733802198e-313)],
      dtype=[('time', [('min', '<i8'), ('sec', '<i8')]), ('te
mp', '<f8')])
```

```
import numpy as np
import os

# platform dependent: difference between Linux and Windows
#data = np.arange(50, dtype=np.int)

data = np.arange(50, dtype=np.int32)
data.tofile("test4.txt")

fh = open("test4.txt", "rb")
# 4 * 32 = 128
fh.seek(128, os.SEEK_SET)
```

```
x = np.fromfile(fh, dtype=np.int32)
print(x)
```

```
[32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
```

Attention:

It can cause problems to use `tofile` and `fromfile` for data storage, because the binary files generated are not platform independent. There is no byte-order or data-type information saved by `tofile`. Data can be stored in the platform independent .npy format using save and load instead.

BEST PRACTICE TO LOAD AND SAVE DATA

The recommended way to store and load data with Numpy in Python consists in using load and save. We also use a temporary file in the following :

```
import numpy as np

print(x)

from tempfile import TemporaryFile

outfile = TemporaryFile()

x = np.arange(10)
np.save(outfile, x)

outfile.seek(0) # Only needed here to simulate closing & reopenin
g file
np.load(outfile)
```

```
[32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49]
```

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

AND YET ANOTHER WAY: GENFROMTXT

There is yet another way to read tabular input from file to create arrays. As the name implies, the input file is supposed to be a text file. The text file can be in the form of an archive file as well. genfromtxt can process the archive formats gzip and bzip2. The type of the archive is determined by the extension of the file, i.e. '.gz' for gzip and bz2' for an bzip2.

genfromtxt is slower than loadtxt, but it is capable of coping with missing data. It processes the file data in two passes. At first it converts the lines of the file into strings. Thereupon it converts the strings into the requested data type. loadtxt on the other hand works in one go, which is the reason, why it is faster.

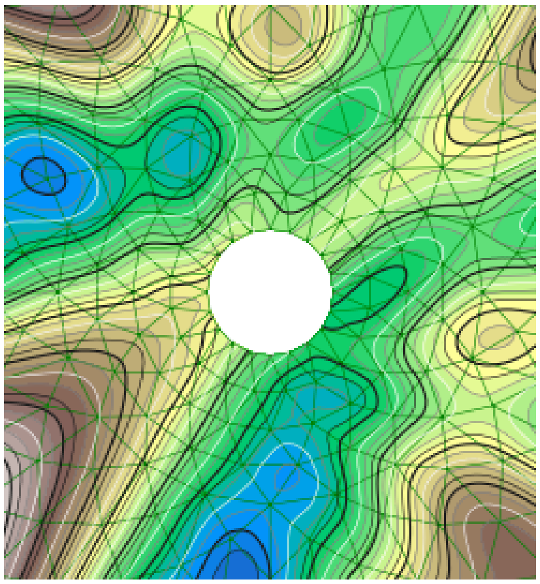
RECFROMCSV(FNAME, **KWARGS)

This is not really another way to read in csv data. 'recfromcsv' basically a shortcut for

np.genfromtxt(filename, delimiter=",", dtype=None)

OVERVIEW OF MATPLOTLIB

INTRODUCTION



Matplotlib is a plotting library like GNUplot. The main advantage towards GNUplot is the fact that Matplotlib is a Python module. Due to the growing interest in python the popularity of matplotlib is continually rising as well.

Another reason for the attractiveness of Matplotlib lies in the fact that it is widely considered to be a perfect alternative to MATLAB, if it is used in combination with Numpy and Scipy. Whereas MATLAB is expensive and closed source, Matplotlib is free and open source code. It is also object-oriented and can be used in an object oriented way. Furthermore it can be used with general-purpose GUI toolkits like wxPython, Qt, and GTK+. There is also a procedural "pylab", which designed to closely resemble that of MATLAB. This can make it extremely easy for MATLAB users to migrate to matplotlib.

Matplotlib can be used to create publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

Another characteristic of matplotlib is its steep learning curve, which means that users usually make rapid progress after having started. The officicial website has to say the following about this: "matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code."

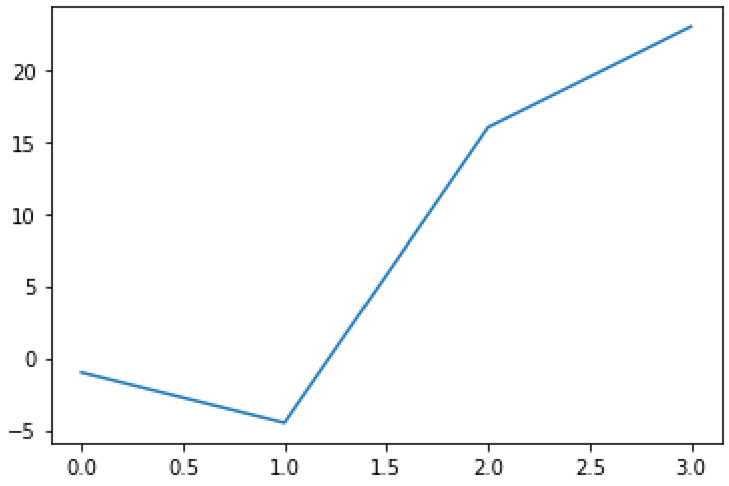
A FIRST EXAMPLE: LINE PLOT

We will start with a simple graph , which is as simple as simple can be. A graph in matplotlib is a two- or three-dimensional drawing showing a relationship by means of points, a curve, or amongst others a series of bars. We have two axis: The horizontal X-axis is representing the independent values and the vertical Y-axis corresponds to the depended values.

We will use the pyplot submodule of matplotlib. pyplot provides a procedural interface to the object-oriented plotting library of matplotlib.
Its plotting commands are chosen in a way that they are similar to Matlab both in naming and with the arguments.

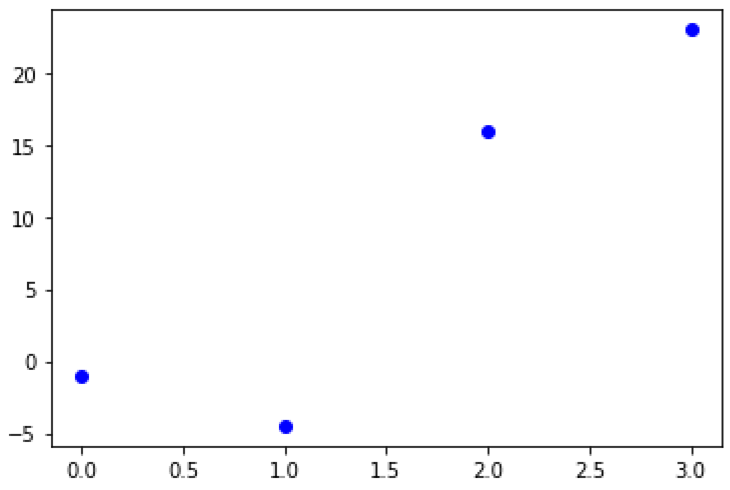
Is is common practice to rename matplotlib.pyplot to plt. We will use the plot function of pyplot in our first example. We will pass a list of values to the plot function. Plot takes these as Y values. The indices of the list are automatically taken as the X values. The command `%matplotlib inline` makes only sense, if you work with Ipython Notebook. It makes sure, that the graphs will be depicted inside of the document and not as independent windows:

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23])
plt.show()
```



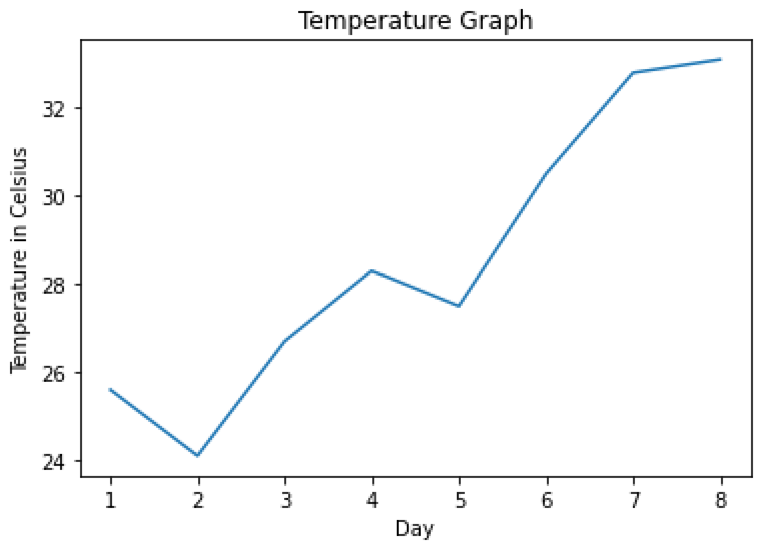
What we see is a continuous graph, even though we provided discrete data for the Y values. By adding a format string to the function call of plot, we can create a graph with discrete values, in our case blue circle markers. The format string defines the way how the discrete points have to be rendered.

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23], "ob")
plt.show()
```



With only a little bit of input we were capable in the previous examples to create plots. Yet, the plots missed some information. Usually, we would like to give a name to the 'x' and 'y' values. Furthermore, the whole graph should have a title. We demonstate this in the following example:

```python
import matplotlib.pyplot as plt

days = range(1, 9)
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

fig, ax = plt.subplots()
ax.plot(days, celsius_values)
ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')

plt.show()
```

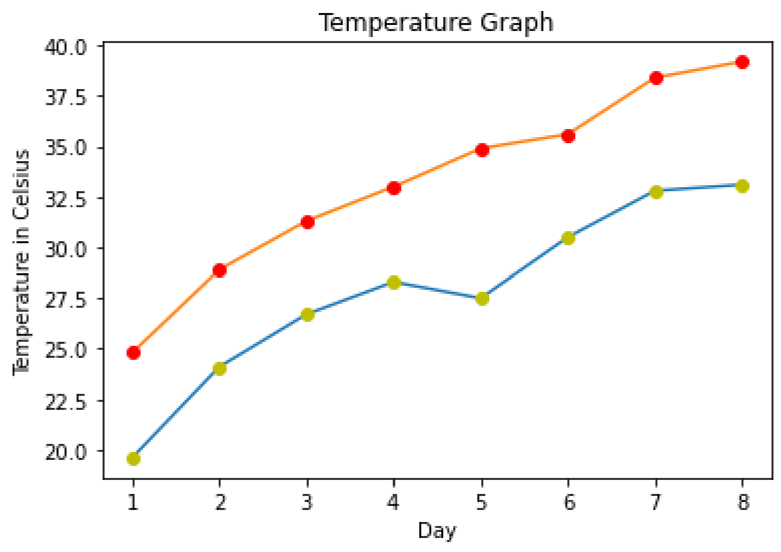


MULTIPLE PLOTS IN ONE GRAPH

We saw that the `plot` function is needed to plot a figure or better the figures, because there may be more than one.

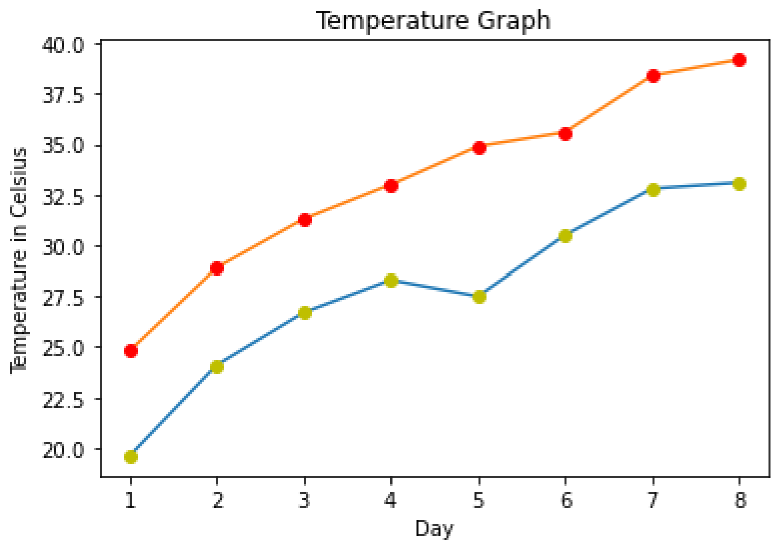
We can specify an arbitrary number of x, y, fmt groups in a one plot function. We will extend our previous temperature example to demonstrate this. We provide two lists with temperature values, one for the minimum and one for the maximum values:

```python
import matplotlib.pyplot as plt

days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]

fig, ax = plt.subplots()

ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')


ax.plot(days, celsius_min,
        days, celsius_min, "oy",
        days, celsius_max,
        days, celsius_max, "or")

plt.show()
```



We could have used four plot calls in the previous code instead of one, even though this is not very attractive:

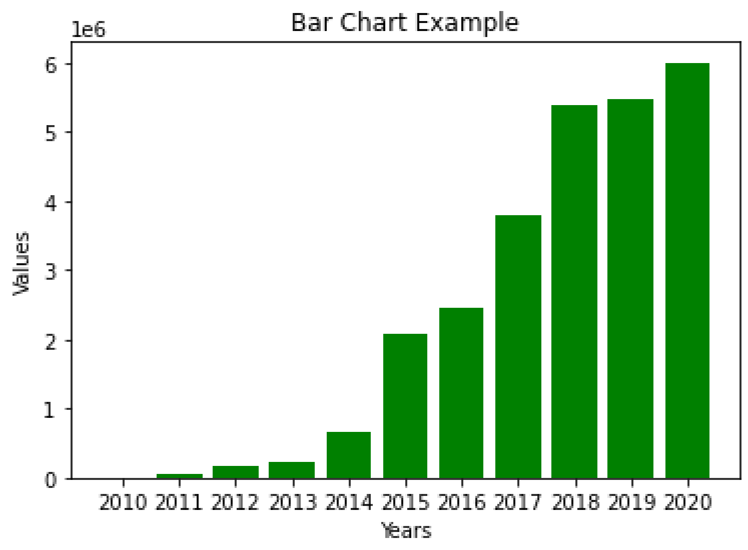
```
import matplotlib.pyplot as plt

days = list(range(1, 9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]

fig, ax = plt.subplots()

ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')

ax.plot(days, celsius_min)
ax.plot(days, celsius_min, "oy")
ax.plot(days, celsius_max)
ax.plot(days, celsius_max, "or")

plt.show()
```



EXAMPLE: BAR PLOTS

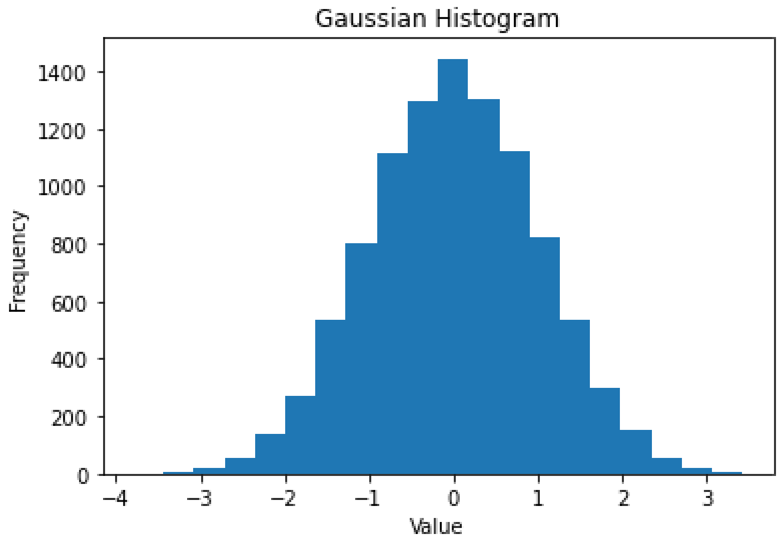
A more formal introduction will follow later in our tutorial on Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np
years = [str(year) for year in range(2010, 2021)]
visitors = (1241, 50927, 162242, 222093,
            665004, 2071987, 2460407, 3799215,
            5399000, 5474016, 6003672)


plt.bar(years, visitors, color="green")

plt.xlabel("Years")
plt.ylabel("Values")
plt.title("Bar Chart Example")

plt.plot()
plt.show()
```



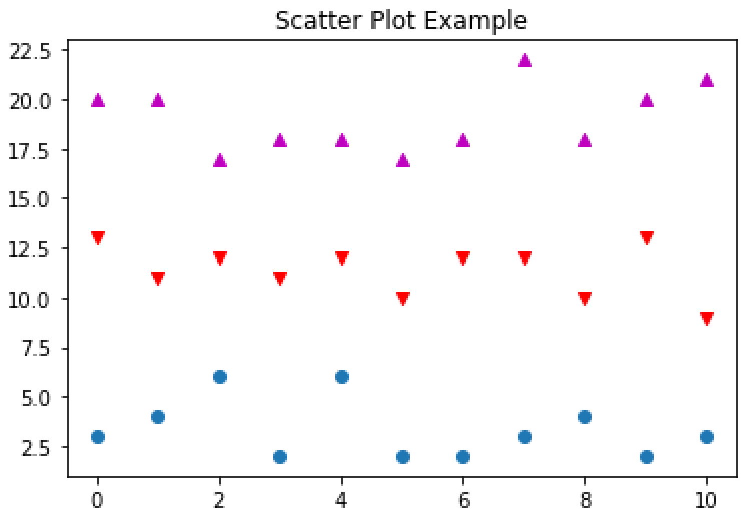
HISTOGRAMS

```
import matplotlib.pyplot as plt
import numpy as np

gaussian_numbers = np.random.normal(size=10000)
gaussian_numbers



plt.hist(gaussian_numbers, bins=20)
plt.title("Gaussian Histogram")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```



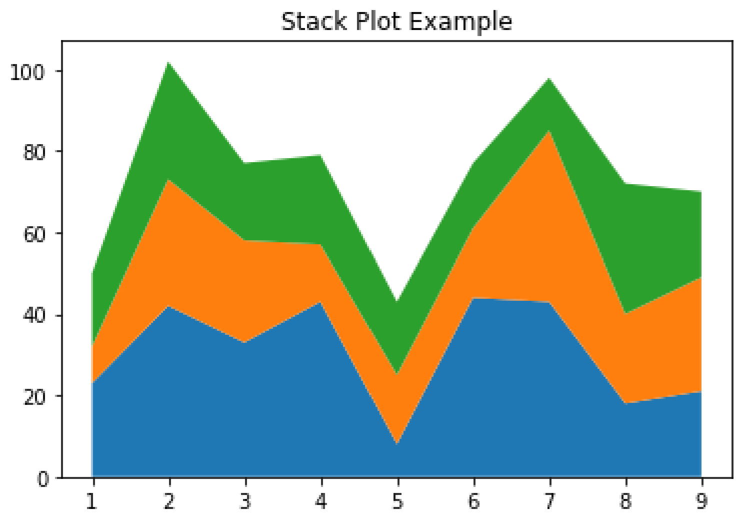
SCATTER PLOTS

```
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0, 11)

y1 = np.random.randint(2, 7, (11,))
y2 = np.random.randint(9, 14, (11,))
y3 = np.random.randint(15, 25, (11,))

# Markers: https://matplotlib.org/api/markers_api.html

plt.scatter(x, y1)
plt.scatter(x, y2, marker='v', color='r')
plt.scatter(x, y3, marker='^', color='m')
plt.title('Scatter Plot Example')
plt.show()
```



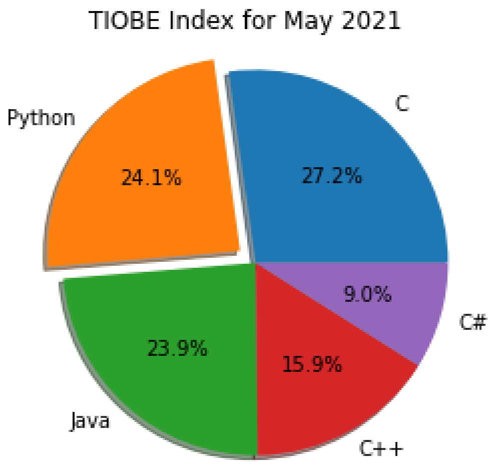
STACK PLOTS

```
import matplotlib.pyplot as plt

idxes = [ 1,  2,  3,  4,  5,  6,  7,  8,  9]
y1  = [23, 42, 33, 43,  8, 44, 43, 18, 21]
y2  = [9, 31, 25, 14, 17, 17, 42, 22, 28]
y3  = [18, 29, 19, 22, 18, 16, 13, 32, 21]


plt.stackplot(idxes,
              y1, y2, y3)
plt.title('Stack Plot Example')

plt.show()
```



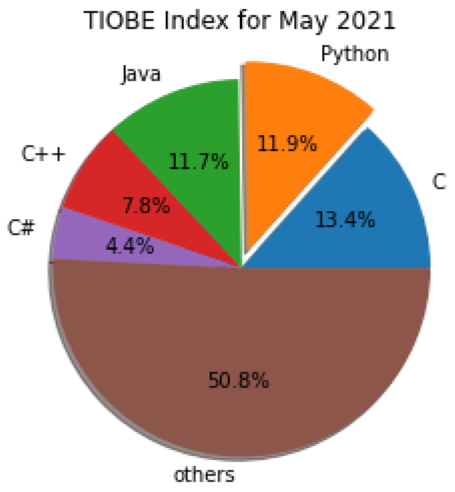
PIE CHARTS

```
import matplotlib.pyplot as plt

# Pie chart, where the slices will be ordered and plotted counter-
clockwise:
labels = 'C', 'Python', 'Java', 'C++', 'C#'
sizes = [13.38, 11.87, 11.74, 7.81, 4.41]
explode = (0, 0.1, 0, 0, 0)  # only "explode" the 2nd slice (i.e.
'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=0)
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn
as a circle.

plt.title('TIOBE Index for May 2021')
plt.show()
```



```
# Pie chart, where the slices will be ordered and plotted counter-
clockwise:
labels = 'C', 'Python', 'Java', 'C++', 'C#', 'others'
sizes = [13.38, 11.87, 11.74, 7.81, 4.41]
sizes.append(100 - sum(sizes))
explode = (0, 0.1, 0, 0, 0, 0)  # only "explode" the 2nd slice
(i.e. 'Hogs')

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=0)
```

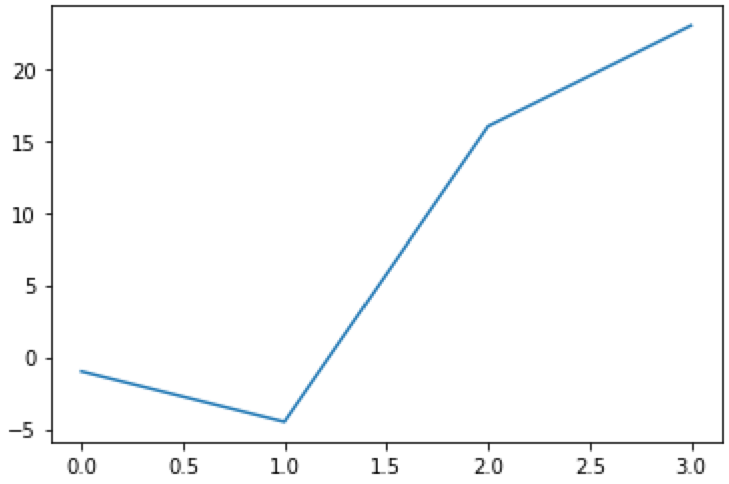
```python
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn
as a circle.

plt.title('TIOBE Index for May 2021')
plt.show()
```



FORMAT PLOTS

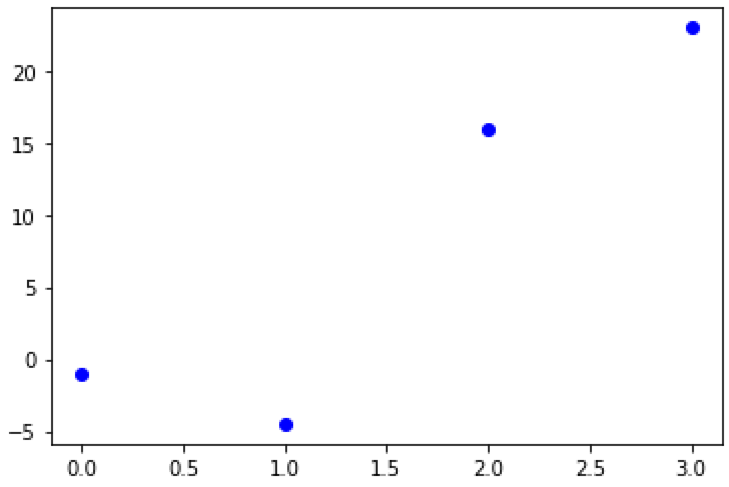
FORMAT PARAMETER

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23])
plt.show()
```



What we see is a continuous graph, even though we provided discrete data for the Y values. By adding a format string to the function call of plot, we can create a graph with discrete values, in our case blue circle markers. The format string defines the way how the discrete points have to be rendered.

```
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23], "ob")
plt.show()
```



THE FORMAT PARAMETER OF PYPLOT.PLOT

We have used "ob" in our previous example as the format parameter. It consists of two characters. The first one defines the line style or the dicrete value style, i.e. the markers, while the second one chooses a colour for the graph. The order of the two characters could have been reversed, i.e. we could have written it as "bo" as well. If the format parameter is not given, as in the first example, the default value is "b-", i.e. a solid blue line.

The following format string characters are accepted to control the line style or marker:

```
=============================================
character       description
=============================================
'-'             solid line style
'--'            dashed line style
'-.'            dash-dot line style
':'             dotted line style
'.'             point marker
','             pixel marker
'o'             circle marker
'v'             triangle_down marker
'^'             triangle_up marker
'<'             triangle_left marker
'>'             triangle_right marker
'1'             tri_down marker
'2'             tri_up marker
'3'             tri_left marker
'4'             tri_right marker
's'             square marker
'p'             pentagon marker
```

```
'*'             star marker
'h'             hexagon1 marker
'H'             hexagon2 marker
'+'             plus marker
'x'             x marker
'D'             diamond marker
'd'             thin_diamond marker
'|'             vline marker
'_'             hline marker
===============================================
```

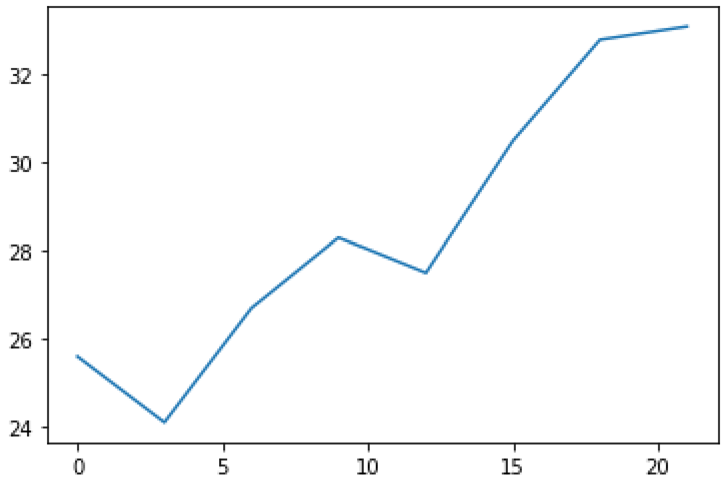
The following color abbreviations are supported:

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

As you may have guessed already, you can add X values to the plot function. We will use the multiples of 3 starting at 3 below 22 as the X values of the plot in the following example:

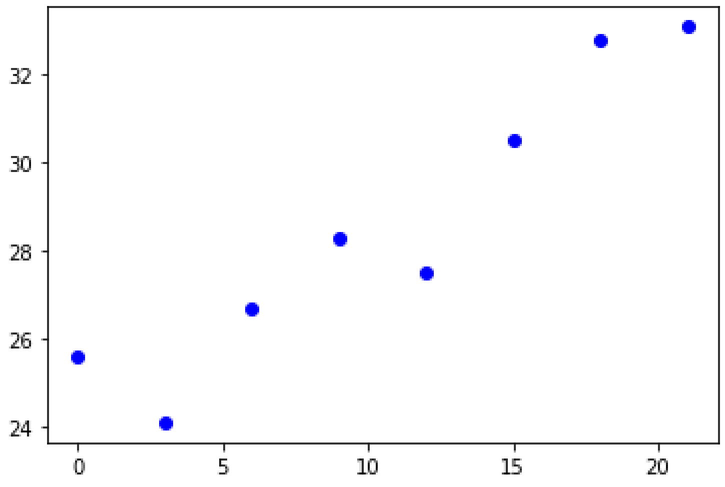
```
import matplotlib.pyplot as plt

# our X values:
days = list(range(0, 22, 3))
print("Values of days:", days)
# our Y values:
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

plt.plot(days, celsius_values)
plt.show()
```

```
Values of days: [0, 3, 6, 9, 12, 15, 18, 21]
```



... and once more with discrete values:

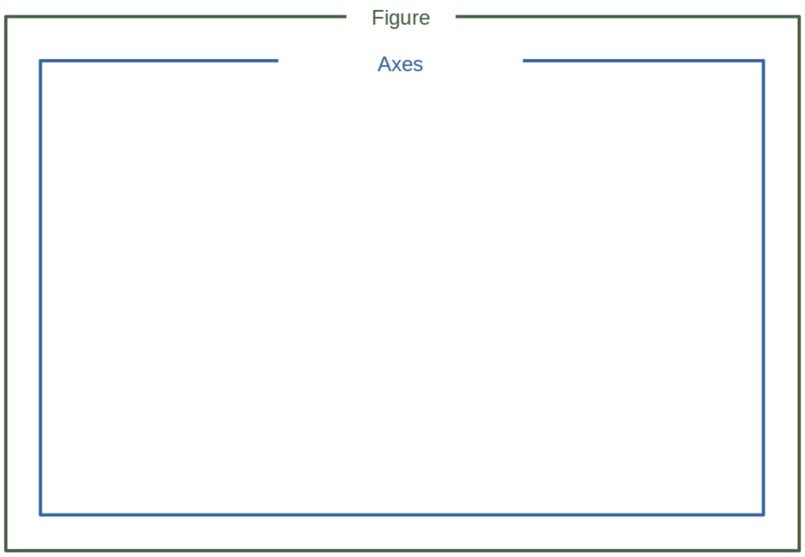
```
plt.plot(days, celsius_values, 'bo')
plt.show()
```



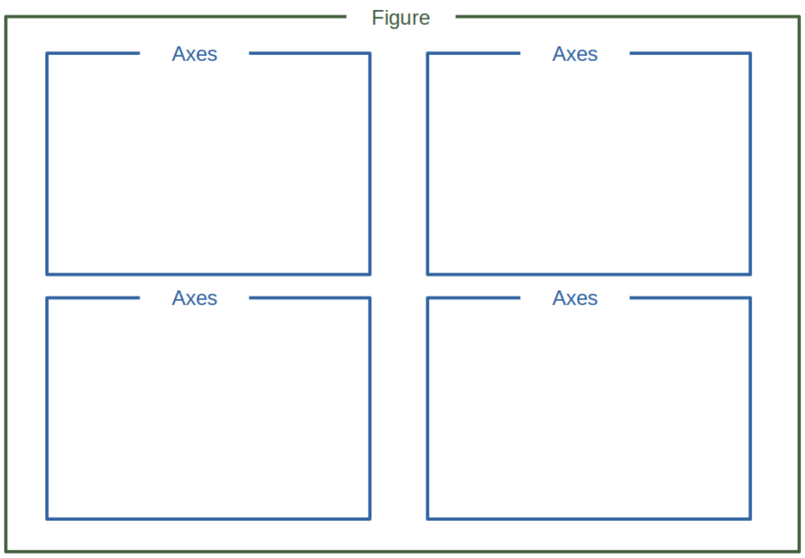
MATPLOTLIB OBJECT HIERARCHY AND IMPORTANT TERMS

Matplotlib is hierarchically organized. In the previous chapter, we didn't made use of this structure. All we used was a simple plot like `plt.plot([2, 5, 9])` which is implicitly building the necessary structures. We didn't have to know about the underlying hierarchical structure. in this chapter of our tutorial we will explain the underlying object hierarchy. We will also demonstrate how to use it to plot graphs and work with the created axes.

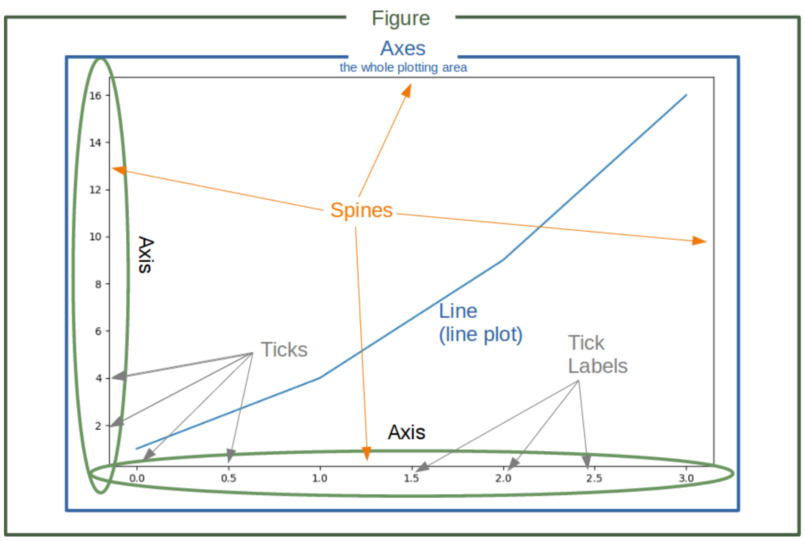
A matplotlib object is a tree-like structure of matplotlib objects which build a "hierarchy". The top of the tree-like structure of matplotlib objects is the `figure` object. A figure can be seen as the container which contains one or more plots. The plots are called `axes` in matplotlib jargon. Axes is the plural of the word "axis", which gives us a misleading idea. We can think about "axes" as a synonym for the word "plot". The following diagram shows a figure with one axes:



As we have already mentioned a figure may also contain more than one axes:



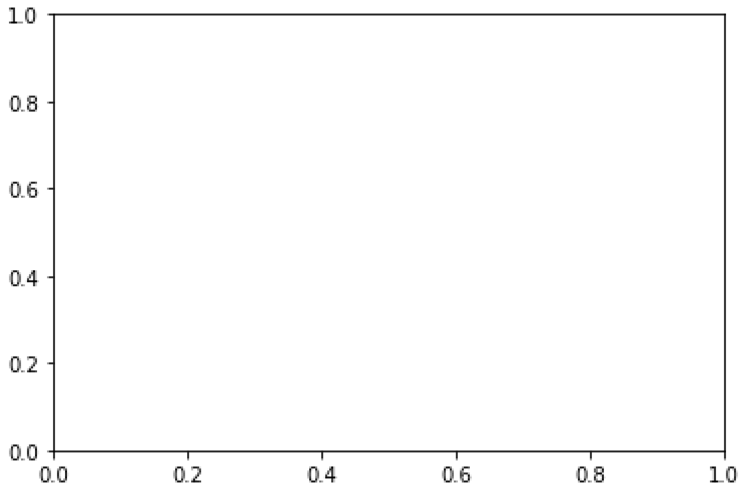
The terms `axes` and `figure` and in particular the relationship between them can be quite confusing to beginners in Matplotlib. Similarly difficult to access and understand are many of the terms `Spine`, `Tick` and `Axis`. Their function and meaning are easy to crasp, if you see look at them in the following diagram:



GENERATE FIGURE AND AXES

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

print(type(fig))
print(type(ax))
```

```
<class 'matplotlib.figure.Figure'>
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

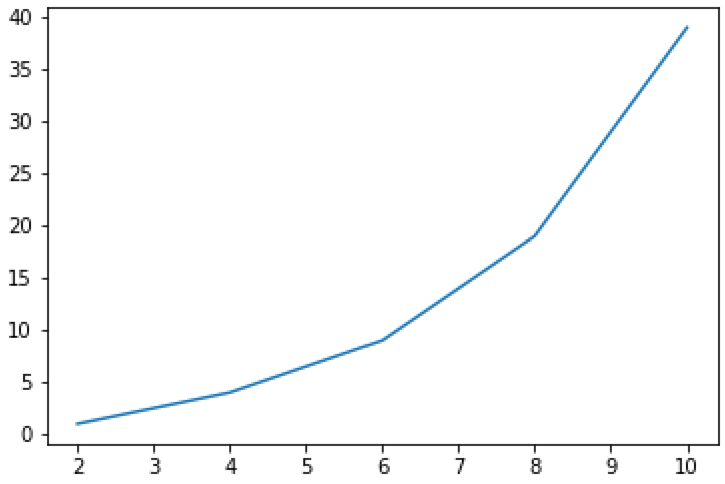


The function `subplots` can be used to create a figure and a set of subplots. In our previous example, we called the function without parameters which creates a figure with a single included axes. we will see later how to create more than one axes with `subplots`.

We will demonstrate in the following example, how we can go on with these objects to create a plot. We can see that we apply `plot` directly on the axis object `ax`. This leaves no possible ambiguity as in the `plt.plot` approach:

```python
import matplotlib.pyplot as plt

# Data for plotting
X = [2, 4, 6, 8, 10]
Y = [1, 4, 9, 19, 39]

fig, ax = plt.subplots()
ax.plot(X, Y)
```

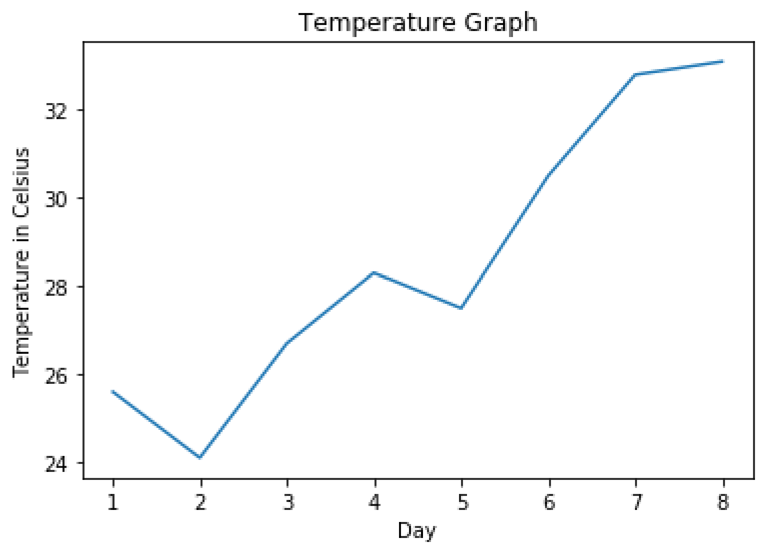
Output:
```
[<matplotlib.lines.Line2D at 0x7f5dbe7b6dd8>]
```



LABELS ON AXES

We can improve the appearance of our graph by adding labels to the axes. We also want to give our plot a headline or let us call it a `title` to stay in the terminology of Matplotlib. We can accomplish this by using the `set` method of the axis object `ax` :

```
import matplotlib.pyplot as plt

days = list(range(1,9))
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

fig, ax = plt.subplots()
ax.plot(days, celsius_values)
ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')

plt.show()
```



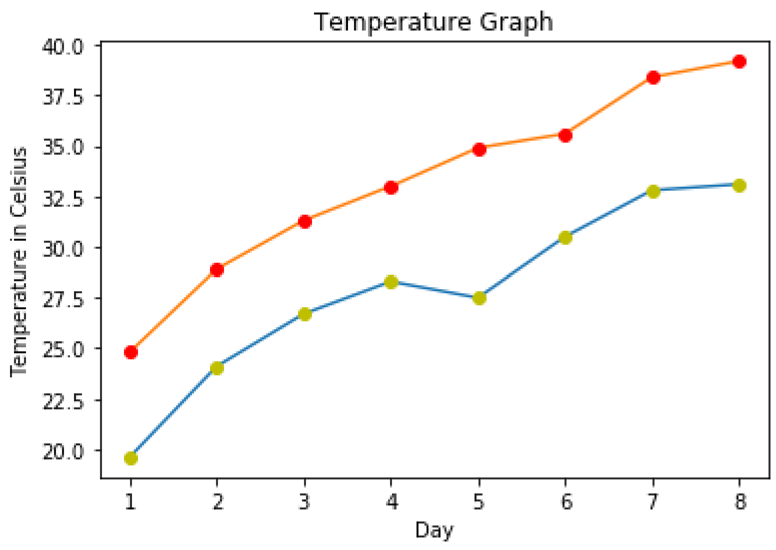
THE PLOT FUNCTION

The `plot` function is needed to plot a figure or better the figures, because there may be more than one. When plot is run in ipython with its pylab mode, it displays all figures and returns to the ipython prompt. When we run it in non-interactive mode, - which is the case, when we run a Python program - it displays all figures and blocks until the figures have been closed.

We can specify an arbitrary number of x, y, fmt groups in a plot function. We will extend our previous temperature example to demonstrate this. We provide two lists with temperature values, one for the minimum and one for the maximum values:

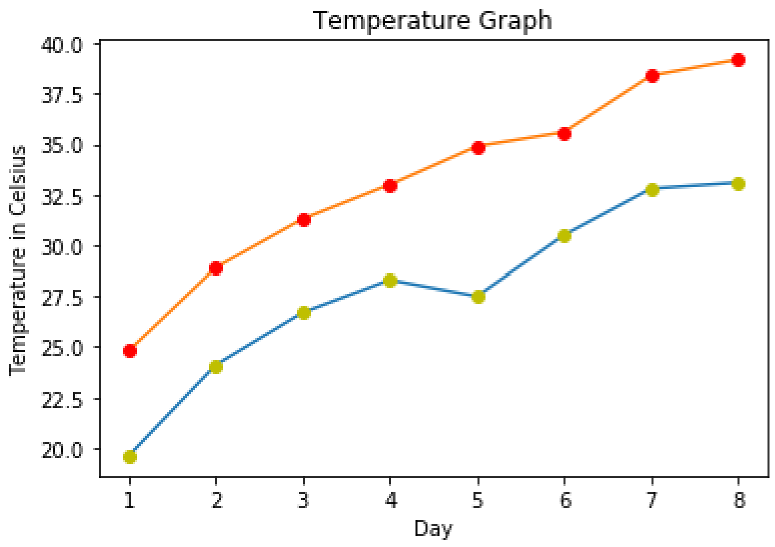
```
import matplotlib.pyplot as plt

days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]

fig, ax = plt.subplots()

ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')


ax.plot(days, celsius_min,
        days, celsius_min, "oy",
        days, celsius_max,
        days, celsius_max, "or")
```

```
plt.show()
```



We could have used for plot calls in the previous code instead of one, even though this is not very attractive:

```
import matplotlib.pyplot as plt

days = list(range(1, 9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]

fig, ax = plt.subplots()

ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')

ax.plot(days, celsius_min)
ax.plot(days, celsius_min, "oy")
ax.plot(days, celsius_max)
ax.plot(days, celsius_max, "or")

plt.show()
```

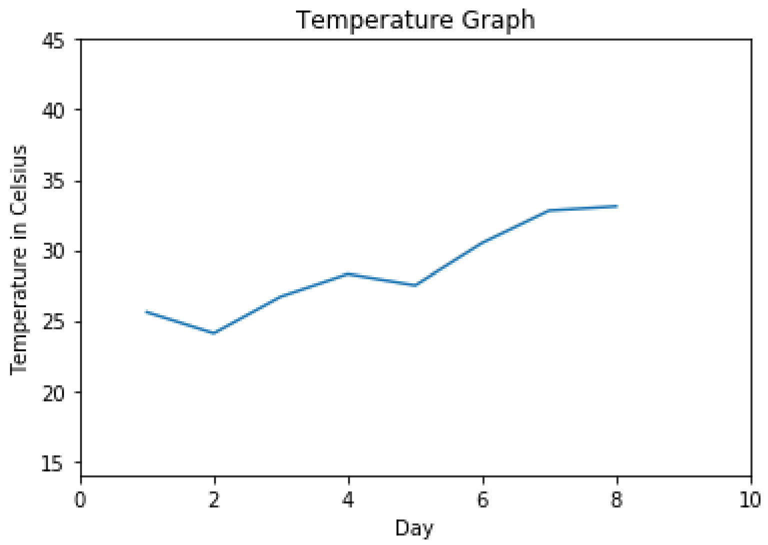


CHECKING AND DEFINING THE RANGE OF AXES

We can also check and define the range of the axes with the function axis. If you call it without arguments it returns the current axis limits:

```
import matplotlib.pyplot as plt

days = list(range(1, 9))
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]

fig, ax = plt.subplots()
ax.plot(days, celsius_values)
ax.set(xlabel='Day',
       ylabel='Temperature in Celsius',
       title='Temperature Graph')

print("The current limits for the axes are:")
print(ax.axis())
print("We set the axes to the following values:")
xmin, xmax, ymin, ymax = 0, 10, 14, 45
print(xmin, xmax, ymin, ymax)
ax.axis([xmin, xmax, ymin, ymax])
plt.show()
```

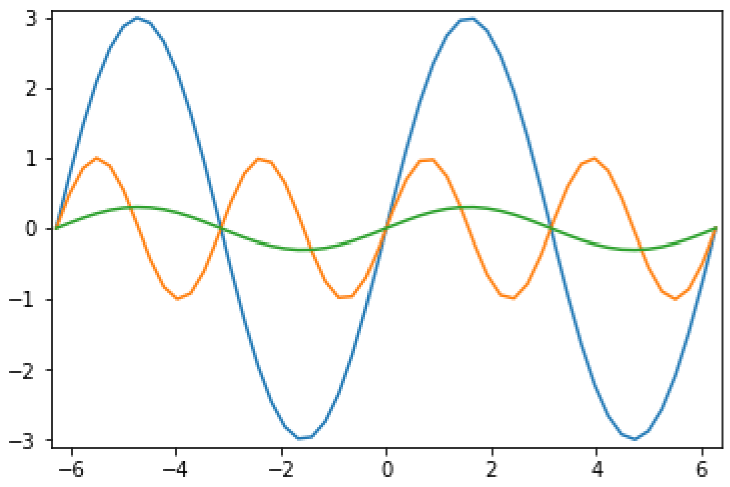
```
The current limits for the axes are:
(0.6499999999999999, 8.35, 23.650000000000002, 33.550000000000004)
We set the axes to the following values:
0 10 14 45
```



"LINSPACE" TO DEFINE X VALUES

We will use the Numpy function linspace in the following example. linspace can be used to create evenly spaced numbers over a specified interval.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)

fig, ax = plt.subplots()

startx, endx = -2 * np.pi - 0.1, 2*np.pi + 0.1
starty, endy = -3.1, 3.1
ax.axis([startx, endx, starty, endy])

ax.plot(X, F1, X, F2, X, F3)
# discrete points:
ax.plot(X, F1, 'ro', X, F2, 'bx')
plt.show()
```

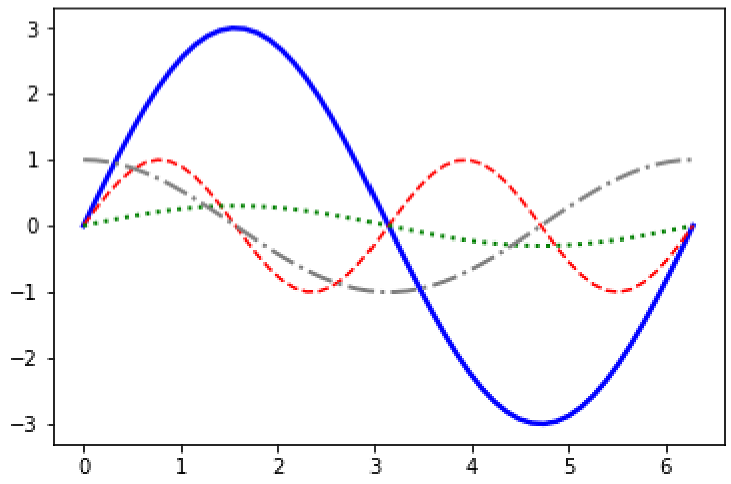


CHANGING THE LINE STYLE

The linestyle of a plot can be influenced with the linestyle or ls parameter of the plot function. It can be set to one of the following values:
`'-', '--', '-.', ':', 'None', ' ', ''`

We can use linewidth to set the width of a line as the name implies.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)
F4 = np.cos(X)

fig, ax = plt.subplots()
ax.plot(X, F1, color="blue", linewidth=2.5, linestyle="-")
ax.plot(X, F2, color="red", linewidth=1.5, linestyle="--")
ax.plot(X, F3, color="green", linewidth=2, linestyle=":")
ax.plot(X, F4, color="grey", linewidth=2, linestyle="-.")
plt.show()
```

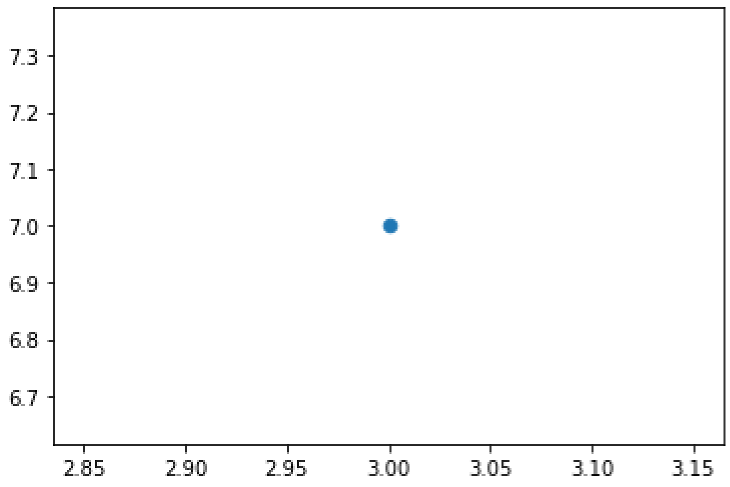


SCATTER PLOTS IN MATPLOTLIB

We will learn now how to draw single points in Matplotlib.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.scatter(3, 7, s=42)
```

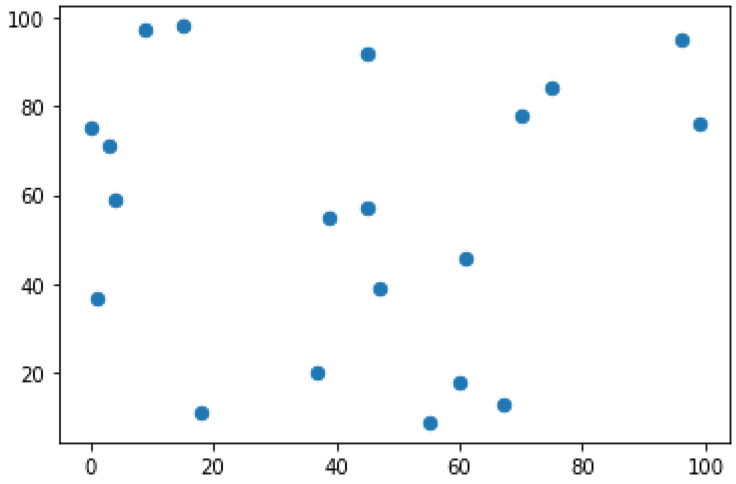
Output:
```
<matplotlib.collections.PathCollection at 0x7fd1ee805e20>
```



```
import matplotlib.pyplot as plt
import numpy as np

X = np.random.randint(0, 100, (20,))
Y = np.random.randint(0, 100, (20,))
```

```
fig, ax = plt.subplots()
ax.scatter(X, Y, s=42)
```

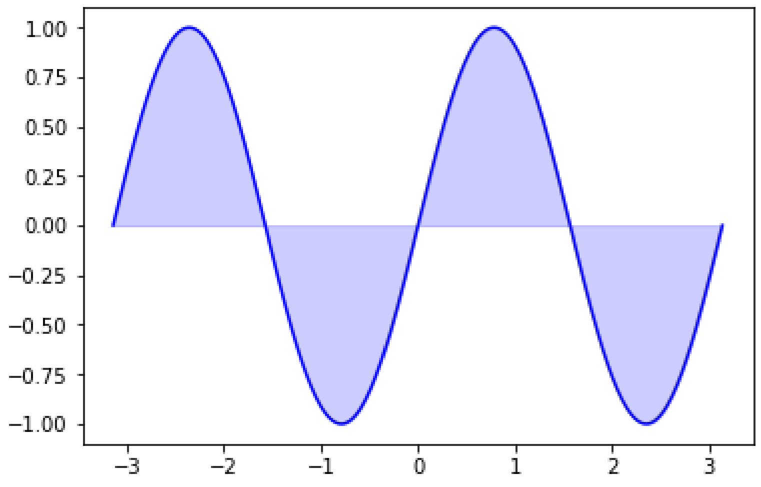
Output:
```
<matplotlib.collections.PathCollection at 0x7fd1edfa16d0>
```



SHADING REGIONS WITH FILL_BETWEEN()

It is possible to shade or colorize regions between two curves. We are filling the region between the X axis and the graph of `sin(2*X)` in the following example:

```
import numpy as np
import matplotlib.pyplot as plt
n = 256
X = np.linspace(-np.pi,np.pi,n,endpoint=True)
Y = np.sin(2*X)

fig, ax = plt.subplots()
ax.plot (X, Y, color='blue', alpha=1.0)
ax.fill_between(X, 0, Y, color='blue', alpha=.2)
plt.show()
```



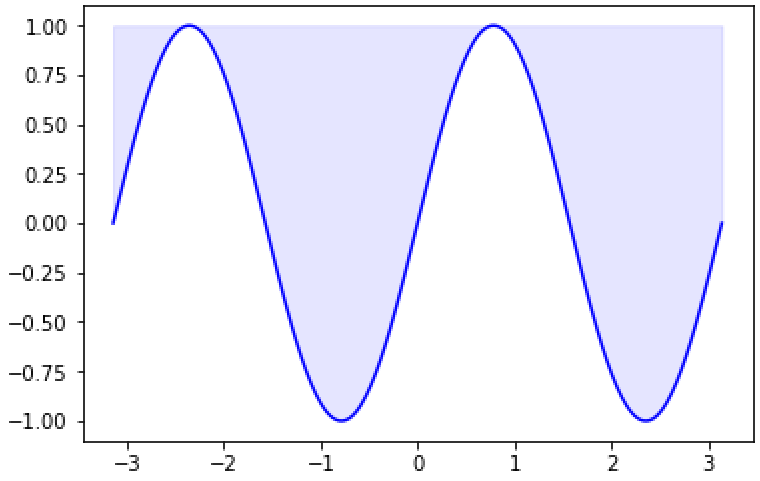
The general syntax of fill_between:

```
fill_between(x, y1, y2=0, where=None, interpolate=False, **kwargs)
```

The parameters of fill_between:

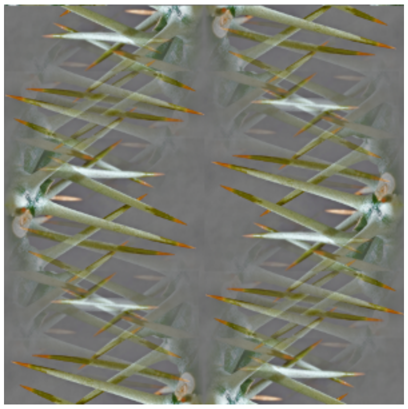
Parameter	Meaning
x	An N-length array of the x data
y1	An N-length array (or scalar) of the y data
y2	An N-length array (or scalar) of the y data
where	If None, default to fill between everywhere. If not None, it is an N-length numpy boolean array and the fill will only happen over the regions where where==True.
interpolate	If True, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the x array.
kwargs	Keyword args passed on to the PolyCollection

```
import numpy as np
import matplotlib.pyplot as plt

n = 256
X = np.linspace(-np.pi,np.pi,n,endpoint=True)
Y = np.sin(2*X)

fig, ax = plt.subplots()

ax.plot (X, Y, color='blue', alpha=1.00)
ax.fill_between(X, Y, 1, color='blue', alpha=.1)
plt.show()
```



MATPLOTLIB TUTORIAL: SPINES AND TICKS

MOVING THE BORDER LINES AND POLISHING UP THE AXES NOTATIONS

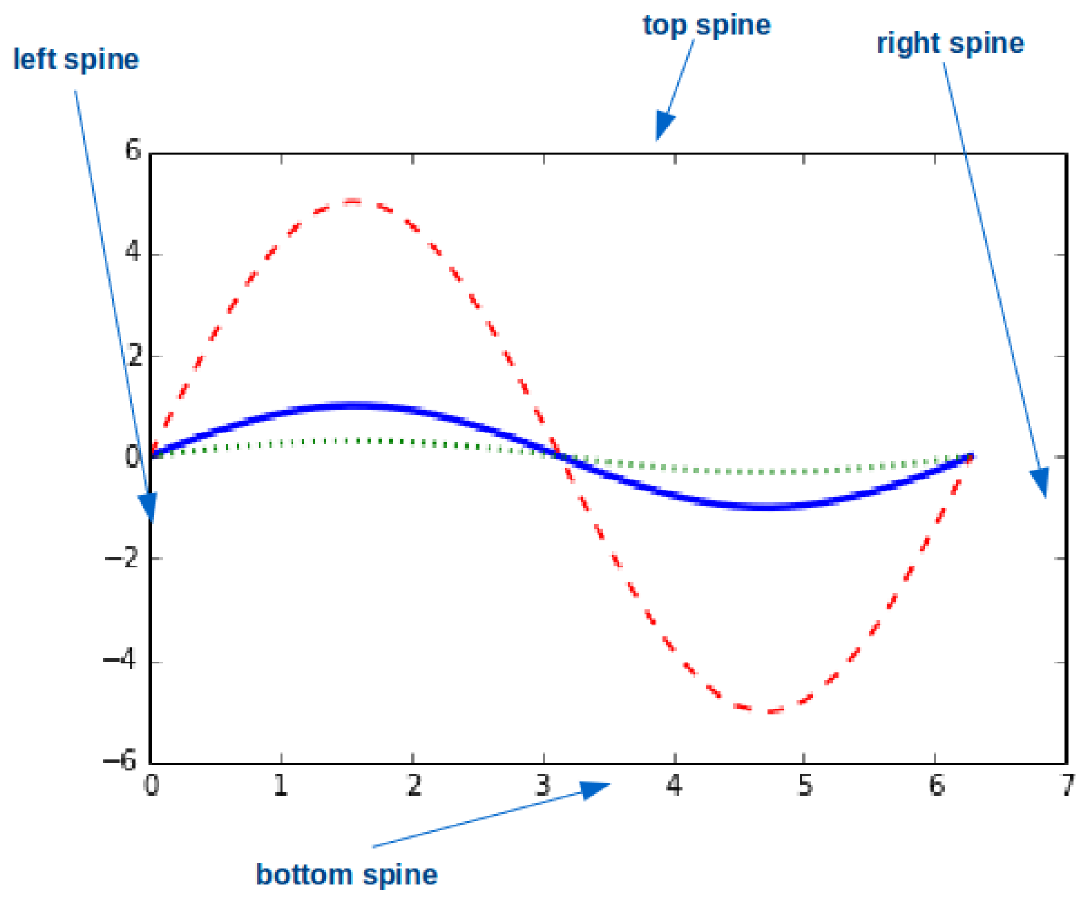


The word spine is most commonly known as the backbone or spinal cord of the human skeleton. Another meaning stands for a book's jacket. Our picture on the right side shows the spines of a cactus, artistically changed into something which looks like a ribcage. You will hardly find the usage of the word `spine` of matplotlib in a dictionary. Spines in matplotlib are the lines connecting the axis tick marks and noting the boundaries of the data area.

We will demonstrate in the following that the spines can be placed at arbitrary positions.

We will move around the spines in the course of this chapter so that the form a 'classical' coordinate syste. One where we have a x axis and a y axis and both go through the origin i.e. the point (0, 0)

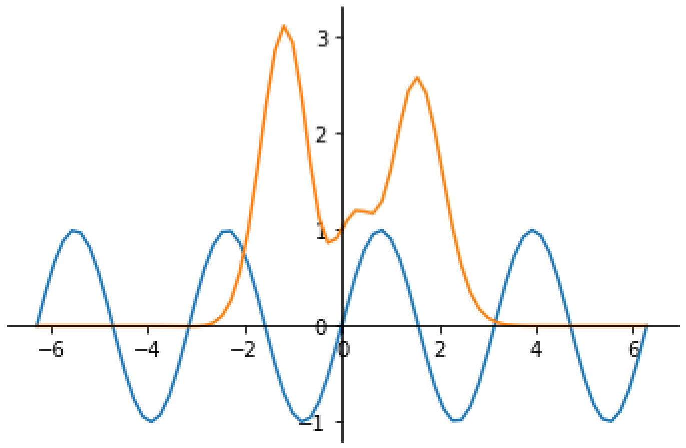
We will show the naming of the spines in the following diagram:



We will move the spines to build a 'classical' coordinate system. To this purpose we turn the top and right spine invisible and move the bottom and left one around:

```
# the next "inline" statement is only needed,
# if you are working with "ipython notebook"
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(2* X)
F2 = (2*X**5 + 4*X**4 - 4.8*X**3 + 1.2*X**2 + X + 1)*np.exp(-X**2)

fig, ax = plt.subplots()

# making the top and right spine invisible:
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
```

```
# moving bottom spine up to y=0 position:
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))

# moving left spine to the right to position x == 0:
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

ax.plot(X, F1, X, F2)

plt.show()
```



CUSTOMIZING TICKS

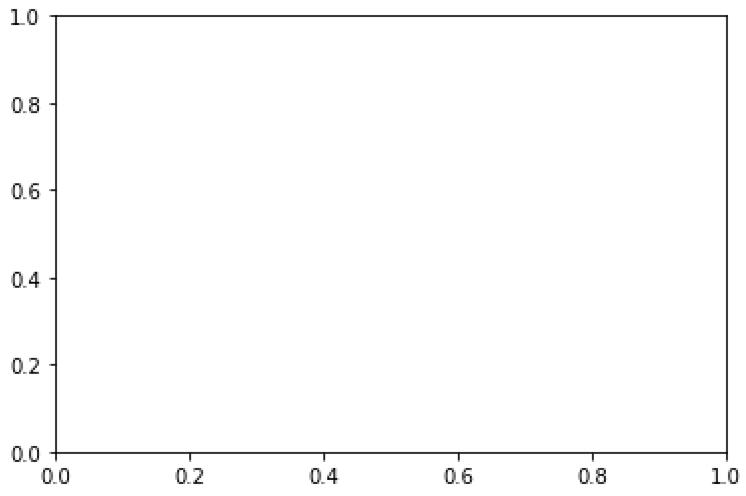
Matplotlib has so far - in all our previous examples - automatically taken over the task of spacing points on the axis. We can see for example that the X axis in our previous example was numbered `-6. -4, -2, 0, 2, 4, 6`, whereas the Y axis was numbered `-1.0,  0, 1.0, 2.0, 3.0`

xticks is a method, which can be used to get or to set the current tick locations and the labels. The same is true for yticks:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

xticks = ax.get_xticks()
xticklabels = ax.get_xticklabels()
print(xticks, xticklabels)
for i in range(6):
    print(xticklabels[i])
```

```
yticks = ax.get_yticks()
print(yticks)
```

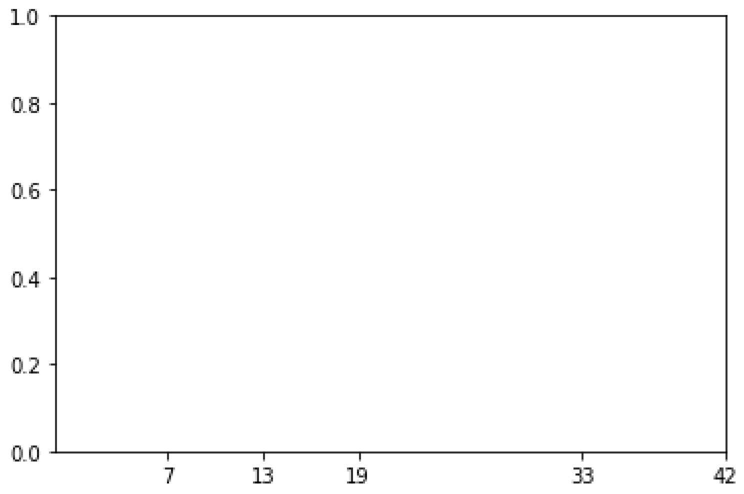
```
[0.  0.2 0.4 0.6 0.8 1. ] <a list of 6 Text xticklabel objects>
Text(0, 0, '')
Text(0, 0, '')
Text(0, 0, '')
Text(0, 0, '')
Text(0, 0, '')
Text(0, 0, '')
[0.  0.2 0.4 0.6 0.8 1. ]
```



As we said before, we can also use xticks to set the location of xticks:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

ax.set_xticks([7, 13, 19, 33, 42])
```

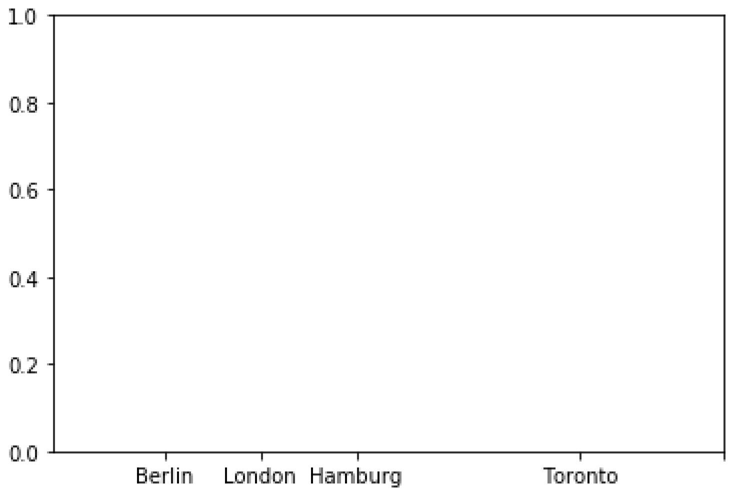
Output:
```
[<matplotlib.axis.XTick at 0x7f8a1e5bd080>,
 <matplotlib.axis.XTick at 0x7f8a1e5bdd68>,
 <matplotlib.axis.XTick at 0x7f8a1e2237b8>,
 <matplotlib.axis.XTick at 0x7f8a1e592a20>,
 <matplotlib.axis.XTick at 0x7f8a1e5921d0>]
```



Now, we will set both the locations and the labels of the xticks:

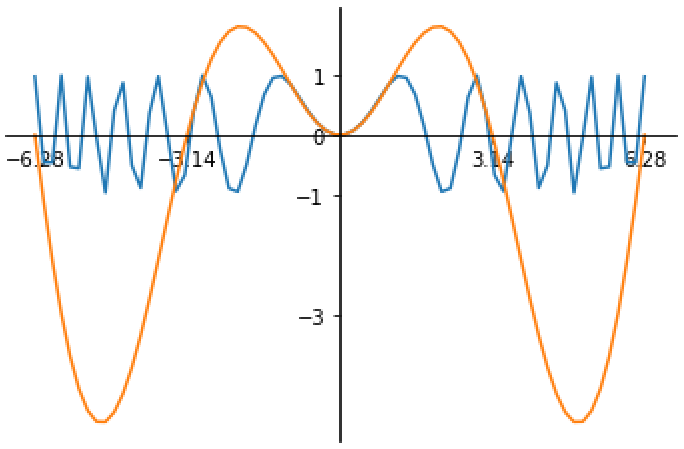
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

ax.set_xticks([7, 13, 19, 33, 42])
ax.set_xticklabels(['Berlin', 'London', 'Hamburg', 'Toronto'])
```

Output:
```
[Text(0, 0, 'Berlin'),
 Text(0, 0, 'London'),
 Text(0, 0, 'Hamburg'),
 Text(0, 0, 'Toronto')]
```



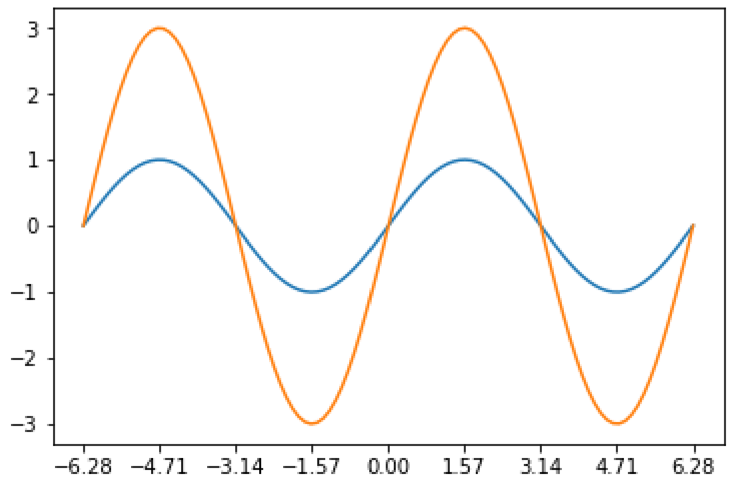
Let's get back to our previous example with the trigonometric functions. Most people might consider factors of Pi to be more appropriate for the X axis than the integer labels:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(X**2)
F2 = X * np.sin(X)

fig, ax = plt.subplots()

# making the top and right spine invisible:
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
# moving bottom spine up to y=0 position:
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
# moving left spine to the right to position x == 0:
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))

ax.set_xticks( [-6.28, -3.14, 3.14, 6.28])
ax.set_yticks([-3, -1, 0, +1, 3])
ax.plot(X, F1)
ax.plot(X, F2)

plt.show()
```



There is an easier way to set the values of the xticks so that we do not have to caculate them manually. We use plt.MultipleLocator with np.pi/2 as argument:

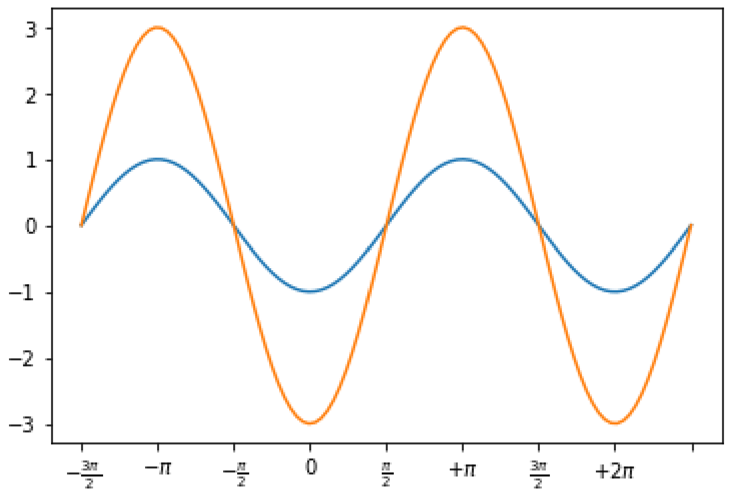
```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 100)
F1 = np.sin(X)
F2 = 3 * np.sin(X)
fig, ax = plt.subplots()

ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))

ax.plot(X, F1, X, F2)

plt.show()
```



SETTING TICK LABELS

We want to rename the xticks now with custom markers. We will use the method xticks again for this purpose as we did in our previous examples. But this time we will call xticks with two parameters: The first one is the same list we used before, i.e. the positions on the x axis, where we want to have the ticks. The second parameter is a list of the same size with corresponding LaTeX tick marks, i.e. the text which we want to see instead of the values. The LaTeX notation has to be a raw string in most cases to suppress the escaping mechanism of Python, because the LaTeX notation heavily uses and relies on the backslash.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 100)
F1 = np.sin(X)
F2 = 3 * np.sin(X)
fig, ax = plt.subplots()

ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
ax.set_xticklabels([r'$-2\pi$', r'$-\frac{3\pi}{2}$', r'$-\pi$',
                    r'$-\frac{\pi}{2}$', 0, r'$\frac{\pi}{2}$',
                    r'$+\pi$', r'$\frac{3\pi}{2}$', r'$+2\pi$'])
ax.plot(X, F1, X, F2)

plt.show()
```



ADJUSTING THE TICKLABELS

We want to increase the legibility of the ticklabels. We will increase the font size, and we will render them on a semi transparant background.

```
print(ax.get_xticklabels())
```

```
<a list of 4 Text xticklabel objects>
```

```
for xtick in ax.get_xticklabels():
    print(xtick)
```

```
Text(-6.28, 0, '$-2\\pi$')
Text(-3.14, 0, '$-\\pi$')
Text(3.14, 0, '$+\\pi$')
Text(6.28, 0, '$+2\\pi$')
```

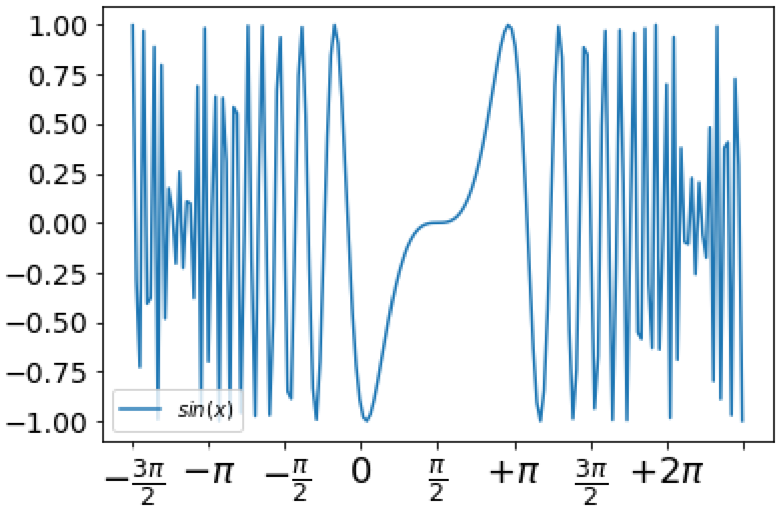
```
labels = [xtick.get_text() for xtick in ax.get_xticklabels()]
print(labels)
```

```
['$-2\\pi$', '$-\\pi$', '$+\\pi$', '$+2\\pi$']
```

Let's increase the fontsize and make the font semi transparant:

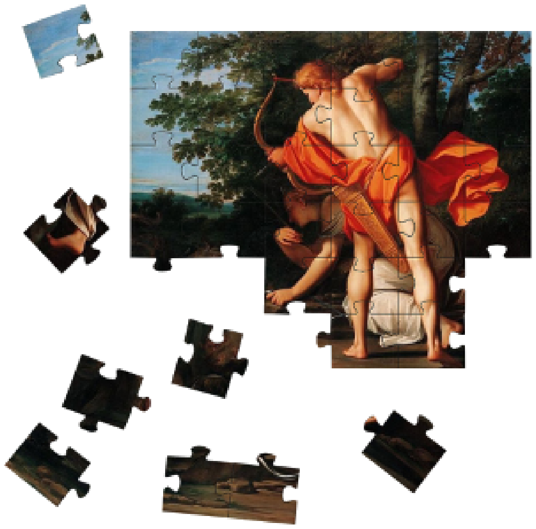
```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 170, endpoint=True)
```

```
F1 = np.sin(X**3 / 2)

fig, ax = plt.subplots()

ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
ax.set_xticklabels([r'$-2\pi$', r'$-\frac{3\pi}{2}$', r'$-\pi$',
                    r'$-\frac{\pi}{2}$', 0, r'$\frac{\pi}{2}$',
                    r'$+\pi$', r'$\frac{3\pi}{2}$', r'$+2\pi$'])


for xtick in ax.get_xticklabels():
    xtick.set_fontsize(18)
    xtick.set_bbox(dict(facecolor='white', edgecolor='None', alph
a=0.7 ))

for ytick in ax.get_yticklabels():
    ytick.set_fontsize(14)
    ytick.set_bbox(dict(facecolor='white', edgecolor='None', alph
a=0.7 ))

ax.plot(X, F1, label="$sin(x)$")

ax.legend(loc='lower left')

plt.show()
```



MATPLOTLIB TUTORIAL, ADDING LEGENDS AND ANNOTATIONS

ADDING A LEGEND



This chapter of our tutorial is about legends. Legends are the classical stories from ancient Greece or other places which are usually devoured by adolescents. They have to read it and this is where the original meaning comes from. The word `legend` stems from Latin and it means in Latin "to be read". So we can say legends are the things in a graph or plot which have to be read to understand the plot. It gives us valuable information about the visualized data.

Before legends have been used in mathematical graphs, they have been used in maps. Legends - as they are found in maps - describe the pictorial language or symbology of the map. Legends are used in line graphs to explain the function or the values underlying the different lines of the graph.

We will demonstrate in the following simple example how we can place a legend on a graph. A legend contains one or more entries. Every entry consists of a key and a label.

The pyplot function

```
legend(*args, **kwargs)
```

places a legend on the axes.

All we have to do to create a legend for lines, which already exist on the axes, is to simply call the function "legend" with an iterable of strings, one for each legend item.

We have mainly used trigonometric functions in our previous chapter. For a change we want to use now polynomials. We will use the Polynomial class which we have defined in our chapter Polynomials.

```python
# the module polynomials can be downloaded from
# https://www.python-course.eu/examples/polynomials.py
from polynomials import Polynomial
import numpy as np
import matplotlib.pyplot as plt

p = Polynomial(-0.8, 2.3, 0.5, 1, 0.2)
p_der = p.derivative()
```

```
fig, ax = plt.subplots()
X = np.linspace(-2, 3, 50, endpoint=True)
F = p(X)
F_derivative = p_der(X)
ax.plot(X, F)
ax.plot(X, F_derivative)

ax.legend(['p', 'derivation of p'])
```

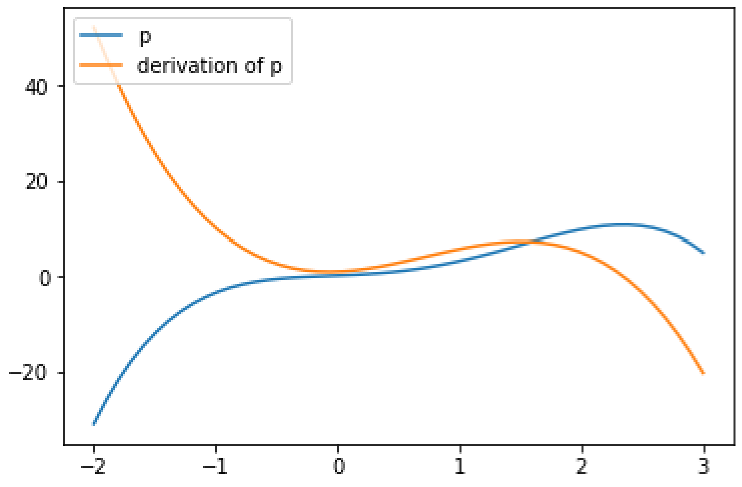
Output:
```
<matplotlib.legend.Legend at 0x7fd2efcad908>
```

If we add a label to the plot function, the value will be used as the label in the legend command. There is anolther argument that we can add to the legend function: We can define the location of the legend inside of the axes plot with the parameter `loc`:

If we add a label to the plot function, the values will be used in the legend command:

```
from polynomials import Polynomial
import numpy as np
import matplotlib.pyplot as plt

p = Polynomial(-0.8, 2.3, 0.5, 1, 0.2)
p_der = p.derivative()

fig, ax = plt.subplots()
X = np.linspace(-2, 3, 50, endpoint=True)
F = p(X)
F_derivative = p_der(X)
ax.plot(X, F, label="p")
ax.plot(X, F_derivative, label="derivation of p")

ax.legend(loc='upper left')
```

Output:
```
<matplotlib.legend.Legend at 0x7fd2efc15cf8>
```



It might be even more interesting to see the actual function in mathematical notation in our legend. Our polynomial class is capable of printing the function in LaTeX notation.

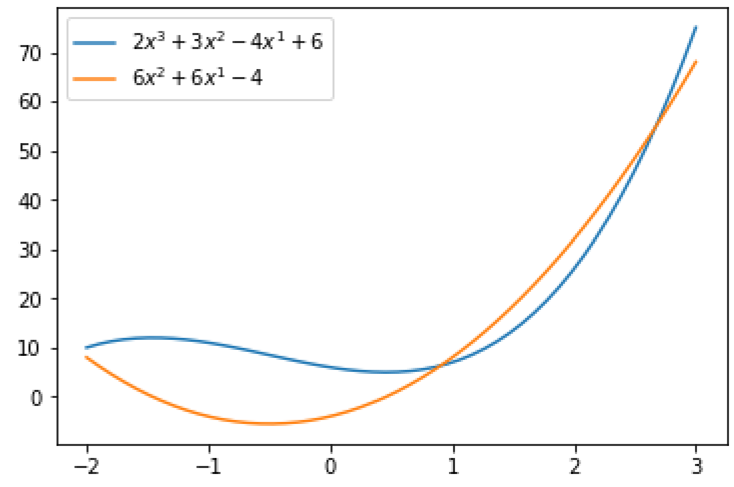
```
print(p)
print(p_der)
```

```
-0.8x^4 + 2.3x^3 + 0.5x^2 + 1x^1 + 0.2
-3.2x^3 + 6.8999999999999995x^2 + 1.0x^1 + 1
```

We can also use LaTeX in our labels, if we include it in '$' signs.

```
from polynomials import Polynomial
import numpy as np
import matplotlib.pyplot as plt

p = Polynomial(2, 3, -4, 6)
p_der = p.derivative()

fig, ax = plt.subplots()
X = np.linspace(-2, 3, 50, endpoint=True)
F = p(X)
F_derivative = p_der(X)
ax.plot(X, F, label="$" + str(p) + "$")
ax.plot(X, F_derivative, label="$" + str(p_der) + "$")

ax.legend(loc='upper left')
```

Output:
```
<matplotlib.legend.Legend at 0x7fd2efb99a58>
```

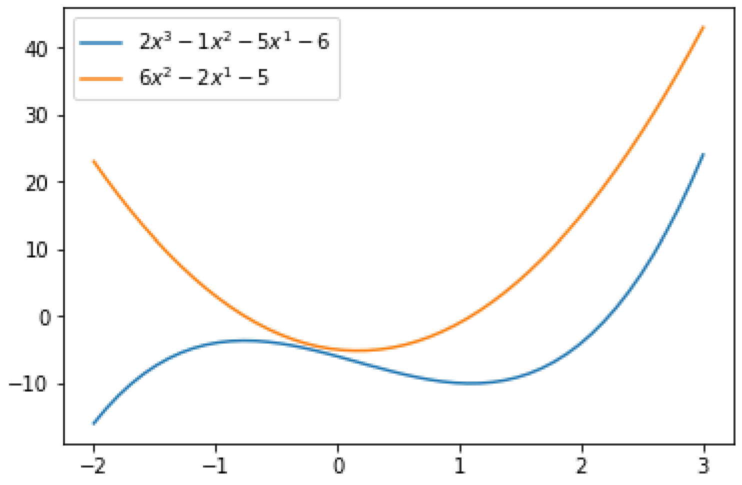


In many cases we don't know what the result may look like before you plot it. It could be for example, that the legend will overshadow an important part of the lines. If you don't know what the data may look like, it may be best to use 'best' as the argument for loc. Matplotlib will automatically try to find the best possible location for the legend:

```
from polynomials import Polynomial
import numpy as np
import matplotlib.pyplot as plt

p = Polynomial(2, -1, -5, -6)
p_der = p.derivative()
print(p_der)

fig, ax = plt.subplots()
X = np.linspace(-2, 3, 50, endpoint=True)
F = p(X)
F_derivative = p_der(X)
ax.plot(X, F, label="$" + str(p) + "$")
ax.plot(X, F_derivative, label="$" + str(p_der) + "$")

ax.legend(loc='best')
```

```
6x^2 - 2x^1 - 5
```

Output:
```
<matplotlib.legend.Legend at 0x7fd2ef370f60>
```

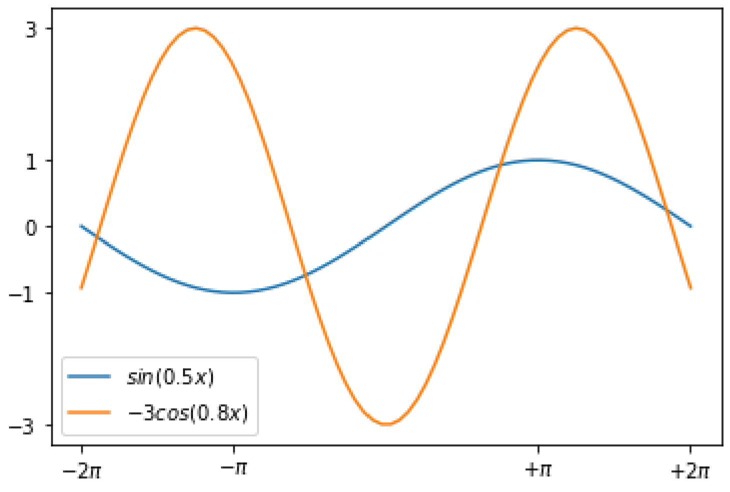


We will go back to trigonometric functions in the following examples. These examples show that

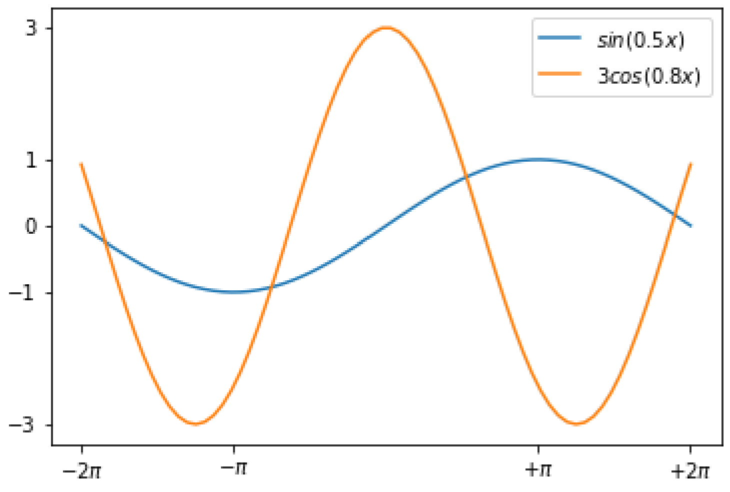
```
loc='best'
```

can work pretty well:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(0.5*X)
F2 = -3 * np.cos(0.8*X)

plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$', r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1, label="$sin(0.5x)$")
plt.plot(X, F2, label="$-3 cos(0.8x)$")
plt.legend(loc='best')

plt.show()
```



```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(0.5*X)
F2 = 3 * np.cos(0.8*X)

plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$', r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1, label="$sin(0.5x)$")
plt.plot(X, F2, label="$3 cos(0.8x)$")
plt.legend(loc='best')

plt.show()
```

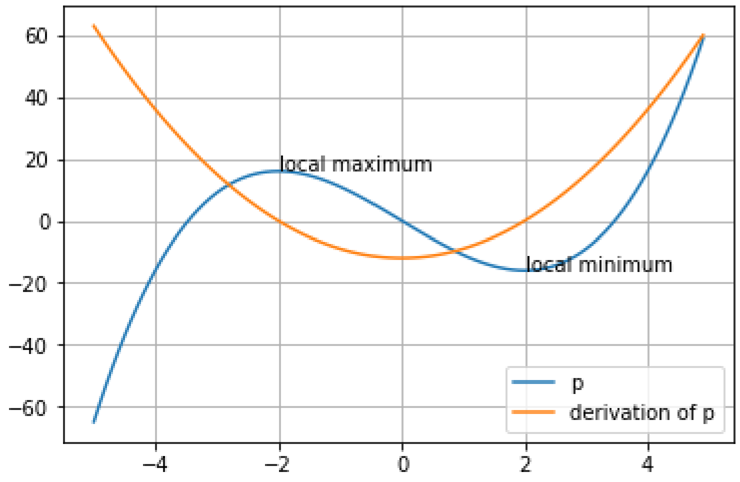


ANNOTATIONS

The visualizations of function plots often makes annotations necessary. This means we draw the readers attentions to important points and areas of the plot. To this purpose we use texts, labels and arrows. We have already used axis labels and titles for this purpose, but these are 'annotations' for the whole plot. We can easily annotate points inside the axis or on the graph with the `annotate method` of an axes object. In an annotation, there are two points to consider: the location being annotated represented by the argument xy and the location of the text xytext. Both of these arguments are (x,y) tuples.

We demonstrate how easy it is in matplotlib to to annotate plots in matplotlib with the annotate method. We will annotate the local maximum and the local minimum of a function. In its simplest form `annotate` method needs two arguments `annotate(s, xy)`, where `s` is the text string for the annotation and `xx` is the position of the point to be annotated:

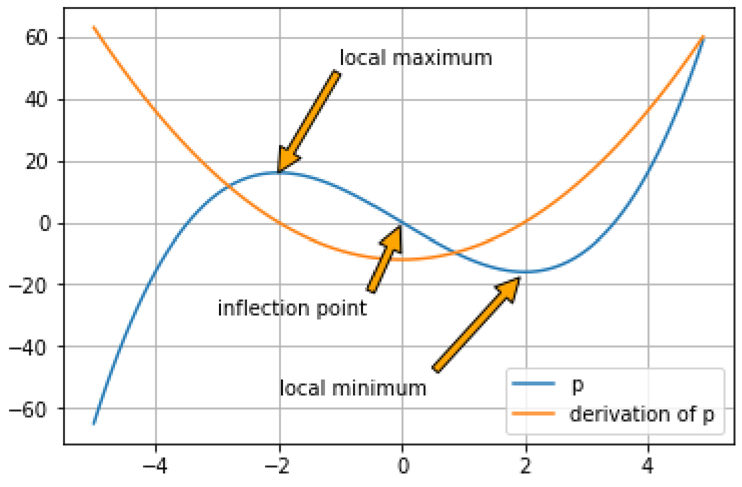
```
from polynomials import Polynomial
import numpy as np
import matplotlib.pyplot as plt

p = Polynomial(1, 0, -12, 0)
p_der = p.derivative()

fig, ax = plt.subplots()
X = np.arange(-5, 5, 0.1)
F = p(X)

F_derivative = p_der(X)
ax.grid()

maximum = (-2, p(-2))
minimum = (2, p(2))
ax.annotate("local maximum", maximum)
ax.annotate("local minimum", minimum)

ax.plot(X, F, label="p")
ax.plot(X, F_derivative, label="derivation of p")

ax.legend(loc='best')
plt.show()
```



If you are not satisfied with the automatic positioning of the text, you can assign a tuple with a position for the text to the keyword parameter xytext:

```
from polynomials import Polynomial
import numpy as np
import matplotlib.pyplot as plt

p = Polynomial(1, 0, -12, 0)
p_der = p.derivative()

fig, ax = plt.subplots()
X = np.arange(-5, 5, 0.1)
F = p(X)

F_derivative = p_der(X)
ax.grid()

ax.annotate("local maximum",
            xy=(-2, p(-2)),
            xytext=(-1, p(-2)+35),
            arrowprops=dict(facecolor='orange'))
ax.annotate("local minimum",
            xy=(2, p(2)),
            xytext=(-2, p(2)-40),
            arrowprops=dict(facecolor='orange', shrink=0.05))
ax.annotate("inflection point",
            xy=(0, p(0)),
            xytext=(-3, -30),
            arrowprops=dict(facecolor='orange', shrink=0.05))
```

```
ax.plot(X, F, label="p")
ax.plot(X, F_derivative, label="derivation of p")

ax.legend(loc='best')
plt.show()
```



We have to provide some informations to the parameters of annotate, we have used in our previous example.

Parameter	Meaning
xy	coordinates of the arrow tip
xytext	coordinates of the text location

The xy and the xytext locations of our example are in data coordinates. There are other coordinate systems available we can choose. The coordinate system of xy and xytext can be specified string values assigned to xycoords and textcoords. The default value is 'data':

String Value	Coordinate System
figure points	points from the lower left corner of the figure
figure pixels	pixels from the lower left corner of the figure
figure fraction	0,0 is lower left of figure and 1,1 is upper right
axes points	points from lower left corner of axes

String Value	Coordinate System
axes pixels	pixels from lower left corner of axes
axes fraction	0,0 is lower left of axes and 1,1 is upper right
data	use the axes data coordinate system

Additionally, we can also specify the properties of the arrow. To do so, we have to provide a dictionary of arrow properties to the parameter arrowprops:

arrowprops key	description
width	the width of the arrow in points
headlength	The length of the arrow head in points
headwidth	the width of the base of the arrow head in points
shrink	move the tip and base some percent away from the annotated point and text
**kwargs	any key for matplotlib.patches.Polygon, e.g., facecolor

Of course, the sinus function has "boring" and interesting values. Let's assume that you are especially interested in the value of $3 * sin(3 * pi/4)$.

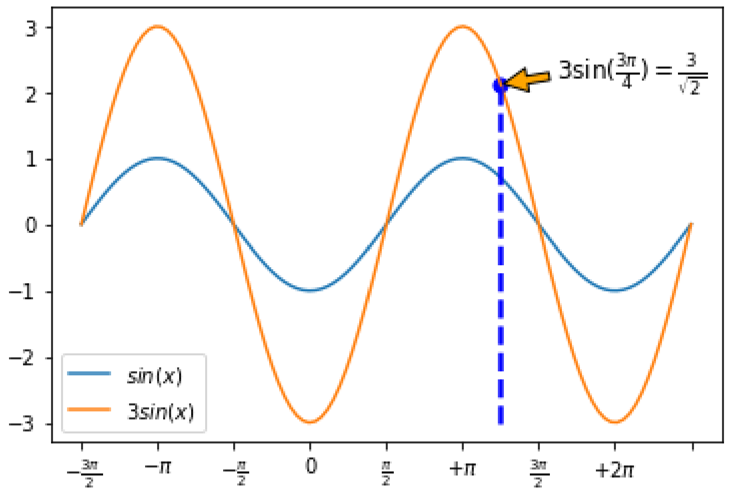
```
import numpy as np

print(3 * np.sin(3 * np.pi/4))
```

```
2.121320343559643
```

The numerical result doesn't look special, but if we do a symbolic calculation for the above expression we get $\frac{3}{\sqrt{2}}$. Now we want to label this point on the graph. We can do this with the annotate function. We want to annotate our graph with this point.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-2 * np.pi, 2 * np.pi, 100)
F1 = np.sin(X)
```

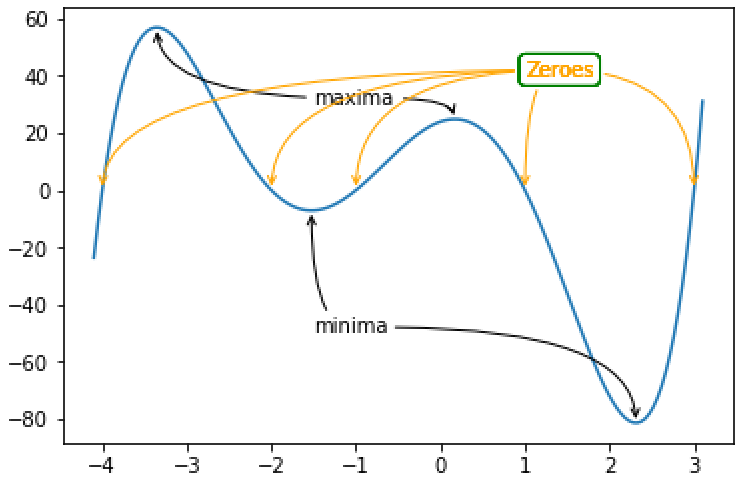
```
F2 = 3 * np.sin(X)
fig, ax = plt.subplots()

ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
#plt.xticks(np.arange(-2 * np.pi, 2.5 * np.pi, np.pi / 2))
ax.set_xticklabels([r'$-2\pi$', r'$-\frac{3\pi}{2}$', r'$-\pi$',
                    r'$-\frac{\pi}{2}$', 0, r'$\frac{\pi}{2}$',
                    r'$+\pi$', r'$\frac{3\pi}{2}$', r'$+2\pi$'])


ax.plot(X, F1, label="$sin(x)$")
ax.plot(X, F2, label="$3 sin(x)$")
ax.legend(loc='lower left')
x = 3 * np.pi / 4

# Plot vertical line:
ax.plot([x, x],[-3, 3 * np.sin(x)], color ='blue', linewidth=2.5,
linestyle="--")
# Print the blue dot:
ax.scatter([x,],[3 * np.sin(x),], 50, color ='blue')

text_x, text_y = (3.5, 2.2)
ax.annotate(r'$3\sin(\frac{3\pi}{4})=\frac{3}{\sqrt{2}}$',
            xy=(x, 3 * np.sin(x)),
            xytext=(text_x, text_y),
            arrowprops=dict(facecolor='orange', shrink=0.05),
            fontsize=12)

plt.show()
```



SOME MORE CURVE SKETCHING

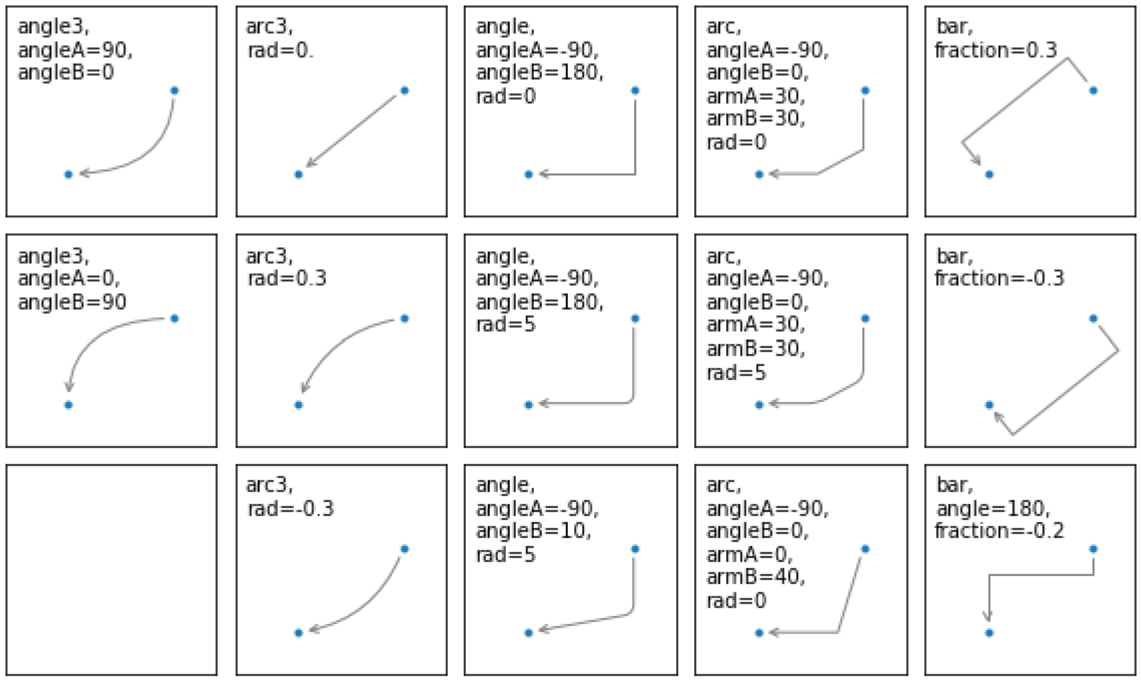
There is anothe example, in which we play around with the arrows and annoate the extrema.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-4.1, 3.1, 150, endpoint=True)
F = X**5 + 3*X**4 - 11*X**3 - 27*X**2 + 10*X + 24

fig, ax = plt.subplots()
ax.plot(X, F)

minimum1 = -1.5264814, -7.051996717492152
minimum2 = 2.3123415793720303, -81.36889464201387
ax.annotate("minima",
            xy=minimum1,
            xytext=(-1.5, -50),
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle3,angleA=0,angle
B=-90"))
ax.annotate(" ",
            xy=minimum2,
            xytext=(-0.7, -50),
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle3,angleA=0,angle
B=-90"))

maximum1 = -3.35475845886632, 56.963107876630595
maximum2 = .16889828232847673,  24.868343482875485
ax.annotate("maxima", xy=maximum1,
            xytext=(-1.5, 30),
            arrowprops=dict(arrowstyle="->",
                       connectionstyle="angle3,angleA=0,angleB=-9
0"))
ax.annotate(" ", xy=maximum2,
            xytext=(-0.6, 30),
            arrowprops=dict(arrowstyle="->",
                      connectionstyle="angle3,angleA=0,angleB=-9
0"))

zeroes = -4, -2, -1, 1, 3
for zero in zeroes:
    zero = zero, 0
    ax.annotate("Zeroes",
                xy=zero,
                color="orange",
                bbox=dict(boxstyle="round", fc="none", ec="gree
n"),
                xytext=(1, 40),
                arrowprops=dict(arrowstyle="->", color="orange",
                      connectionstyle="angle3,angleA=0,angleB=-9
0"))

plt.show()
```



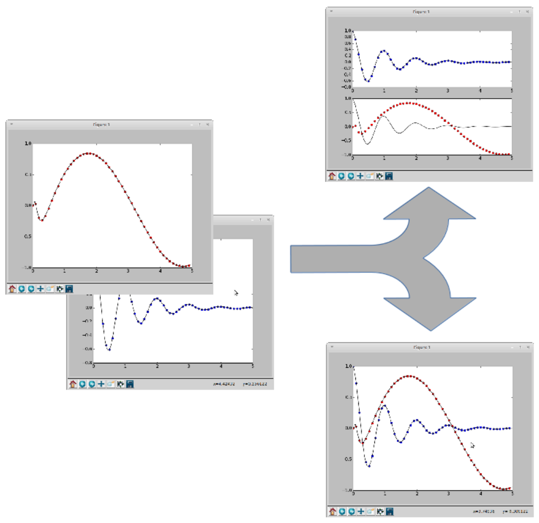
The following program visualizes the various arrowstyles:

```
import matplotlib.pyplot as plt


def demo_con_style(ax, connectionstyle):
    x1, y1 = 0.3, 0.2
    x2, y2 = 0.8, 0.6

    ax.plot([x1, x2], [y1, y2], ".")
    ax.annotate("",
                xy=(x1, y1), xycoords='data',
                xytext=(x2, y2), textcoords='data',
                arrowprops=dict(arrowstyle="->", color="0.5",
                                shrinkA=5, shrinkB=5,
                                patchA=None, patchB=None,
                                connectionstyle=connectionstyle))
    ax.text(.05, .95, connectionstyle.replace(",", ",\n"),
            transform=ax.transAxes, ha="left", va="top")


fig, axs = plt.subplots(3, 5, figsize=(8, 4.8))
demo_con_style(axs[0, 0], "angle3,angleA=90,angleB=0")
demo_con_style(axs[1, 0], "angle3,angleA=0,angleB=90")
demo_con_style(axs[0, 1], "arc3,rad=0.")
demo_con_style(axs[1, 1], "arc3,rad=0.3")
demo_con_style(axs[2, 1], "arc3,rad=-0.3")
demo_con_style(axs[0, 2], "angle,angleA=-90,angleB=180,rad=0")
demo_con_style(axs[1, 2], "angle,angleA=-90,angleB=180,rad=5")
demo_con_style(axs[2, 2], "angle,angleA=-90,angleB=10,rad=5")
demo_con_style(axs[0, 3], "arc,angleA=-90,angleB=0,armA=30,armB=3
0,rad=0")
demo_con_style(axs[1, 3], "arc,angleA=-90,angleB=0,armA=30,armB=3
0,rad=5")
demo_con_style(axs[2, 3], "arc,angleA=-90,angleB=0,armA=0,armB=4
0,rad=0")
demo_con_style(axs[0, 4], "bar,fraction=0.3")
demo_con_style(axs[1, 4], "bar,fraction=-0.3")
demo_con_style(axs[2, 4], "bar,angle=180,fraction=-0.2")

for ax in axs.flat:
    ax.set(xlim=(0, 1), ylim=(0, 1), xticks=[], yticks=[], aspec
t=1)
fig.tight_layout(pad=0.2)

plt.show()
```


angle3, angleA=90, angleB=0
arc3, rad=0.
angle, angleA=-90, angleB=180, rad=0
arc, angleA=-90, angleB=0, armA=30, armB=30, rad=0
bar, fraction=0.3
angle3, angleA=0, angleB=90
arc3, rad=0.3
angle, angleA=-90, angleB=180, rad=5
arc, angleA=-90, angleB=0, armA=30, armB=30, rad=5
bar, fraction=-0.3
arc3, rad=-0.3
angle, angleA=-90, angleB=10, rad=5
arc, angleA=-90, angleB=0, armA=0, armB=40, rad=0
bar, angle=180, fraction=-0.2

MATPLOTLIB TUTORIAL: SUBPLOTS

We have given so far lots of examples for plotting graphs in the previous chapters of our Python tutorial on Matplotlib. A frequently asked question is how to have multiple plots in one graph?



In the simplest case this might mean, that you have one curve and you want another curve printed over it. This is not a problem, because it will be enough to put the two plots in your scripts, as we have seen before. The more interesting case is, if you want two plots beside of each other for example. In one figure but in two subplots. The idea is to have more than one graph in one window and each graph appears in its own subplot.

We will demonstrate in our examples how this can be accomplished with the funtion `subplots` .

CREATING SUBPLOTS WITH SUBPLOTS

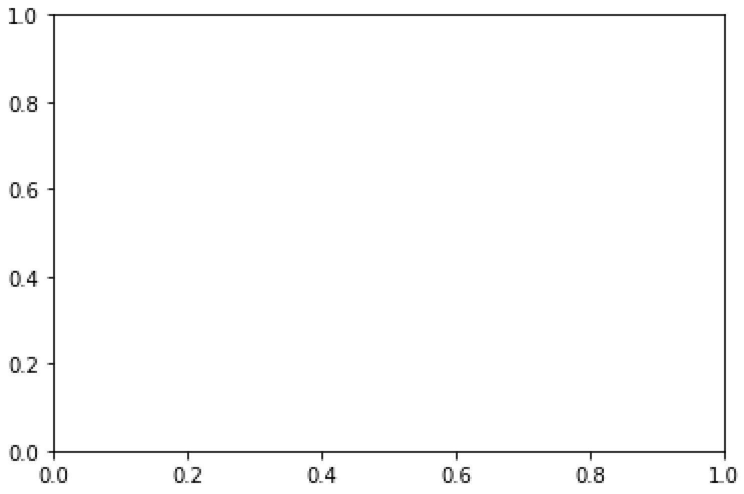
The function `subplot` create a figure and a set of subplots. It is a wrapper function to make it convenient to create common layouts of subplots, including the enclosing figure object, in a single call.

This function returns a figure and an Axes object or an array of Axes objects.

If we call this function without any parameters - like we do in the following example - a Figure object and one Axes object will be returned:

```python
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
print(fig, ax)
```

```
Figure(432x288) AxesSubplot(0.125,0.125;0.775x0.755)
```



The parameter of subplots function are:

`subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplot_kw=None, gridspec_kw=None, **fig_kw)`

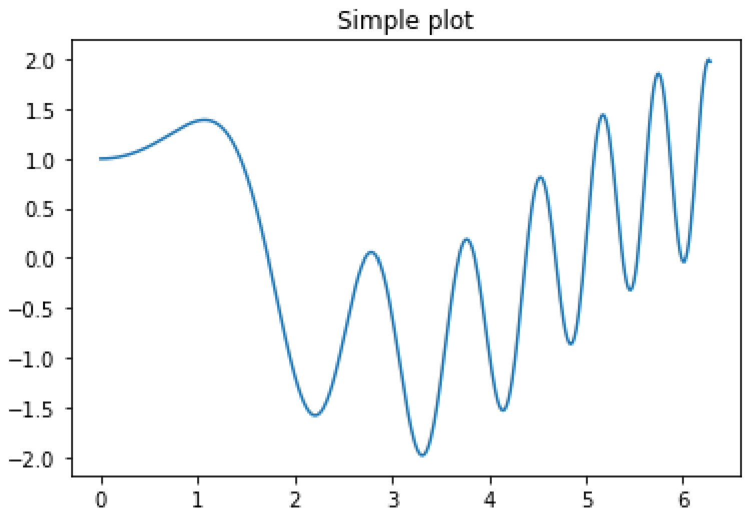
Parameter	Meaning
nrows, ncols	int, optional, default: 1 Number of rows/columns of the subplot grid.
sharex	bool or {'none', 'all', 'row', 'col'}, default: False Controls sharing of properties among x (`sharex`) axis: If `sharex` is set to `True` or `all` , the x-axis will be shared among all subplots. If `sharex` is set to `False` or `none` , each x-axis of a subplot will be independent. If it is set to `row` , each subplot row will share an x-axis. If it is set to `col` , each subplot column will share an x-axis.
sharey	analogue to sharex When subplots have a shared x-axis along a column, only the x tick labels of the bottom subplot are created. Similarly, when subplots have a shared y-axis along a row, only the y tick labels of the first column subplot are created.
squeeze	bool, optional, default: True If `squeeze` is set to `True` , extra dimensions are squeezed out from the returned Axes object
num	integer or string, optional, default: None A `.pyplot.figure` keyword that sets the figure number or label.
subplot_kw	dict, optional Dict with keywords passed to the `~matplotlib.figure.Figure.add_subplot` call used to create each

Parameter	Meaning
	subplot.
gridspec_kw	dict, optional Dict with keywords passed to the `~matplotlib.gridspec.GridSpec` constructor used to create the grid the subplots are placed on.
**fig_kw	All additional keyword arguments are passed to the `.pyplot.figure` call.

In the previous chapters of our tutorial, we saw already the simple case of creating one figure and one axes.

```python
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2) + np.cos(x)

#Creates just a figure and only one subplot
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')
```

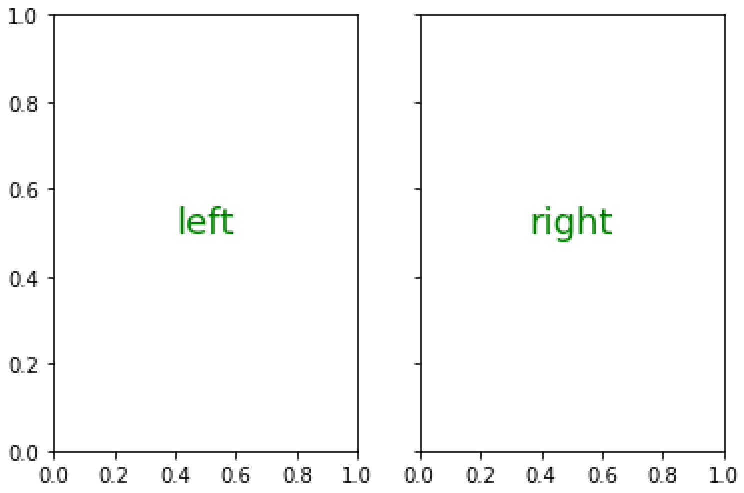
Output:
```
Text(0.5, 1.0, 'Simple plot')
```



We will demonstrate now, how to create to subplots beside of each other. By setting the parameter `sharey` to `True` , we make sure that the y labels are not repeated on the right subplot:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 4))

fig, (ax1, ax2)  = plt.subplots(1, 2,
                                sharey='row')

ax1.text(0.5, 0.5,
              "left",
              color="green",
              fontsize=18,
              ha='center')

ax2.text(0.5, 0.5,
              "right",
              color="green",
              fontsize=18,
              ha='center')

plt.show()
```

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

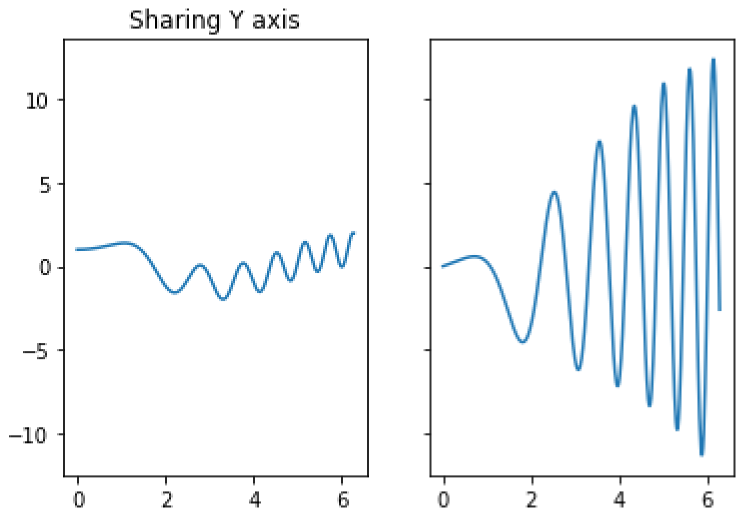


Now with some 'real' data:

```
f, (ax1, ax2) = plt.subplots(1, 2,
                             sharey=True)
derivative = 2 * x * np.cos(x**2) - np.sin(x)
```

```
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.plot(x, derivative)
```

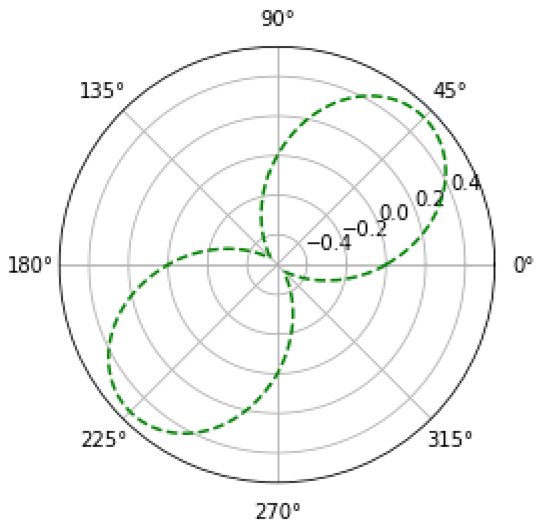
Output:
```
[<matplotlib.lines.Line2D at 0x7fc9509f7a90>]
```



We demonstrate in the following example how to create a subplot for polar plotting. We achieve this by creating a key `polar` in the the `subplot_kw` dictionary and set it to `True` :

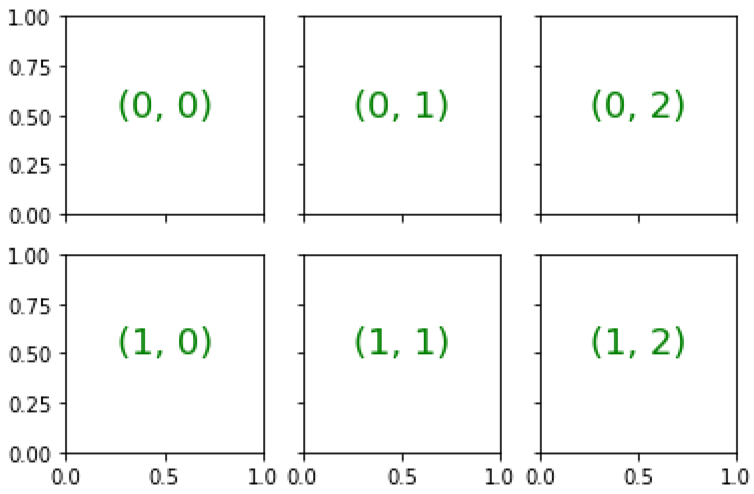
```
fig, ax = plt.subplots(1, subplot_kw=dict(polar=True))
ax.plot(x, np.sin(x) * np.cos(x), "--g")
```

Output:
```
[<matplotlib.lines.Line2D at 0x7fc9519bb650>]
```

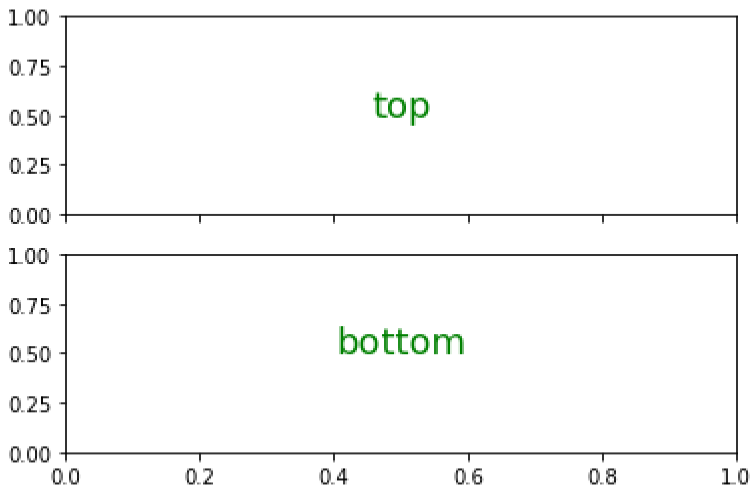


The first two parameters of `subplots` define the numbers of rows and columns respectively. We demonstrate this in the following example. To demonstrate the structure we use the `text` method of the axis objects. We use it to put the

```
import matplotlib.pyplot as plt

rows, cols = 2, 3
fig, ax = plt.subplots(rows, cols,
                       sharex='col',
                       sharey='row')

for row in range(rows):
    for col in range(cols):
        ax[row, col].text(0.5, 0.5,
                          str((row, col)),
                          color="green",
                          fontsize=18,
                          ha='center')

plt.show()
```



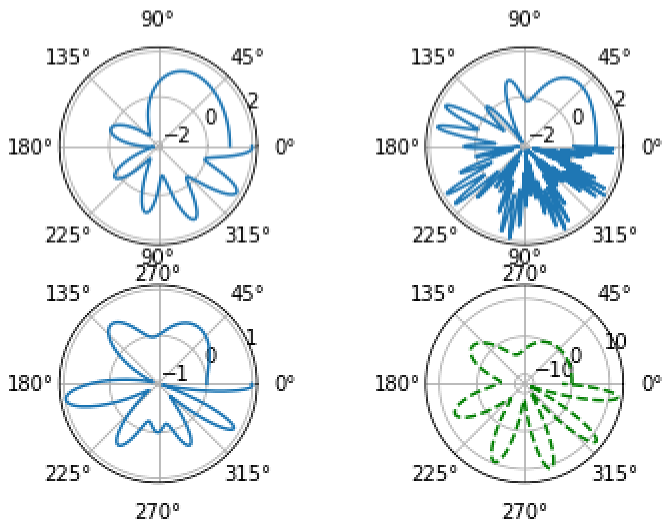
```
import matplotlib.pyplot as plt
#plt.figure(figsize=(6, 4))

fig, ax = plt.subplots(2,
           sharex='col', sharey='row')

ax[0].text(0.5, 0.5,
              "top",
              color="green",
              fontsize=18,
              ha='center')

ax[1].text(0.5, 0.5,
              "bottom",
              color="green",
              fontsize=18,
              ha='center')

plt.show()
```



We will create now a similar structure with two subplots on top of each other containing polar plots:

```
fig, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
axes[0, 0].plot(x, y)
axes[0, 1].plot(x, np.sin(x**2) + np.cos(x**3))
axes[1, 0].plot(x, np.cos(x) * np.sin(x**2))
axes[1, 1].plot(x, derivative, "g--")
```

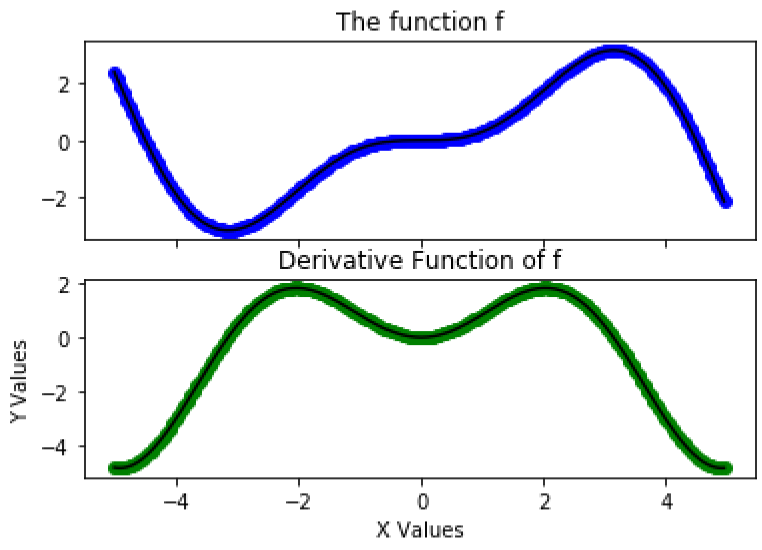
Output:
```
[<matplotlib.lines.Line2D at 0x7fc9506cdad0>]
```



Now with titles for subplots:

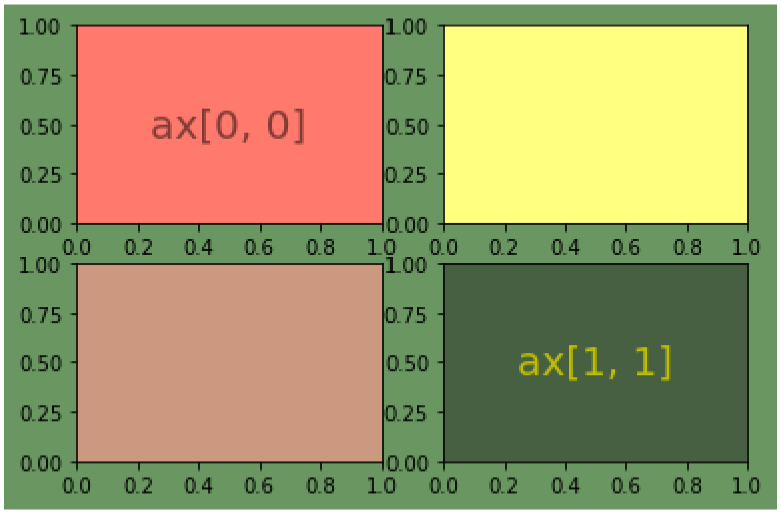
```
import numpy as np
import matplotlib.pyplot as plt
plt.figure(figsize=(6, 4))

def f(x):
    return np.sin(x) - x * np.cos(x)

def fp(x):
    """ The derivative of f """
    return x * np.sin(x)

X = np.arange(-5, 5.0, 0.05)

fig, ax = plt.subplots(2,
           sharex='col', sharey='row')

ax[0].plot(X, f(X), 'bo', X, f(X), 'k')
ax[0].set(title='The function f')

ax[1].plot(X, fp(X), 'go', X, fp(X), 'k')
ax[1].set(xlabel='X Values', ylabel='Y Values',
       title='Derivative Function of f')

plt.show()
```

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



```
import matplotlib.pyplot as plt
python_course_green = "#476042"
python_course_orange = "#f17c0b"
python_course_green_light = "#6a9662"
#plt.figure(figsize=(6, 4))
fig, ax = plt.subplots(2, 2,
                       figsize=(6, 4),
                       facecolor=python_course_green_light)


ax[0, 0].text(0.5, # x-Koordinate, 0 ganz links, 1 ganz rechts
              0.5, # y-Koordinate, 0 ganz oben, 1 ganz unten
              'ax[0, 0]', # der Text der ausgegeben wird
               horizontalalignment='center', # abgekürzt 'ha'
               verticalalignment='center', # abgekürzt 'va'
               fontsize=20,
               alpha=.5 )

ax[0, 0].set_facecolor('xkcd:salmon')

ax[1,1].text(0.5, 0.5,
         'ax[1, 1]',
         ha='center', va='center',
         fontsize=20,
         color="y")
ax[1, 0].set_facecolor((0.8, 0.6, 0.5))
ax[0, 1].set_facecolor((1, 1, 0.5))
```

```
ax[1, 1].set_facecolor(python_course_green)
plt.show()
```

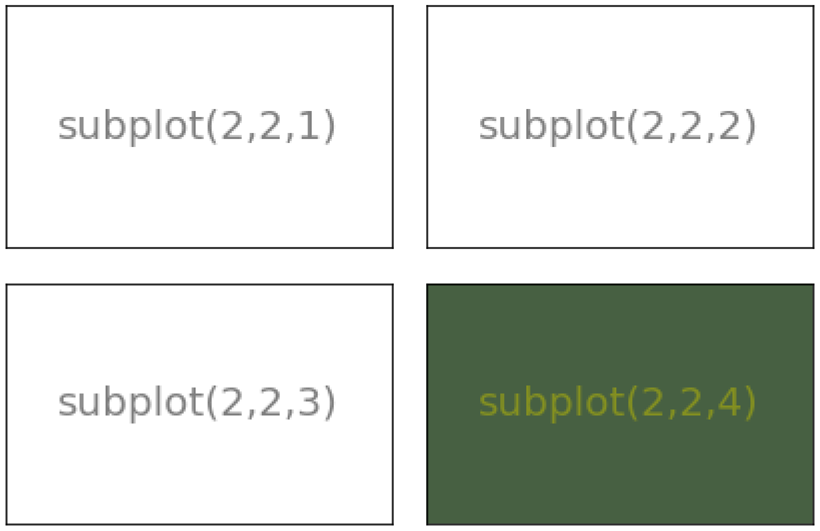


Let us get rid of the ticks again. This time we cannot use `plt.xticks(())` and `plt.yticks(())`. We have to use the `set_xticks(())` and `set_yticks(())` methods instead.

Activating all subplot of the 2x2 grid looks like this:

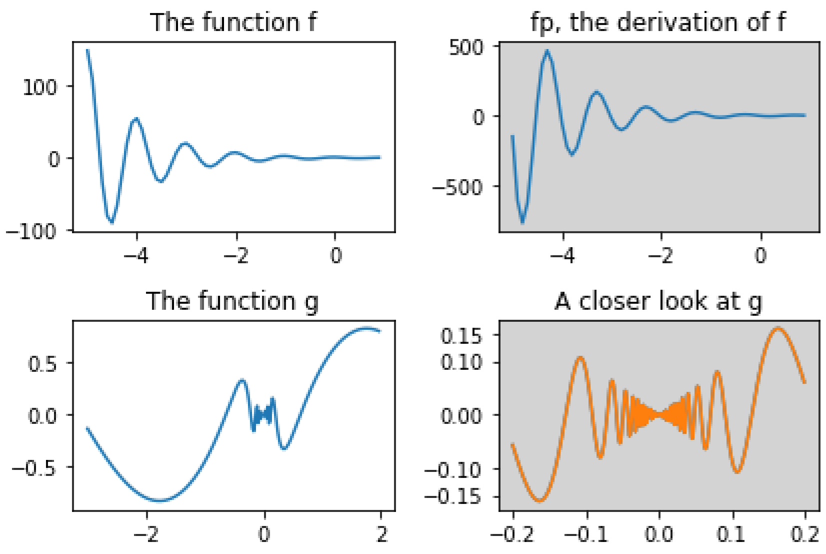
```
import matplotlib.pyplot as plt

python_course_green = "#476042"
fig = plt.figure(figsize=(6, 4))
sub1 = plt.subplot(2, 2, 1)
sub1.set_xticks(())
sub1.set_yticks(())
sub1.text(0.5, 0.5, 'subplot(2,2,1)', ha='center', va='center',
        size=20, alpha=.5)

sub2 = plt.subplot(2, 2, 2)
sub2.set_xticks(())
sub2.set_yticks(())
sub2.text(0.5, 0.5, 'subplot(2,2,2)', ha='center', va='center',
        size=20, alpha=.5)

sub3 = plt.subplot(2, 2, 3)
sub3.set_xticks(())
sub3.set_yticks(())
sub3.text(0.5, 0.5, 'subplot(2,2,3)', ha='center', va='center',
        size=20, alpha=.5)
```

```
sub4 = plt.subplot(2, 2, 4, facecolor=python_course_green)
sub4.set_xticks(())
sub4.set_yticks(())
sub4.text(0.5, 0.5, 'subplot(2,2,4)', ha='center', va='center',
        size=20, alpha=.5, color="y")

fig.tight_layout()
plt.show()
```



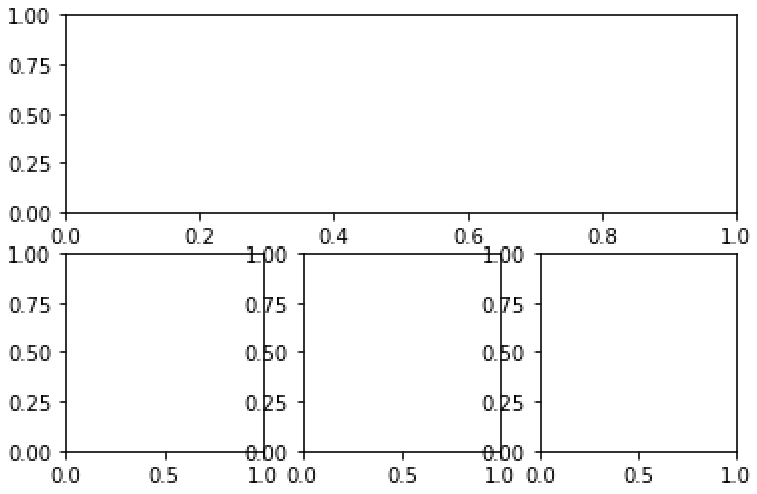
The previous examples are solely showing how to create a subplot design. Usually, you want to write Python programs using Matplotlib and its subplot features to depict some graphs. We will demonstrate how to populate the previous subplot design with some example graphs:

```
import numpy as np
from numpy import e, pi, sin, exp, cos
import matplotlib.pyplot as plt

def f(t):
    return exp(-t) * cos(2*pi*t)

def fp(t):
    return -2*pi * exp(-t) * sin(2*pi*t) - e**(-t)*cos(2*pi*t)

def g(t):
    return sin(t) * cos(1/(t+0.1))

def g(t):
    return sin(t) * cos(1/(t))


python_course_green = "#476042"
fig = plt.figure(figsize=(6, 4))

t = np.arange(-5.0, 1.0, 0.1)

sub1 = fig.add_subplot(221) # instead of plt.subplot(2, 2, 1)
sub1.set_title('The function f') # non OOP: plt.title('The functio
n f')
sub1.plot(t, f(t))


sub2 = fig.add_subplot(222, facecolor="lightgrey")
sub2.set_title('fp, the derivation of f')
sub2.plot(t, fp(t))


t = np.arange(-3.0, 2.0, 0.02)
sub3 = fig.add_subplot(223)
sub3.set_title('The function g')
sub3.plot(t, g(t))

t = np.arange(-0.2, 0.2, 0.001)
sub4 = fig.add_subplot(224, facecolor="lightgrey")
sub4.set_title('A closer look at g')
sub4.set_xticks([-0.2, -0.1, 0, 0.1, 0.2])
sub4.set_yticks([-0.15, -0.1, 0, 0.1, 0.15])
sub4.plot(t, g(t))

plt.plot(t, g(t))

plt.tight_layout()
plt.show()
```



Another example:

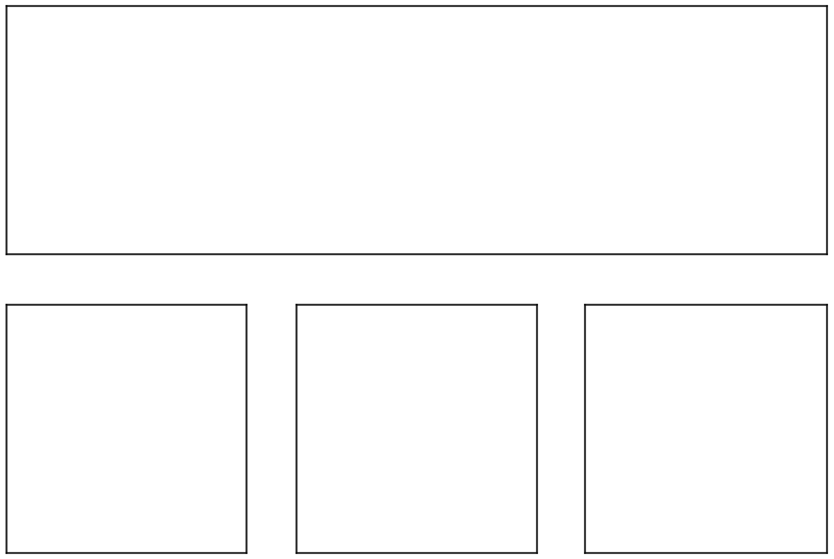
```
import  matplotlib.pyplot as plt

X = [ (2,1,1), (2,3,4), (2,3,5), (2,3,6) ]
for nrows, ncols, plot_number in X:
    plt.subplot(nrows, ncols, plot_number)
```



The following example shows nothing special. We will remove the xticks and play around with the size of the figure and the subplots. To do this we introduce the keyword paramter figsize of 'figure' and the function 'subplot_adjust' along with its keyword parameters bottom, left, top, right:

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

```
fig =plt.figure(figsize=(6,4))
fig.subplots_adjust(bottom=0.025, left=0.025, top = 0.975, righ
t=0.975)

X = [ (2,1,1), (2,3,4), (2,3,5), (2,3,6) ]
for nrows, ncols, plot_number in X:
    sub = fig.add_subplot(nrows, ncols, plot_number)
    sub.set_xticks([])
    sub.set_yticks([])
```



Alternative Solution:

As the first three three elements of 2x3 grid have to be joined, we can choose a tuple notation, inour case (1,3) in (2,3,(1,3)) to define that the first three elements of a notional 2x3 grid are joined:

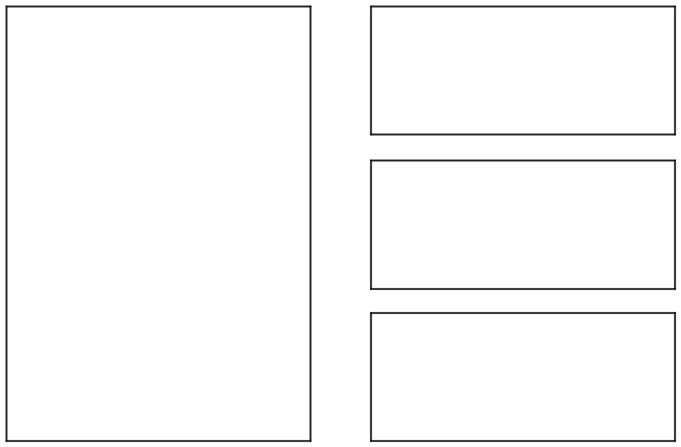
```
import  matplotlib.pyplot as plt

fig =plt.figure(figsize=(6,4))
fig.subplots_adjust(bottom=0.025, left=0.025, top = 0.975, righ
t=0.975)

X = [ (2,3,(1,3)), (2,3,4), (2,3,5), (2,3,6) ]
for nrows, ncols, plot_number in X:
    sub = fig.add_subplot(nrows, ncols, plot_number)
    sub.set_xticks([])
    sub.set_yticks([])
```

EXERCISE

How can you create a subplotdesign of a 3x2 design, where the complete first column is spanned?

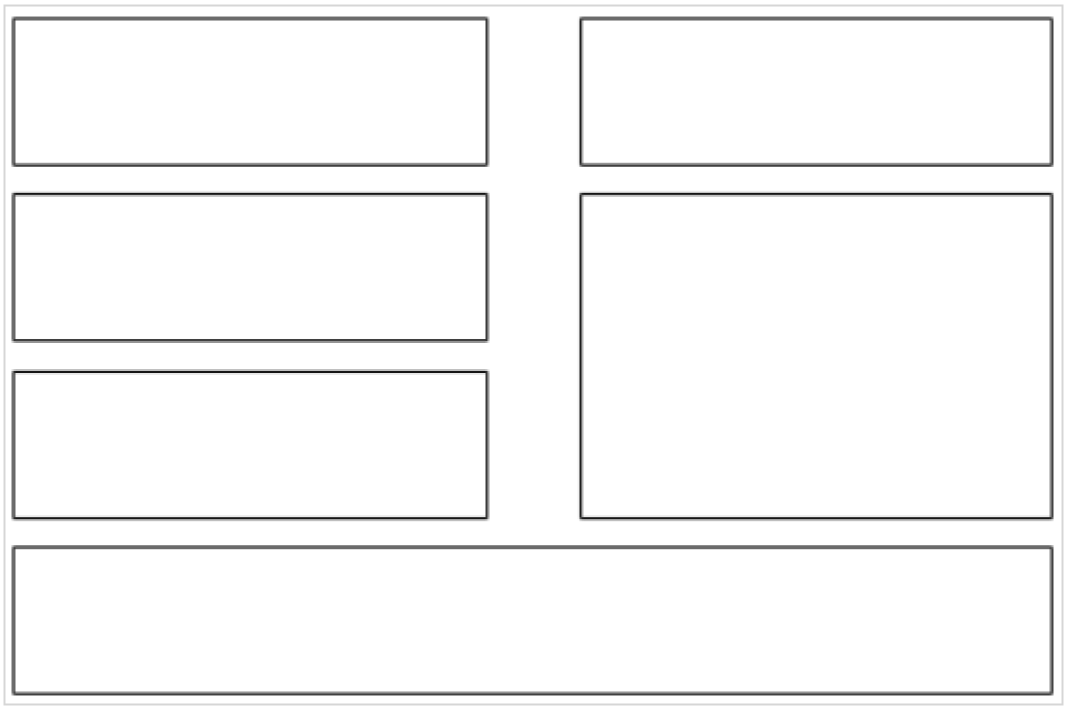
Solution:

```
import  matplotlib.pyplot as plt

X = [ (1,2,1), (3,2,2), (3,2,4), (3,2,6) ]
for nrows, ncols, plot_number in X:
    plt.subplot(nrows, ncols, plot_number)
    plt.xticks([])
    plt.yticks([])
```



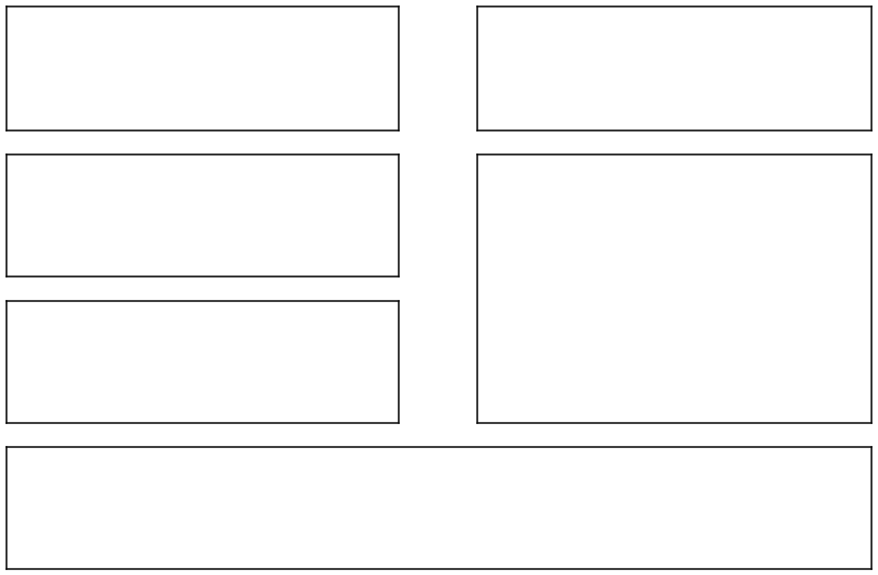
EXERCISE

Create a subplot layout for the following design:



Solution:

```
import  matplotlib.pyplot as plt

X = [  (4,2,1),(4,2,2), (4,2,3), (4,2,5), (4,2,(4,6)), (4,1,4)]
plt.subplots_adjust(bottom=0, left=0, top = 0.975, right=1)
for nrows, ncols, plot_number in X:
    plt.subplot(nrows, ncols, plot_number)
    plt.xticks([])
    plt.yticks([])

plt.show()
```

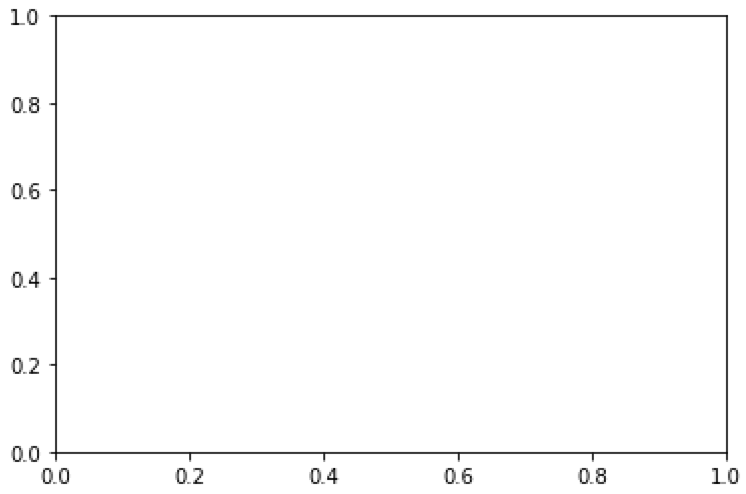


SUBPLOTS WITH GRIDSPEC

'matplotlib.gridspec' contains a class GridSpec. It can be used as an alternative to subplot to specify the geometry of the subplots to be created. The basic idea behind GridSpec is a 'grid'. A grid is set up with a number of rows and columns. We have to define after this, how much of the grid a subplot should span.

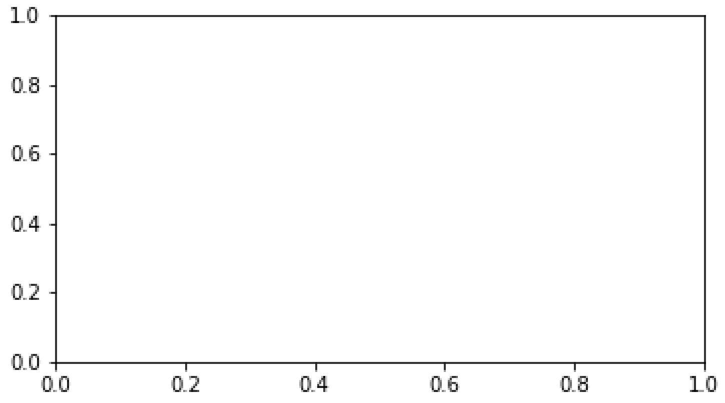
The following example shows the the trivial or simplest case, i.e. a 1x1 grid

```
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

fig = plt.figure()
gs = GridSpec(1, 1)
ax = fig.add_subplot(gs[0,0])

plt.show()
```



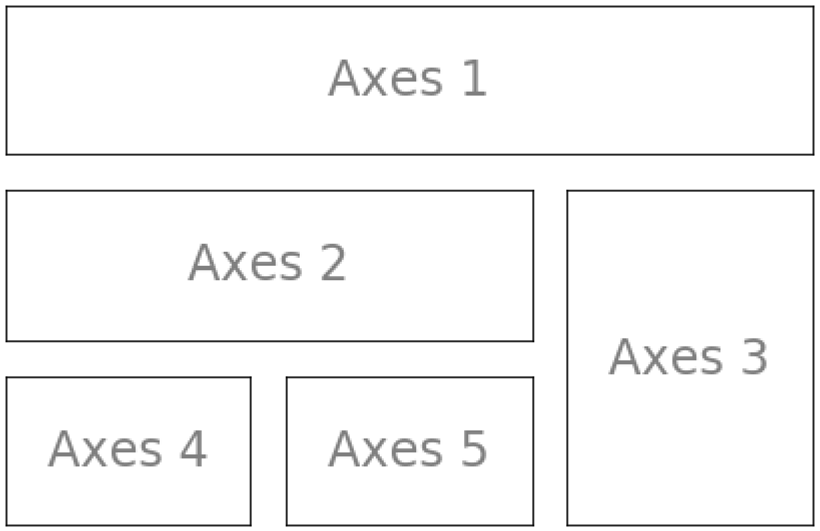
We could have used some of the parameters of Gridspec, e.g. we can define, that our graph should begin at 20 % from the bottom and 15 % to the left side of the available figure area:

```
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

fig = plt.figure()
gs = GridSpec(1, 1,
              bottom=0.2,
              left=0.15,
              top=0.8)
ax = fig.add_subplot(gs[0,0])

plt.show()
```



The next example shows a complexer example with a more elaborate grid design:

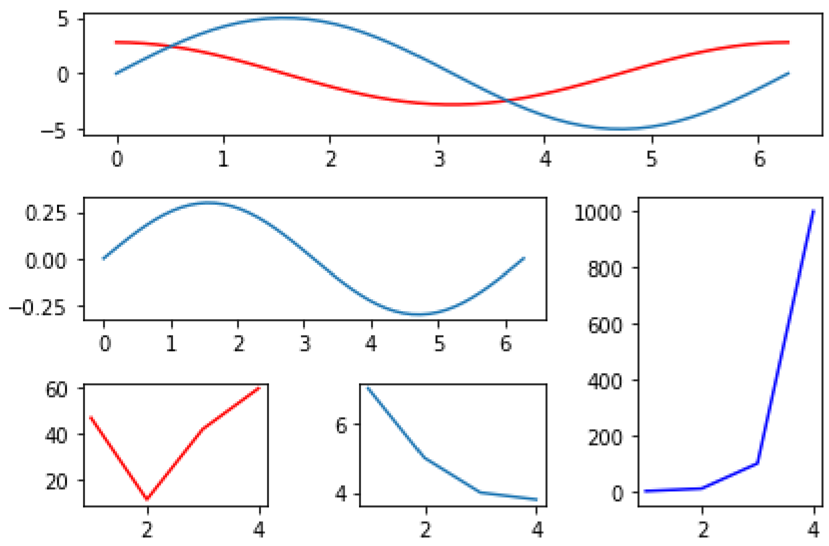
```python
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
G = gridspec.GridSpec(3, 3)

axes_1 = plt.subplot(G[0, :])
axes_1.set_xticks(())
axes_1.set_yticks(())
axes_1.text(0.5, 0.5, 'Axes 1',
            ha='center', va='center',
            size=24, alpha=.5)

axes_2 = plt.subplot(G[1, :-1])
axes_2.set_xticks(())
axes_2.set_yticks(())
axes_2.text(0.5, 0.5, 'Axes 2',
            ha='center', va='center',
            size=24, alpha=.5)
axes_3 = plt.subplot(G[1:, -1])
axes_3.set_xticks(())
axes_3.set_yticks(())
axes_3.text(0.5, 0.5, 'Axes 3',
            ha='center', va='center',
            size=24, alpha=.5)
axes_4 = plt.subplot(G[-1, 0])
axes_4.set_xticks(())
axes_4.set_yticks(())
axes_4.text(0.5, 0.5, 'Axes 4',
            ha='center', va='center',
            size=24, alpha=.5)
axes_5 = plt.subplot(G[-1, -2])
axes_5.set_xticks(())
axes_5.set_yticks(())
axes_5.text(0.5, 0.5, 'Axes 5',
            ha='center', va='center',
            size=24, alpha=.5)

plt.tight_layout()
plt.show()
```



We will use now the grid specification from the previous example to populate it with the graphs of some functions:

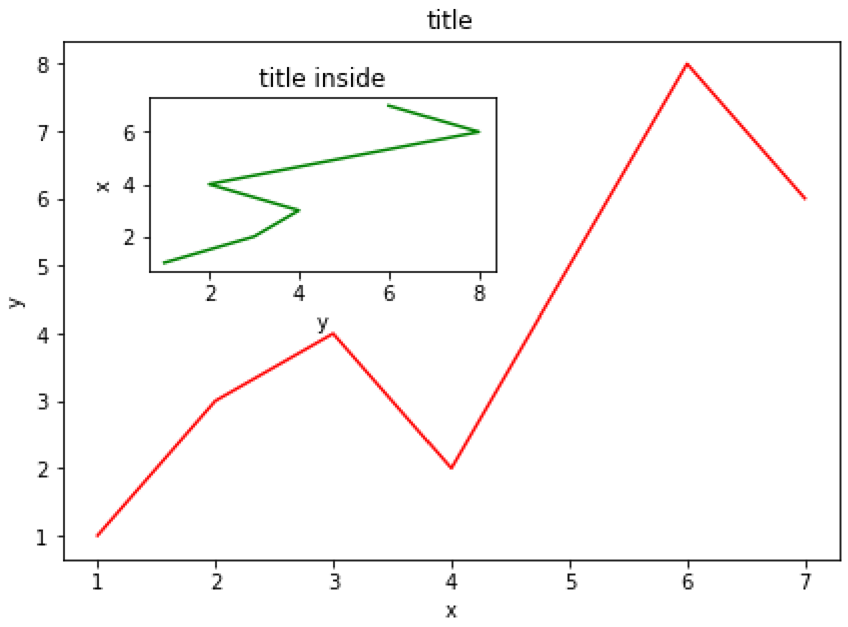
```python
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(6, 4))
G = gridspec.GridSpec(3, 3)

X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F1 = 2.8 * np.cos(X)
F2 = 5 * np.sin(X)
F3 = 0.3 * np.sin(X)

axes_1 = plt.subplot(G[0, :])
axes_1.plot(X, F1, 'r-', X, F2)

axes_2 = plt.subplot(G[1, :-1])
axes_2.plot(X, F3)

axes_3 = plt.subplot(G[1:, -1])
axes_3.plot([1,2,3,4], [1,10,100,1000], 'b-')

axes_4 = plt.subplot(G[-1, 0])
axes_4.plot([1,2,3,4], [47, 11, 42, 60], 'r-')

axes_5 = plt.subplot(G[-1, -2])
```

```
axes_5.plot([1,2,3,4], [7, 5, 4, 3.8])

plt.tight_layout()
plt.show()
```



A PLOT INSIDE OF ANOTHER PLOT

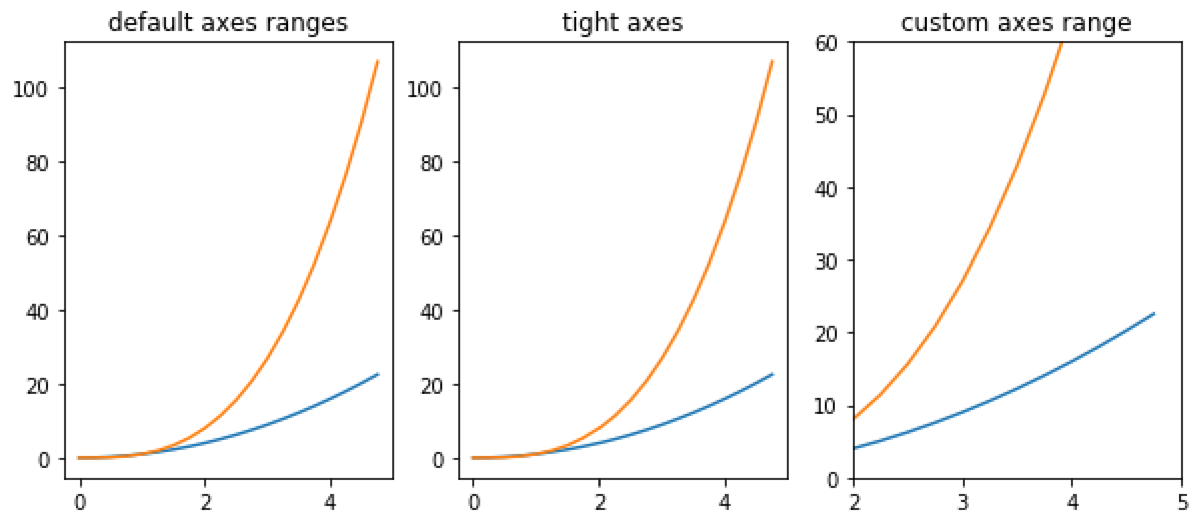
```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()

X = [1, 2, 3, 4, 5, 6, 7]
Y = [1, 3, 4, 2, 5, 8, 6]

axes1 = fig.add_axes([0.1, 0.1, 0.9, 0.9]) # main axes
axes2 = fig.add_axes([0.2, 0.6, 0.4, 0.3]) # inset axes

# main figure
axes1.plot(X, Y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('title')

# insert
axes2.plot(Y, X, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('title inside');
```



SETTING THE PLOT RANGE

It's possible to configure the ranges of the axes. This can be done by using the set_ylim and set_xlim methods in the axis object. With axis('tight') we create automatrically "tightly fitted" axes ranges:

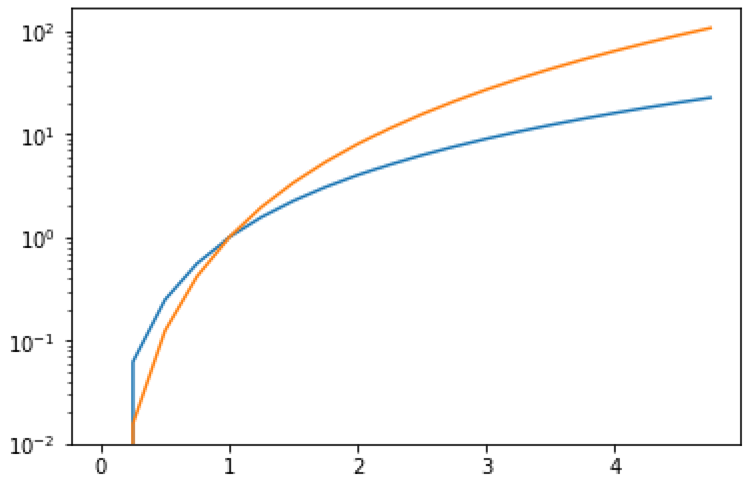
```
import numpy as np
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 3, figsize=(10, 4))

x = np.arange(0, 5, 0.25)

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```



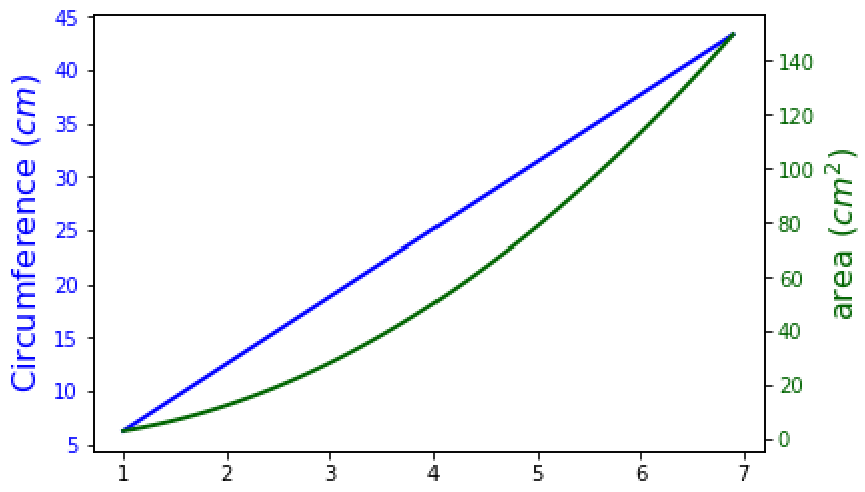
LOGARITHMIC SCALE

It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using set_xscale and set_yscale methods which accept one parameter (with the value "log" in this case):

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

x = np.arange(0, 5, 0.25)

ax.plot(x, x**2, x, x**3)


ax.set_yscale("log")

plt.show()
```



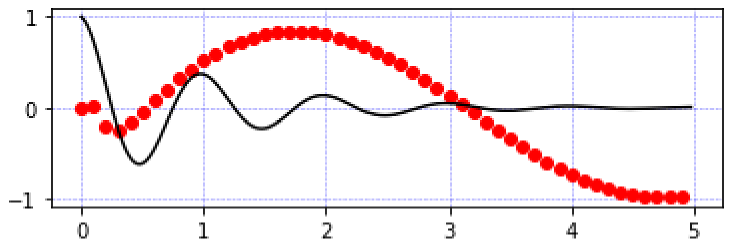
```python
import numpy as np
import matplotlib.pyplot as plt

fig, ax1 = plt.subplots()

x = np.arange(1,7,0.1)
ax1.plot(x, 2 * np.pi * x, lw=2, color="blue")
ax1.set_ylabel(r"Circumference $(cm)$", fontsize=16, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, np.pi * x ** 2, lw=2, color="darkgreen")
ax2.set_ylabel(r"area $(cm^2)$", fontsize=16, color="darkgreen")
for label in ax2.get_yticklabels():
    label.set_color("darkgreen")
```



The following topics are not directly related to subplotting, but we want to present them to round up the introduction into the basic possibilities of matplotlib. The first one shows how to define grid lines and the second one is quite important. It is about saving plots in image files.

GRID LINES

```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)
def g(t):
    return np.sin(t) * np.cos(1/(t+0.1))

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
plt.subplot(212)
plt.plot(t1, g(t1), 'ro', t2, f(t2), 'k')
plt.grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
plt.show()
```



SAVING FIGURES

The savefig method can be used to save figures to a file:

fig.savefig("filename.png")

It is possible to optionally specify the DPI and to choose between different output formats:

fig.savefig("filename.png", dpi=200)

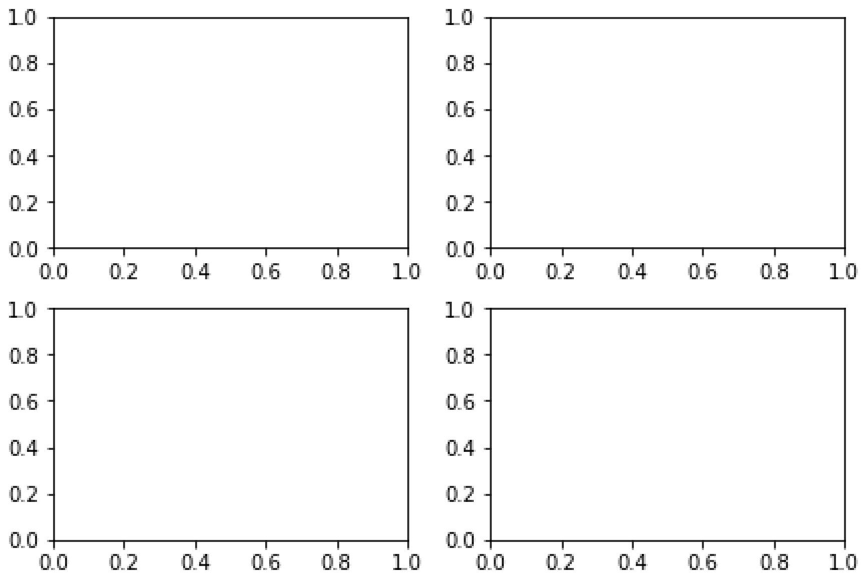
Output can be generated in the formats PNG, JPG, EPS, SVG, PGF and PDF.

MATPLOTLIB TUTORIAL: GRIDSPEC

We have seen in the last chapter of our Python tutorial on Matplotlib how to create a figure with multiple axis or subplot. To create such figures we used the `subplots` function. We will demonstrate in this chapter how the submodule of matplotlib `gridspec` can be used to specify the location of the subplots in a figure. In other words, it specifies the location of the subplots in a given `GridSpec`. `GridSpec` provides us with additional control over the placements of subplots, also the the margins and the spacings between the individual subplots. It also allows us the creation of axes which can spread over multiple grid areas.

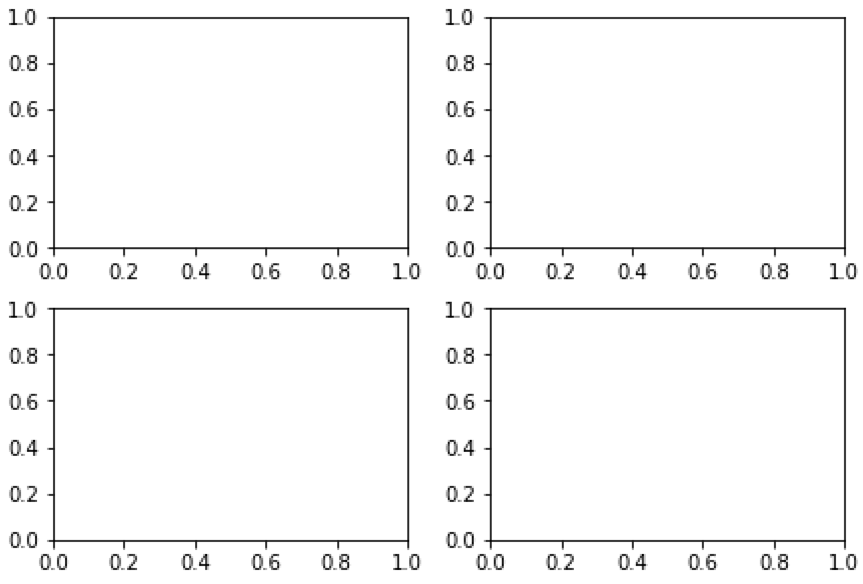
We create a figure and four containing axes in the following code. We have covered this in our previous chapter.

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

fig, axes = plt.subplots(ncols=2, nrows=2, constrained_layout=Tru
e)
```



We will create the previous example now by using GridSpec. At first, we have to create a figure object and after this a GridSpec object. We pass the figure object to the parameter `figure` of `GridSpec`. The axes are created by using `add_subplot`. The elements of the gridspec are accessed the same way as numpy arrays.

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

fig = plt.figure(constrained_layout=True)
spec = gridspec.GridSpec(ncols=2, nrows=2, figure=fig)
ax1 = fig.add_subplot(spec[0, 0])
ax2 = fig.add_subplot(spec[0, 1])
ax3 = fig.add_subplot(spec[1, 0])
ax4 = fig.add_subplot(spec[1, 1])
```



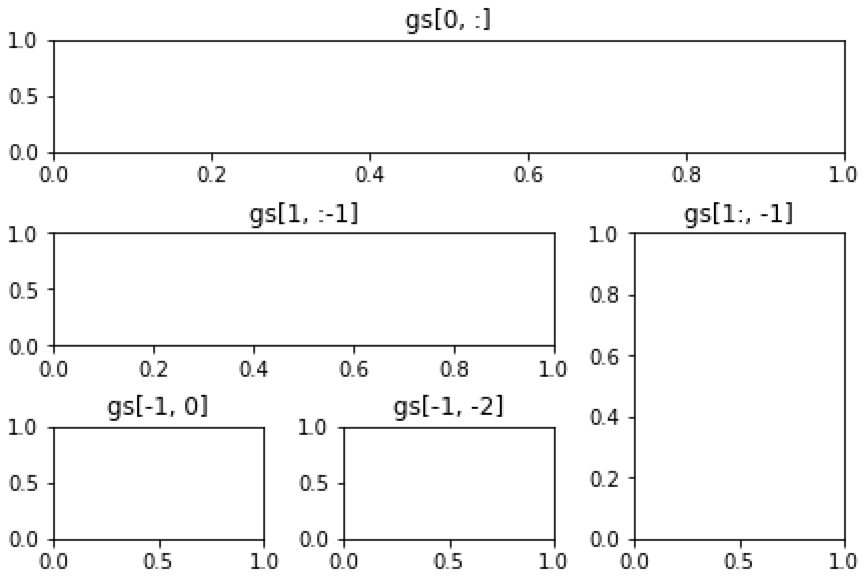
The above example is not a good usecase of `GridSpec`. It does not give us any benefit over the use of `subplots`. In principle, it is only more complicated in this case.

The importance and power of Gridspec unfolds, if we create subplots that span rows and columns.

We show this in the following example:

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

fig = plt.figure(constrained_layout=True)
gs = fig.add_gridspec(3, 3)
ax1 = fig.add_subplot(gs[0, :])
ax1.set_title('gs[0, :]')
ax2 = fig.add_subplot(gs[1, :-1])
ax2.set_title('gs[1, :-1]')
ax3 = fig.add_subplot(gs[1:, -1])
ax3.set_title('gs[1:, -1]')
ax4 = fig.add_subplot(gs[-1, 0])
ax4.set_title('gs[-1, 0]')
ax5 = fig.add_subplot(gs[-1, -2])
ax5.set_title('gs[-1, -2]')
```

Output:
```
Text(0.5, 1.0, 'gs[-1, -2]')
```

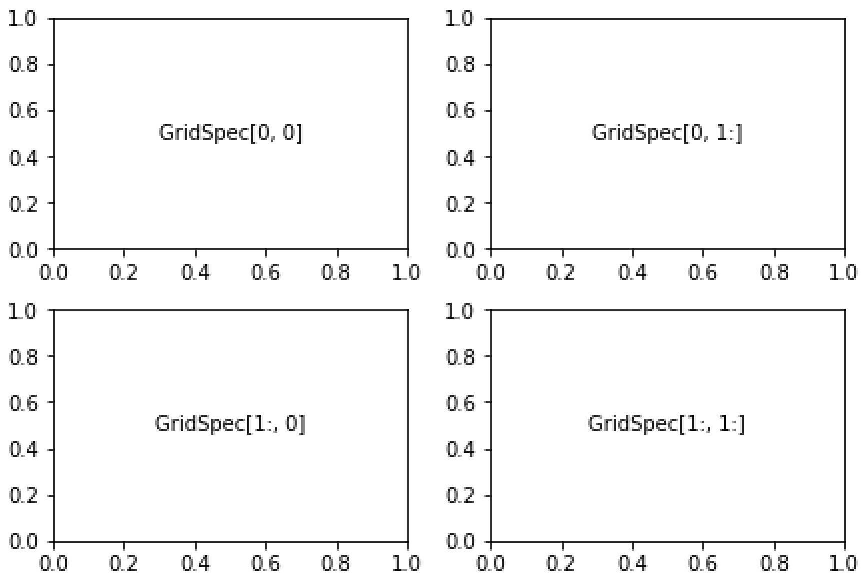


:mod: `~matplotlib.gridspec` is also indispensable for creating subplots of different widths via a couple of methods.

The method shown here is similar to the one above and initializes a uniform grid specification, and then uses numpy indexing and slices to allocate multiple "cells" for a given subplot.

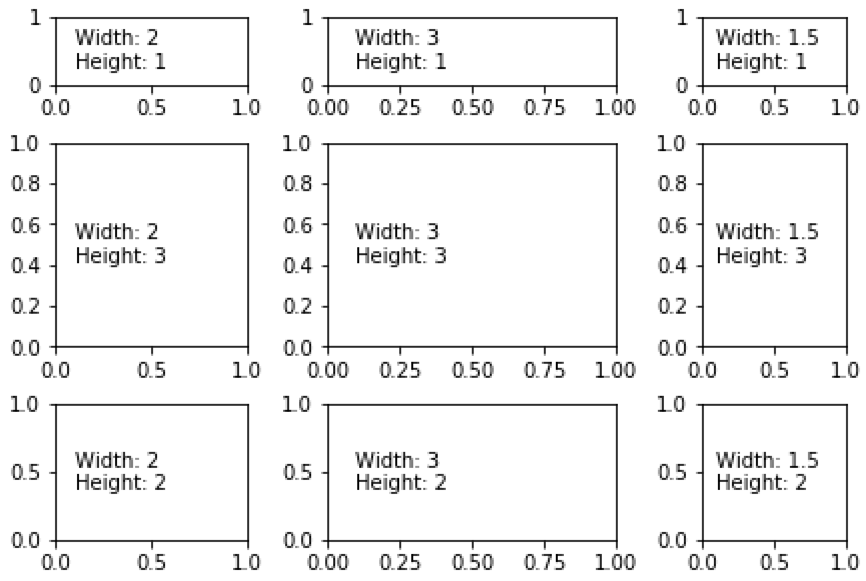
```
fig = plt.figure(constrained_layout=True)
spec = fig.add_gridspec(ncols=2, nrows=2)
optional_params = dict(xy=(0.5, 0.5),
                       xycoords='axes fraction',
                       va='center',
                       ha='center')

ax = fig.add_subplot(spec[0, 0])
ax.annotate('GridSpec[0, 0]', **optional_params)
fig.add_subplot(spec[0, 1]).annotate('GridSpec[0, 1:]', **optiona
l_params)
fig.add_subplot(spec[1, 0]).annotate('GridSpec[1:, 0]', **optiona
l_params)
fig.add_subplot(spec[1, 1]).annotate('GridSpec[1:, 1:]', **optiona
l_params)
```

Output:
```
Text(0.5, 0.5, 'GridSpec[1:, 1:]')
```



Another option is to use the `width_ratios` and `height_ratios` parameters. These keyword arguments are lists of numbers. Note that absolute values are meaningless, only their relative ratios matter. That means that `width_ratios=[2, 4, 8]` is equivalent to `width_ratios=[1, 2, 4]` within equally wide figures. For the sake of demonstration, we'll blindly create the axes within `for` loops since we won't need them later.

```
fig5 = plt.figure(constrained_layout=True)
widths = [2, 3, 1.5]
heights = [1, 3, 2]
spec5 = fig5.add_gridspec(ncols=3, nrows=3, width_ratios=widths,
                          height_ratios=heights)
for row in range(3):
    for col in range(3):
        ax = fig5.add_subplot(spec5[row, col])
        label = 'Width: {}\nHeight: {}'.format(widths[col], height
s[row])
        ax.annotate(label, (0.1, 0.5), xycoords='axes fraction', v
a='center')
```

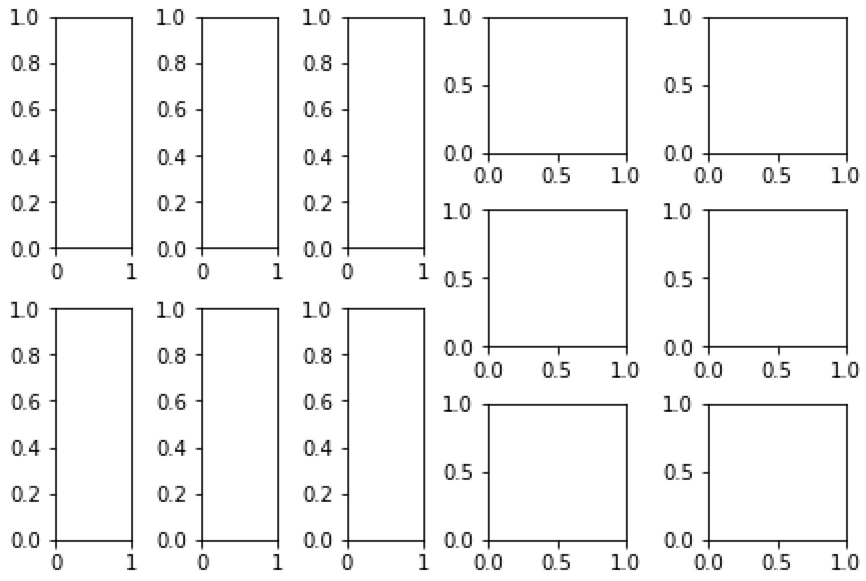


GridSpec using SubplotSpec

You can create GridSpec from the :class: `~matplotlib.gridspec.SubplotSpec`, in which case its layout parameters are set to that of the location of the given SubplotSpec.

Note this is also available from the more verbose `.gridspec.GridSpecFromSubplotSpec`.

```
fig10 = plt.figure(constrained_layout=True)
gs0 = fig10.add_gridspec(1, 2)

gs00 = gs0[0].subgridspec(2, 3)
gs01 = gs0[1].subgridspec(3, 2)

for a in range(2):
    for b in range(3):
        fig10.add_subplot(gs00[a, b])
        fig10.add_subplot(gs01[b, a])
```

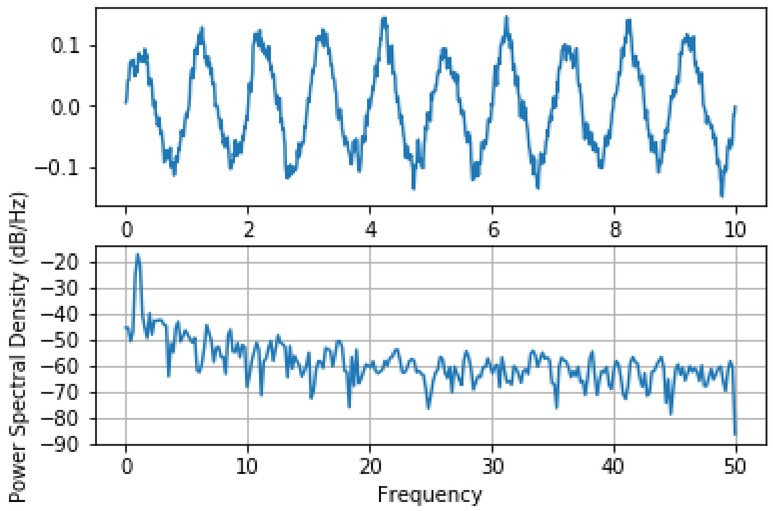


PSD DEMO

Plotting Power Spectral Density (PSD) in Matplotlib.

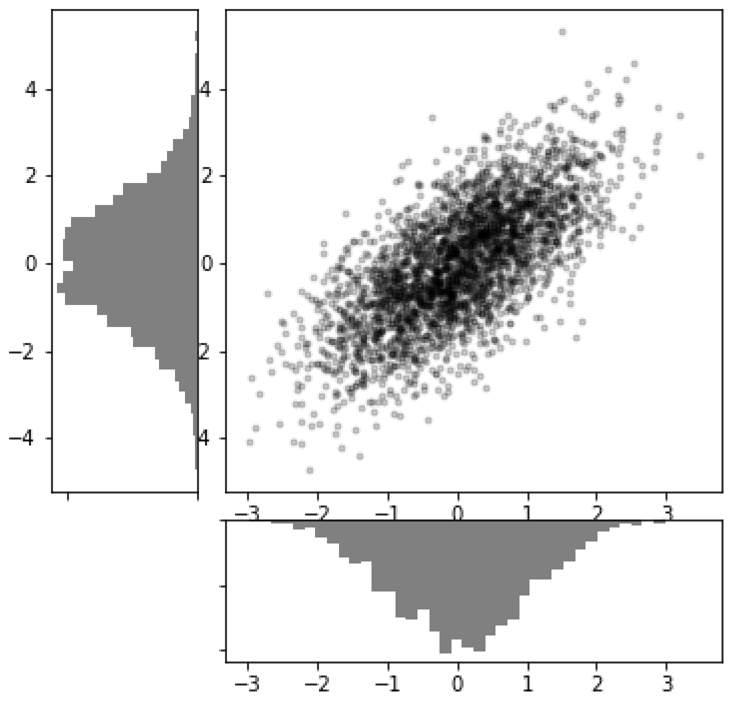
The PSD is a common plot in the field of signal processing. NumPy has many useful libraries for computing a PSD. Below we demo a few examples of how this can be accomplished and visualized with Matplotlib.

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.gridspec as gridspec

# Fixing random state for reproducibility
np.random.seed(42)

dt = 0.01
t = np.arange(0, 10, dt)
nse = np.random.randn(len(t))
r = np.exp(-t / 0.05)

cnse = np.convolve(nse, r) * dt
cnse = cnse[:len(t)]
s = 0.1 * np.sin(2 * np.pi * t) + cnse

plt.subplot(211)
plt.plot(t, s)
plt.subplot(212)
plt.psd(s, 512, 1 / dt)
plt.show()
```

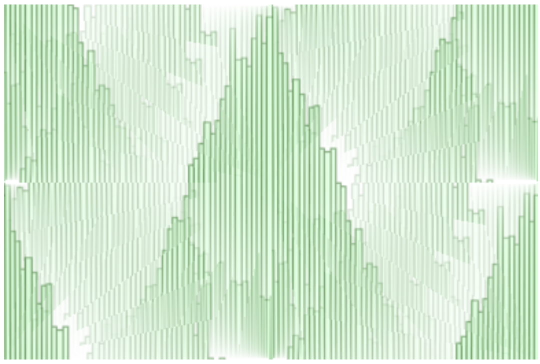


This type of flexible grid alignment has a wide range of uses. I most often use it when creating multi-axes histogram plots like the ones shown here:

```
# Create some normally distributed data
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 3000).T

# Set up the axes with gridspec
fig = plt.figure(figsize=(6, 6))
grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)
main_ax = fig.add_subplot(grid[:-1, 1:])
y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=mai
n_ax)
x_hist = fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=mai
n_ax)

# scatter points on the main axes
main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)

# histogram on the attached axes
x_hist.hist(x, 40, histtype='stepfilled',
            orientation='vertical', color='gray')
x_hist.invert_yaxis()

y_hist.hist(y, 40, histtype='stepfilled',
                 orientation='horizontal', color='gray')
y_hist.invert_xaxis()
```



MATPLOTLIB TUTORIAL: HISTOGRAMS AND BAR PLOTS



It's hard to imagine that you open a newspaper or magazin without seeing some bar charts or histograms telling you about the number of smokers in certain age groups, the number of births per year and so on. It's a great way to depict facts without having to use too many words, but on the downside they can be used to manipulate or lie with statistics as well. They provide us with quantitative information on a wide range of topics. Bar charts and column charts clearly show us the ranking of our top politicians. They also inform about consequences of certain behavior: smoking or not smoking. Advantages and disadvantages of various activities. Income distributions and so on. On the one hand, they serve as a source of information for us to see our own thinking and acting in statistical comparison with others, on the other hand they also - by perceiving them - change our thinking and acting in many cases.

However, we are primarily interested in how to create charts and histograms in this chapter. A splendid way to create such charts consists in using Python in combination with Matplotlib.

What is a histogram? A formal definition can be: It's a graphical representation of a frequency distribution of some numerical data. Rectangles with equal width have heights with the associated frequencies.

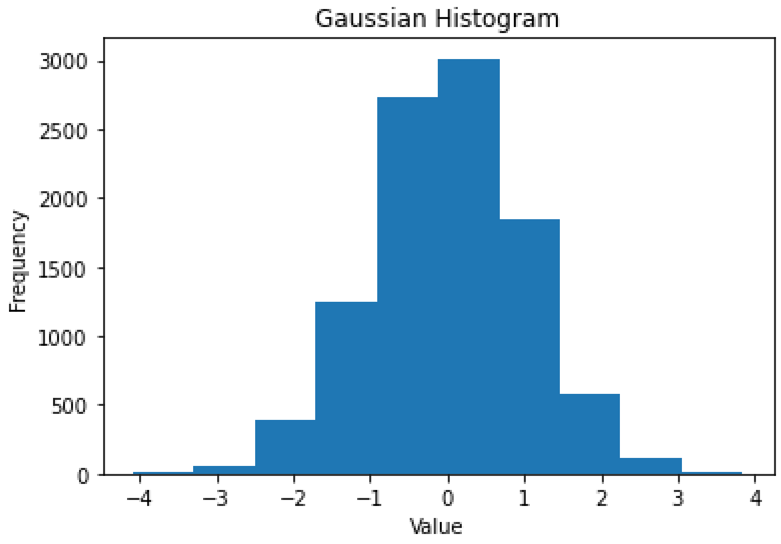
If we construct a histogram, we start with distributing the range of possible x values into usually equal sized and adjacent intervals or bins.

We start now with a practical Python program. We create a histogram with random numbers:

```
import matplotlib.pyplot as plt
import numpy as np
gaussian_numbers = np.random.normal(size=10000)
gaussian_numbers
```

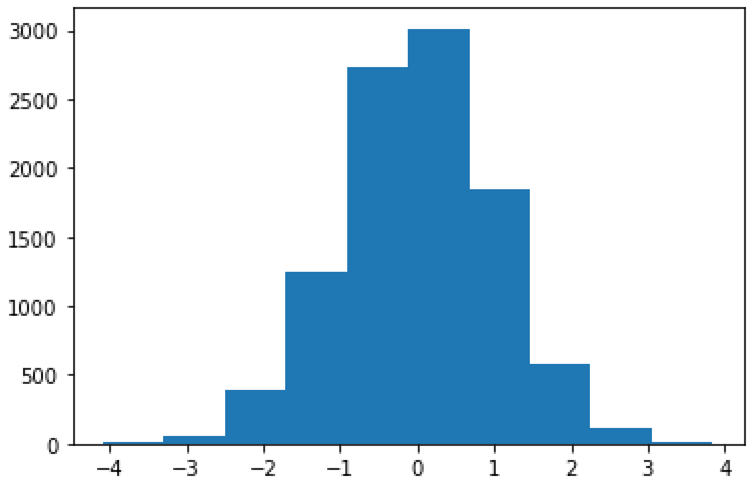
Output:
```
array([-0.80404388, -2.77483865, -2.26510843, ..., -0.698508
2 ,
        1.21066701,  0.4700107 ])
```

```
plt.hist(gaussian_numbers)
plt.title("Gaussian Histogram")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```



We have seen that the function hist (actually matplotlib.pyplot.hist) computes the histogram values and plots the graph. It also returns a tuple of three objects (n, bins, patches):

```
n, bins, patches = plt.hist(gaussian_numbers)
```



n[i] contains the number of values of gaussian numbers that lie within the interval with the boundaries bins [i] and bins [i + 1]:

```
print("n: ", n, sum(n))
```

```
n:  [   5.    61.   385. 1254. 2728. 3012. 1839.   585.   115.    16.]
10000.0
```

So n is an array of frequencies. The last return value of hist is a list of patches, which corresponds to the rectangles with their properties:

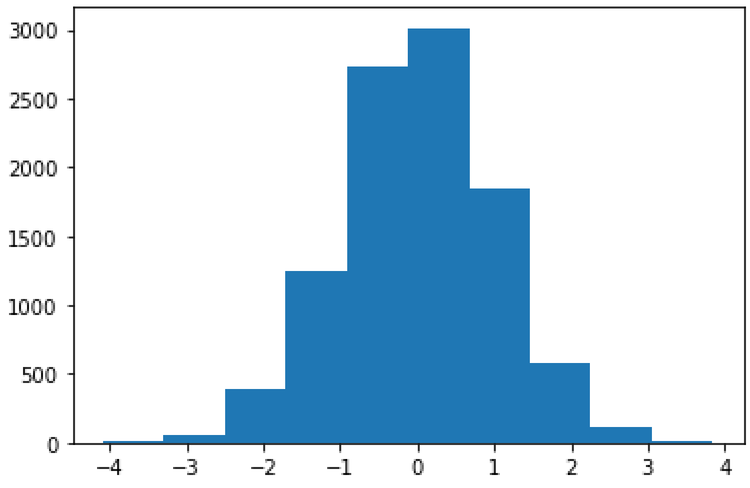
```
print("patches: ", patches)
for i in range(10):
    print(patches[i])
```

```
patches:  <BarContainer object of 10 artists>
Rectangle(xy=(-4.08653, 0), width=0.79308, height=5, angle=0)
Rectangle(xy=(-3.29345, 0), width=0.79308, height=61, angle=0)
Rectangle(xy=(-2.50037, 0), width=0.79308, height=385, angle=0)
Rectangle(xy=(-1.70729, 0), width=0.79308, height=1254, angle=0)
Rectangle(xy=(-0.914206, 0), width=0.79308, height=2728, angle=0)
Rectangle(xy=(-0.121126, 0), width=0.79308, height=3012, angle=0)
Rectangle(xy=(0.671954, 0), width=0.79308, height=1839, angle=0)
Rectangle(xy=(1.46503, 0), width=0.79308, height=585, angle=0)
Rectangle(xy=(2.25811, 0), width=0.79308, height=115, angle=0)
Rectangle(xy=(3.05119, 0), width=0.79308, height=16, angle=0)
```

Let's take a closer look at the return values. To create the histogram array gaussian_numbers are divided into equal intervals, i.e. the "bins". The interval limits calculated by hist are obtained in the second component of the return tuple. In our example, they are denoted by the variable bins:

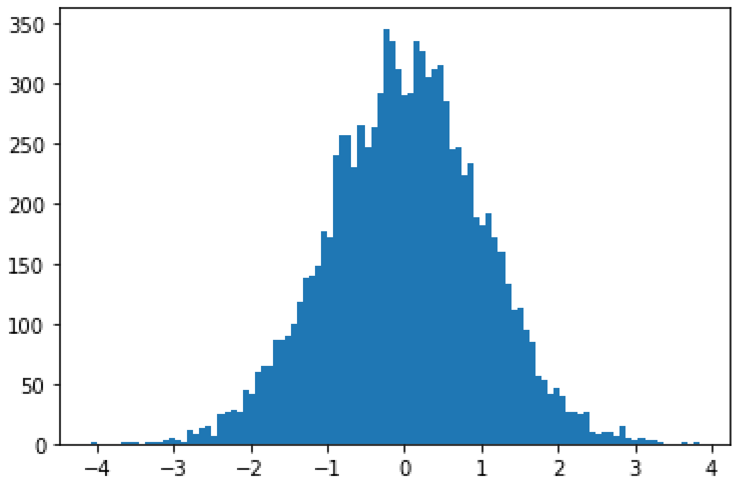
```
n, bins, patches = plt.hist(gaussian_numbers)
print("n: ", n, sum(n))
print("bins: ", bins)
for i in range(len(bins)-1):
    print(bins[i+1] -bins[i])
print("patches: ", patches)
print(patches[1])
print(patches[2])
```

```
n:  [   5.   61.  385. 1254. 2728. 3012. 1839.  585.  115.   16.]
10000.0
bins:  [-4.08652622 -3.29344612 -2.50036601 -1.7072859  -0.914205
8  -0.12112569
  0.67195442  1.46503452  2.25811463  3.05119474  3.84427484]
0.7930801067329991
0.7930801067329987
0.7930801067329987
0.7930801067329991
0.7930801067329991
0.7930801067329982
0.7930801067329991
0.7930801067329991
0.7930801067329991
0.7930801067329982
patches:  <BarContainer object of 10 artists>
Rectangle(xy=(-3.29345, 0), width=0.79308, height=61, angle=0)
Rectangle(xy=(-2.50037, 0), width=0.79308, height=385, angle=0)
```



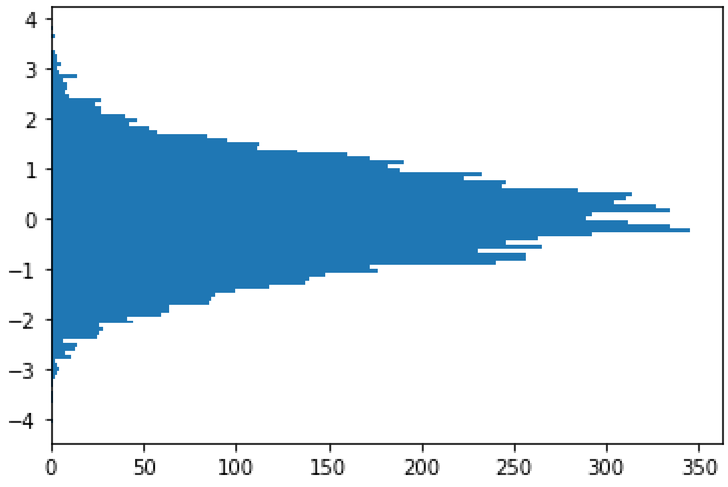
Let's increase the number of bins. 10 bins is not a lot, if you imagine, that we have 10,000 random values. To do so, we set the keyword parameter bins to 100:

```
plt.hist(gaussian_numbers, bins=100)
plt.show()
```



Indem wir den Parameter orientation auf vertical setzen, können wir das Histogramm auch seitwärts ausgeben:

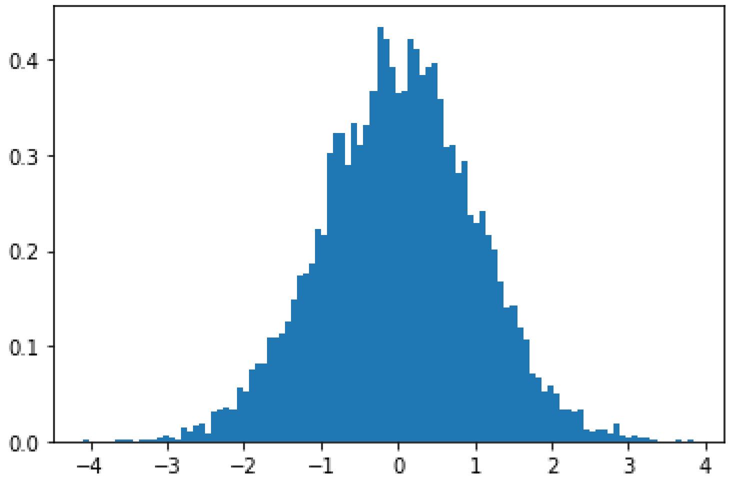
```
plt.hist(gaussian_numbers,
         bins=100,
         orientation="horizontal")
plt.show()
```



Another important keyword parameter of hist is density, which replaces the deprecated normed parameter. If set to true, the first component - that is, the frequencies - of the return tuple is normalized to form a probability density, i. the area (or the integral) under the histogram makes the sum 1

```
n, bins, patches = plt.hist(gaussian_numbers,
                            bins=100,
                            density=True)
plt.show()
```

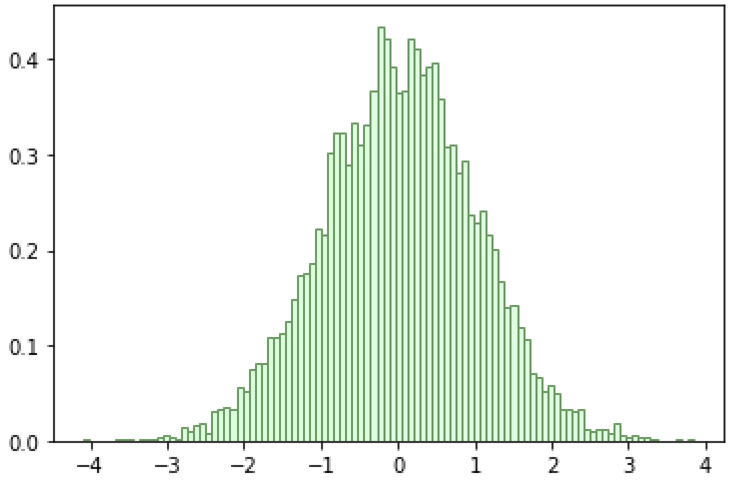
```
print("Area below the integral: ", np.sum(n * np.diff(bins)))
```



```
Area below the integral:  1.0
```

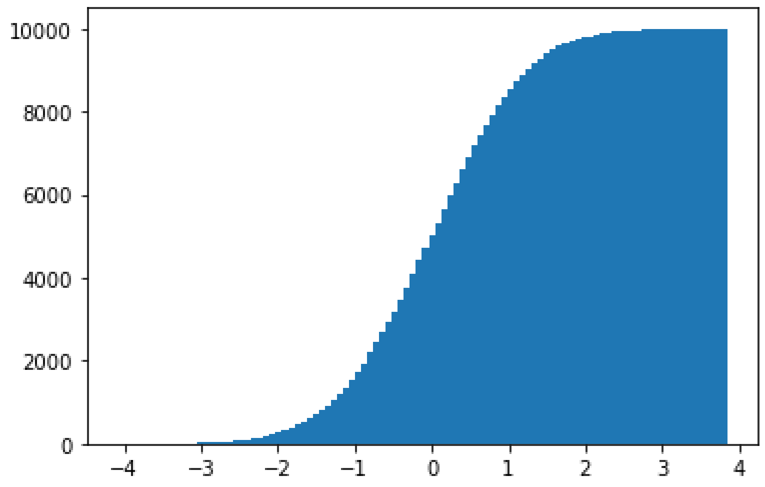
If both the parameters 'density' and 'stacked' are set to 'True', the sum of the histograms is normalized to 1. With the parameters edgecolor and color we can define the line color and the color of the surfaces:

```
plt.hist(gaussian_numbers,
         bins=100,
         density=True,
         stacked=True,
         edgecolor="#6A9662",
         color="#DDFFDD")
plt.show()
```



Okay, you want to see the data depicted as a plot of cumulative values? We can plot it as a cumulative distribution function by setting the parameter 'cumulative'.

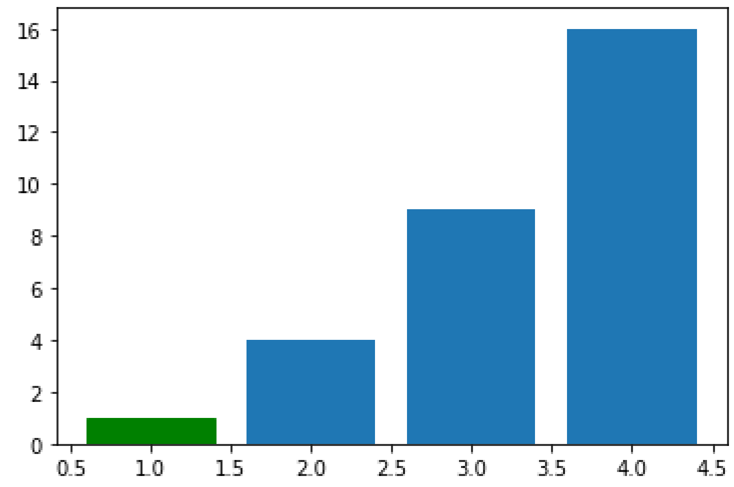
```
plt.hist(gaussian_numbers,
         bins=100,
         stacked=True,
         cumulative=True)

plt.show()
```



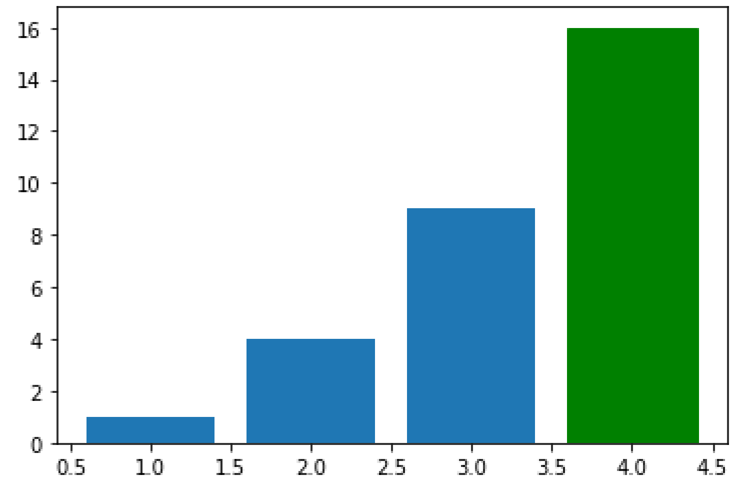
BAR PLOTS

Now we come to one of the most commonly used chart types, well known even among non-scientists. A bar chart is composed of rectangles that are perpendicular to the x-axis and that rise up like columns. The width of the rectangles has no mathematical meaning.

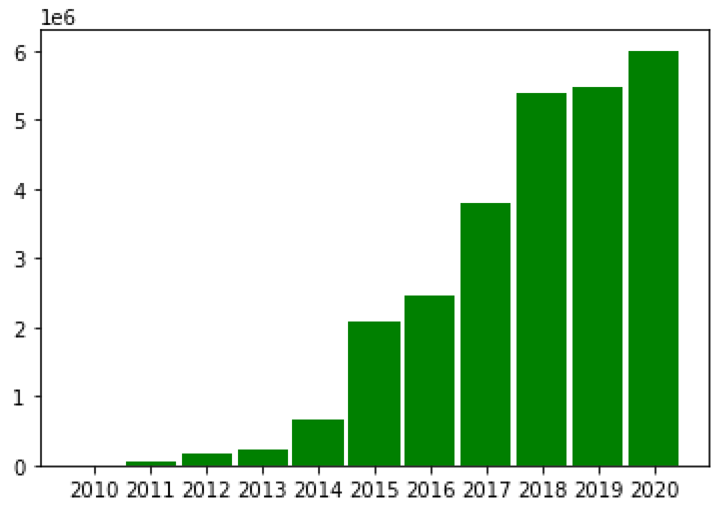
```
bars = plt.bar([1, 2, 3, 4], [1, 4, 9, 16])
bars[0].set_color('green')
plt.show()
```



```
f=plt.figure()
ax=f.add_subplot(1,1,1)
ax.bar([1,2,3,4], [1,4,9,16])
children = ax.get_children()
children[3].set_color('g')
```

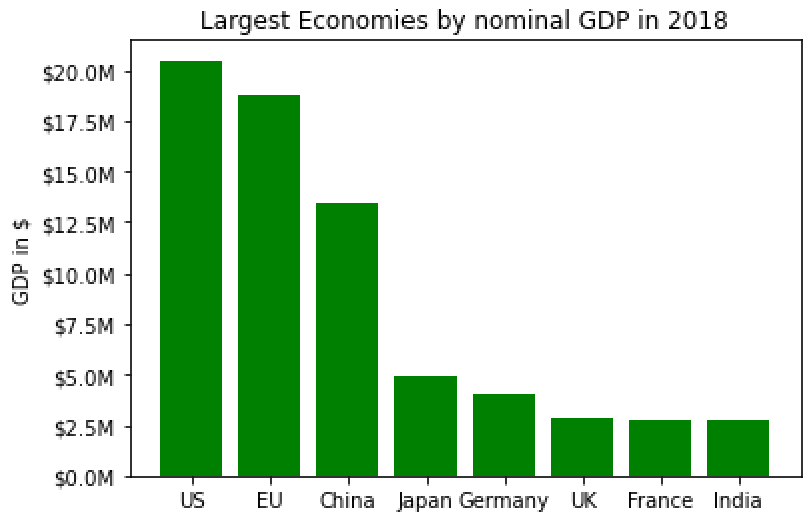


```
import matplotlib.pyplot as plt
import numpy as np
years = [str(year) for year in range(2010, 2021)]
visitors = (1241, 50927, 162242, 222093, 665004,
            2071987, 2460407, 3799215, 5399000, 5475016, 6003672)
index = np.arange(len(years))
bar_width = 0.9
plt.bar(index, visitors, bar_width,  color="green")
plt.xticks(index, years) # labels get centered
plt.show()
```



BARPLOTS WITH CUSTOMIZED TICKS

```
from matplotlib.ticker import FuncFormatter
import matplotlib.pyplot as plt
import numpy as np

def millions(x, pos):
    'The two args are the value and tick position'
    #return '$%1.1fM' % (x * 1e-6)
    return f'${x * 1e-6:1.1f}M'
formatter = FuncFormatter(millions)

years = ('US', 'EU', 'China', 'Japan',
          'Germany', 'UK', 'France', 'India')
GDP = (20494050, 18750052, 13407398, 4971929,
       4000386, 2828644, 2775252, 2716746)

fig, ax = plt.subplots()
ax.yaxis.set_major_formatter(formatter)
ax.bar(x=np.arange(len(GDP)), # The x coordinates of the bars.
       height=GDP, # the height(s) of the vars
       color="green",
       align="center",
       tick_label=years)
ax.set_ylabel('GDP in $')
ax.set_title('Largest Economies by nominal GDP in 2018')
plt.show()
```

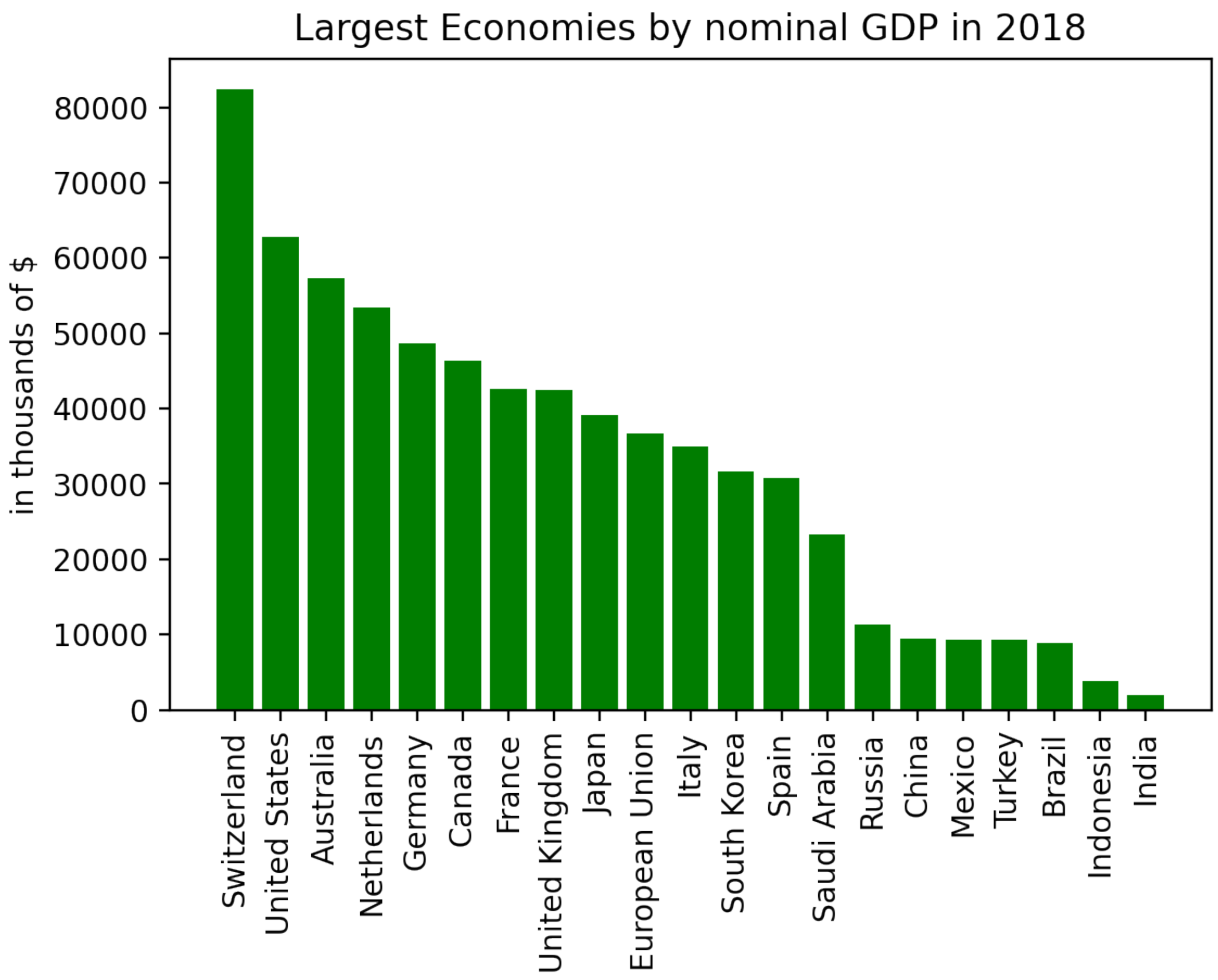


The file 'data/GDP.txt' contains a listing of countries, GDP and the population numbers of 2018 in the following format:

1 United States 20,494,050 326,766,748
— European Union 18,750,052 511,522,671 2 China 13,407,398 1,415,045,928 3 Japan 4,971,929 127,185,332
4 Germany 4,000,386 82,293,457
5 United Kingdom 2,828,644 66,573,504 6 France 2,775,252 65,233,271
7 India 2,716,746 1,354,051,854 8 Italy 2,072,201 59,290,969 9 Brazil 1,868,184 210,867,954
10 Canada 1,711,387 36,953,765
11 Russia 1,630,659 143,964,709 12 South Korea 1,619,424 51,164,435 13 Spain 1,425,865 46,397,452
14 Australia 1,418,275 24,772,247
15 Mexico 1,223,359 130,759,074 16 Indonesia 1,022,454 266,794,980 17 Netherlands 912,899 17,084,459
18 Saudi Arabia 782,483 33,554,343 19 Turkey 766,428 81,916,871
20 Switzerland 703,750 8,544,034

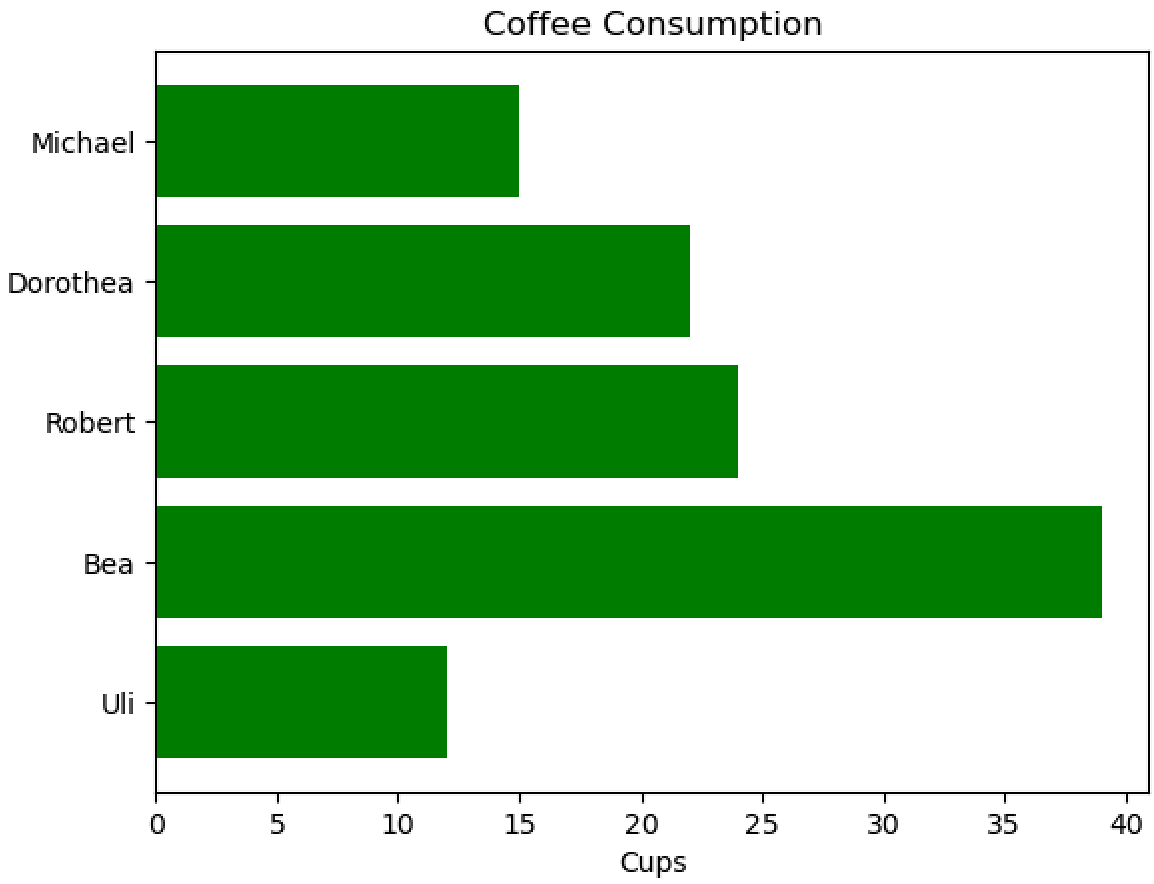
Create a bar plot with the per capita nominal GDP.

```
import matplotlib.pyplot as plt
import numpy as np

land_GDP_per_capita = []
with open('data/GDP.txt') as fh:
    for line in fh:
        index, *land, gdp, population = line.split()
        land = " ".join(land)
        gdp = int(gdp.replace(',', ''))
        population = int(population.replace(',', ''))
        per_capita = int(round(gdp * 1000000 / population, 0))
        land_GDP_per_capita.append((land, per_capita))
land_GDP_per_capita.sort(key=lambda x: x[1], reverse=True)
countries, GDP_per_capita = zip(*land_GDP_per_capita)

fig = plt.figure(figsize=(6,5), dpi=200)
left, bottom, width, height = 0.1, 0.3, 0.8, 0.6
ax = fig.add_axes([left, bottom, width, height])

ax.bar(x=np.arange(len(GDP_per_capita)), # The x coordinates of th
e bars.
       height=GDP_per_capita, # the height(s) of the vars
       color="green", align="center",
       tick_label=countries)
ax.set_ylabel('in thousands of $')
#ax.set_xticks(rotation='vertical')
ax.set_title('Largest Economies by nominal GDP in 2018')
plt.xticks(rotation=90)
plt.show()
```



VERTICAL BAR CHARTS (LINE CHARTS)

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.pyplot as plt
# restore default parameters:
plt.rcdefaults()
fig, ax = plt.subplots()
personen = ('Michael', 'Dorothea', 'Robert', 'Bea', 'Uli')
y_pos = np.arange(len(personen))
cups = (15, 22, 24, 39, 12)
ax.barh(y_pos, cups, align='center',
        color='green', ecolor='black')
ax.set_yticks(y_pos)
ax.set_yticklabels(personen)
ax.invert_yaxis()
ax.set_xlabel('Cups')
ax.set_title('Coffee Consumption')
plt.show()
```

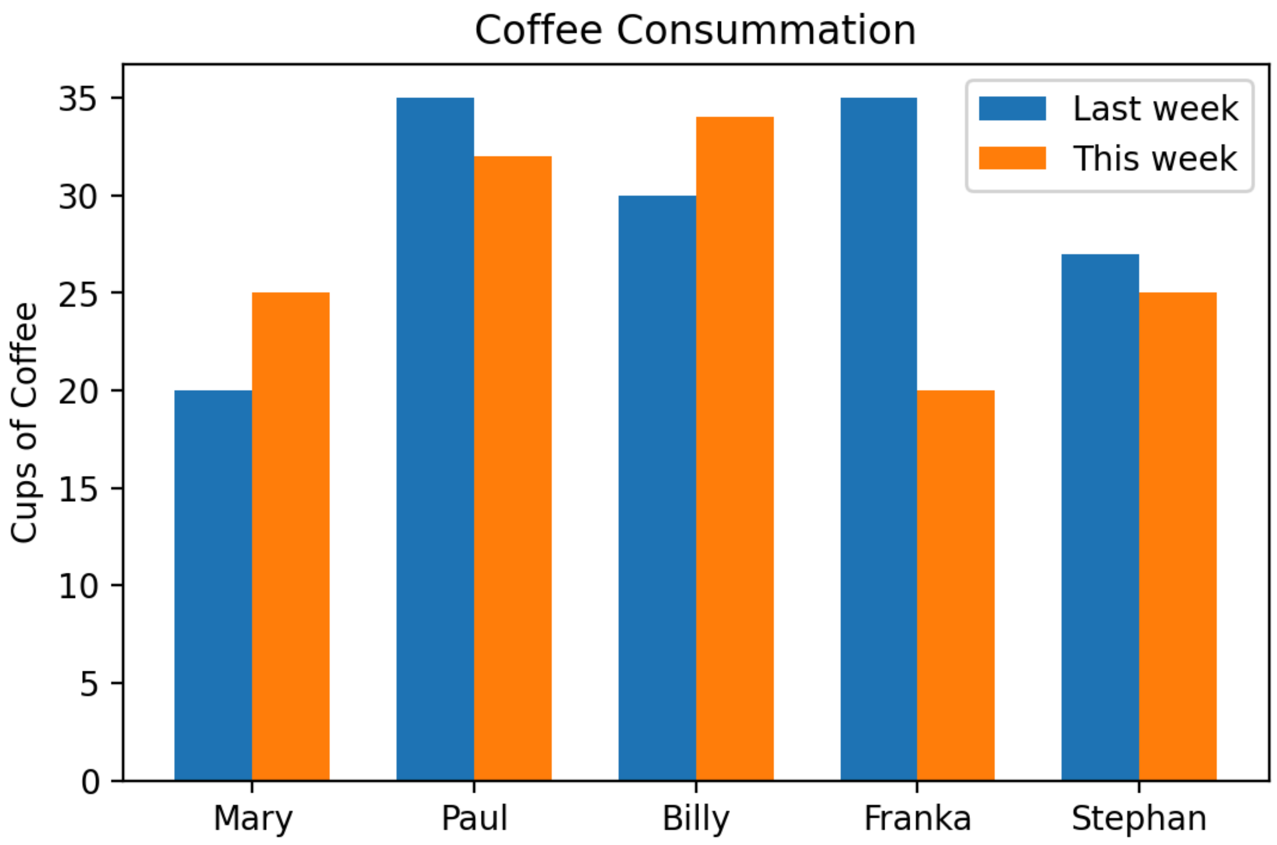


GROUPED BAR CHARTS

So far we used in our bar plots for each categorical group one bar. I.e., for each country in the previous example we had one bar for the "per capita GDP" of one year. We could think of a graph representing these values for different years. This can be accomplished with grouped bar charts. A grouped bar chart contains two or more bars for each categorical group. These bars are color-coded to represent a particular grouping. For example, a business owner, running a production line with two main products might make a grouped bar chart with different colored bars to represent each product. The horizontal axis would show the months of the year and the vertical axis would show the revenue.

```
import matplotlib.pyplot as plt
import numpy as np

last_week_cups = (20, 35, 30, 35, 27)
this_week_cups = (25, 32, 34, 20, 25)
names = ['Mary', 'Paul', 'Billy', 'Franka', 'Stephan']
```

```
fig = plt.figure(figsize=(6,5), dpi=200)
left, bottom, width, height = 0.1, 0.3, 0.8, 0.6
ax = fig.add_axes([left, bottom, width, height])

width = 0.35
ticks = np.arange(len(names))
ax.bar(ticks, last_week_cups, width, label='Last week')
ax.bar(ticks + width, this_week_cups, width, align="center",
    label='This week')

ax.set_ylabel('Cups of Coffee')
ax.set_title('Coffee Consummation')
ax.set_xticks(ticks + width/2)
ax.set_xticklabels(names)

ax.legend(loc='best')
plt.show()
```

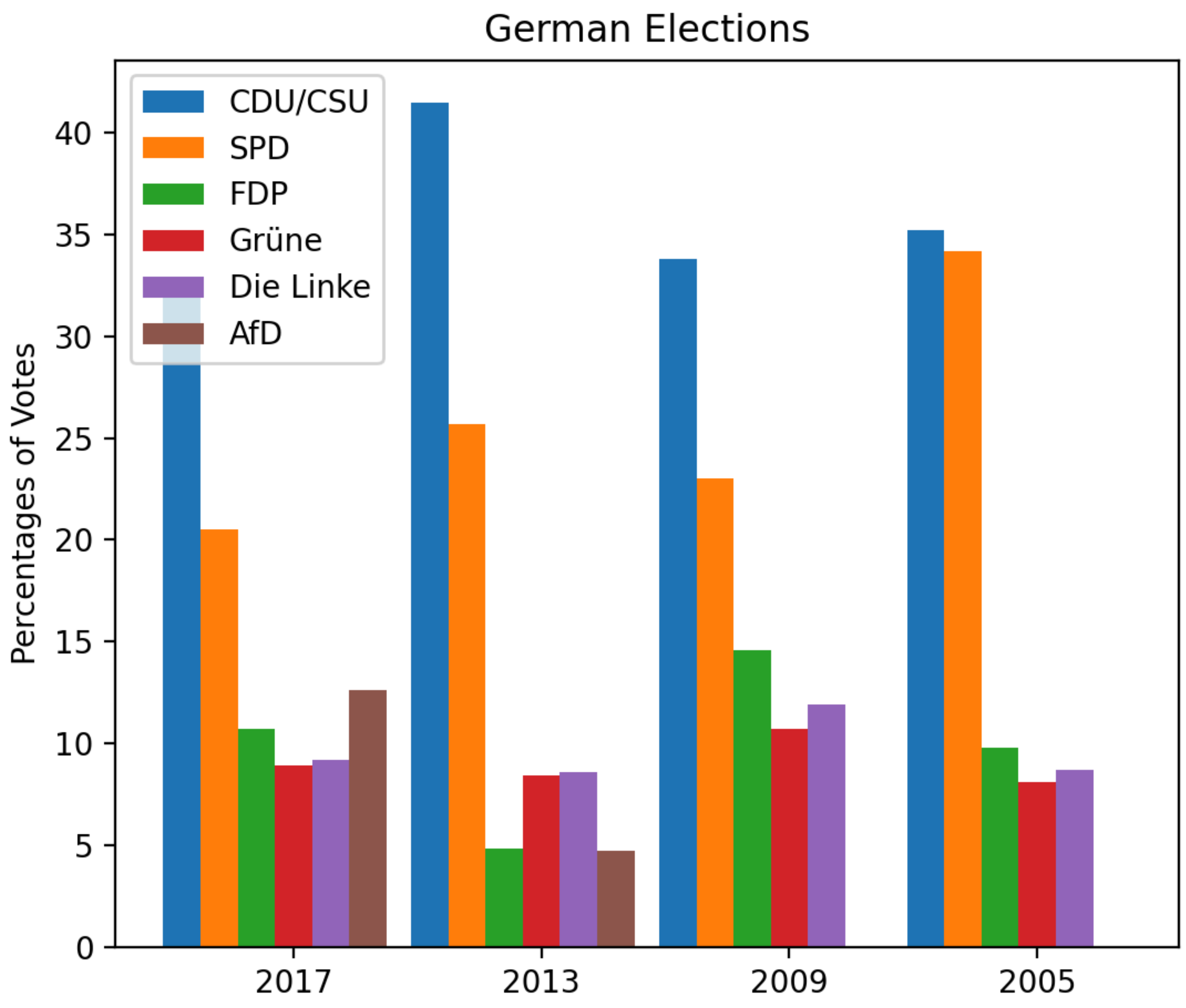


EXERCISE

The file data/german_election_results.txt contains four election result of Germany.

Create a bar chart graph with data.

```
import matplotlib.pyplot as plt
import numpy as np

parties = ('CDU/CSU', 'SPD', 'FDP', 'Grüne', 'Die Linke', 'AfD')

election_results_per_year = {}
with open('data/german_election_results.txt') as fh:
    fh.readline()
    for line in fh:
        year, *results = line.rsplit()
        election_results_per_year[year] = [float(x) for x in resul
ts]


election_results_per_party = list(zip(*election_results_per_year.v
alues()))

fig = plt.figure(figsize=(6,5), dpi=200)
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
ax = fig.add_axes([left, bottom, width, height])

years = list(election_results_per_year.keys())
width = 0.9 / len(parties)
ticks = np.arange(len(years))

for index, party in enumerate(parties):
    ax.bar(ticks+index*width, election_results_per_party[index], w
idth, label=party)

ax.set_ylabel('Percentages of Votes')
ax.set_title('German Elections')
ax.set_xticks(ticks + 0.45)
ax.set_xticklabels(years)

ax.legend(loc='best')
plt.show()
```



We change the previous code by adding the following dictionary:

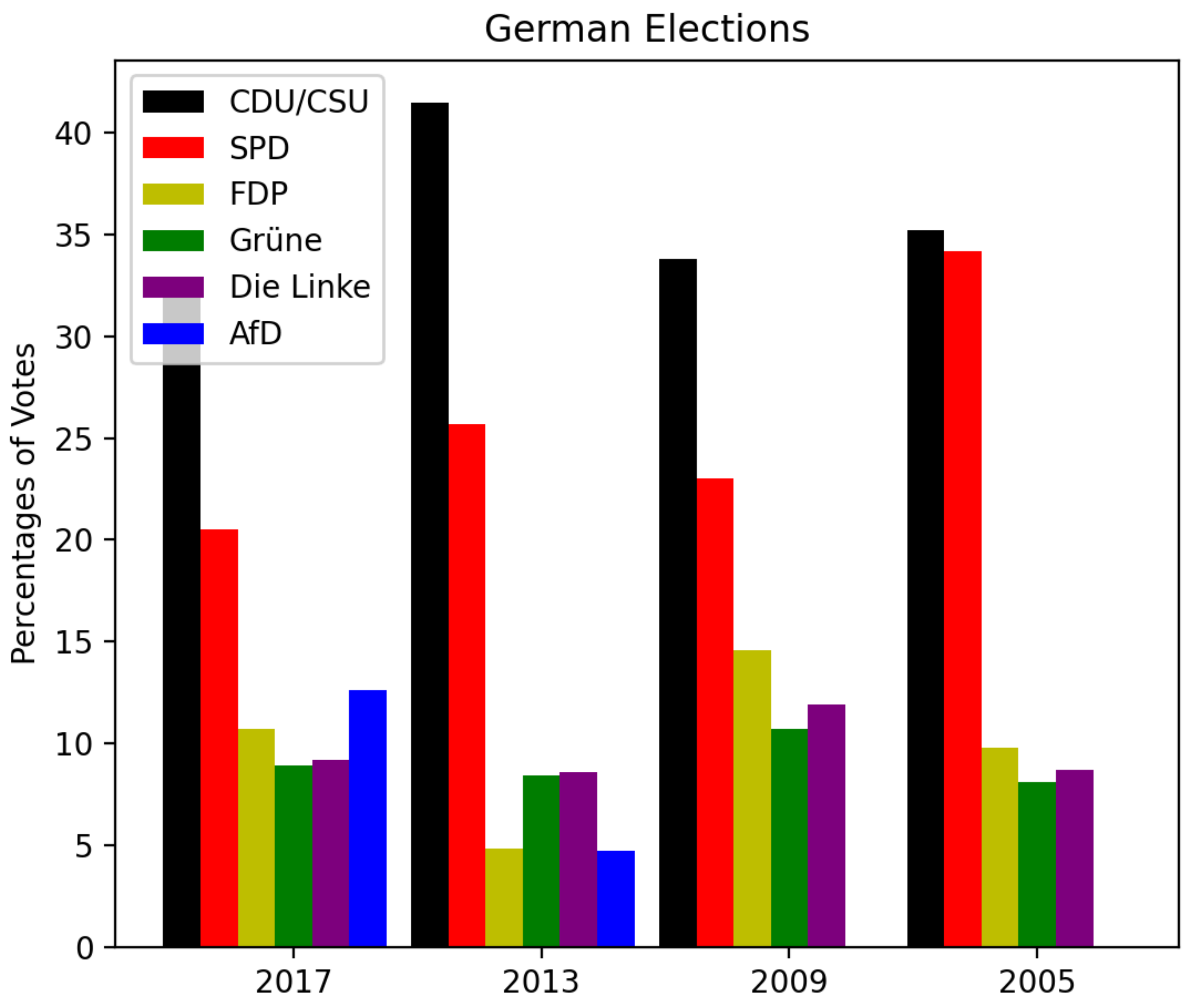
```
colors = {'CDU/CSU': "black", 'SPD': "r", 'FDP': "y",
           'Grüne': "green", 'Die Linke': "purple", 'AfD': "blue"}
```

We also change the creation of the bar code by assigning a color value to the parameter ‚color‘:

```
for index, party in enumerate(parties):
        ax.bar(ticks+index*width,
           election_results_per_party[index],
           width,
           label=party,
           color=colors[parties[index]])
```

Now the complete program with the customized colors:

```
import matplotlib.pyplot as plt
import numpy as np

parties = ('CDU/CSU', 'SPD', 'FDP', 'Grüne', 'Die Linke', 'AfD')
colors = {'CDU/CSU': "black", 'SPD': "r", 'FDP': "y",
           'Grüne': "green", 'Die Linke': "purple", 'AfD': "blue"}

election_results_per_year = {}
with open('data/german_election_results.txt') as fh:
    fh.readline()
    for line in fh:
        year, *results = line.rsplit()
        election_results_per_year[year] = [float(x) for x in resul
ts]


election_results_per_party = list(zip(*election_results_per_year.v
alues()))

fig = plt.figure(figsize=(6,5), dpi=200)
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
ax = fig.add_axes([left, bottom, width, height])

years = list(election_results_per_year.keys())
width = 0.9 / len(parties)
ticks = np.arange(len(years))

for index, party in enumerate(parties):
        ax.bar(ticks+index*width,
           election_results_per_party[index],
           width,
           label=party,
           color=colors[parties[index]])

ax.set_ylabel('Percentages of Votes')
ax.set_title('German Elections')
ax.set_xticks(ticks + 0.45)
ax.set_xticklabels(years)

ax.legend(loc='best')
plt.show()
```



STACKED BAR CHARS

As an alternative to grouped bar charts stacked bar charts can be used.

The stacked bar chart stacks bars that represent different groups on top of each other. The height of the resulting bar shows the combined result or summation of the individual groups.

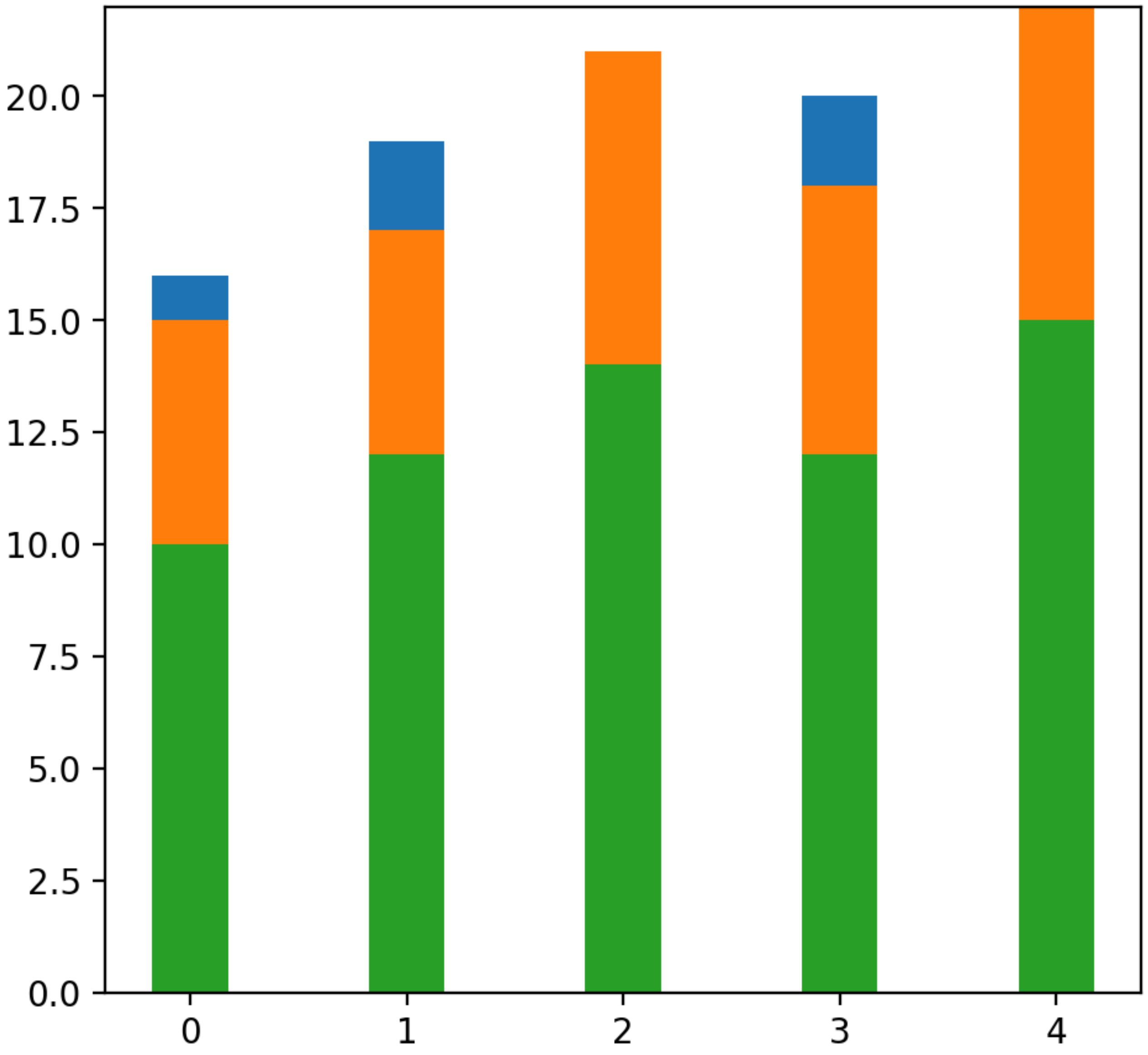
Stacked bar charts are great to depict the total and at the same time providing a view of how the single parts are related to the sum.

Stacked bar charts are not suited for datasets where some groups have negative values. In such cases, grouped bar charts are the better choice.

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

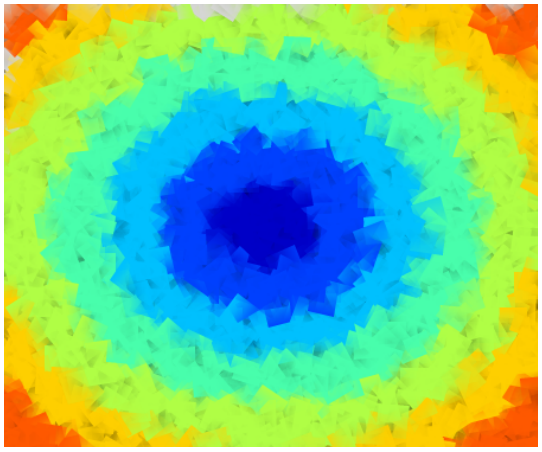
```python
import numpy as np

coffee = np.array([5, 5, 7, 6, 7])
tea = np.array([1, 2, 0, 2, 0])
water = np.array([10, 12, 14, 12, 15])
names = ['Mary', 'Paul', 'Billy', 'Franka', 'Stephan']

fig = plt.figure(figsize=(6,5), dpi=200)
left, bottom, width, height = 0.2, 0.1, 0.7, 0.8
ax = fig.add_axes([left, bottom, width, height])

width = 0.35
ticks = np.arange(len(names))
ax.bar(ticks, tea, width, label='Coffee', bottom=water+coffee)
ax.bar(ticks, coffee, width, align="center", label='Tea',
       bottom=water)
ax.bar(ticks, water, width, align="center", label='Water')
```

Output:
```
<BarContainer object of 5 artists>
```



MATPLOTLIB TUTORIAL: CONTOUR PLOTS

CONTOUR PLOT



A contour line or isoline of a function of two variables is a curve along which the function has a constant value.

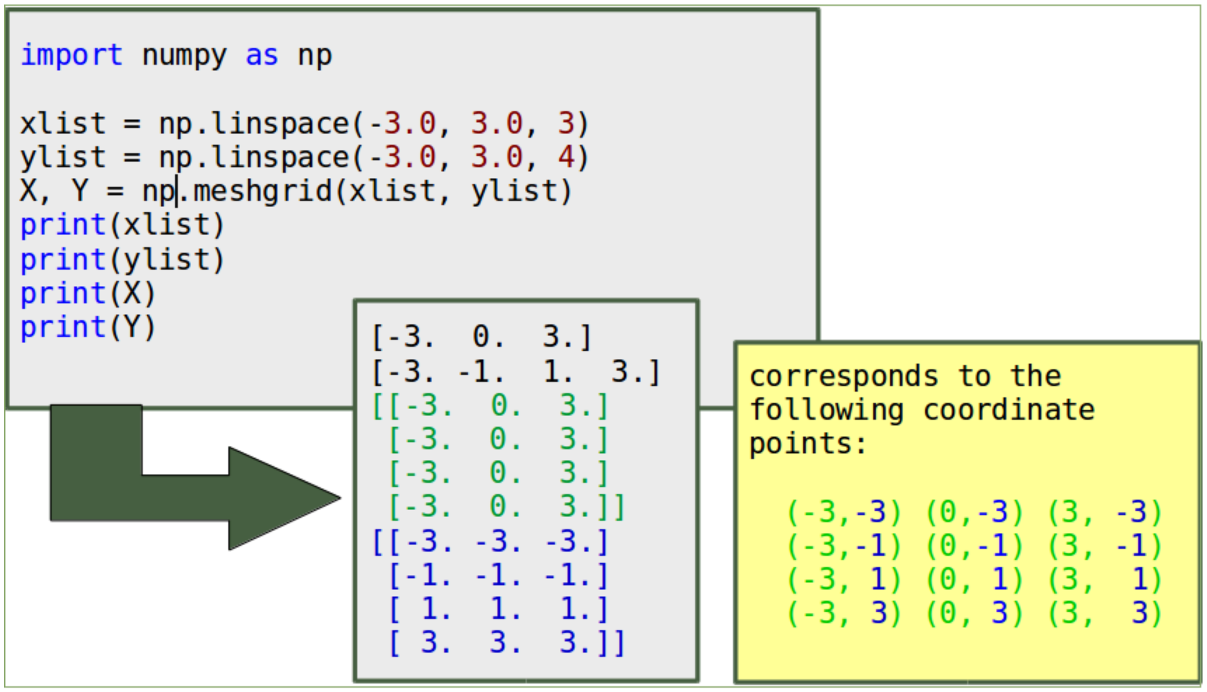
It is a cross-section of the three-dimensional graph of the function f(x, y) parallel to the x, y plane.

Contour lines are used e.g. in geography and meteorology.

In cartography, a contour line joins points of equal elevation (height) above a given level, such as mean sea level.

We can also say in a more general way that a contour line of a function with two variables is a curve which connects points with the same values.

CREATING A "MESHGRID"



```
import numpy as np

xlist = np.linspace(-3.0, 3.0, 3)
ylist = np.linspace(-3.0, 3.0, 4)
X, Y = np.meshgrid(xlist, ylist)
print(xlist)
print(ylist)
print(X)
print(Y)
```

```
[-3.  0.  3.]
[-3. -1.  1.  3.]
[[-3.  0.  3.]
 [-3.  0.  3.]
 [-3.  0.  3.]
 [-3.  0.  3.]]
[[-3. -3. -3.]
 [-1. -1. -1.]
 [ 1.  1.  1.]
 [ 3.  3.  3.]]
```

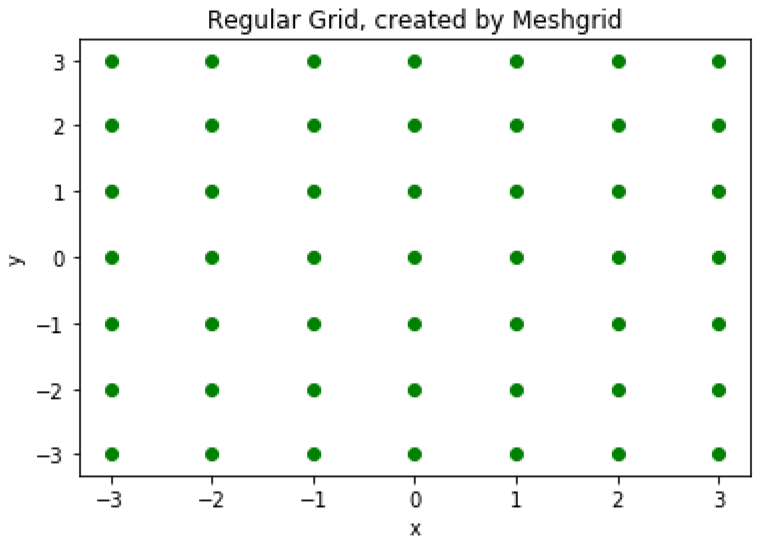
```
corresponds to the
following coordinate
points:

  (-3,-3) (0,-3) (3, -3)
  (-3,-1) (0,-1) (3, -1)
  (-3, 1) (0, 1) (3,  1)
  (-3, 3) (0, 3) (3,  3)
```

```
# the following line is only necessary if working with "ipython no
tebook"
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

n, m = 7, 7
start = -3

x_vals = np.arange(start, start+n, 1)
y_vals = np.arange(start, start+m, 1)
X, Y = np.meshgrid(x_vals, y_vals)

print(X)
print(Y)
```

```
[[-3 -2 -1  0  1  2  3]
 [-3 -2 -1  0  1  2  3]
 [-3 -2 -1  0  1  2  3]
 [-3 -2 -1  0  1  2  3]
 [-3 -2 -1  0  1  2  3]
 [-3 -2 -1  0  1  2  3]
 [-3 -2 -1  0  1  2  3]]
[[-3 -3 -3 -3 -3 -3 -3]
 [-2 -2 -2 -2 -2 -2 -2]
 [-1 -1 -1 -1 -1 -1 -1]
 [ 0  0  0  0  0  0  0]
 [ 1  1  1  1  1  1  1]
 [ 2  2  2  2  2  2  2]
 [ 3  3  3  3  3  3  3]]
```

We can visulize our meshgridif we add the following code to our previous program:

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

ax.scatter(X, Y, color="green")
ax.set_title('Regular Grid, created by Meshgrid')
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```

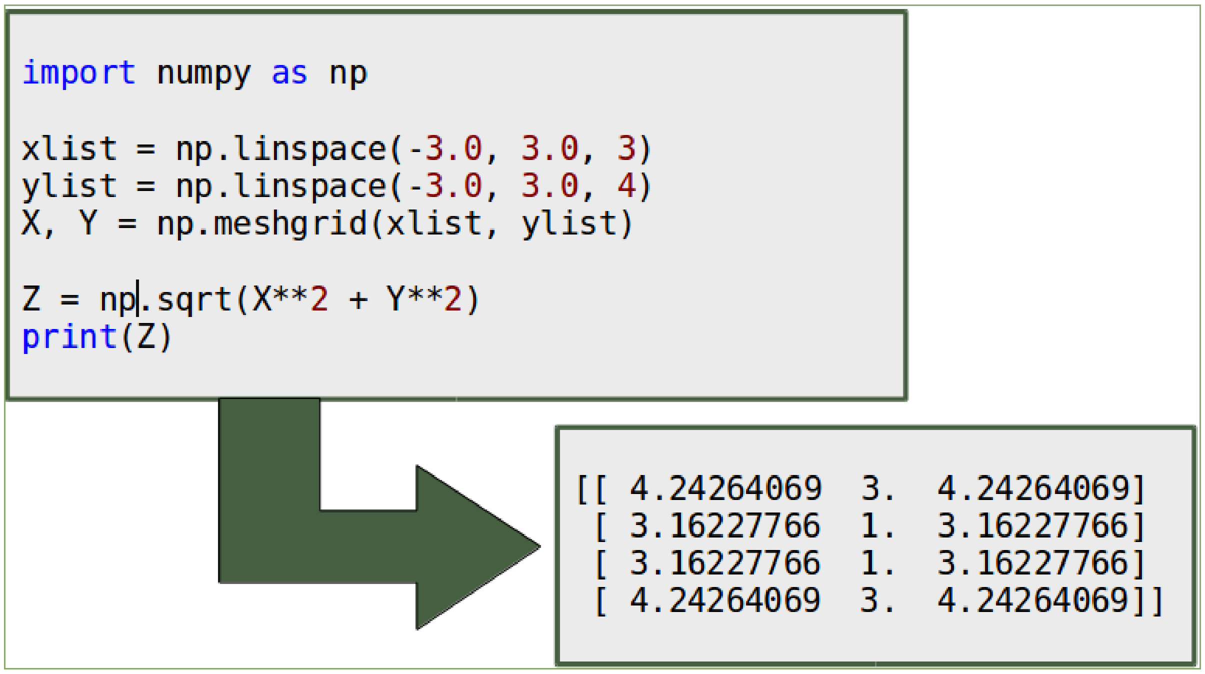


```
import numpy as np

xlist = np.linspace(-3.0, 3.0, 3)
ylist = np.linspace(-3.0, 3.0, 4)
```

```
X, Y = np.meshgrid(xlist, ylist)

Z = np.sqrt(X**2 + Y**2)
print(Z)
```

```
[[4.24264069 3.         4.24264069]
 [3.16227766 1.         3.16227766]
 [3.16227766 1.         3.16227766]
 [4.24264069 3.         4.24264069]]
```

CALCULATION OF THE VALUES



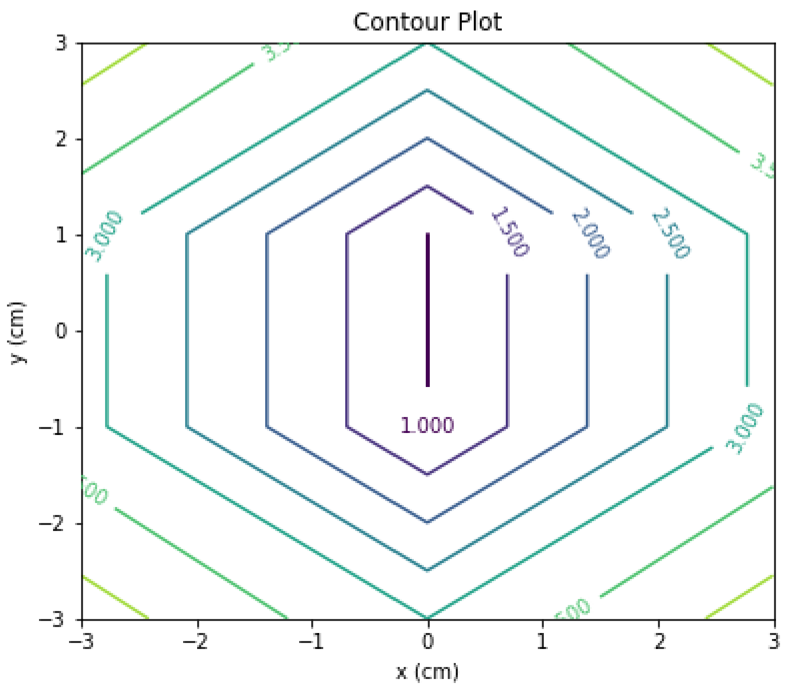
```
import numpy as np

xlist = np.linspace(-3.0, 3.0, 3)
ylist = np.linspace(-3.0, 3.0, 4)
X, Y = np.meshgrid(xlist, ylist)

Z = np.sqrt(X**2 + Y**2)
print(Z)
```

```
[[ 4.24264069  3.  4.24264069]
 [ 3.16227766  1.  3.16227766]
 [ 3.16227766  1.  3.16227766]
 [ 4.24264069  3.  4.24264069]]
```

```
import numpy as np

xlist = np.linspace(-3.0, 3.0, 3)
ylist = np.linspace(-3.0, 3.0, 4)
X, Y = np.meshgrid(xlist, ylist)

Z = np.sqrt(X**2 + Y**2)
print(Z)
```

```
[[4.24264069 3.         4.24264069]
 [3.16227766 1.         3.16227766]
 [3.16227766 1.         3.16227766]
 [4.24264069 3.         4.24264069]]
```

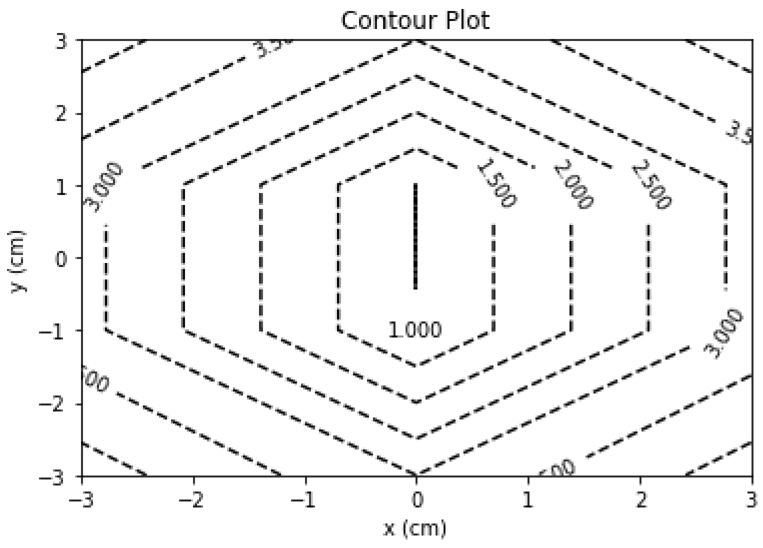
```
fig = plt.figure(figsize=(6,5))
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
ax = fig.add_axes([left, bottom, width, height])


Z = np.sqrt(X**2 + Y**2)
cp = ax.contour(X, Y, Z)
ax.clabel(cp, inline=True,
          fontsize=10)
ax.set_title('Contour Plot')
ax.set_xlabel('x (cm)')
ax.set_ylabel('y (cm)')
plt.show()
```



CHANGING THE COLOURS AND THE LINE STYLE

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

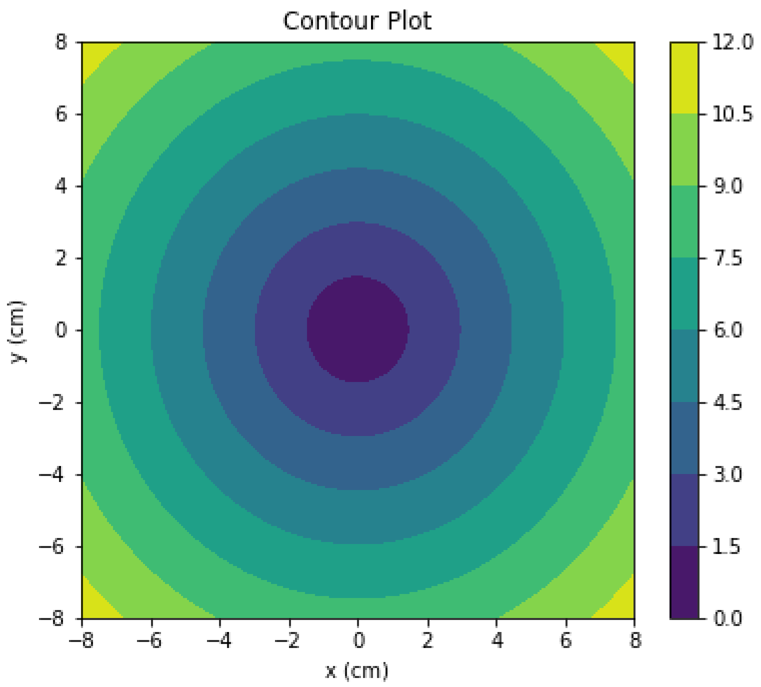
```
plt.figure()
cp = plt.contour(X, Y, Z, colors='black', linestyles='dashed')
plt.clabel(cp, inline=True,
          fontsize=10)
plt.title('Contour Plot')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```



FILLED CONTOURS

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize=(6,5))
left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
ax = fig.add_axes([left, bottom, width, height])

start, stop, n_values = -8, 8, 800

x_vals = np.linspace(start, stop, n_values)
y_vals = np.linspace(start, stop, n_values)
X, Y = np.meshgrid(x_vals, y_vals)
```

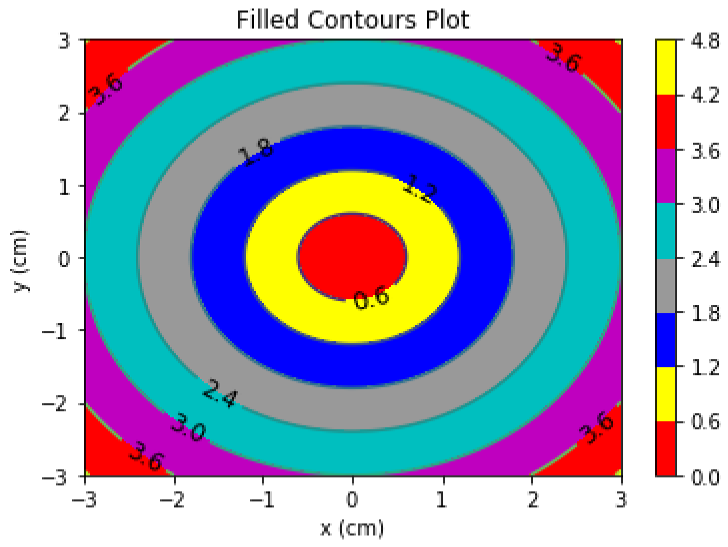
```
Z = np.sqrt(X**2 + Y**2)

cp = plt.contourf(X, Y, Z)
plt.colorbar(cp)

ax.set_title('Contour Plot')
ax.set_xlabel('x (cm)')
ax.set_ylabel('y (cm)')
plt.show()
```



INDIVIDUAL COLOURS

```
import numpy as np
import matplotlib.pyplot as plt

xlist = np.linspace(-3.0, 3.0, 100)
ylist = np.linspace(-3.0, 3.0, 100)
X, Y = np.meshgrid(xlist, ylist)
Z = np.sqrt(X**2 + Y**2)

plt.figure()
```

```
contour = plt.contour(X, Y, Z)
plt.clabel(contour, colors = 'k', fmt = '%2.1f', fontsize=12)
c = ('#ff0000', '#ffff00', '#0000FF', '0.6', 'c', 'm')
contour_filled = plt.contourf(X, Y, Z, colors=c)
plt.colorbar(contour_filled)

plt.title('Filled Contours Plot')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.savefig('contourplot_own_colours.png', dpi=300)
plt.show()
```



LEVELS

The levels were decided automatically by contour and contourf so far. They can be defined manually, by providing a list of levels as a fourth parameter. Contour lines will be drawn for each value in the list, if we use contour. For contourf, there will be filled colored regions between the values in the list.

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt

xlist = np.linspace(-3.0, 3.0, 100)
ylist = np.linspace(-3.0, 3.0, 100)
```

```
X, Y = np.meshgrid(xlist, ylist)

Z = np.sqrt(X ** 2 + Y ** 2 )
plt.figure()

levels = [0.0, 0.2, 0.5, 0.9, 1.5, 2.5, 3.5]
contour = plt.contour(X, Y, Z, levels, colors='k')
plt.clabel(contour, colors = 'k', fmt = '%2.1f', fontsize=12)
contour_filled = plt.contourf(X, Y, Z, levels)
plt.colorbar(contour_filled)

plt.title('Plot from level list')
plt.xlabel('x (cm)')
plt.ylabel('y (cm)')
plt.show()
```

The last example of this chapter will be a "lovely" contour plot:

In [ ]:

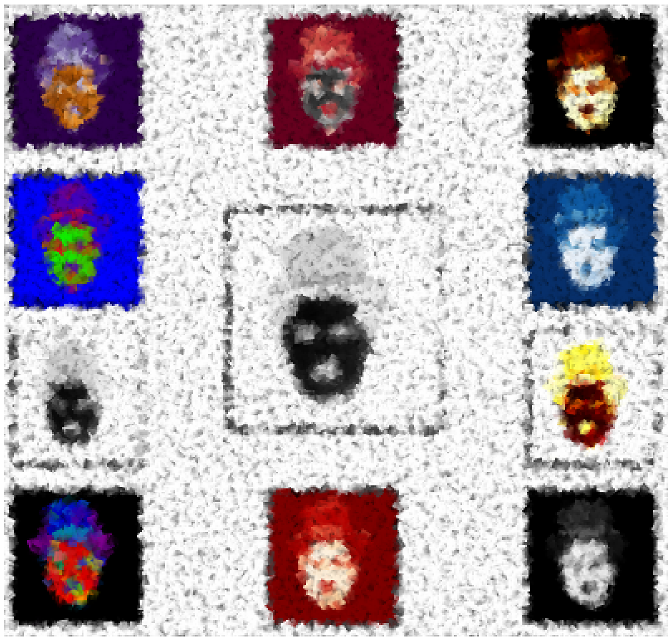
```
import matplotlib.pyplot as plt
import numpy as np

y, x = np.ogrid[-1:2:100j, -1:1:100j]
plt.contour(x.ravel(),
            y.ravel(),
            x**2 + (y-((x**2)**(1.0/3)))**2,
            [1],
            colors='red',)
plt.axis('equal')
plt.show()
```

In [ ]:

```

```

IMAGE PROCESSING

INTRODUCTION



It has never been easier to take a picture than it is today. All you normally need is a cell phone. These are the essentials to take and view a picture. Taking photos is free if we don't include the cost of the mobile phone, which is often bought for other purposes anyway. A generation ago, amateur and real artists needed specialized and often expensive equipment, and the cost per image was far from free.

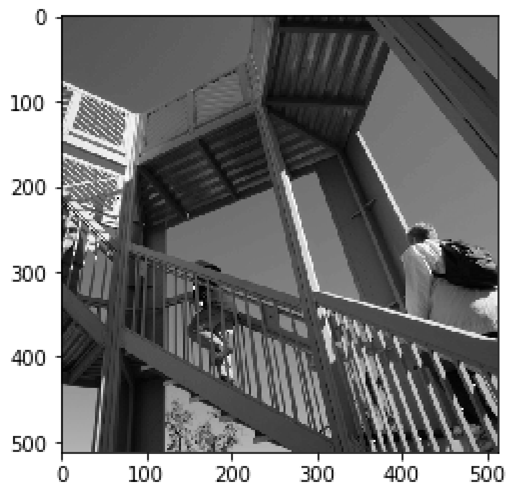
We take photos to preserve great moments of our life in time. "Pickled memories" ready to be "opened" in the future at will.

Similar to pickling, we need to use the right preservatives. Of course, the mobile phone also offers us a range of image processing software, but as soon as we have to process a large amount of photos, we need other tools. This is when programming and Python come into play. Python and its modules such as Numpy, Scipy, Matplotlib and other special modules offer the optimal functionality to cope with the flood of images.

In order to provide you with the necessary knowledge, this chapter of our Python tutorial deals with basic image processing and manipulation. For this purpose we use the modules NumPy, Matplotlib and SciPy.

We start with the `scipy` package `misc`. The helpfile says that scipy.misc contains "various utilities that don't have another home". For example, it also contains a few images, such as the following:

```
from scipy import misc
import matplotlib.pyplot as plt

ascent = misc.ascent()
plt.gray()
plt.imshow(ascent)
plt.show()
```



Additionally to the image, we can see the axis with the ticks. This may be very interesting, if you need some orientations about the size and the pixel position, but in most cases, you want to see the image without this information. We can get rid of the ticks and the axis by adding the command `plt.axis("off")`:

```
from scipy import misc

ascent = misc.ascent()
import matplotlib.pyplot as plt
plt.axis("off") # removes the axis and the ticks
plt.gray()
plt.imshow(ascent)
plt.show()
```

We can see that the type of this image is an integer array:

```
ascent.dtype
```

Output:
```
dtype('int64')
```

We can also check the size of the image:

```
ascent.shape
```

Output:
```
(512, 512)
```

The misc package contains an image of a racoon as well:

```
import scipy.misc
face = scipy.misc.face()
print(face.shape)
print(face.max)
print(face.dtype)
plt.axis("off")
plt.gray()
plt.imshow(face)
plt.show()
```

```
(768, 1024, 3)
<built-in method max of numpy.ndarray object at 0x7fa59c3e4710>
uint8
```

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

Only png images are supported by matplotlib

```
img = plt.imread('frankfurt.png')
```

```
print(img[:3])
```

```
[[[0.4117647  0.5686275  0.8       ]
  [0.40392157 0.56078434 0.7921569 ]
  [0.40392157 0.5686275  0.79607844]
  ...
  [0.48235294 0.62352943 0.81960785]
  [0.47843137 0.627451   0.81960785]
  [0.47843137 0.62352943 0.827451  ]]

 [[0.40784314 0.5647059  0.79607844]
  [0.40392157 0.56078434 0.7921569 ]
  [0.40392157 0.5686275  0.79607844]
  ...
  [0.48235294 0.62352943 0.81960785]
  [0.47843137 0.627451   0.81960785]
  [0.48235294 0.627451   0.83137256]]

 [[0.40392157 0.5686275  0.79607844]
  [0.40392157 0.5686275  0.79607844]
  [0.40392157 0.5686275  0.79607844]
  ...
  [0.48235294 0.62352943 0.81960785]
  [0.48235294 0.62352943 0.81960785]
  [0.4862745  0.627451   0.83137256]]]
```

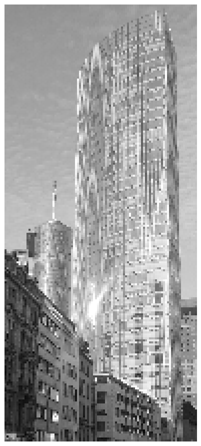
```
plt.axis("off")
imgplot = plt.imshow(img)
```

```
lum_img = img[:,:,1]
```

```
print(lum_img)
```

```
[[0.5686275  0.56078434 0.5686275  ... 0.62352943 0.627451   0.623
52943]
 [0.5647059  0.56078434 0.5686275  ... 0.62352943 0.627451   0.627
451  ]
 [0.5686275  0.5686275  0.5686275  ... 0.62352943 0.62352943 0.627
451  ]
 ...
 [0.31764707 0.32941177 0.32941177 ... 0.30588236 0.3137255  0.317
64707]
 [0.31764707 0.3137255  0.32941177 ... 0.3019608  0.32156864 0.337
2549 ]
 [0.31764707 0.3019608  0.33333334 ... 0.30588236 0.32156864 0.333
33334]]
```

```
plt.axis("off")
imgplot = plt.imshow(lum_img)
```



TINT, SHADE AND TONE

Now, we will show how to tint an image. Tint is an expression from colour theory and an often used technique by painters. Thinking about painters and not think about the Netherlands is hard to imagine. So we will use a picture with Dutch windmills in our next example. (The image has been taken at Kinderdijk, a village in the Netherlands, about 15 km east of Rotterdam and about 50 kilometres from Den Haag (The Hague). It's a UNESCO World Heritage Site since 1997.)

```
windmills = plt.imread('windmills.png')
```

```
plt.axis("off")
plt.imshow(windmills)
```

Output:
```
<matplotlib.image.AxesImage at 0x7fa59c2479d0>
```

We want to tint the image now. This means we will "mix" our colours with white. This will increase the lightness of our image. For this purpose, we write a Python function, which takes an image and a percentage value as a parameter. Setting 'percentage' to 0 will not change the image, setting it to one means that the image will be completely whitened:

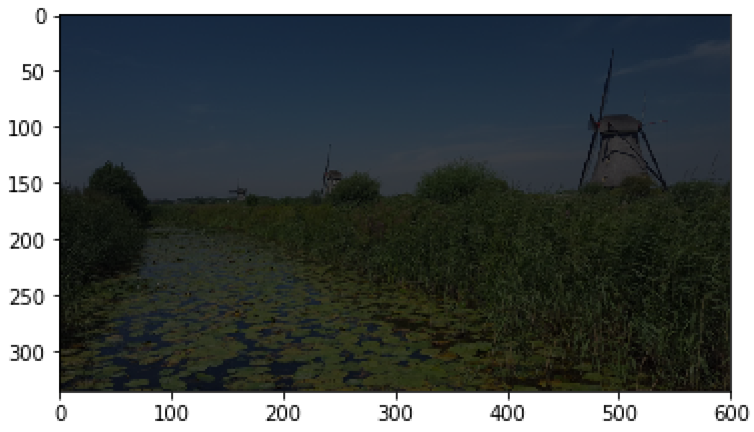
```
import numpy as np
import matplotlib.pyplot as plt

def tint(imag, percent):
    """
    imag: the image which will be shaded
    percent: a value between 0 (image will remain unchanged
             and 1 (image will completely white)
    """
    tinted_imag = imag + (np.ones(imag.shape) - imag) * percent
    return tinted_imag

windmills = plt.imread('windmills.png')

tinted_windmills = tint(windmills, 0.8)
plt.axis("off")
plt.imshow(tinted_windmills)
```

Output:
```
<matplotlib.image.AxesImage at 0x7fa59c30f350>
```

A shade is the mixture of a color with black, which reduces lightness.

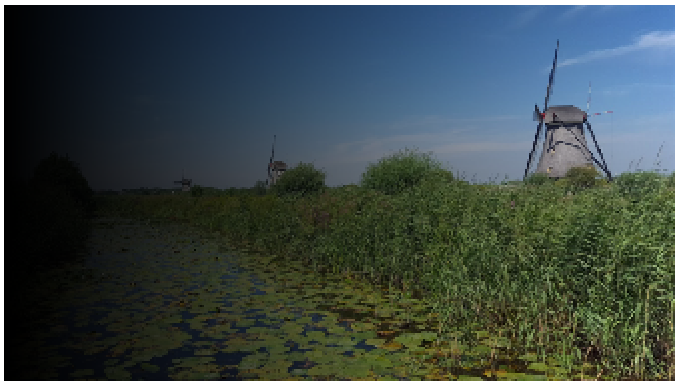
```
import numpy as np
import matplotlib.pyplot as plt


def shade(imag, percent):
    """
    imag: the image which will be shaded
    percent: a value between 0 (image will remain unchanged
             and 1 (image will be blackened)
    """
    tinted_imag = imag * (1 - percent)
    return tinted_imag

windmills = plt.imread('windmills.png')

tinted_windmills = shade(windmills, 0.7)
plt.imshow(tinted_windmills)
```

Output:
```
<matplotlib.image.AxesImage at 0x7fa59c3b8050>
```



```
def vertical_gradient_line(image, reverse=False):
    """
    We create a horizontal gradient line with the shape (1, imag
e.shape[1], 3))
    The values are incremented from 0 to 1, if reverse is False,
    otherwise the values are decremented from 1 to 0.
    """
    number_of_columns = image.shape[1]
    if reverse:
        C = np.linspace(1, 0, number_of_columns)
    else:
        C = np.linspace(0, 1, number_of_columns)
    C = np.dstack((C, C, C))
    return C

horizontal_brush = vertical_gradient_line(windmills)
tinted_windmills =  windmills * horizontal_brush
plt.axis("off")
plt.imshow(tinted_windmills)
```

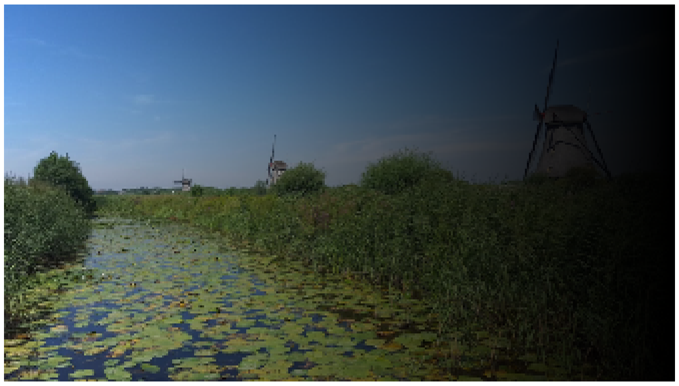
Output: `<matplotlib.image.AxesImage at 0x7fa59e769b50>`



We will tint the image now from right to left by setting the reverse parameter of our Python function to "True":

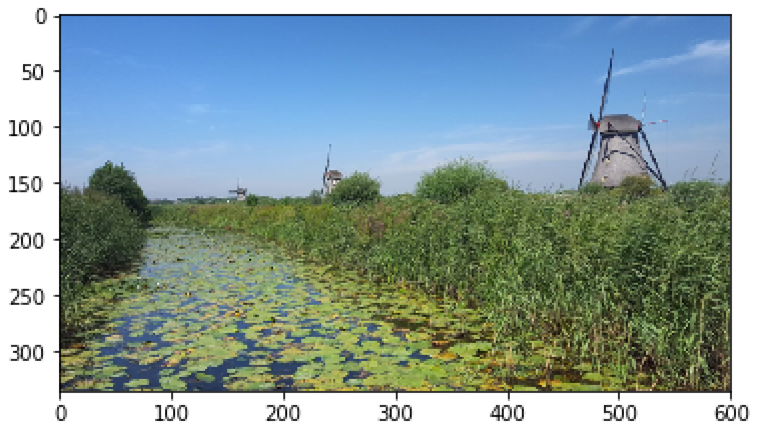
```
def vertical_gradient_line(image, reverse=False):
    """
    We create a horizontal gradient line with the shape (1, imag
e.shape[1], 3))
    The values are incremented from 0 to 1, if reverse is False,
    otherwise the values are decremented from 1 to 0.
    """
    number_of_columns = image.shape[1]
    if reverse:
        C = np.linspace(1, 0, number_of_columns)
    else:
        C = np.linspace(0, 1, number_of_columns)
    C = np.dstack((C, C, C))
    return C

horizontal_brush = vertical_gradient_line(windmills, reverse=True)
tinted_windmills =  windmills * horizontal_brush
plt.axis("off")
plt.imshow(tinted_windmills)
```

Output:
```
<matplotlib.image.AxesImage at 0x7fa59cc89250>
```



```
def horizontal_gradient_line(image, reverse=False):
    """
    We create a vertical gradient line with the shape (image.shap
e[0], 1, 3))
    The values are incremented from 0 to 1, if reverse is False,
    otherwise the values are decremented from 1 to 0.
    """
    number_of_rows, number_of_columns = image.shape[:2]
    C = np.linspace(1, 0, number_of_rows)
    C = C[np.newaxis,:]
    C = np.concatenate((C, C, C)).transpose()
    C = C[:, np.newaxis]
    return C

vertical_brush = horizontal_gradient_line(windmills)
tinted_windmills =  windmills
plt.imshow(tinted_windmills)
```

Output:
```
<matplotlib.image.AxesImage at 0x7fa59c227dd0>
```

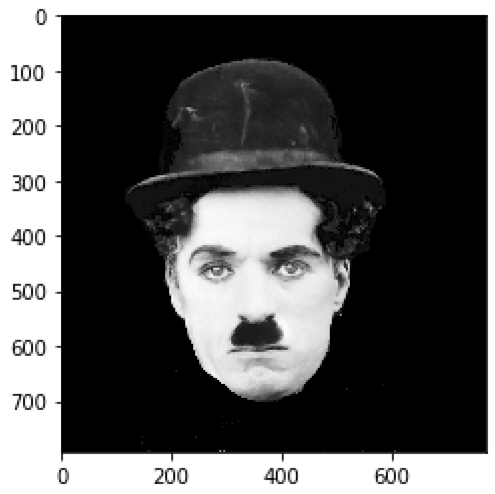


A tone is produced either by the mixture of a color with gray, or by both tinting and shading.

```
charlie = plt.imread('Chaplin.png')
plt.gray()
print(charlie)
plt.imshow(charlie)
```

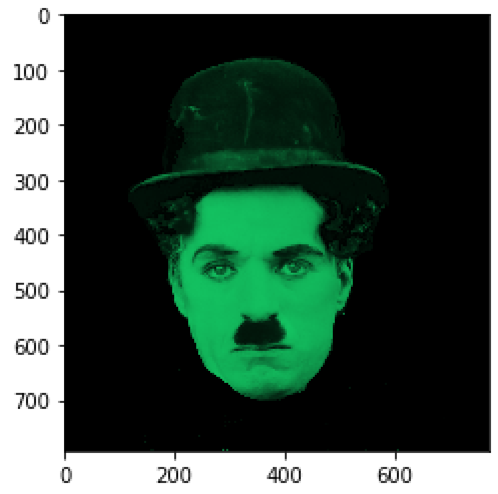
```
[[0.16470589 0.16862746 0.1764706  ... 0.         0.
0.        ]
 [0.16078432 0.16078432 0.16470589 ... 0.         0.
0.        ]
 [0.15686275 0.15686275 0.16078432 ... 0.         0.
0.        ]
 ...
 [0.         0.         0.         ... 0.         0.
0.        ]
 [0.         0.         0.         ... 0.         0.
0.        ]
 [0.         0.         0.         ... 0.         0.
0.        ]]
```

Output:
```
<matplotlib.image.AxesImage at 0x7fa59c18f810>
```



```
colored = np.dstack((charlie*0.1, charlie*1, charlie*0.5))

plt.imshow(colored)
```

Output:
```
<matplotlib.image.AxesImage at 0x7fa59c0f9890>
```



tinting gray scale images: http://scikit-image.org/docs/dev/auto_examples/plot_tinting_grayscale_images.html

We will use different colormaps in the following example. The colormaps can be found in matplotlib.pyplot.cm.datad:

```
plt.cm.datad.keys()
```

Output:

```
dict_keys(['Blues', 'BrBG', 'BuGn', 'BuPu', 'CMRmap', 'GnB
u', 'Greens', 'Greys', 'OrRd', 'Oranges', 'PRGn', 'PiYG', 'Pu
Bu', 'PuBuGn', 'PuOr', 'PuRd', 'Purples', 'RdBu', 'RdGy', 'Rd
Pu', 'RdYlBu', 'RdYlGn', 'Reds', 'Spectral', 'Wistia', 'YlG
n', 'YlGnBu', 'YlOrBr', 'YlOrRd', 'afmhot', 'autumn', 'binar
y', 'bone', 'brg', 'bwr', 'cool', 'coolwarm', 'copper', 'cube
helix', 'flag', 'gist_earth', 'gist_gray', 'gist_heat', 'gis
t_ncar', 'gist_rainbow', 'gist_stern', 'gist_yarg', 'gnuplo
t', 'gnuplot2', 'gray', 'hot', 'hsv', 'jet', 'nipy_spectra
l', 'ocean', 'pink', 'prism', 'rainbow', 'seismic', 'sprin
g', 'summer', 'terrain', 'winter', 'Accent', 'Dark2', 'Paire
d', 'Pastel1', 'Pastel2', 'Set1', 'Set2', 'Set3', 'tab10', 't
ab20', 'tab20b', 'tab20c', 'Blues_r', 'BrBG_r', 'BuGn_r', 'Bu
Pu_r', 'CMRmap_r', 'GnBu_r', 'Greens_r', 'Greys_r', 'OrR
d_r', 'Oranges_r', 'PRGn_r', 'PiYG_r', 'PuBu_r', 'PuBuGn_r',
'PuOr_r', 'PuRd_r', 'Purples_r', 'RdBu_r', 'RdGy_r', 'RdP
u_r', 'RdYlBu_r', 'RdYlGn_r', 'Reds_r', 'Spectral_r', 'Wisti
a_r', 'YlGn_r', 'YlGnBu_r', 'YlOrBr_r', 'YlOrRd_r', 'afmho
t_r', 'autumn_r', 'binary_r', 'bone_r', 'brg_r', 'bwr_r', 'co
ol_r', 'coolwarm_r', 'copper_r', 'cubehelix_r', 'flag_r', 'gi
st_earth_r', 'gist_gray_r', 'gist_heat_r', 'gist_ncar_r', 'gi
st_rainbow_r', 'gist_stern_r', 'gist_yarg_r', 'gnuplot_r', 'g
nuplot2_r', 'gray_r', 'hot_r', 'hsv_r', 'jet_r', 'nipy_spectr
al_r', 'ocean_r', 'pink_r', 'prism_r', 'rainbow_r', 'seismi
c_r', 'spring_r', 'summer_r', 'terrain_r', 'winter_r', 'Accen
t_r', 'Dark2_r', 'Paired_r', 'Pastel1_r', 'Pastel2_r', 'Set
1_r', 'Set2_r', 'Set3_r', 'tab10_r', 'tab20_r', 'tab20b_r',
'tab20c_r'])
```

```python
import numpy as np
import matplotlib.pyplot as plt

charlie = plt.imread('Chaplin.png')

#  colormaps plt.cm.datad
# cmaps = set(plt.cm.datad.keys())
cmaps = {'afmhot', 'autumn', 'bone', 'binary', 'bwr', 'brg',
         'CMRmap', 'cool', 'copper', 'cubehelix', 'Greens'}

X = [(4, 3, 1, (1, 0, 0)),
     (4, 3, 2, (0.5, 0.5, 0)),
     (4, 3, 3, (0, 1, 0)),
     (4, 3, 4, (0, 0.5, 0.5)),
     (4, 3, (5, 8), (0, 0, 1)),
     (4, 3, 6, (1, 1, 0)),
     (4, 3, 7, (0.5, 1, 0) ),
     (4, 3, 9, (0, 0.5, 0.5)),
     (4, 3, 10, (0, 0.5, 1)),
     (4, 3, 11, (0, 1, 1)),
     (4, 3, 12, (0.5, 1, 1))]

fig = plt.figure(figsize=(6, 5))

#fig.subplots_adjust(bottom=0, left=0, top = 0.975, right=1)
for nrows, ncols, plot_number, factor in X:
    sub = fig.add_subplot(nrows, ncols, plot_number)
    sub.set_xticks([])

    sub.imshow(charlie*0.0002, cmap=cmaps.pop())
    sub.set_yticks([])
```

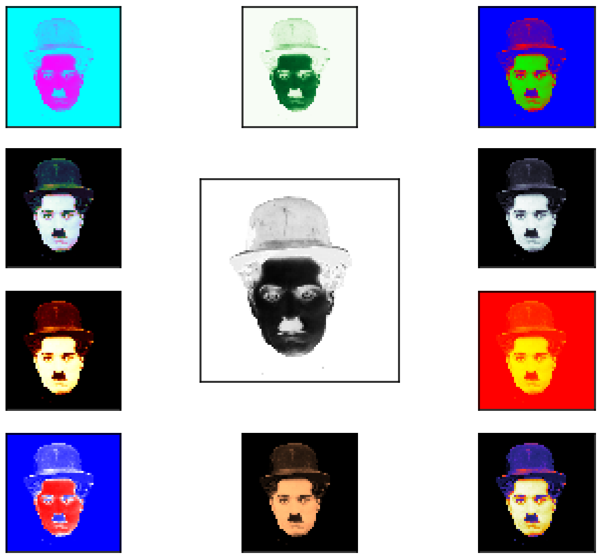
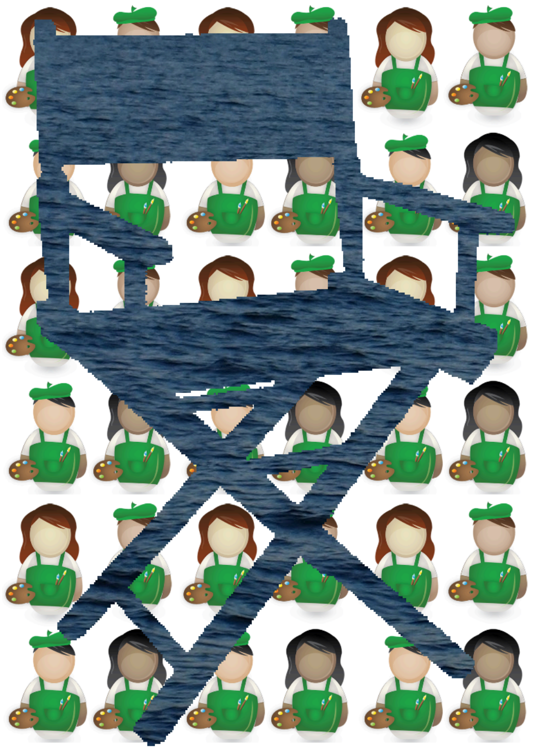


IMAGE PROCESSING TECHNIQUES

INTRODUCTION



As you may have noticed, each of our pages in our various tutorials are introduced by eye candy pictures, which have created with great care to enrich the content. One of those images has been the raison d'être of this chapter. We want to demonstrate how we created the picture for our chapter on Decorators. The idea was to play with decorators in "real life", small icons with images of small workers painting a room and on the other hand blending this with the "at" sign, the Python symbol for decorator. It is also a good example of how to create a watermark.

We will demonstrate in this chapter the whole process chain of how we created this image. The picture on the right side of the current page has also been created the same way but uses a director's chair on a small painters background as a watermark instead of the at sign.

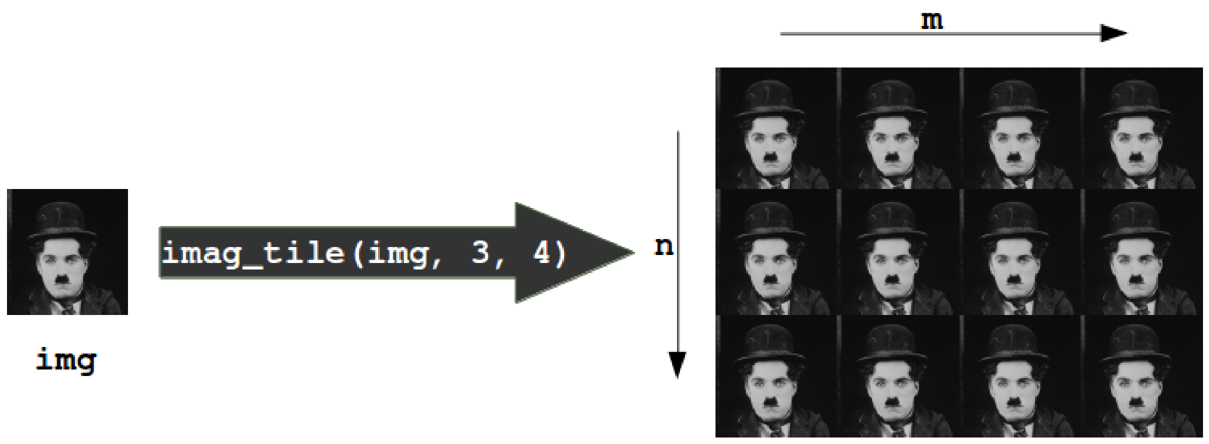
At first, we write a function "imag_tile" for tiling images both in horizontal and in vertical direction. We will use this to create the background of our image.

Then we show how to cut out with slicing a cutout or an excerpt of an image. We will use the shade function, which we introduced in our previous chapter on image processing, to shade our image.

Finally, we will use the original image, the shaded image, plus an image with a binary at sign with the conditional numpy where function to create the final image. The final image contains the at sign as a watermark, cut out from the shaded image.

TILING AN IMAGE

The function imag_tile, which we are going to design, can be best explained with the following diagram:



The function imag_tile

```
imag_tile(img, n, m)
```

creates a tiled image by appending an image "img" m times in horizontal direction. After this we append the strip image consisting of m img images n times in vertical direction.

In the following code, we use a picture of painting decorators as the tile image:

```
%matplotlib inline

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

import numpy as np

def imag_tile(img, n, m=1):
    """
    The image "img" will be repeated n times in
    vertical and m times in horizontal direction.
    """

    if n == 1:
        tiled_img = img
    else:
        lst_imgs = []
        for i in range(n):
             lst_imgs.append(img)
        tiled_img = np.concatenate(lst_imgs, axis=1 )
    if m > 1:
        lst_imgs = []
        for i in range(m):
             lst_imgs.append(tiled_img)
        tiled_img = np.concatenate(lst_imgs, axis=0 )

    return tiled_img

basic_pattern = mpimg.imread('decorators_b2.png')

decorators_img = imag_tile(basic_pattern, 3, 3)

plt.axis("off")
plt.imshow(decorators_img)
```

Output:
```
<matplotlib.image.AxesImage at 0x7f0c1a6ff390>
```

An image is a 3-dimensional numpy ndarray.

```
type(basic_pattern)
```

Output:
```
numpy.ndarray
```

The first three rows of our image basic_pattern look like this:

```
basic_pattern[:3]
```

Output:
```
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.],
        [1., 1., 1.],
        ...,
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        ...,
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]], dtype=float32)
```

The innermost lists of our image contain the pixels. We have three values corresponding the the R, G, and B values, this means that we have a 24-bit RGB PNG image, eight bits for each of R, G, B.

PNG images might also consist of 32-bit images (RGBA). The fourth value "A" will be used for transparancy, single channel grayscale.

It's easy to access indivual pixels by indexing, e.g. the pixel in row 100 and column 20:

```
basic_pattern[100, 28]
```

Output:
```
array([0.9019608, 0.8901961, 0.8627451], dtype=float32)
```

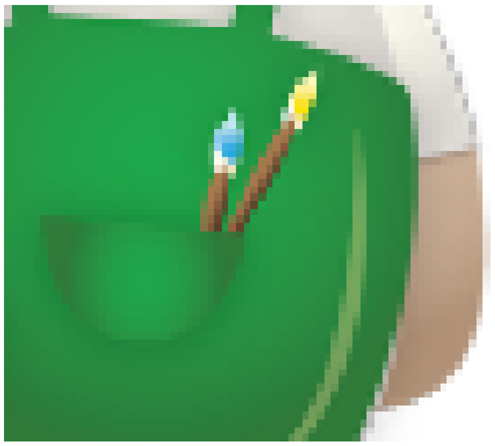
As we have seen, the pixels are float (float32) values between 0 and 1. Matplotlib plotting can handle both float32 and uint8 for PNG images. For all other formats it will be only uint8.

CROP IMAGES

We can also crop subimages with the slicing function. We crop the image from (90, 50), i.e. row 90 and column 50, to (50, 120) in the following example:

```
cropped = basic_pattern[90:150, 50:120]
plt.axis("off")
plt.imshow(cropped)
```

Output:
```
<matplotlib.image.AxesImage at 0x7f0c18e64950>
```



We will need this technique in the following.

We will load the image of an at sign in the following script:

We can use the slicing function to crop parts of an image. We will use this to make sure that both images have the same size.

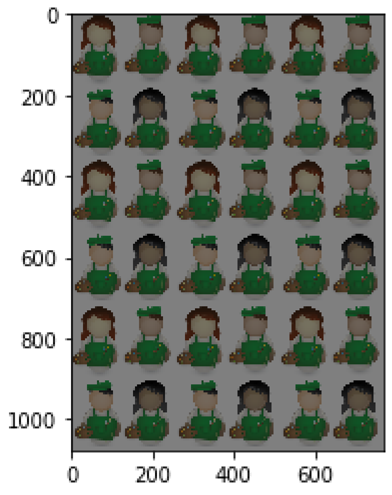
```
at_img=mpimg.imread('at_sign.png')

# at_img and decorators_img have to be of equal size:
d_shape = decorators_img.shape
at_shape = at_img.shape
height, width, colours = [min(x) for x in zip(*(d_shape, at_shap
e))]
at_img = at_img[0:height, 0:width]
```

SHADE AN IMAGE

We define a function "shade" in the following script. "shade" takes two parameters. The first one "imag" is the image, which will be shaded and the second one is the shading factor. It can be a value between 0 and 1. If the factor is set to 0, imag will remain unchanged. If set to one, the image will be completetely blackened.

```
def shade(imag, percent):
    """
    imag: the image which will be shaded
    percent: a value between 0 (image will remain unchanged
             and 1 (image will be blackened)
    """
    tinted_imag = imag * (1 - percent)
    return tinted_imag

tinted_decorator_img = shade(decorators_img, 0.5)
plt.imshow(tinted_decorator_img)
print(tinted_decorator_img[:3])
```

```
[[[0.5 0.5 0.5]
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]
  ...
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]]

 [[0.5 0.5 0.5]
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]
  ...
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]]

 [[0.5 0.5 0.5]
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]
  ...
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]
  [0.5 0.5 0.5]]]
```



BLEND IMAGES

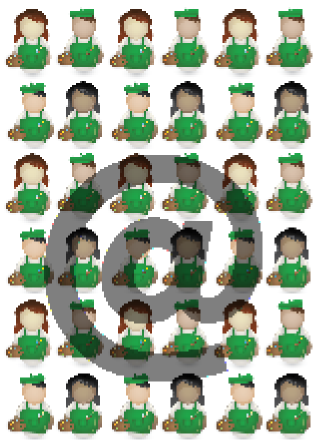
FIRST EXAMPLE

We have everything together now to create the blended image. Our at sign picture consists of black and white pixels. The blended image is constructed like this: Let p=(n, m) be an arbitrary pixel in the n-th row and m-th column of the image at_image. If the value of this pixel is not black or dark gray, we will use the pixel at position (n, m) from the picture decorators_img, otherwise, we will use the corresponding pixel from tinted_decorator_img. The where function of numpy is ideal for this task:

```
print(at_img.shape,
      decorators_img.shape,
      tinted_decorator_img.shape)
#basic_pattern = mpimg.imread('decorators2.png')
img2 = np.where(at_img > [0.1, 0.1, 0.1],
                decorators_img,
                tinted_decorator_img)

plt.axis("off")
plt.imshow(img2)
```

```
(1077, 771, 3) (1077, 771, 3) (1077, 771, 3)
```

Output:
```
<matplotlib.image.AxesImage at 0x7f0c18ce3a10>
```



All there is left to do is save the newly created image:

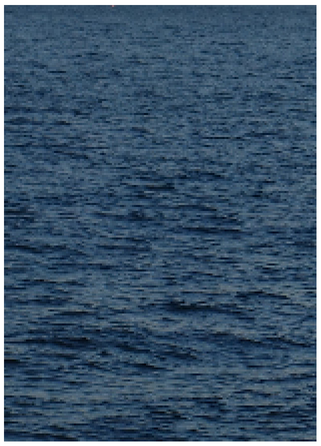
```
mpimg.imsave('decorators_with_at.png', img2)
```

SECOND EXAMPLE

We want to use now a different image as a "watermark". Instead of the at sign, we want to use now a director's chair. We will create the image from the top of this page.

```
images = [mpimg.imread(fname) for fname in ["director_chair.png",
"the_sea.png", "the_sky.png"]]
director_chair, sea, sky = images


plt.axis("off")
plt.imshow(sea)
```

Output:
```
<matplotlib.image.AxesImage at 0x7f0c18cc4450>
```

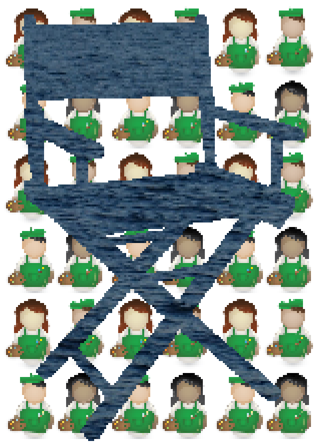


```
plt.axis("off")
plt.imshow(director_chair)
```

Output:
```
<matplotlib.image.AxesImage at 0x7f0c18423850>
```

In the following, we blend together the images director_chair, decorators_img and sea by using where of numpy once more:

```
#sea2 = mpimg.imread('the_sea2.png')
img = np.where(director_chair > [0.9, 0.9, 0.9],
               decorators_img,
               sea)
plt.axis("off")
plt.imshow(img)
mpimg.imsave('decorators_with_chair.png', img)
```



We could have used "Image.open" from PIL instead of mpimg.imread from matplotlib to read in the pictures. There is a crucial difference or a potential "problem" between these two ways: The image we get from imread has values between 0 and 1, whereas Image.open consists of values between 0 and 255. So we might have to divide all the pixels by 255, if we have to work with an image read in by mpimg.imread:

```python
from PIL import Image


img = Image.open("director_chair.jpg")
img = img.resize((at_img.shape[1], at_img.shape[0]))
img = np.asarray(img)


plt.axis("off")
plt.imshow(img)

print(img[100, 129])
```

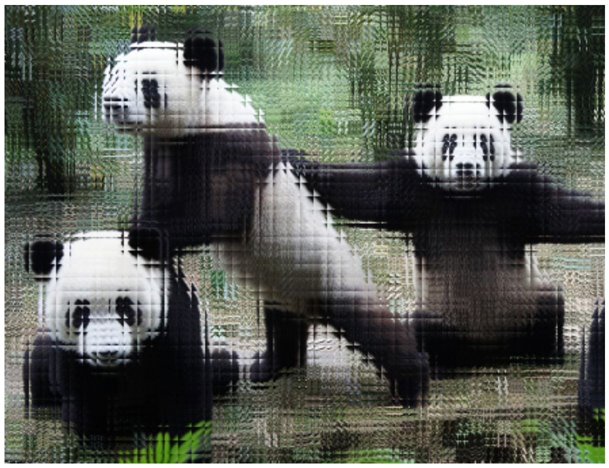
```
[27 27 27]
```

```python
# PIL: Pixel are within range 0 and 255
# mpimg: range 0 bis 1

img = np.asarray(img, np.float)
img = img / 255

print(img[100, 129])
```

```
[0.10588235 0.10588235 0.10588235]
```

Introduction into Pandas

The pandas we are writing about in this chapter have nothing to do with the cute panda bears. Endearing bears are not what our visitors expect in a Python tutorial. Pandas is the name for a Python module, which is rounding up the capabilities of Numpy, Scipy and Matplotlab. The word pandas is an acronym which is derived from "Python and data analysis" and "panel data".



There is often some confusion about whether Pandas is an alternative to Numpy, SciPy and Matplotlib. The truth is that it is built on top of Numpy. This means that Numpy is required by pandas. Scipy and Matplotlib on the other hand are not required by pandas but they are extremely useful. That's why the Pandas project lists them as "optional dependency".

Pandas is a software library written for the Python programming language. It is used for data manipulation and analysis. It provides special data structures and operations for the manipulation of numerical tables and time series. Pandas is free software released under the three-clause BSD license.

DATA STRUCTURES

We will start with the following two important data structures of Pandas:

- Series and
- DataFrame

SERIES

A Series is a one-dimensional labelled array-like object. It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on. It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data.

We define a simple Series object in the following example by instantiating a Pandas Series object with a list. We will later see that we can use other data objects for example Numpy arrays and dictionaries as well to instantiate a Series object.

```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
S
```

Output:

```
0    11
1    28
2    72
3     3
4     5
5     8
dtype: int64
```

We haven't defined an index in our example, but we see two columns in our output: The right column contains our data, whereas the left column contains the index. Pandas created a default index starting with 0 going to 5, which is the length of the data minus 1.

We can directly access the index and the values of our Series S:

```
print(S.index)
print(S.values)
```

```
RangeIndex(start=0, stop=6, step=1)
[11 28 72  3  5  8]
```

If we compare this to creating an array in numpy, we will find lots of similarities:

```
import numpy as np
X = np.array([11, 28, 72, 3, 5, 8])
print(X)
print(S.values)
# both are the same type:
print(type(S.values), type(X))
```

```
[11 28 72  3  5  8]
[11 28 72  3  5  8]
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

So far our Series have not been very different to ndarrays of Numpy. This changes, as soon as we start defining Series objects with individual indices:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
S
```

Output:

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

A big advantage to NumPy arrays is obvious from the previous example: We can use arbitrary indices.

If we add two series with the same indices, we get a new series with the same index and the correponding values will be added:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("sum of S: ", sum(S))
```

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
sum of S:  115
```

The indices do not have to be the same for the Series addition. The index will be the "union" of both indices. If an index doesn't occur in both Series, the value for this Series will be NaN:

```
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

```
cherries       83.0
oranges        46.0
peaches         NaN
pears          42.0
raspberries     NaN
dtype: float64
```

In principle, the indices can be completely different, as in the following example. We have two indices. One is the Turkish translation of the English fruit names:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']

fruits_tr = ['elma', 'portakal', 'kiraz', 'armut']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits_tr)
print(S + S2)
```

```
apples     NaN
armut      NaN
cherries   NaN
elma       NaN
kiraz      NaN
oranges    NaN
pears      NaN
portakal   NaN
dtype: float64
```

INDEXING

It's possible to access single values of a Series.

```
print(S['apples'])
```

```
20
```

This lookes like accessing the values of dictionaries through keys.

However, Series objects can also be accessed by multiple indexes at the same time. This can be done by packing the indexes into a list. This type of access returns a Pandas Series again:

```
print(S[['apples', 'oranges', 'cherries']])
```

```
apples      20
oranges     33
cherries    52
dtype: int64
```

Similar to Numpy we can use scalar operations or mathematical functions on a series:

```
import numpy as np
print((S + 3) * 4)
print("======================")
print(np.sin(S))
```

```
apples       92
oranges     144
cherries    220
pears        52
dtype: int64
======================
apples      0.912945
oranges     0.999912
cherries    0.986628
pears      -0.544021
dtype: float64
```

PANDAS.SERIES.APPLY

Series.apply(func, convert_dtype=True, args=(), **kwds)

The function "func" will be applied to the Series and it returns either a Series or a DataFrame, depending on "func".

Parameter	Meaning
func	a function, which can be a NumPy function that will be applied to the entire Series or a Python function that will be applied to every single value of the series
convert_dtype	A boolean value. If it is set to True (default), apply will try to find better dtype for elementwise function results.

Parameter	Meaning
	If False, leave as dtype=object
args	Positional arguments which will be passed to the function "func" additionally to the values from the series.
**kwds	Additional keyword arguments will be passed as keywords to the function

Example:

```
S.apply(np.log)
```

Output:
```
apples      2.995732
oranges     3.496508
cherries    3.951244
pears       2.302585
dtype: float64
```

We can also use Python lambda functions. Let's assume, we have the following task. The test the amount of fruit for every kind. If there are less than 50 available, we will augment the stock by 10:

```
S.apply(lambda x: x if x > 50 else x+10 )
```

Output:
```
apples      30
oranges     43
cherries    52
pears       20
dtype: int64
```

FILTERING WITH A BOOLEAN ARRAY

Similar to numpy arrays, we can filter Pandas Series with a Boolean array:

```
S[S>30]
```

Output:
```
oranges     33
cherries    52
dtype: int64
```

A series can be seen as an ordered Python dictionary with a fixed length.

```
"apples" in S
```

Output:
```
True
```

CREATING SERIES OBJECTS FROM DICTIONARIES

We can even use a dictionary to create a Series object. The resulting Series contains the dict's keys as the indices and the values as the values.

```
cities = {"London":    8615246,
          "Berlin":    3562166,
          "Madrid":    3165235,
          "Rome":      2874038,
          "Paris":     2273305,
          "Vienna":    1805681,
          "Bucharest": 1803425,
          "Hamburg":   1760433,
          "Budapest":  1754000,
          "Warsaw":    1740119,
          "Barcelona": 1602386,
          "Munich":    1493900,
          "Milan":     1350680}

city_series = pd.Series(cities)
print(city_series)
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

NAN - MISSING DATA

One problem in dealing with data analysis tasks consists in missing data. Pandas makes it as easy as possible to work with missing data.

If we look once more at our previous example, we can see that the index of our series is the same as the keys of the dictionary we used to create the cities_series. Now, we want to use an index which is not overlapping with the dictionary keys. We have already seen that we can pass a list or a tuple to the keyword argument 'index' to define the index. In our next example, the list (or tuple) passed to the keyword parameter 'index' will not be equal to the keys. This means that some cities from the dictionary will be missing and two cities ("Zurich" and "Stuttgart") don't occur in the dictionary.

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

Output:
```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

Due to the Nan values the population values for the other cities are turned into floats. There is no missing data in the following examples, so the values are int:

```
my_cities = ["London", "Paris", "Berlin", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

Output:
```
London      8615246
Paris       2273305
Berlin      3562166
Hamburg     1760433
dtype: int64
```

THE METHODS ISNULL() AND NOTNULL()

We can see, that the cities, which are not included in the dictionary, get the value NaN assigned. NaN stands for "not a number". It can also be seen as meaning "missing" in our example.

We can check for missing values with the methods isnull and notnull:

```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
print(my_city_series.isnull())
```

```
London       False
Paris        False
Zurich        True
Berlin       False
Stuttgart     True
Hamburg      False
dtype: bool
```

```
print(my_city_series.notnull())
```

```
London        True
Paris         True
Zurich       False
Berlin        True
Stuttgart    False
Hamburg       True
dtype: bool
```

CONNECTION BETWEEN NAN AND NONE

We get also a NaN, if a value in the dictionary has a None:

```
d = {"a":23, "b":45, "c":None, "d":0}
S = pd.Series(d)
print(S)
```

```
a     23.0
b     45.0
c      NaN
d      0.0
dtype: float64
```

```
pd.isnull(S)
```

Output:
```
a     False
b     False
c      True
d     False
dtype: bool
```

```
pd.notnull(S)
```

Output:
```
a      True
b      True
c     False
d      True
dtype: bool
```

FILTERING OUT MISSING DATA

It's possible to filter out missing data with the Series method dropna. It returns a Series which consists only of non-null data:

```
print(my_city_series.dropna())
```

```
London      8615246.0
Paris       2273305.0
Berlin      3562166.0
Hamburg     1760433.0
dtype: float64
```

FILLING IN MISSING DATA

In many cases you don't want to filter out missing data, but you want to fill in appropriate data for the empty gaps. A suitable method in many situations will be fillna:

```
print(my_city_series.fillna(0))
```

```
London        8615246.0
Paris         2273305.0
Zurich              0.0
Berlin        3562166.0
Stuttgart           0.0
Hamburg       1760433.0
dtype: float64
```

Okay, that's not what we call "fill in appropriate data for the empty gaps". If we call fillna with a dict, we can provide the appropriate data, i.e. the population of Zurich and Stuttgart:

```
missing_cities = {"Stuttgart":597939, "Zurich":378884}
my_city_series.fillna(missing_cities)
```

Output:

```
London        8615246.0
Paris         2273305.0
Zurich         378884.0
Berlin        3562166.0
Stuttgart      597939.0
Hamburg       1760433.0
dtype: float64
```

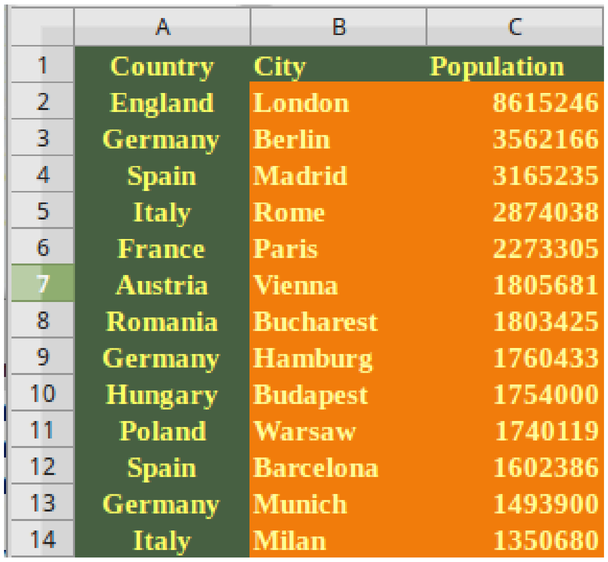
We still have the problem that integer values - which means values which should be integers like number of people - are converted to float as soon as we have NaN values. We can solve this problem now with the method 'fillna':

```
cities = {"London":   8615246,
          "Berlin":   3562166,
          "Madrid":   3165235,
          "Rome":     2874038,
          "Paris":    2273305,
          "Vienna":   1805681,
          "Bucharest":1803425,
          "Hamburg":  1760433,
          "Budapest": 1754000,
          "Warsaw":   1740119,
          "Barcelona":1602386,
          "Munich":   1493900,
          "Milan":    1350680}

my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                            index=my_cities)
my_city_series = my_city_series.fillna(0).astype(int)
print(my_city_series)
```

```
London       8615246
Paris        2273305
Zurich             0
Berlin       3562166
Stuttgart          0
Hamburg      1760433
dtype: int64
```

DATAFRAME

The underlying idea of a DataFrame is based on spreadsheets. We can see the data structure of a DataFrame as tabular and spreadsheet-like. A DataFrame logically corresponds to a "sheet" of an Excel document. A DataFrame has both a row and a column index.



	A	B	C
1	Country	City	Population
2	England	London	8615246
3	Germany	Berlin	3562166
4	Spain	Madrid	3165235
5	Italy	Rome	2874038
6	France	Paris	2273305
7	Austria	Vienna	1805681
8	Romania	Bucharest	1803425
9	Germany	Hamburg	1760433
10	Hungary	Budapest	1754000
11	Poland	Warsaw	1740119
12	Spain	Barcelona	1602386
13	Germany	Munich	1493900
14	Italy	Milan	1350680

Like a spreadsheet or Excel sheet, a DataFrame object contains an ordered collection of columns. Each column consists of a unique data typye, but different columns can have different types, e.g. the first column may consist of integers, while the second one consists of boolean values and so on.

There is a close connection between the DataFrames and the Series of Pandas. A DataFrame can be seen as a concatenation of Series, each Series having the same index, i.e. the index of the DataFrame.

We will demonstrate this in the following example.

We define the following three Series:

```
import pandas as pd

years = range(2014, 2018)

shop1 = pd.Series([2409.14, 2941.01, 3496.83, 3119.55], index=year
s)
shop2 = pd.Series([1203.45, 3441.62, 3007.83, 3619.53], index=year
s)
shop3 = pd.Series([3412.12, 3491.16, 3457.19, 1963.10], index=year
s)
```

What happens, if we concatenate these "shop" Series? Pandas provides a concat function for this purpose:

```
pd.concat([shop1, shop2, shop3])
```

Output:

```
2014    2409.14
2015    2941.01
2016    3496.83
2017    3119.55
2014    1203.45
2015    3441.62
2016    3007.83
2017    3619.53
2014    3412.12
2015    3491.16
2016    3457.19
2017    1963.10
dtype: float64
```

This result is not what we have intended or expected. The reason is that concat used 0 as the default for the axis parameter. Let's do it with "axis=1":

```
shops_df = pd.concat([shop1, shop2, shop3], axis=1)
shops_df
```

Output:

	0	**1**	**2**
2014	2409.14	1203.45	3412.12
2015	2941.01	3441.62	3491.16
2016	3496.83	3007.83	3457.19
2017	3119.55	3619.53	1963.10

Let's do some fine sanding by giving names to the columns:

```
cities = ["Zürich", "Winterthur", "Freiburg"]
shops_df.columns = cities
print(shops_df)

# alternative way: give names to series:
shop1.name = "Zürich"
```

```
shop2.name = "Winterthur"
shop3.name = "Freiburg"

print("------")
shops_df2 = pd.concat([shop1, shop2, shop3], axis=1)
print(shops_df2)
```

```
       Zürich  Winterthur  Freiburg
2014  2409.14     1203.45   3412.12
2015  2941.01     3441.62   3491.16
2016  3496.83     3007.83   3457.19
2017  3119.55     3619.53   1963.10
------
       Zürich  Winterthur  Freiburg
2014  2409.14     1203.45   3412.12
2015  2941.01     3441.62   3491.16
2016  3496.83     3007.83   3457.19
2017  3119.55     3619.53   1963.10
```

This was nice, but what kind of data type is our result?

```
print(type(shops_df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

This means, we can arrange or concat Series into DataFrames!

DATAFRAMES FROM DICTIONARIES

A DataFrame has a row and column index; it's like a dict of Series with a common index.

```
cities = {"name": ["London", "Berlin", "Madrid", "Rome",
                   "Paris", "Vienna", "Bucharest", "Hamburg",
                   "Budapest", "Warsaw", "Barcelona",
                   "Munich", "Milan"],
          "population": [8615246, 3562166, 3165235, 2874038,
                         2273305, 1805681, 1803425, 1760433,
                         1754000, 1740119, 1602386, 1493900,
                         1350680],
          "country": ["England", "Germany", "Spain", "Italy",
                      "France", "Austria", "Romania",
                      "Germany", "Hungary", "Poland", "Spain",
                      "Germany", "Italy"]}
```

```
city_frame = pd.DataFrame(cities)
city_frame
```

Output:

	name	population	country
0	London	8615246	England
1	Berlin	3562166	Germany
2	Madrid	3165235	Spain
3	Rome	2874038	Italy
4	Paris	2273305	France
5	Vienna	1805681	Austria
6	Bucharest	1803425	Romania
7	Hamburg	1760433	Germany
8	Budapest	1754000	Hungary
9	Warsaw	1740119	Poland
10	Barcelona	1602386	Spain
11	Munich	1493900	Germany
12	Milan	1350680	Italy

RETRIEVING THE COLUMN NAMES

It's possible to get the names of the columns as a list:

```
city_frame.columns.values
```

Output:

```
array(['name', 'population', 'country'], dtype=object)
```

CUSTOM INDEX

We can see that an index (0,1,2, ...) has been automatically assigned to the DataFrame. We can also assign a custom index to the DataFrame object:

```
ordinals = ["first", "second", "third", "fourth",
            "fifth", "sixth", "seventh", "eigth",
            "ninth", "tenth", "eleventh", "twelvth",
            "thirteenth"]
city_frame = pd.DataFrame(cities, index=ordinals)
city_frame
```

Output:

	name	population	country
first	London	8615246	England
second	Berlin	3562166	Germany
third	Madrid	3165235	Spain
fourth	Rome	2874038	Italy
fifth	Paris	2273305	France
sixth	Vienna	1805681	Austria
seventh	Bucharest	1803425	Romania
eigth	Hamburg	1760433	Germany
ninth	Budapest	1754000	Hungary
tenth	Warsaw	1740119	Poland
eleventh	Barcelona	1602386	Spain
twelvth	Munich	1493900	Germany
thirteenth	Milan	1350680	Italy

REARRANGING THE ORDER OF COLUMNS

We can also define and rearrange the order of the columns at the time of creation of the DataFrame. This makes also sure that we will have a defined ordering of our columns, if we create the DataFrame from a dictionary. Dictionaries are not ordered, as you have seen in our chapter on Dictionaries in our Python tutorial, so we cannot know in advance what the ordering of our columns will be:

```
city_frame = pd.DataFrame(cities,
                          columns=["name",
                                   "country",
                                   "population"])

city_frame
```

Output:

	name	country	population
0	London	England	8615246
1	Berlin	Germany	3562166
2	Madrid	Spain	3165235
3	Rome	Italy	2874038
4	Paris	France	2273305
5	Vienna	Austria	1805681
6	Bucharest	Romania	1803425
7	Hamburg	Germany	1760433
8	Budapest	Hungary	1754000
9	Warsaw	Poland	1740119
10	Barcelona	Spain	1602386
11	Munich	Germany	1493900
12	Milan	Italy	1350680

We change both the column order and the ordering of the index with the function `reindex` with the following code:

```
city_frame.reindex(index=[0, 2, 4, 6,  8, 10, 12, 1, 3, 5, 7, 9, 1
1],
                  columns=['country', 'name', 'population'])
```

Output:

	country	name	population
0	England	London	8615246
2	Spain	Madrid	3165235
4	France	Paris	2273305
6	Romania	Bucharest	1803425
8	Hungary	Budapest	1754000
10	Spain	Barcelona	1602386
12	Italy	Milan	1350680
1	Germany	Berlin	3562166
3	Italy	Rome	2874038
5	Austria	Vienna	1805681
7	Germany	Hamburg	1760433
9	Poland	Warsaw	1740119
11	Germany	Munich	1493900

Now, we want to rename our columns. For this purpose, we will use the DataFrame method 'rename'. This method supports two calling conventions

- (index=index_mapper, columns=columns_mapper, ...)
- (mapper, axis={'index', 'columns'}, ...)

We will rename the columns of our DataFrame into Romanian names in the following example. We set the parameter inplace to True so that our DataFrame will be changed instead of returning a new DataFrame, if inplace is set to False, which is the default!

```
city_frame.rename(columns={"name":"Soyadı",
                           "country":"Ülke",
                                        "population":"Nüfus"},
                          inplace=True)
city_frame
```

Output:

	Soyadı	Ülke	Nüfus
0	London	England	8615246
1	Berlin	Germany	3562166
2	Madrid	Spain	3165235
3	Rome	Italy	2874038
4	Paris	France	2273305
5	Vienna	Austria	1805681
6	Bucharest	Romania	1803425
7	Hamburg	Germany	1760433
8	Budapest	Hungary	1754000
9	Warsaw	Poland	1740119
10	Barcelona	Spain	1602386
11	Munich	Germany	1493900
12	Milan	Italy	1350680

EXISTING COLUMN AS THE INDEX OF A DATAFRAME

We want to create a more useful index in the following example. We will use the country name as the index, i.e. the list value associated to the key "country" of our cities dictionary:

```
city_frame = pd.DataFrame(cities,
                          columns=["name", "population"],
                          index=cities["country"])

city_frame
```

Output:

	name	population
England	London	8615246
Germany	Berlin	3562166
Spain	Madrid	3165235
Italy	Rome	2874038
France	Paris	2273305
Austria	Vienna	1805681
Romania	Bucharest	1803425
Germany	Hamburg	1760433
Hungary	Budapest	1754000
Poland	Warsaw	1740119
Spain	Barcelona	1602386
Germany	Munich	1493900
Italy	Milan	1350680

Alternatively, we can change an existing DataFrame. We can us the method set_index to turn a column into an index. "set_index" does not work in-place, it returns a new data frame with the chosen column as the index:

```
city_frame = pd.DataFrame(cities)
city_frame2 = city_frame.set_index("country")
print(city_frame2)
```

```
              name  population
country
England     London     8615246
Germany     Berlin     3562166
Spain       Madrid     3165235
Italy         Rome     2874038
France       Paris     2273305
Austria     Vienna     1805681
Romania  Bucharest     1803425
Germany    Hamburg     1760433
Hungary   Budapest     1754000
Poland      Warsaw     1740119
Spain    Barcelona     1602386
Germany     Munich     1493900
Italy        Milan     1350680
```

We saw in the previous example that the set_index method returns a new DataFrame object and doesn't change the original DataFrame. If we set the optional parameter "inplace" to True, the DataFrame will be changed in place, i.e. no new object will be created:

```
city_frame = pd.DataFrame(cities)
city_frame.set_index("country", inplace=True)
print(city_frame)
```

```
              name  population
country
England     London     8615246
Germany     Berlin     3562166
Spain       Madrid     3165235
Italy         Rome     2874038
France       Paris     2273305
Austria     Vienna     1805681
Romania  Bucharest     1803425
Germany    Hamburg     1760433
Hungary   Budapest     1754000
Poland      Warsaw     1740119
Spain    Barcelona     1602386
Germany     Munich     1493900
Italy        Milan     1350680
```

ACCESSING ROWS VIA INDEX VALUES

So far we have accessed DataFrames via the columns. It is often necessary to select certain rows via the index names. We will demonstrate now, how we can access rows from DataFrames via the locators 'loc' and 'iloc'. We will not cover 'ix' because it is deprecated and will be removed in the future.

We select all the German cities in the following example by using 'loc'. The result is a DataFrame:

```
city_frame = pd.DataFrame(cities,
                          columns=("name", "population"),
                          index=cities["country"])
print(city_frame.loc["Germany"])
```

```
            name  population
Germany   Berlin     3562166
Germany  Hamburg     1760433
Germany   Munich     1493900
```

It is also possible to simultaneously extracting rows by chosen more than on index labels. To do this we use a list of indices:

```
print(city_frame.loc[["Germany", "France"]])
```

```
            name  population
Germany   Berlin     3562166
Germany  Hamburg     1760433
Germany   Munich     1493900
France     Paris     2273305
```

We will also need to select pandas DataFrame rows based on conditions, which are applied to column values. We can use the operators '>', '=', '=', '<=', '!=' for this purpose. We select all cities with a population of more than two million in the following example:

```
condition = city_frame.population>2000000
condition
```

Output:
```
England     True
Germany     True
Spain       True
Italy       True
France      True
Austria    False
Romania    False
Germany    False
Hungary    False
Poland     False
Spain      False
Germany    False
Italy      False
Name: population, dtype: bool
```

We can use this Boolean DataFrame `condition` with `loc` to finally create the selection:

```
print(city_frame.loc[condition])
```

```
            name  population
England   London     8615246
Germany   Berlin     3562166
Spain     Madrid     3165235
Italy       Rome     2874038
France     Paris     2273305
```

It is also possible to logically combine more than one condition with `&` and `|`:

```
condition1 = (city_frame.population>1500000)
condition2 = (city_frame['name'].str.contains("m"))
print(city_frame.loc[condition1 & condition2])
```

```
             name  population
Italy        Rome     2874038
Germany   Hamburg     1760433
```

We use a logical or `|` in the following example to see all cities of the Pandas DataFrame, where either the city name contains the letter 'm' or the population number is greater than three million:

```
condition1 = (city_frame.population>3000000)
condition2 = (city_frame['name'].str.contains("m"))
print(city_frame.loc[condition1 | condition2])
```

```
             name  population
England    London     8615246
Germany    Berlin     3562166
Spain      Madrid     3165235
Italy        Rome     2874038
Germany   Hamburg     1760433
```

ADDING ROWS TO A DATAFRAME

```
milan = ['Milan', 1399860]
city_frame.iloc[-1] = milan
city_frame.loc['Switzerland'] = ['Zurich', 415215]
city_frame
```

Output:

	name	population
England	London	8615246
Germany	Berlin	3562166
Spain	Madrid	3165235
Italy	Rome	2874038
France	Paris	2273305
Austria	Vienna	1805681
Romania	Bucharest	1803425
Germany	Hamburg	1760433
Hungary	Budapest	1754000
Poland	Warsaw	1740119
Spain	Barcelona	1602386
Germany	Munich	1493900
Italy	Milan	1399860
Switzerland	Zurich	415215

ACCESSING ROWS BY POSITION

The `iloc` method of a Pandas DataFrame object can be used to select rows and columns by number, i.e. in the order that they appear in the data frame. `iloc` allows selections of the rows, as if they were numbered by integers `0` , `1` , `2` , ...

We demonstrate this in the following example:

```
df = city_frame.iloc[3]
print(df)
```

```
name                Rome
population       2874038
Name: Italy, dtype: object
```

To get a DataFrame with selected rows by numbers, we use a list of integers. We can see that we can change the order of the rows and we are also able to select rows multiple times:

```
df = city_frame.iloc[[3, 2, 0, 5, 0]]
print(df)
```

```
            name  population
Italy       Rome     2874038
Spain     Madrid     3165235
England   London     8615246
Austria   Vienna     1805681
England   London     8615246
```

SUM AND CUMULATIVE SUM

The DataFrame object of Pandas provides a method to sum both columns and rows. Before we will explain the usage of the sum method, we will create a new DataFrame object on which we will apply our examples. We will start by creating an empty DataFrame without columns but an index. We populate this DataFrame by adding columns with random values:

```
years = range(2014, 2019)
cities = ["Zürich", "Freiburg", "München", "Konstanz", "Saarbrücke
n"]
shops = pd.DataFrame(index=years)
for city in cities:
    shops.insert(loc=len(shops.columns),
                 column=city,
                 value=(np.random.uniform(0.7, 1, (5,)) * 1000).ro
und(2))

print(shops)
```

```
      Zürich  Freiburg  München  Konstanz  Saarbrücken
2014  872.51    838.22   961.17    934.99       796.42
2015  944.47    943.27   862.66    784.23       770.94
2016  963.22    859.97   818.13    965.38       995.74
2017  866.11    731.42   811.37    955.21       836.36
2018  790.95    837.39   941.92    735.93       837.23
```

Let's apply `sum` to the DataFrame `shops` :

```
shops.sum()
```

Output:
```
Zürich         4504.85
Freiburg       4182.09
München        3999.67
Konstanz       4557.86
Saarbrücken    4608.05
dtype: float64
```

We can see that it summed up all the columns of our DataFrame. What about calculating the sum of the rows? We can do this by using the `axis` parameter of `sum`.

```
shops.sum(axis=1)
```

Output:
```
2014    4226.30
2015    4458.91
2016    4696.46
2017    4186.20
2018    4284.65
dtype: float64
```

You only want to the the sums for the first, third and the last column and for all the years:

```
s = shops.iloc[:, [0, 2, -1]]
print(s)
print("and now the sum:")
print(s.sum())
```

```
      Zürich  München  Saarbrücken
2014  780.38   952.88       854.93
2015  968.05   807.03       894.87
2016  990.00   803.41       991.15
2017  815.45   734.82       950.40
2018  950.97   701.53       916.70
and now the sum:
Zürich         4504.85
München        3999.67
Saarbrücken    4608.05
dtype: float64
```

Of course, you could have also have achieved it in the following way, if the column names are known:

```
shops[["Zürich", "München", "Saarbrücken"]].sum()
```

Output:
```
Zürich          4504.85
München         3999.67
Saarbrücken     4608.05
dtype: float64
```

We can use "cumsum" to calculate the cumulative sum over the years:

```
x = shops.cumsum()
print(x)
```

```
       Zürich  Freiburg  München  Konstanz  Saarbrücken
2014   780.38    908.77   952.88    729.34       854.93
2015  1748.43   1719.42  1759.91   1707.65      1749.80
2016  2738.43   2698.64  2563.32   2640.33      2740.95
2017  3553.88   3424.87  3298.14   3599.63      3691.35
2018  4504.85   4182.09  3999.67   4557.86      4608.05
```

Using the keyword parameter `axis` with the value 1, we can build the cumulative sum over the rows:

```
x = shops.cumsum(axis=1)
print(x)
```

```
      Zürich  Freiburg  München  Konstanz  Saarbrücken
2014  780.38   1689.15  2642.03   3371.37      4226.30
2015  968.05   1778.70  2585.73   3564.04      4458.91
2016  990.00   1969.22  2772.63   3705.31      4696.46
2017  815.45   1541.68  2276.50   3235.80      4186.20
2018  950.97   1708.19  2409.72   3367.95      4284.65
```

ASSIGNING NEW VALUES TO COLUMNS

x is a Pandas Series. We can reassign the previously calculated cumulative sums to the population column:

```
city_frame["population"] = x
print(city_frame)
```

```
               name  population
England      London     8615246
Germany      Berlin    12177412
Spain        Madrid    15342647
Italy          Rome    18216685
France        Paris    20489990
Austria      Vienna    22295671
Romania   Bucharest    24099096
Germany     Hamburg    25859529
Hungary    Budapest    27613529
Poland       Warsaw    29353648
Spain     Barcelona    30956034
Germany      Munich    32449934
Italy         Milan    33800614
```

Instead of replacing the values of the population column with the cumulative sum, we want to add the cumulative population sum as a new culumn with the name "cum_population".

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "population",
                                   "cum_population"],
                          index=cities["name"])

city_frame
```

Output:

	country	population	cum_population
London	England	8615246	NaN
Berlin	Germany	3562166	NaN
Madrid	Spain	3165235	NaN
Rome	Italy	2874038	NaN
Paris	France	2273305	NaN
Vienna	Austria	1805681	NaN
Bucharest	Romania	1803425	NaN
Hamburg	Germany	1760433	NaN
Budapest	Hungary	1754000	NaN
Warsaw	Poland	1740119	NaN
Barcelona	Spain	1602386	NaN
Munich	Germany	1493900	NaN
Milan	Italy	1350680	NaN

We can see that the column "cum_population" is set to Nan, as we haven't provided any data for it.

We will assign now the cumulative sums to this column:

```
city_frame["cum_population"] = city_frame["population"].cumsum()
city_frame
```

Output:

	country	population	cum_population
London	England	8615246	8615246
Berlin	Germany	3562166	12177412
Madrid	Spain	3165235	15342647
Rome	Italy	2874038	18216685
Paris	France	2273305	20489990
Vienna	Austria	1805681	22295671
Bucharest	Romania	1803425	24099096
Hamburg	Germany	1760433	25859529
Budapest	Hungary	1754000	27613529
Warsaw	Poland	1740119	29353648
Barcelona	Spain	1602386	30956034
Munich	Germany	1493900	32449934
Milan	Italy	1350680	33800614

We can also include a column name which is not contained in the dictionary, when we create the DataFrame from the dictionary. In this case, all the values of this column will be set to NaN:

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "area",
                                   "population"],
                          index=cities["name"])
print(city_frame)
```

```
           country area  population
London     England  NaN     8615246
Berlin     Germany  NaN     3562166
Madrid       Spain  NaN     3165235
Rome         Italy  NaN     2874038
Paris       France  NaN     2273305
Vienna     Austria  NaN     1805681
Bucharest  Romania  NaN     1803425
Hamburg    Germany  NaN     1760433
Budapest   Hungary  NaN     1754000
Warsaw      Poland  NaN     1740119
Barcelona    Spain  NaN     1602386
Munich     Germany  NaN     1493900
Milan        Italy  NaN     1350680
```

ACCESSING THE COLUMNS OF A DATAFRAME

There are two ways to access a column of a DataFrame. The result is in both cases a Series:

```
# in a dictionary-like way:
print(city_frame["population"])
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
Name: population, dtype: int64
```

```
# as an attribute
print(city_frame.population)
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
Name: population, dtype: int64
```

```
print(type(city_frame.population))
```

```
<class 'pandas.core.series.Series'>
```

```
city_frame.population
```

Output:

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
Name: population, dtype: int64
```

From the previous example, we can see that we have not copied the population column. "p" is a view on the data of city_frame.

ASSIGNING NEW VALUES TO A COLUMN

The column area is still not defined. We can set all elements of the column to the same value:

```
city_frame["area"] = 1572
print(city_frame)
```

```
           country  area  population
London     England  1572     8615246
Berlin     Germany  1572     3562166
Madrid       Spain  1572     3165235
Rome         Italy  1572     2874038
Paris       France  1572     2273305
Vienna     Austria  1572     1805681
Bucharest  Romania  1572     1803425
Hamburg    Germany  1572     1760433
Budapest   Hungary  1572     1754000
Warsaw      Poland  1572     1740119
Barcelona    Spain  1572     1602386
Munich     Germany  1572     1493900
Milan        Italy  1572     1350680
```

In this case, it will be definitely better to assign the exact area to the cities. The list with the area values needs to have the same length as the number of rows in our DataFrame.

```
# area in square km:
area = [1572, 891.85, 605.77, 1285,
        105.4, 414.6, 228, 755,
        525.2, 517, 101.9, 310.4,
        181.8]
# area could have been designed as a list, a Series, an array or
a scalar

city_frame["area"] = area
print(city_frame)
```

```
           country     area  population
London     England  1572.00     8615246
Berlin     Germany   891.85     3562166
Madrid       Spain   605.77     3165235
Rome         Italy  1285.00     2874038
Paris       France   105.40     2273305
Vienna     Austria   414.60     1805681
Bucharest  Romania   228.00     1803425
Hamburg    Germany   755.00     1760433
Budapest   Hungary   525.20     1754000
Warsaw      Poland   517.00     1740119
Barcelona    Spain   101.90     1602386
Munich     Germany   310.40     1493900
Milan        Italy   181.80     1350680
```

SORTING DATAFRAMES

Let's sort our DataFrame according to the city area:

```
city_frame = city_frame.sort_values(by="area", ascending=False)
print(city_frame)
```

```
           country     area  population
London     England  1572.00     8615246
Rome         Italy  1285.00     2874038
Berlin     Germany   891.85     3562166
Hamburg    Germany   755.00     1760433
Madrid       Spain   605.77     3165235
Budapest   Hungary   525.20     1754000
Warsaw      Poland   517.00     1740119
Vienna     Austria   414.60     1805681
Munich     Germany   310.40     1493900
Bucharest  Romania   228.00     1803425
Milan        Italy   181.80     1350680
Paris       France   105.40     2273305
Barcelona    Spain   101.90     1602386
```

Let's assume, we have only the areas of London, Hamburg and Milan. The areas are in a series with the correct indices. We can assign this series as well:

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "area",
                                   "population"],
                          index=cities["name"])

some_areas = pd.Series([1572, 755, 181.8],
                    index=['London', 'Hamburg', 'Milan'])

city_frame['area'] = some_areas
print(city_frame)
```

```
             country    area  population
London       England  1572.0     8615246
Berlin       Germany     NaN     3562166
Madrid         Spain     NaN     3165235
Rome           Italy     NaN     2874038
Paris         France     NaN     2273305
Vienna       Austria     NaN     1805681
Bucharest    Romania     NaN     1803425
Hamburg      Germany   755.0     1760433
Budapest     Hungary     NaN     1754000
Warsaw        Poland     NaN     1740119
Barcelona      Spain     NaN     1602386
Munich       Germany     NaN     1493900
Milan          Italy   181.8     1350680
```

INSERTING NEW COLUMNS INTO EXISTING DATAFRAMES

In the previous example we have added the column area at creation time. Quite often it will be necessary to add or insert columns into existing DataFrames. For this purpose the DataFrame class provides a method "insert", which allows us to insert a column into a DataFrame at a specified location:

```
insert(self, loc, column, value, allow_duplicates=False)`
```

The parameters are specified as:

Parameter	Meaning
loc	int
	This value should be within the range 0 <= loc <= len(columns)
column	the column name
value	can be a list, a Series an array or a scalar
allow_duplicates	If `allow_duplicates` is False, an Exception will be raised, if column is already contained in the DataFrame.

```
city_frame = pd.DataFrame(cities,
                          columns=["country",
                                   "population"],
                          index=cities["name"])
```

```
idx = 1
city_frame.insert(loc=idx, column='area', value=area)
city_frame
```

Output:

	country	area	population
London	England	1572.00	8615246
Berlin	Germany	891.85	3562166
Madrid	Spain	605.77	3165235
Rome	Italy	1285.00	2874038
Paris	France	105.40	2273305
Vienna	Austria	414.60	1805681
Bucharest	Romania	228.00	1803425
Hamburg	Germany	755.00	1760433
Budapest	Hungary	525.20	1754000
Warsaw	Poland	517.00	1740119
Barcelona	Spain	101.90	1602386
Munich	Germany	310.40	1493900
Milan	Italy	181.80	1350680

CREATING A DATAFRAME BY APPENDING ROWS

```
import pandas as pd
from numpy.random import randint

df = pd.DataFrame(columns=['lib', 'qty1', 'qty2'])
for i in range(5):
     df.loc[i] = ['name' + str(i)] + list(randint(10, size=2))
```

```
df
```

Output:

	lib	qty1	qty2
0	name0	1	8
1	name1	5	6
2	name2	9	9
3	name3	8	8
4	name4	7	0

DATAFRAME FROM NESTED DICTIONARIES

A nested dictionary of dicts can be passed to a DataFrame as well. The indices of the outer dictionary are taken as the the columns and the inner keys. i.e. the keys of the nested dictionaries, are used as the row indices:

```
growth = {"Switzerland": {"2010": 3.0, "2011": 1.8, "2012": 1.1,
"2013": 1.9},
          "Germany": {"2010": 4.1, "2011": 3.6, "2012":
0.4, "2013": 0.1},
          "France": {"2010":2.0,  "2011":2.1, "2012": 0.3, "201
3": 0.3},
          "Greece": {"2010":-5.4, "2011":-8.9, "2012":-6.6, "201
3":        -3.3},
          "Italy": {"2010":1.7, "2011":        0.6, "2012":-2.3,
"2013":-1.9}
          }
```

```
growth_frame = pd.DataFrame(growth)
growth_frame
```

Output:

	Switzerland	Germany	France	Greece	Italy
2010	3.0	4.1	2.0	-5.4	1.7
2011	1.8	3.6	2.1	-8.9	0.6
2012	1.1	0.4	0.3	-6.6	-2.3
2013	1.9	0.1	0.3	-3.3	-1.9

You like to have the years in the columns and the countries in the rows? No problem, you can transpose the data:

```
growth_frame.T
```

Output:

	2010	2011	2012	2013
Switzerland	3.0	1.8	1.1	1.9
Germany	4.1	3.6	0.4	0.1
France	2.0	2.1	0.3	0.3
Greece	-5.4	-8.9	-6.6	-3.3
Italy	1.7	0.6	-2.3	-1.9

```
growth_frame = growth_frame.T

growth_frame2 = growth_frame.reindex(["Switzerland",
                                      "Italy",
                                      "Germany",
                                      "Greece"])
print(growth_frame2)
```

```
             2010  2011  2012  2013
Switzerland   3.0   1.8   1.1   1.9
Italy         1.7   0.6  -2.3  -1.9
Germany       4.1   3.6   0.4   0.1
Greece       -5.4  -8.9  -6.6  -3.3
```

FILLING A DATAFRAME WITH RANDOM VALUES:

```
import numpy as np

names = ['Frank', 'Eve', 'Stella', 'Guido', 'Lara']
index = ["January", "February", "March",
         "April", "May", "June",
         "July", "August", "September",
         "October", "November", "December"]
df = pd.DataFrame((np.random.randn(12, 5)*1000).round(2),
                  columns=names,
                  index=index)
df
```

Output:

	Frank	Eve	Stella	Guido	Lara
January	-2452.01	-91.03	-122.31	-750.05	931.99
February	-874.08	-189.72	628.49	-392.53	342.06
March	-1222.37	-1125.60	-826.90	-107.74	-831.32
April	1333.69	1468.56	-1537.82	-743.23	624.10
May	802.27	-1444.17	84.38	-376.80	256.64
June	-762.07	207.63	-476.06	-2068.54	-1614.25
July	-1172.28	-2582.39	-351.16	-555.88	-21.40
August	-468.52	-665.97	833.53	-1650.37	-426.07
September	-788.57	408.96	-652.73	346.99	875.78
October	92.81	699.84	251.47	872.04	415.53
November	558.65	-1699.90	-949.95	212.92	-1874.80
December	-646.58	-35.22	-1018.20	-644.35	776.56

ACCESSING AND CHANGING VALUES OF DATAFRAMES

We have seen in the previous chapters of our tutorial many ways to create Series and DataFrames. We also learned how to access and replace complete columns. This chapter of our Pandas and Python tutorial will show various ways to access and change selectively values in Pandas DataFrames and Series. We will show ways how to change single value or values matching strings or regular expressions.

For this purpose we will learn to know the methods `loc`, `at` and `replace`

We will work with the following DataFrame structure in our examples:

```
import pandas as pd

first = ('Mike', 'Dorothee', 'To
m', 'Bill', 'Pete', 'Kate')
last = ('Meyer', 'Maier', 'Meyer', 'Mayer', 'Meyr', 'Mair')
job = ('data analyst', 'programmer', 'computer scientist',
       'data scientist', 'accountant', 'psychiatrist')
language = ('Python', 'Perl', 'Java', 'Java', 'Cobol', 'Brainfuc
k')

df = pd.DataFrame(list(zip(last, job, language)),
                  columns =['last', 'job', 'language'],
                  index=first)
df
```

Output:

	last	job	language
Mike	Meyer	data analyst	Python
Dorothee	Maier	programmer	Perl
Tom	Meyer	computer scientist	Java
Bill	Mayer	data scientist	Java
Pete	Meyr	accountant	Cobol
Kate	Mair	psychiatrist	Brainfuck

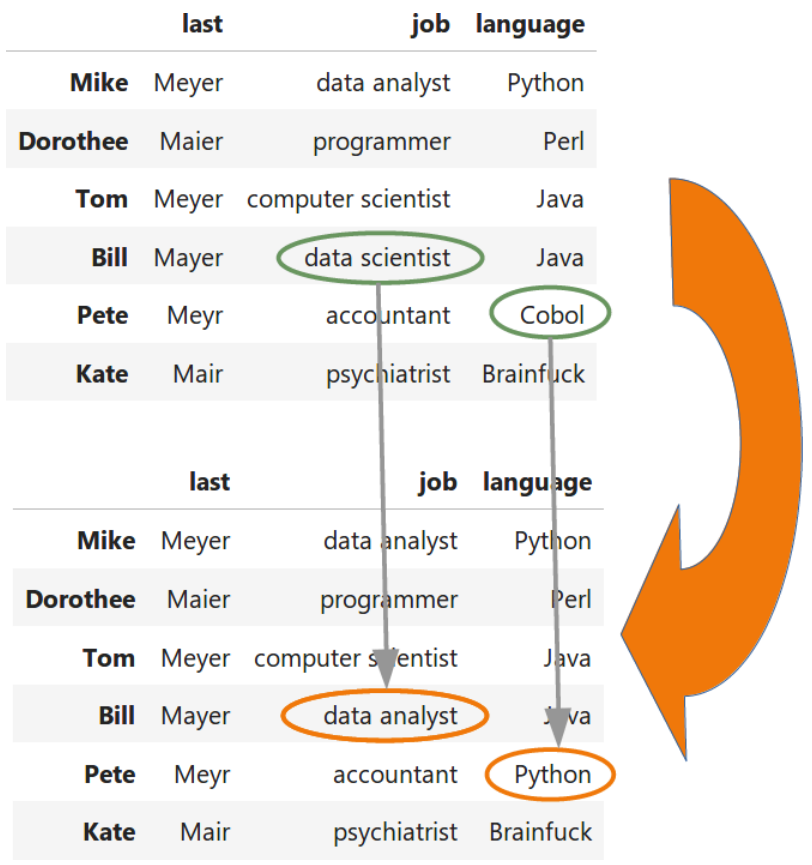
CHANGING ONE VALUE IN DATAFRAME

Pandas provides two ways, i.e. `loc` and `at`, to access or change a single value of a DataFrame. We will experiment with the height of Bill in the following Python code:

```
# accessing the job of Bill:
print(df.loc['Bill', 'job'])
# alternative way to access it with at:
print(df.at['Bill', 'job'])

# setting the job of Bill to 'data analyst' with 'loc'
df.loc['Bill', 'job'] = 'data analyst'
# let us check it:
print(df.loc['Bill', 'job'])

# setting the job of Bill to 'computer scientist' with 'at'
df.at['Pete', 'language'] = 'Python'
```

```
data scientist
data scientist
data analyst
```

The following image shows what we have done:



	last	job	language
Mike	Meyer	data analyst	Python
Dorothee	Maier	programmer	Perl
Tom	Meyer	computer scientist	Java
Bill	Mayer	data scientist	Java
Pete	Meyr	accountant	Cobol
Kate	Mair	psychiatrist	Brainfuck

	last	job	language
Mike	Meyer	data analyst	Python
Dorothee	Maier	programmer	Perl
Tom	Meyer	computer scientist	Java
Bill	Mayer	data analyst	Java
Pete	Meyr	accountant	Python
Kate	Mair	psychiatrist	Brainfuck

You will ask yourself now which one you should use? The help on the `at` method says the following: "Access a single value for a row/column label pair. Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series." `loc` on the other hand can be used to access a single value but also to access a group of rows and columns by a label or labels.

Another intestering question is about the speed of both methods in comparison. We will measure the time behaviour in the following code examples:

```
%timeit df.loc['Bill', 'language'] = 'Python'
```

```
129 µs ± 2.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
```

```
%timeit df.at['Bill', 'language'] = 'Python'
```

```
4.69 µs ± 209 ns per loop (mean ± std. dev. of 7 runs, 100000 loop
s each)
```

When it comes to speed the answer is clear: we should definitely use `at`.

REPLACE

The syntax of `replace`:

```
replace(self,
        to_replace=None,
        value=None,
        inplace=False,
        limit=None,
        regex=False,
        method='pad')
```

This method replaces values given in `to_replace` with `value`. This differs from updating with `.loc` or `.iloc`, which requires you to specify a location to update with some value. With replace it is possible to replace values in a Series or DataFrame without knowing where they occur.

`replace` works both with Series and DataFrames.

We will explain the way of working of the method `replace` by discussing the different data types of the parameter `to_replace` individually:

TO_REPLACE CAN BE NUMERIC VALUE, STRING OR REGULAR EXPRESSION:

Data Type	processing
numeric	numeric values in the DataFrame (or Series) which are equal to `to_replace` will be replaced with parameter `value`
str	If the parameter `to_replace` consists of a string, all the strings of the DataFrame (or Series) which exactly match this string will be replaced by the value of the parameter `value`
regex	All strings inside of the DataFrame (or Series) which match the regular expression of `to_replace` will be replaced with `value`

Let's start with a Series example. We will change one value into another one. Be aware of the fact that `replace` by default creates a copy of the object in which all the values are replaced. This means that the parameter `inplace` is set to `False` by default.

```
s = pd.Series([27, 33, 13, 19])
s.replace(13, 42)
```

Output:
```
0     27
1     33
2     42
3     19
dtype: int64
```

If we really want to change the object `s` is referencing, we should set the `inplace` parameter to `True`:

```
s = pd.Series([27, 33, 13, 19])
s.replace(13, 42, inplace=True)
s
```

Output:
```
0     27
1     33
2     42
3     19
dtype: int64
```

We can also change multiple values into one single value, as you can see in the following example.

```
s = pd.Series([0, 1, 2, 3, 4])
s.replace([0, 1, 2], 42, inplace=True)
s
```

Output:
```
0     42
1     42
2     42
3      3
4      4
dtype: int64
```

We will show now, how replace can be used on DataFrames. For this purpose we will recreate the example from the beginning of this chapter of our tutorial. It will not be exactly the same though. More people have learned Python now, but unfortunately somebody has put in some spelling errors in the word Python. Pete has become a programmer now:

```
import pandas as pd

first = ('Mike', 'Dorothee', 'Tom', 'Bill', 'Pete', 'Kate')
last = ('Meyer', 'Maier', 'Meyer', 'Mayer', 'Meyr', 'Mair')
job = ('data analyst', 'programmer', 'computer scientist',
       'data scientist', 'programmer', 'psychiatrist')
language = ('Python', 'Perl', 'Java', 'Pithon', 'Pythen', 'Brainfu
ck')
```

```
df = pd.DataFrame(list(zip(last, job, language)),
                  columns =['last', 'job', 'language'],
                  index=first)

df
```

Output:

	last	job	language
Mike	Meyer	data analyst	Python
Dorothee	Maier	programmer	Perl
Tom	Meyer	computer scientist	Java
Bill	Mayer	data scientist	Pithon
Pete	Meyr	programmer	Pythen
Kate	Mair	psychiatrist	Brainfuck

Both ambitious Dorothee and enthusiastic Pete have finished a degree in Computer Science in the meantime. So we have to change our DataFrame accodingly:

```
df.replace("programmer",
           "computer scientist",
           inplace=True)
df
```

Output:

	last	job	language
Mike	Meyer	data analyst	Python
Dorothee	Maier	computer scientist	Perl
Tom	Meyer	computer scientist	Java
Bill	Mayer	data scientist	Pithon
Pete	Meyr	computer scientist	Pythen
Kate	Mair	psychiatrist	Brainfuck

TO_REPLACE CAN BE A LIST CONSISTING OF STR, REGEX OR NUMERIC OBJECTS:

`to_replace` can be a list consisting of str, regex or numeric objects. If both the values of `to_replace` and `value` are both lists, they **must** be the same length.

First, if `to_replace` and `value` are both lists, they **must** be the same length. Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. We slightly changed our DataFrame for the next example. The first names build a column now!

```
df = pd.DataFrame(list(zip(first, last, job, language)),
                  columns =['first', 'last', 'job', 'language'])
df
```

Output:

	first	last	job	language
0	Mike	Meyer	data analyst	Python
1	Dorothee	Maier	programmer	Perl
2	Tom	Meyer	computer scientist	Java
3	Bill	Mayer	data scientist	Pithon
4	Pete	Meyr	programmer	Pythen
5	Kate	Mair	psychiatrist	Brainfuck

'Mike' and 'Tom' are abbreviations of the real first names, so we will change them now. We are also glad to announce that Dorothee finally migrated from Perl to Python:

```
df.replace(to_replace=['Mike', 'Tom', 'Perl'],
           value= ['Michael', 'Thomas', 'Python'],
           inplace=True)
df
```

Output:

	first	last	job	language
0	Michael	Meyer	data analyst	Python
1	Dorothee	Maier	programmer	Python
2	Thomas	Meyer	computer scientist	Java
3	Bill	Mayer	data scientist	Pithon
4	Pete	Meyr	programmer	Pythen
5	Kate	Mair	psychiatrist	Brainfuck

USING REGULAR EXPRESSIONS

Now it is time to do something about the surnames in our DataFrame. All these names sound the same - at least for German speakers. Let us assume that we just found out that they should all be spelled the same, i.e. Mayer. Now the time has come for regular expression. The parameter `to_replace` can also be used with regular expressions. We will use a regular expression to unify the surnames. We will also fix the misspellings of Python:

```
df.replace(to_replace=[r'M[ea][iy]e?r', r'P[iy]th[eo]n'],
           value=['Mayer', 'Python'],
           regex=True,
           inplace=True)
df
```

Output:

	first	last	job	language
0	Michael	Mayer	data analyst	Python
1	Dorothee	Mayer	programmer	Python
2	Thomas	Mayer	computer scientist	Java
3	Bill	Mayer	data scientist	Python
4	Pete	Mayer	programmer	Python
5	Kate	Mayer	psychiatrist	Brainfuck

TO_REPLACE CAN BE A DICT

Dictionaries can be used to specify different replacement values for different existing values. For example, `{'a': 'b', 'y': 'z'}` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the `value` parameter should be `None` .

Ìf `replace` is applied on a DataFrame, a dict can specify that different values should be replaced in different columns. For example, `{'a': 1, 'b': 'z'}` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in `value` . The `value` parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.

If we use nested dictionaries like `{'A': {'foo': 'bar'}}` , they are interpreted like this: look in column 'A' for the value 'foo' and replace it with 'bar'. The `value` parameter has to be `None` in this case.

```
df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
                   'B': ['foo', 'bar', 'bloo', 'blee', 'bloo'],
                   'C': ['green', 'red', 'blue', 'yellow', 'gree
n']})

df.replace(to_replace={"A": {0: 42, 3: 33}, 'B': {'bloo': 'vlo
o'}},
           inplace=True)
df
```

Output:

	A	B	C
0	42	foo	green
1	1	bar	red
2	2	vloo	blue
3	33	blee	yellow
4	4	vloo	green

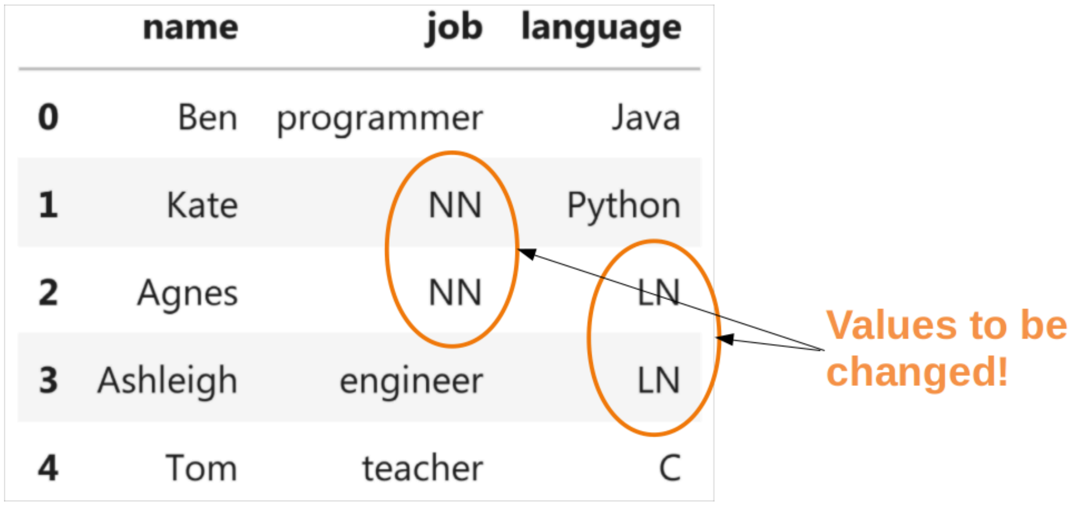
THE METHOD PARAMETER OF REPLACE:

When the parameter `value` is `None` and the parameter `to_replace` is a scalar, list or tuple, the method `replace` will use the parameter `method` to decide which replacement to perform. The possible values for `method` are `pad`, `ffill`, `bfill`, `None`. The default value is `pad`.

First of all, you have to know that `pad` and `ffill` are equivalent.

We will explain the way of working with the following Python example code. We will replace the values `NN` and `LN` of the following DataFrame in different ways by using the parameter `method`.

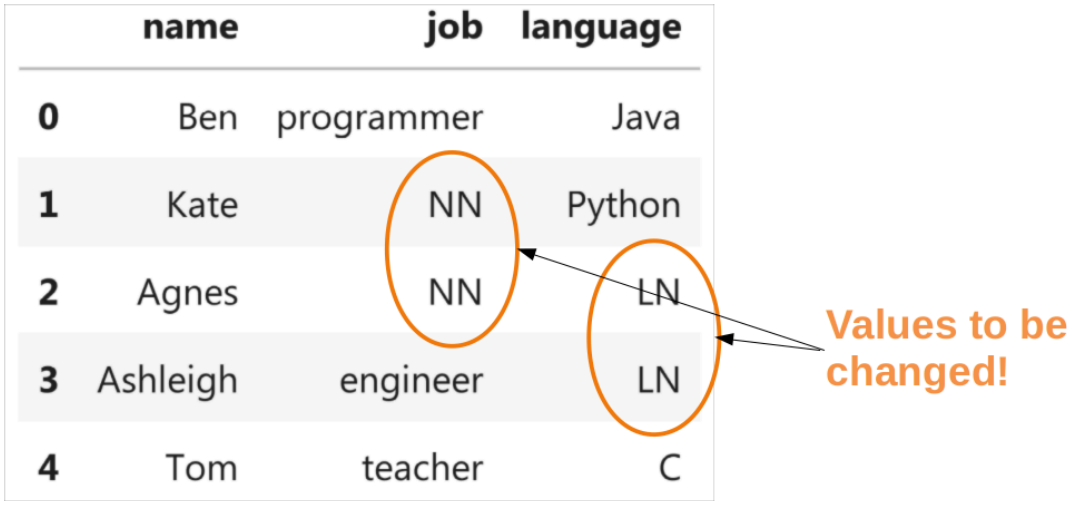
```
import pandas as pd

df = pd.DataFrame({
    'name':['Ben', 'Kate', 'Agnes', 'Ashleigh', 'Tom'],
    'job':['programmer', 'NN', 'NN', 'engineer', 'teacher'],
    'language':['Java', 'Python', 'LN', 'LN', 'C']})
```



	name	job	language
0	Ben	programmer	Java
1	Kate	NN	Python
2	Agnes	NN	LN
3	Ashleigh	engineer	LN
4	Tom	teacher	C

Values to be changed!

We will start with changing the 'NN' values by using `ffill` in the parameter `method`:



	name	job	language
0	Ben	programmer	Java
1	Kate	NN	Python
2	Agnes	NN	LN
3	Ashleigh	engineer	LN
4	Tom	teacher	C

Values to be changed!

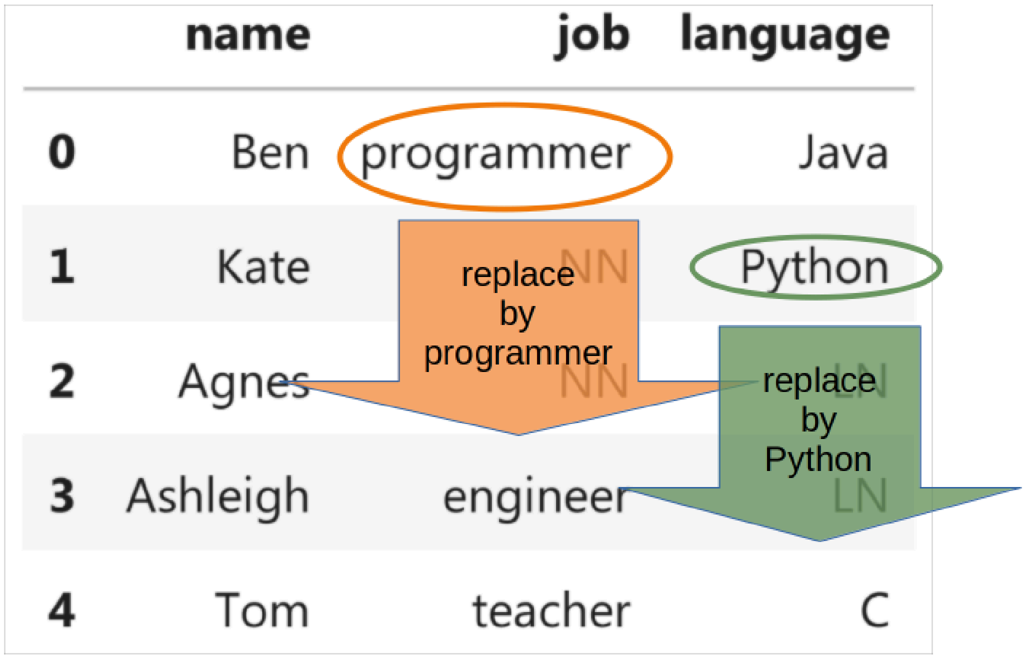
The picture above shows how the following call to `replace`. In words: Every occurrence of `NN` in the `job` column will have to be replaced. The value is defined by the parameter `method`. `ffill` means 'forward fill', i.e. we will take the preceding value to the first `NN` as the fill value. This is why we will replace the values by 'programmer':

```
# method is backfill
df.replace(to_replace='NN',
           value=None,
           method='ffill')
```

Output:

	name	job	language
0	Ben	programmer	Java
1	Kate	programmer	Python
2	Agnes	programmer	LN
3	Ashleigh	engineer	LN
4	Tom	teacher	C

Instead of using a single value for `to_replace` we can also use a list or tuple. We will replace in the following all occurrences of `NN` and `LN` accordingly, as we can see in the following picture:

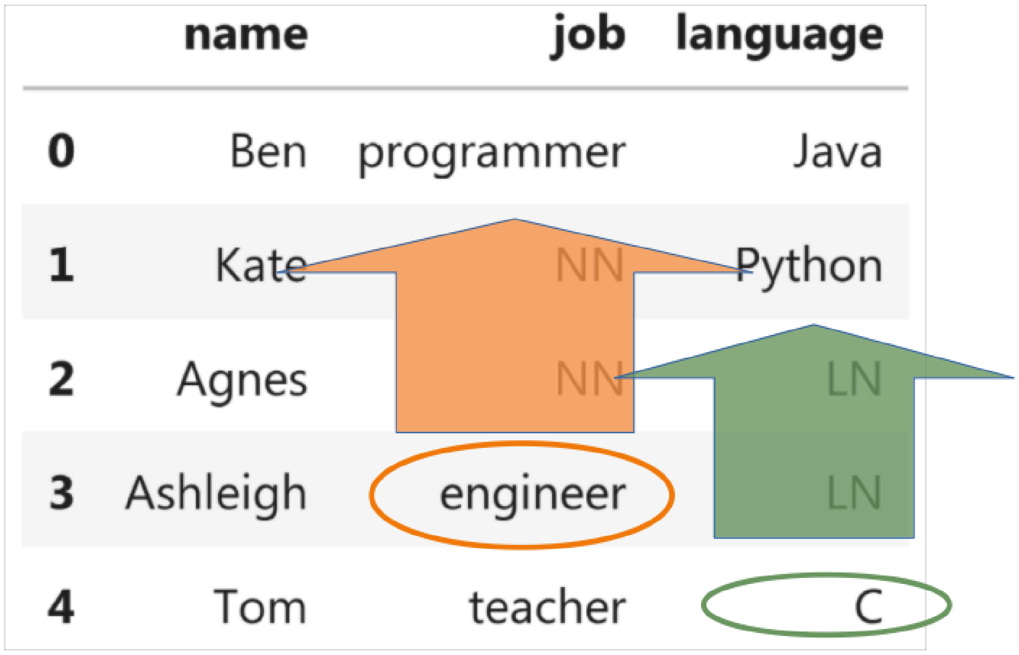


```
df.replace(to_replace=['NN', 'LN'],
           value=None,
           method='ffill')
```

Output:

	name	job	language
0	Ben	programmer	Java
1	Kate	programmer	Python
2	Agnes	programmer	Python
3	Ashleigh	engineer	Python
4	Tom	teacher	C

We will show now what happens, if we use `bfill` (backward fill). Now the occurrences of 'LN' become 'C' instead of Python. We also turn the 'NN's into engineers instead of programmers.



	name	job	language
0	Ben	programmer	Java
1	Kate	NN	Python
2	Agnes	NN	LN
3	Ashleigh	engineer	LN
4	Tom	teacher	C

```
df.replace(['NN', 'LN'], value=None, method='bfill')
```

Output:

	name	job	language
0	Ben	programmer	Java
1	Kate	engineer	Python
2	Agnes	engineer	C
3	Ashleigh	engineer	C
4	Tom	teacher	C

```
df.replace('NN',
           value=None,
           inplace=True,
           method='bfill')
df.replace('LN',
           value=None,
           inplace=True,
           method='ffill')
df
```

Output:

	name	job	language
0	Ben	programmer	Java
1	Kate	engineer	Python
2	Agnes	engineer	Python
3	Ashleigh	engineer	Python
4	Tom	teacher	C

OTHER EXAMPLES

We will present some more examples, which are taken from the help file of `loc`:

```
df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
                  index=['cobra', 'viper', 'sidewinder'],
                  columns=['max_speed', 'shield'])
```

```
df
```

Output:

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

Single label. Note this returns the row as a Series:

```
df.loc['viper']
```

Output:

```
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame:

```
df.loc[['viper', 'sidewinder']]
```

Output:

	max_speed	shield
viper	4	5
sidewinder	7	8

Single label for row and column

```
df.loc['cobra', 'shield']
```

Output:

```
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
df.loc['cobra':'viper', 'max_speed']
```

Output:
```
cobra     1
viper     4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
df.loc[[False, False, True]]
```

Output:

	max_speed	shield
sidewinder	7	8

Conditional that returns a boolean Series

```
df.loc[df['shield'] > 6]
```

Output:

	max_speed	shield
sidewinder	7	8

Conditional that returns a boolean Series with column labels specified

```
df.loc[df['shield'] > 6, ['max_speed']]
```

Output:

	max_speed
sidewinder	7

Callable that returns a boolean Series

```
df.loc[lambda df: df['shield'] == 8]
```

Output:

	max_speed	shield
sidewinder	7	8

Set value for all items matching the list of labels

```
df.loc[['viper', 'sidewinder'], ['shield']] = 50
df
```

Output:

	max_speed	shield
cobra	1	2
viper	4	50
sidewinder	7	50

Set value for an entire row

```
df.loc['cobra'] = 10
df
```

Output:

	max_speed	shield
cobra	10	10
viper	4	50
sidewinder	7	50

Set value for an entire column

```
df.loc[:, 'max_speed'] = 30
df
```

Output:

	max_speed	shield
cobra	30	10
viper	30	50
sidewinder	30	50

Set value for rows matching callable condition

```
df.loc[df['shield'] > 35] = 0
df
```

Output:

	max_speed	shield
cobra	30	10
viper	0	0
sidewinder	0	0

PANDAS: GROUPBY

This chapter of our Pandas tutorial deals with an extremely important functionality, i.e. `groupby`. It is not really complicated, but it is not obvious at first glance and is sometimes found to be difficult. Completely wrong, as we shall see. It is also very important to become familiar with 'groupby' because it can be used to solve important problems that would not be possible without it. The Pandas `groupby` operation involves some combination of splitting the object, applying a function, and combining the results. We can split a DataFrame object into groups based on various criteria and row and column-wise, i.e. using `axis`.

'Applying' means

- to filter the data,
- transform the data or
- aggregate the data.

`groupby` can be applied to Pandas Series objects and DataFrame objects! We will learn to understand how it works with many small practical examples in this tutorial.

GOUPBY WITH SERIES

We create with the following Python program a Series object with an index of size `nvalues`. The index will not be unique, because the strings for the index are taken from the list `fruits`, which has less elements than `nvalues`:

```
import pandas as pd
import numpy as np
import random

nvalues = 30
# we create random values, which will be used as the Series value
s:
values = np.random.randint(1, 20, (nvalues,))
fruits = ["bananas", "oranges", "apples", "clementines", "cherrie
s", "pears"]
fruits_index = np.random.choice(fruits, (nvalues,))

s = pd.Series(values, index=fruits_index)
```

```
print(s[:10])
```

```
bananas        19
oranges        12
clementines     6
oranges         6
clementines    11
bananas        17
clementines     5
apples          5
clementines    12
bananas         9
dtype: int64
```

```
grouped = s.groupby(s.index)
grouped
```

Output:

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7fda33
1c1050>
```

We can see that we get a `SeriesGroupBy` object, if we apply `groupby` on the index of our series object `s`. The result of this operation `grouped` is iterable. In every step we get a tuple object returned, which consists of an index label and a series object. The series object is `s` reduced to this label.

```
grouped = s.groupby(s.index)

for fruit, s_obj in grouped:
    print(f"===== {fruit} =====")
    print(s_obj)
```

```
===== apples =====
apples     5
apples    17
apples     9
apples    16
apples     9
dtype: int64
===== bananas =====
bananas    19
bananas    17
bananas     9
bananas    13
bananas     7
bananas    16
bananas    11
bananas    18
bananas    13
dtype: int64
===== cherries =====
cherries    12
dtype: int64
===== clementines =====
clementines     6
clementines    11
clementines     5
clementines    12
clementines    12
clementines     6
dtype: int64
===== oranges =====
oranges    12
oranges     6
oranges     9
dtype: int64
===== pears =====
pears    18
pears     9
pears    10
pears    10
pears     1
pears    16
dtype: int64
```

We could have got the same result - except for the order - without using `` groupby '' with the following Python code.

```python
for fruit in set(s.index):
    print(f"===== {fruit} =====")
    print(s[fruit])
```

```
===== cherries =====
12
===== oranges =====
oranges    12
oranges     6
oranges     9
dtype: int64
===== pears =====
pears    18
pears     9
pears    10
pears    10
pears     1
pears    16
dtype: int64
===== clementines =====
clementines     6
clementines    11
clementines     5
clementines    12
clementines    12
clementines     6
dtype: int64
===== bananas =====
bananas    19
bananas    17
bananas     9
bananas    13
bananas     7
bananas    16
bananas    11
bananas    18
bananas    13
dtype: int64
===== apples =====
apples     5
apples    17
apples     9
apples    16
apples     9
dtype: int64
```

GROUPBY WITH DATAFRAMES

We will start with a very simple DataFrame. The DataFRame has two columns one containing names `Name` and the other one `Coffee` contains integers which are the number of cups of coffee the person drank.

```
import pandas as pd
beverages = pd.DataFrame({'Name': ['Robert', 'Melinda', 'Brenda',
                                   'Samantha', 'Melinda', 'Rober
t',
                                   'Melinda', 'Brenda', 'Samanth
a'],
                          'Coffee': [3, 0, 2, 2, 0, 2, 0, 1, 3],
                          'Tea':    [0, 4, 2, 0, 3, 0, 3, 2, 0]})

beverages
```

Output:

	Name	Coffee	Tea
0	Robert	3	0
1	Melinda	0	4
2	Brenda	2	2
3	Samantha	2	0
4	Melinda	0	3
5	Robert	2	0
6	Melinda	0	3
7	Brenda	1	2
8	Samantha	3	0

It's simple, and we've already seen in the previous chapters of our tutorial how to calculate the total number of coffee cups. The task is to sum a column of a DatFrame, i.e. the 'Coffee' column:

```
beverages['Coffee'].sum()
```

Output:
```
13
```

Let's compute now the total number of coffees and teas:

```
beverages[['Coffee', 'Tea']].sum()
```

Output:
```
Coffee     13
Tea        14
dtype: int64
```

'groupby' has not been necessary for the previous tasks. Let's have a look at our DataFrame again. We can see that some of the names appear multiple times. So it will be very interesting to see how many cups of coffee and tea each person drank in total. That means we are applying 'groupby' to the 'Name' column. Thereby we split the DatFrame. Then we apply 'sum' to the results of 'groupby':

```
res = beverages.groupby(['Name']).sum()
print(res)
```

```
          Coffee  Tea
Name
Brenda         3    4
Melinda        0   10
Robert         5    0
Samantha       5    0
```

We can see that the names are now the index of the resulting DataFrame:

```
print(res.index)
```

```
Index(['Brenda', 'Melinda', 'Robert', 'Samantha'], dtype='objec
t', name='Name')
```

There is only one column left, i.e. the `Coffee` column:

```
print(res.columns)
```

```
Index(['Coffee', 'Tea'], dtype='object')
```

We can also calculate the average number of coffee and tea cups the persons had:

```
beverages.groupby(['Name']).mean()
```

Output:

	Coffee	Tea
Name		
Brenda	1.5	2.000000
Melinda	0.0	3.333333
Robert	2.5	0.000000
Samantha	2.5	0.000000

ANOTHER EXAMPLE

The following Python code is used to create the data, we will use in our next `groupby` example. It is not necessary to understand the following Python code for the content following afterwards. The module `faker` has to be installed. In cae of an Anaconda installation this can be done by executing one of the following commands in a shell:

```
conda install -c conda-forge faker
conda install -c conda-forge/label/gcc7 faker
conda install -c conda-forge/label/cf201901 faker
conda install -c conda-forge/label/cf202003 faker
```

```
from faker import Faker
import numpy as np
from itertools import chain

fake = Faker('de_DE')

number_of_names = 10
names = []
for _ in range(number_of_names):
    names.append(fake.first_name())


data = {}
workweek = ("Monday", "Tuesday", "Wednesday", "Thursday", "Frida
y")
weekend = ("Saturday", "Sunday")

for day in chain(workweek, weekend):
    data[day] = np.random.randint(0, 10, (number_of_names,))
```

```
data_df = pd.DataFrame(data, index=names)
data_df
```

Output:

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Kenan	0	0	5	9	2	2	9
Jovan	9	1	5	1	0	0	0
Stanislaus	6	7	5	1	1	5	3
Adelinde	2	2	5	8	7	4	9
Cengiz	2	7	3	8	4	6	9
Edeltraud	4	7	9	9	7	9	7
Sara	7	1	7	0	7	8	3
Gerda	9	8	7	0	8	5	8
Tilman	5	1	9	4	7	5	5
Roswita	1	8	5	3	5	3	9

```
print(names)
```

```
['Kenan', 'Jovan', 'Stanislaus', 'Adelinde', 'Cengiz', 'Edeltrau
d', 'Sara', 'Gerda', 'Tilman', 'Roswita']
```

```
names = ('Ortwin', 'Mara', 'Siegrun', 'Sylvester', 'Metin', 'Adeli
ne', 'Utz', 'Susan', 'Gisbert', 'Senol')
data = {'Monday': np.array([0, 9, 2, 3, 7, 3, 9, 2, 4, 9]),
        'Tuesday': np.array([2, 6, 3, 3, 5, 5, 7, 7, 1, 0]),
        'Wednesday': np.array([6, 1, 1, 9, 4, 0, 8, 6, 8, 8]),
        'Thursday': np.array([1, 8, 6, 9, 9, 4, 1, 7, 3, 2]),
        'Friday': np.array([3, 5, 6, 6, 5, 2, 2, 4, 6, 5]),
        'Saturday': np.array([8, 4, 8, 2, 3, 9, 3, 4, 9, 7]),
        'Sunday': np.array([0, 8, 7, 8, 9, 7, 2, 0, 5, 2])}

data_df = pd.DataFrame(data, index=names)
data_df
```

Output:

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Ortwin	0	2	6	1	3	8	0
Mara	9	6	1	8	5	4	8
Siegrun	2	3	1	6	6	8	7
Sylvester	3	3	9	9	6	2	8
Metin	7	5	4	9	5	3	9
Adeline	3	5	0	4	2	9	7
Utz	9	7	8	1	2	3	2
Susan	2	7	6	7	4	4	0
Gisbert	4	1	8	3	6	9	5
Senol	9	0	8	2	5	7	2

We will demonstrate with this DataFrame how to combine columns by a function.

```
def is_weekend(day):
    if day in {'Saturday', 'Sunday'}:
        return "Weekend"
    else:
        return "Workday"
```

```
for res_func, df in data_df.groupby(by=is_weekend, axis=1):
    print(df)
```

```
           Saturday  Sunday
Ortwin            8       0
Mara              4       8
Siegrun           8       7
Sylvester         2       8
Metin             3       9
Adeline           9       7
Utz               3       2
Susan             4       0
Gisbert           9       5
Senol             7       2
           Monday  Tuesday  Wednesday  Thursday  Friday
Ortwin          0        2          6         1       3
Mara            9        6          1         8       5
Siegrun         2        3          1         6       6
Sylvester       3        3          9         9       6
Metin           7        5          4         9       5
Adeline         3        5          0         4       2
Utz             9        7          8         1       2
Susan           2        7          6         7       4
Gisbert         4        1          8         3       6
Senol           9        0          8         2       5
```

```
data_df.groupby(by=is_weekend, axis=1).sum()
```

Output:

	Weekend	Workday
Ortwin	8	12
Mara	12	29
Siegrun	15	18
Sylvester	10	30
Metin	12	30
Adeline	16	14
Utz	5	27
Susan	4	26
Gisbert	14	22
Senol	9	24

EXERCISES

EXERCISE 1

Calculate the average prices of the products of the following DataFrame:

```
import pandas as pd

d = {"products": ["Oppilume", "Dreaker", "Lotadilo",
                  "Crosteron", "Wazzasoft", "Oppilume",
                  "Dreaker", "Lotadilo", "Wazzasoft"],
     "colours": ["blue", "blue", "blue",
                 "green", "blue", "green",
                 "green", "green", "red"],
     "customer_price": [2345.89, 2390.50, 1820.00,
                        3100.00, 1784.50, 2545.89,
                        2590.50, 2220.00, 2084.50],
     "non_customer_price": [2445.89, 2495.50, 1980.00,
                            3400.00, 1921.00, 2645.89,
                                 2655.50, 2140.00, 2190.00]}

product_prices = pd.DataFrame(d)
product_prices
```

Output:

	products	colours	customer_price	non_customer_price
0	Oppilume	blue	2345.89	2445.89
1	Dreaker	blue	2390.50	2495.50
2	Lotadilo	blue	1820.00	1980.00
3	Crosteron	green	3100.00	3400.00
4	Wazzasoft	blue	1784.50	1921.00
5	Oppilume	green	2545.89	2645.89
6	Dreaker	green	2590.50	2655.50
7	Lotadilo	green	2220.00	2140.00
8	Wazzasoft	red	2084.50	2190.00

EXERCISE 2

Calculate the sum of the price according to the colours.

EXERCISE 3

Read in the `project_times.txt` file from the `data1` directory. This rows of this file contain comma separated the date, the name of the programmer, the name of the project, the time the programmer spent on the project.

Calculate the time spend on all the projects per day

EXERCISE 4

Create a DateFrame containing the total times spent on a project per day by all the programmers

EXERCISE 5

Calculate the total times spent on the projects over the whole month.

EXERCISE 6

Calculate the monthly times of each programmer regardless of the projects

EXERCISE 7

Rearrange the DataFrame with a MultiIndex consisting of the date and the project names, the columns should be the programmer names and the data of the columns the time of the programmers spent on the projects.

```
                    time
programmer          Antonie  Elise  Fatima  Hella  Mariola
date      project
2020-01-01 BIRDY    NaN      NaN    NaN     1.50   1.75
           NSTAT    NaN      NaN    0.25    NaN    1.25
           XTOR     NaN      NaN    NaN     1.00   3.50
2020-01-02 BIRDY    NaN      NaN    NaN     1.75   2.00
           NSTAT    0.5      NaN    NaN     NaN    1.75
```

Replace the NaN values by 0.

SOLUTIONS

SOLUTION TO EXERCISE 1

```
x = product_prices.groupby("products").mean()
x
```

Output:

	customer_price	non_customer_price
products		
Crosteron	3100.00	3400.00
Dreaker	2490.50	2575.50
Lotadilo	2020.00	2060.00
Oppilume	2445.89	2545.89
Wazzasoft	1934.50	2055.50

SOLUTION TO EXERCISE 2

```
x = product_prices.groupby("colours").sum()
x
```

Output:

	customer_price	non_customer_price
colours		
blue	8340.89	8842.39
green	10456.39	10841.39
red	2084.50	2190.00

SOLUTION TO EXERCISE 3

```
import pandas as pd

df = pd.read_csv("data1/project_times.txt", index_col=0)
df
```

Output:

	programmer	project	time
date			
2020-01-01	Hella	XTOR	1.00
2020-01-01	Hella	BIRDY	1.50
2020-01-01	Fatima	NSTAT	0.25
2020-01-01	Mariola	NSTAT	0.50
2020-01-01	Mariola	BIRDY	1.75
...	...	...	...
2030-01-30	Antonie	XTOR	0.50
2030-01-31	Hella	BIRDY	1.25
2030-01-31	Hella	BIRDY	1.75
2030-01-31	Mariola	BIRDY	1.00
2030-01-31	Hella	BIRDY	1.00

17492 rows × 3 columns

```
times_per_day = df.groupby(df.index).sum()
print(times_per_day[:10])
```

```
             time
date
2020-01-01   9.25
2020-01-02   6.00
2020-01-03   2.50
2020-01-06   5.75
2020-01-07  15.00
2020-01-08  13.25
2020-01-09  10.25
2020-01-10  17.00
2020-01-13   4.75
2020-01-14  10.00
```

SOLUTION TO EXERCISE 4

```
times_per_day_project = df.groupby([df.index, 'project']).sum()
print(times_per_day_project[:10])
```

```
                      time
date       project
2020-01-01 BIRDY      3.25
           NSTAT      1.50
           XTOR       4.50
2020-01-02 BIRDY      3.75
           NSTAT      2.25
2020-01-03 BIRDY      1.00
           NSTAT      0.25
           XTOR       1.25
2020-01-06 BIRDY      2.75
           NSTAT      0.75
```

SOLUTION TO EXERCISE 5

```
df.groupby(['project']).sum()
```

Output:

	time
project	
BIRDY	9605.75
NSTAT	8707.75
XTOR	6427.50

SOLUTION TO EXERCISE 6

```
df.groupby(['programmer']).sum()
```

Output:

	time
programmer	
Antonie	1511.25
Elise	80.00
Fatima	593.00
Hella	10642.00
Mariola	11914.75

SOLUTION TO EXERCISE 7

```
x = df.groupby([df.index, 'project', 'programmer']).sum()

x = x.unstack()
x
```

Output:

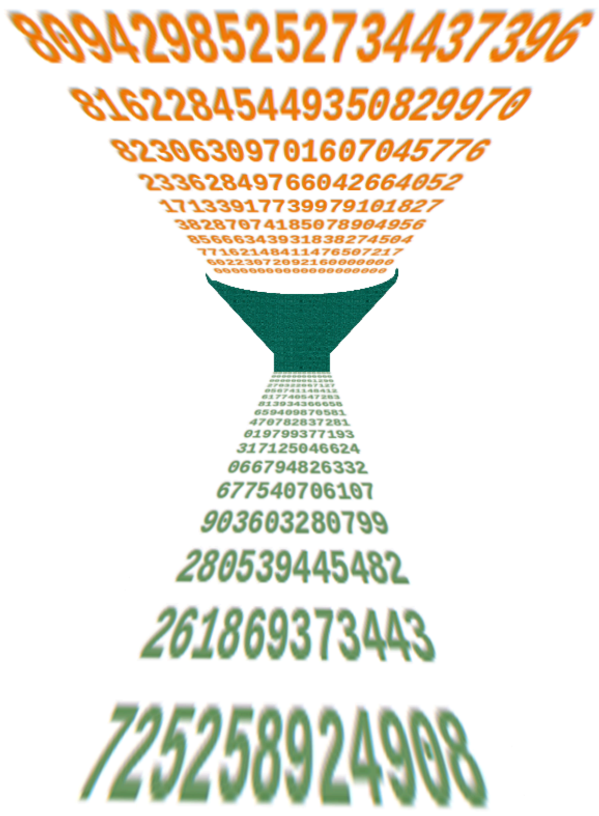
		time				
	programmer	**Antonie**	**Elise**	**Fatima**	**Hella**	**Mariola**
date	**project**					
2020-01-01	**BIRDY**	NaN	NaN	NaN	1.50	1.75
	NSTAT	NaN	NaN	0.25	NaN	1.25
	XTOR	NaN	NaN	NaN	1.00	3.50
2020-01-02	**BIRDY**	NaN	NaN	NaN	1.75	2.00
	NSTAT	0.5	NaN	NaN	NaN	1.75
...	**...**	...	...	...	...	...
2030-01-29	**XTOR**	NaN	NaN	NaN	1.00	5.50
2030-01-30	**BIRDY**	NaN	NaN	NaN	0.75	4.75
	NSTAT	NaN	NaN	NaN	3.75	NaN
	XTOR	0.5	NaN	NaN	0.75	NaN
2030-01-31	**BIRDY**	NaN	NaN	NaN	4.00	1.00

7037 rows × 5 columns

```
x = x.fillna(0)
print(x[:10])
```

```
                      time                                 
programmer         Antonie Elise Fatima Hella Mariola
date       project                                  
2020-01-01 BIRDY      0.00   0.0   0.00  1.50    1.75
           NSTAT      0.00   0.0   0.25  0.00    1.25
           XTOR       0.00   0.0   0.00  1.00    3.50
2020-01-02 BIRDY      0.00   0.0   0.00  1.75    2.00
           NSTAT      0.50   0.0   0.00  0.00    1.75
2020-01-03 BIRDY      0.00   0.0   1.00  0.00    0.00
           NSTAT      0.25   0.0   0.00  0.00    0.00
           XTOR       0.00   0.0   0.00  0.50    0.75
2020-01-06 BIRDY      0.00   0.0   0.00  2.50    0.25
           NSTAT      0.00   0.0   0.00  0.00    0.75
```

READING AND WRITING DATA



All the powerful data structures like the Series and the DataFrames would avail to nothing, if the Pandas module wouldn't provide powerful functionalities for reading in and writing out data. It is not only a matter of having a functions for interacting with files. To be useful to data scientists it also needs functions which support the most important data formats like

- Delimiter-separated files, like e.g. csv
- Microsoft Excel files
- HTML
- XML
- JSON

DELIMITER-SEPARATED VALUES

Most people take csv files as a synonym for delimter-separated values files. They leave the fact out of account that csv is an acronym for "comma separated values", which is not the case in many situations. Pandas also uses "csv" and contexts, in which "dsv" would be more appropriate.

Delimiter-separated values (DSV) are defined and stored two-dimensional arrays (for example strings) of data by separating the values in each row with delimiter characters defined for this purpose. This way of implementing data is often used in combination of spreadsheet programs, which can read in and write out data as DSV. They are also used as a general data exchange format.

We call a text file a "delimited text file" if it contains text in DSV format.

For example, the file dollar_euro.txt is a delimited text file and uses tabs (\t) as delimiters.

READING CSV AND DSV FILES

Pandas offers two ways to read in CSV or DSV files to be precise:

- DataFrame.from_csv
- read_csv

There is no big difference between those two functions, e.g. they have different default values in some cases and read_csv has more paramters. We will focus on read_csv, because DataFrame.from_csv is kept inside Pandas for reasons of backwards compatibility.

```
import pandas as pd

exchange_rates = pd.read_csv("data1/dollar_euro.txt",
                             sep="\t")
print(exchange_rates)
```

```
    Year   Average  Min USD/EUR  Max USD/EUR  Working days
0   2016  0.901696     0.864379     0.959785           247
1   2015  0.901896     0.830358     0.947688           256
2   2014  0.753941     0.716692     0.823655           255
3   2013  0.753234     0.723903     0.783208           255
4   2012  0.778848     0.743273     0.827198           256
5   2011  0.719219     0.671953     0.775855           257
6   2010  0.755883     0.686672     0.837381           258
7   2009  0.718968     0.661376     0.796495           256
8   2008  0.683499     0.625391     0.802568           256
9   2007  0.730754     0.672314     0.775615           255
10  2006  0.797153     0.750131     0.845594           255
11  2005  0.805097     0.740357     0.857118           257
12  2004  0.804828     0.733514     0.847314           259
13  2003  0.885766     0.791766     0.963670           255
14  2002  1.060945     0.953562     1.165773           255
15  2001  1.117587     1.047669     1.192748           255
16  2000  1.085899     0.962649     1.211827           255
17  1999  0.939475     0.848176     0.998502           261
```

As we can see, read_csv used automatically the first line as the names for the columns. It is possible to give other names to the columns. For this purpose, we have to skip the first line by setting the parameter "header" to 0 and we have to assign a list with the column names to the parameter "names":

```
import pandas as pd

exchange_rates = pd.read_csv("data1/dollar_euro.txt",
                             sep="\t",
                             header=0,
                             names=["year", "min", "max", "days"])
print(exchange_rates)
```

```
          year       min       max  days
2016  0.901696  0.864379  0.959785   247
2015  0.901896  0.830358  0.947688   256
2014  0.753941  0.716692  0.823655   255
2013  0.753234  0.723903  0.783208   255
2012  0.778848  0.743273  0.827198   256
2011  0.719219  0.671953  0.775855   257
2010  0.755883  0.686672  0.837381   258
2009  0.718968  0.661376  0.796495   256
2008  0.683499  0.625391  0.802568   256
2007  0.730754  0.672314  0.775615   255
2006  0.797153  0.750131  0.845594   255
2005  0.805097  0.740357  0.857118   257
2004  0.804828  0.733514  0.847314   259
2003  0.885766  0.791766  0.963670   255
2002  1.060945  0.953562  1.165773   255
2001  1.117587  1.047669  1.192748   255
2000  1.085899  0.962649  1.211827   255
1999  0.939475  0.848176  0.998502   261
```

EXERCISE

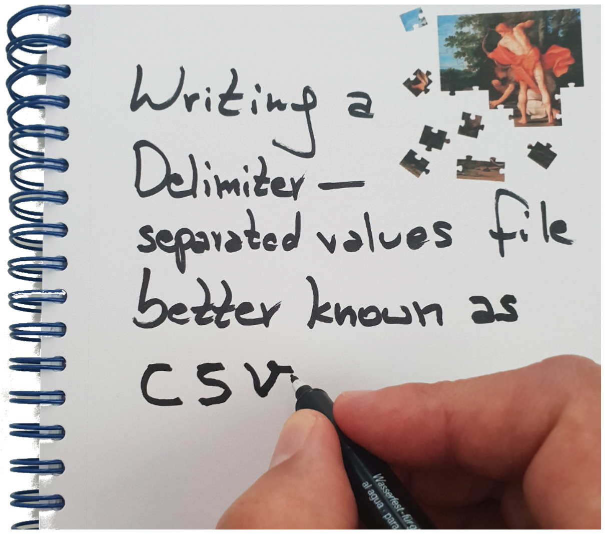
The file "countries_population.csv" is a csv file, containing the population numbers of all countries (July 2014). The delimiter of the file is a space and commas are used to separate groups of thousands in the numbers. The method 'head(n)' of a DataFrame can be used to give out only the first n rows or lines. Read the file into a DataFrame.

Solution:

```
pop = pd.read_csv("data1/countries_population.csv",
                  header=None,
                  names=["Country", "Population"],
                  index_col=0,
                  quotechar="'",
                  sep=" ",
                  thousands=",")
print(pop.head(5))
```

```
                Population
Country
China           1355692576
India           1236344631
European Union   511434812
United States    318892103
Indonesia        253609643
```

WRITING CSV FILES


Writing a Delimiter — separated values file better known as CSV

We can create csv (or dsv) files with the method "to_csv". Before we do this, we will prepare some data to output, which we will write to a file. We have two csv files with population data for various countries. countries_male_population.csv contains the figures of the male populations and countries_female_population.csv correspondingly the numbers for the female populations. We will create a new csv file with the sum:

```
column_names = ["Country"] + list(range(2002, 2013))
male_pop = pd.read_csv("data1/countries_male_population.csv",
                       header=None,
                       index_col=0,
                       names=column_names)

female_pop = pd.read_csv("data1/countries_female_population.csv",
                         header=None,
                         index_col=0,
                         names=column_names)


population = male_pop + female_pop
```

```
population
```

Output:

	2002	2003	2004	2005	2006	2007	2008	2009
Country								
Australia	19640979.0	19872646	20091504	20339759	20605488	21015042	21431781	21874
Austria	8139310.0	8067289	8140122	8206524	8265925	8298923	8331930	8355
Belgium	10309725.0	10355844	10396421	10445852	10511382	10584534	10666866	10753
Canada	NaN	31361611	31372587	31989454	32299496	32649482	32927372	33327
Czech Republic	10269726.0	10203269	10211455	10220577	10251079	10287189	10381130	10467
Denmark	5368354.0	5383507	5397640	5411405	5427459	5447084	5475791	5511
Finland	5194901.0	5206295	5219732	5236611	5255580	5276955	5300484	5326
France	59337731.0	59630121	59900680	62518571	62998773	63392140	63753140	64366
Germany	82440309.0	82536680	82531671	82500849	82437995	82314906	82217837	82002
Greece	10988000.0	11006377	11040650	11082751	11125179	11171740	11213785	11260
Hungary	10174853.0	10142362	10116742	10097549	10076581	10066158	10045401	10030
Iceland	286575.0	288471	290570	293577	299891	307672	315459	319
Ireland	3882683.0	3963636	4027732	4109173	4209019	4239848	4401335	4450
Italy	56993742.0	57321070	57888245	58462375	58751711	59131287	59619290	60045
Japan	127291000.0	127435000	127620000	127687000	127767994	127770000	127771000	127692
Korea	47639618.0	47925318	48082163	48138077	48297184	48456369	48606787	48746
Luxembourg	444050.0	448300	451600	455000	469086	476187	483799	493
Mexico	101826249.0	103039964	104213503	103001871	103946866	104874282	105790725	106682
Netherlands	16105285.0	16192572	16258032	16305526	16334210	16357992	16405399	16485

	2002	2003	2004	2005	2006	2007	2008	2009
Country								
New Zealand	3939130.0	4009200	4062500	4100570	4139470	4228280	4268880	4315
Norway	4524066.0	4552252	4577457	4606363	4640219	4681134	4737171	4799
Poland	38632453.0	38218531	38190608	38173835	38157055	38125479	38115641	38135
Portugal	10335559.0	10407465	10474685	10529255	10569592	10599095	10617575	10627
Slovak Republic	5378951.0	5379161	5380053	5384822	5389180	5393637	5400998	5412
Spain	40409330.0	41550584	42345342	43038035	43758250	44474631	45283259	45828
Sweden	8909128.0	8940788	8975670	9011392	9047752	9113257	9182927	9256
Switzerland	7261210.0	7313853	7364148	7415102	7459128	7508739	7593494	7701
Turkey	NaN	70171979	70689500	71607500	72519974	72519974	70586256	71517
United Kingdom	58706905.0	59262057	59699828	60059858	60412870	60781346	61179260	61595
United States	277244916.0	288774226	290810719	294442683	297308143	300184434	304846731	305127

```python
population.to_csv("data1/countries_total_population.csv")
```

We want to create a new DataFrame with all the information, i.e. female, male and complete population. This means that we have to introduce an hierarchical index. Before we do it on our DataFrame, we will introduce this problem in a simple example:

```python
import pandas as pd

shop1 = {"foo":{2010:23, 2011:25}, "bar":{2010:13, 2011:29}}
shop2 = {"foo":{2010:223, 2011:225}, "bar":{2010:213, 2011:229}}

shop1 = pd.DataFrame(shop1)
shop2 = pd.DataFrame(shop2)
```

```
both_shops = shop1 + shop2
print("Sales of shop1:\n", shop1)
print("\nSales of both shops\n", both_shops)
```

```
Sales of shop1:
       foo  bar
2010   23   13
2011   25   29

Sales of both shops
       foo  bar
2010  246  226
2011  250  258
```

```
shops = pd.concat([shop1, shop2], keys=["one", "two"])
shops
```

Output:

		foo	bar
one	**2010**	23	13
	2011	25	29
two	**2010**	223	213
	2011	225	229

We want to swap the hierarchical indices. For this we will use 'swaplevel':

```
shops.swaplevel()
shops.sort_index(inplace=True)
shops
```

Output:

		foo	bar
one	**2010**	23	13
	2011	25	29
two	**2010**	223	213
	2011	225	229

We will go back to our initial problem with the population figures. We will apply the same steps to those DataFrames:

```
pop_complete = pd.concat([population.T,
                          male_pop.T,
                          female_pop.T],
                          keys=["total", "male", "female"])
df = pop_complete.swaplevel()
df.sort_index(inplace=True)
df[["Austria", "Australia", "France"]]
```

Output:

	Country	Austria	Australia	France
2002	**female**	4179743.0	9887846.0	30510073.0
	male	3959567.0	9753133.0	28827658.0
	total	8139310.0	19640979.0	59337731.0
2003	**female**	4158169.0	9999199.0	30655533.0
	male	3909120.0	9873447.0	28974588.0
	total	8067289.0	19872646.0	59630121.0
2004	**female**	4190297.0	10100991.0	30789154.0
	male	3949825.0	9990513.0	29111526.0
	total	8140122.0	20091504.0	59900680.0
2005	**female**	4220228.0	10218321.0	32147490.0
	male	3986296.0	10121438.0	30371081.0
	total	8206524.0	20339759.0	62518571.0
2006	**female**	4246571.0	10348070.0	32390087.0
	male	4019354.0	10257418.0	30608686.0
	total	8265925.0	20605488.0	62998773.0
2007	**female**	4261752.0	10570420.0	32587979.0
	male	4037171.0	10444622.0	30804161.0
	total	8298923.0	21015042.0	63392140.0
2008	**female**	4277716.0	10770864.0	32770860.0
	male	4054214.0	10660917.0	30982280.0

	Country	Austria	Australia	France
	total	8331930.0	21431781.0	63753140.0
2009	female	4287213.0	10986535.0	33208315.0
	male	4068047.0	10888385.0	31158647.0
	total	8355260.0	21874920.0	64366962.0
2010	female	4296197.0	11218144.0	33384930.0
	male	4079093.0	11124254.0	31331380.0
	total	8375290.0	22342398.0	64716310.0
2011	female	4308915.0	11359807.0	33598633.0
	male	4095337.0	11260747.0	31531113.0
	total	8404252.0	22620554.0	65129746.0
2012	female	4324983.0	11402769.0	33723892.0
	male	4118035.0	11280804.0	31670391.0
	total	8443018.0	22683573.0	65394283.0

```
df.to_csv("data1/countries_total_population.csv")
```

EXERCISE

- Read in the dsv file (csv) bundeslaender.txt. Create a new file with the columns 'land', 'area', 'female', 'male', 'population' and 'density' (inhabitants per square kilometres.
- print out the rows where the area is greater than 30000 and the population is greater than 10000
- Print the rows where the density is greater than 300

```
lands = pd.read_csv('data1/bundeslaender.txt', sep=" ")
print(lands.columns.values)
```

```
['land' 'area' 'male' 'female']
```

```
# swap the columns of our DataFrame:
lands = lands.reindex(columns=['land', 'area', 'female', 'male'])
lands[:2]
```

Output:

	land	area	female	male
0	Baden-Württemberg	35751.65	5465	5271
1	Bayern	70551.57	6366	6103

```
lands.insert(loc=len(lands.columns),
             column='population',
             value=lands['female'] + lands['male'])
```

```
lands[:3]
```

Output:

	land	area	female	male	population
0	Baden-Württemberg	35751.65	5465	5271	10736
1	Bayern	70551.57	6366	6103	12469
2	Berlin	891.85	1736	1660	3396

```
lands.insert(loc=len(lands.columns),
             column='density',
             value=(lands['population'] * 1000 / lands['area']).ro
und(0))

lands[:4]
```

Output:

	land	area	female	male	population	density
0	Baden-Württemberg	35751.65	5465	5271	10736	300.0
1	Bayern	70551.57	6366	6103	12469	177.0
2	Berlin	891.85	1736	1660	3396	3808.0
3	Brandenburg	29478.61	1293	1267	2560	87.0

```
print(lands.loc[(lands.area>30000) & (lands.population>10000)])
```

```
                  land      area  female  male  population  densit
y
0    Baden-Württemberg  35751.65    5465  5271       10736    30
0.0
1               Bayern  70551.57    6366  6103       12469    17
7.0
9  Nordrhein-Westfalen  34085.29    9261  8797       18058    53
0.0
```

READING AND WRITING EXCEL FILES

It is also possible to read and write Microsoft Excel files. The Pandas functionalities to read and write Excel files use the modules 'xlrd' and 'openpyxl'. These modules are not automatically installed by Pandas, so you may have to install them manually!

We will use a simple Excel document to demonstrate the reading capabilities of Pandas. The document sales.xls contains two sheets, one called 'week1' and the other one 'week2'.
An Excel file can be read in with the Pandas function "read_excel". This is demonstrated in the following example Python code:

```
excel_file = pd.ExcelFile("data1/sales.xls")

sheet = pd.read_excel(excel_file)
sheet
```

Output:

	Weekday	Sales
0	Monday	123432.980000
1	Tuesday	122198.650200
2	Wednesday	134418.515220
3	Thursday	131730.144916
4	Friday	128173.431003

The document "sales.xls" contains two sheets, but we only have been able to read in the first one with "read_excel". A complete Excel document, which can consist of an arbitrary number of sheets, can be completely read in like this:

```
docu = {}
for sheet_name in excel_file.sheet_names:
    docu[sheet_name] = excel_file.parse(sheet_name)

for sheet_name in docu:
    print("\n" + sheet_name + ":\n", docu[sheet_name])
```

```
week1:
       Weekday          Sales
0       Monday  123432.980000
1      Tuesday  122198.650200
2    Wednesday  134418.515220
3     Thursday  131730.144916
4       Friday  128173.431003

week2:
       Weekday          Sales
0       Monday  223277.980000
1      Tuesday  234441.879000
2    Wednesday  246163.972950
3     Thursday  241240.693491
4       Friday  230143.621590
```

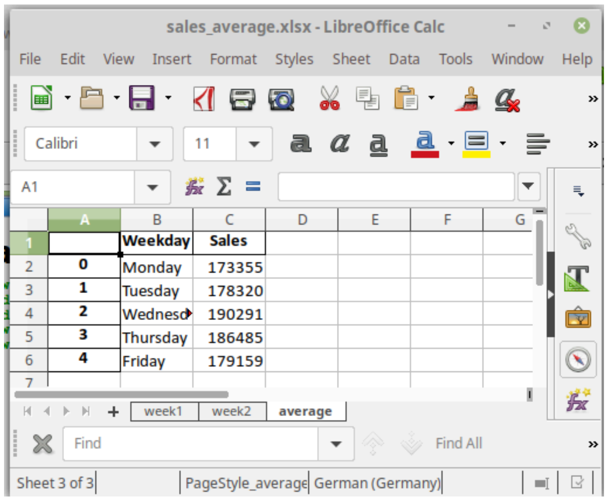
We will calculate now the avarage sales numbers of the two weeks:

```
average = docu["week1"].copy()
average["Sales"] = (docu["week1"]["Sales"] + docu["week2"]["Sale
s"]) / 2
print(average)
```

```
      Weekday          Sales
0       Monday  173355.480000
1      Tuesday  178320.264600
2    Wednesday  190291.244085
3     Thursday  186485.419203
4       Friday  179158.526297
```

We will save the DataFrame 'average' in a new document with 'week1' and 'week2' as additional sheets as well:

```
writer = pd.ExcelWriter('data1/sales_average.xlsx')
document['week1'].to_excel(writer,'week1')
document['week2'].to_excel(writer,'week2')
average.to_excel(writer,'average')
writer.save()
writer.close()
```

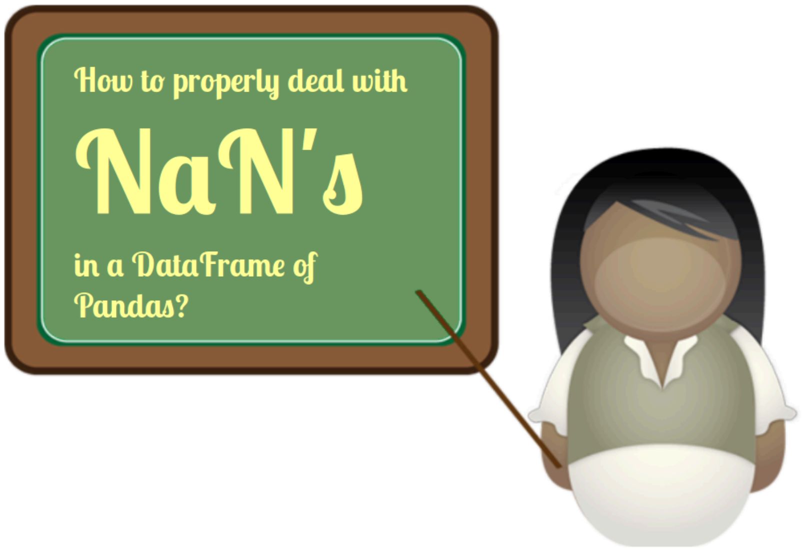

sales_average.xlsx - LibreOffice Calc
File Edit View Insert Format Styles Sheet Data Tools Window Help
Calibri
11
A1
week1
week2
average
Find
Find All
Sheet 3 of 3
PageStyle_average
German (Germany)

	A	B	C
1		Weekday	Sales
2	0	Monday	173355
3	1	Tuesday	178320
4	2	Wednesd	190291
5	3	Thursday	186485
6	4	Friday	179159

DEALING WITH NAN

INTRODUCTION

NaN was introduced, at least officially, by the IEEE Standard for Floating-Point Arithmetic (IEEE 754). It is a technical standard for floating-point computation established in 1985 - many years before Python was invented, and even a longer time befor Pandas was created - by the Institute of Electrical and Electronics Engineers (IEEE). It was introduced to solve problems found in many floating point implementations that made them difficult to use reliably and portably.



This standard added NaN to the arithmetic formats: "arithmetic formats: sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special 'not a number' values (NaNs)"

'NAN' IN PYTHON

Python knows NaN values as well. We can create it with "float":

```
n1 = float("nan")
n2 = float("Nan")
n3 = float("NaN")
n4 = float("NAN")
print(n1, n2, n3, n4)
```

```
nan nan nan nan
```

"nan" is also part of the math module since Python 3.5:

```
import math
n1 = math.nan
print(n1)
print(math.isnan(n1))
```

```
nan
True
```

Warning: Do not perform comparison between "NaN" values or "Nan" values and regular numbers. A simple or simplified reasoning is this: Two things are "not a number", so they can be anything but most probably not the same. Above all there is no way of ordering NaNs:

```
print(n1 == n2)
print(n1 == 0)
print(n1 == 100)
print(n2 < 0)
```

```
False
False
False
False
```

NAN IN PANDAS

EXAMPLE WITHOUT NANS

Before we will work with NaN data, we will process a file without any NaN values. The data file temperatures.csv contains the temperature data of six sensors taken every 15 minuts between 6:00 to 19.15 o'clock.

Reading in the data file can be done with the read_csv function:

```
import pandas as pd

df = pd.read_csv("data1/temperatures.csv",
                 sep=";",
                 decimal=",")
df.loc[:3]
```

Output:

	time	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6
0	06:00:00	14.3	13.7	14.2	14.3	13.5	13.6
1	06:15:00	14.5	14.5	14.0	15.0	14.5	14.7
2	06:30:00	14.6	15.1	14.8	15.3	14.0	14.2
3	06:45:00	14.8	14.5	15.6	15.2	14.7	14.6

We want to calculate the avarage temperatures per measuring point over all the sensors. We can use the DataFrame method 'mean'. If we use 'mean' without parameters it will sum up the sensor columns, which isn't what we want, but it may be interesting as well:

```
df.mean()
```

Output:

```
sensor1    19.775926
sensor2    19.757407
sensor3    19.840741
sensor4    20.187037
sensor5    19.181481
sensor6    19.437037
dtype: float64
```

```
average_temp_series = df.mean(axis=1)
print(average_temp_series[:8])
```

```
0    13.933333
1    14.533333
2    14.666667
3    14.900000
4    15.083333
5    15.116667
6    15.283333
7    15.116667
dtype: float64
```

```
sensors = df.columns.values[1:]
# all columns except the time column will be removed:
df = df.drop(sensors, axis=1)
print(df[:5])
```

```
        time
0   06:00:00
1   06:15:00
2   06:30:00
3   06:45:00
4   07:00:00
```

We will assign now the average temperature values as a new column 'temperature':

```
# best practice:
df = df.assign(temperature=average_temp_series)  # inplace option
not available

# alternatively:
#df.loc[:,"temperature"] = average_temp_series
```

```
df[:3]
```

Output:

	time	temperature
0	06:00:00	13.933333
1	06:15:00	14.533333
2	06:30:00	14.666667

EXAMPLE WITH NANS

We will use now a data file similar to the previous temperature csv, but this time we will have to cope with NaN data, when the sensors malfunctioned.

We will create a temperature DataFrame, in which some data is not defined, i.e. NaN.

We will use and change the data from the the temperatures.csv file:

```
import pandas as pd
temp_df = pd.read_csv("data1/temperatures.csv",
                      sep=";",
                      index_col=0,
                      decimal=",")

temp_df[:8]
```

Output:

	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6
time						
06:00:00	14.3	13.7	14.2	14.3	13.5	13.6
06:15:00	14.5	14.5	14.0	15.0	14.5	14.7
06:30:00	14.6	15.1	14.8	15.3	14.0	14.2
06:45:00	14.8	14.5	15.6	15.2	14.7	14.6
07:00:00	15.0	14.9	15.7	15.6	14.0	15.3
07:15:00	15.2	15.2	14.6	15.3	15.5	14.9
07:30:00	15.4	15.3	15.6	15.6	14.7	15.1
07:45:00	15.5	14.8	15.4	15.5	14.6	14.9

We will randomly assign some NaN values into the data frame. For this purpose, we will use the where method from DataFrame. If we apply where to a DataFrame object df, i.e. df.where(cond, other_df), it will return an object of same shape as df and whose corresponding entries are from df where the corresponding element of cond is True and otherwise are taken from other_df.

Before we continue with our task, we will demonstrate the way of working of where with some simple examples:

s = pd.Series(range(5)) s.where(s > 0)

```
import numpy as np

A = np.random.randint(1, 30, (4, 2))

df = pd.DataFrame(A, columns=['Foo', 'Bar'])
m = df % 2 == 0
df.where(m, -df, inplace=True)
df
```

Output:

	Foo	Bar
0	18	-5
1	22	-7
2	14	-3
3	16	-23

For our task, we need to create a DataFrame 'nan_df', which consists purely of NaN values and has the same shape as our temperature DataFrame 'temp_df'. We will use this DataFrame in 'where'. We also need a DataFrame with the conditions "df_bool" as True values. For this purpose we will create a DataFrame with random values between 0 and 1 and by applying 'random_df < 0.8' we get the df_bool DataFrame, in which about 20 % of the values will be True:

```
random_df = pd.DataFrame(np.random.random(size=temp_df.shape),
                         columns=temp_df.columns.values,
                         index=temp_df.index)
nan_df = pd.DataFrame(np.nan,
                      columns=temp_df.columns.values,
                      index=temp_df.index)
df_bool = random_df<0.8

df_bool[:5]
```

Output:

	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6
time						
06:00:00	True	True	True	True	True	True
06:15:00	True	True	True	False	True	True
06:30:00	True	True	True	True	True	False
06:45:00	True	True	True	True	False	True
07:00:00	True	True	True	True	True	True

Finally, we have everything toghether to create our DataFrame with distrubed measurements:

```
disturbed_data = temp_df.where(df_bool, nan_df)

disturbed_data.to_csv("data1/temperatures_with_NaN.csv")
disturbed_data[:10]
```

Output:

	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6
time						
06:00:00	14.3	13.7	14.2	14.3	13.5	13.6
06:15:00	14.5	14.5	14.0	NaN	14.5	14.7
06:30:00	14.6	15.1	14.8	15.3	14.0	NaN
06:45:00	14.8	14.5	15.6	15.2	NaN	14.6
07:00:00	15.0	14.9	15.7	15.6	14.0	15.3
07:15:00	NaN	15.2	14.6	15.3	15.5	14.9
07:30:00	15.4	15.3	15.6	15.6	14.7	NaN
07:45:00	15.5	14.8	15.4	15.5	14.6	14.9
08:00:00	15.7	15.6	15.9	16.2	15.4	15.4
08:15:00	15.9	15.8	15.9	NaN	16.0	16.2

USING DROPNA ON THE DATAFRAME

'dropna' is a DataFrame method. If we call this method without arguments, it will return an object where every row is ommitted, in which data are missing, i.e. some value is NaN:

```
df = disturbed_data.dropna()
df
```

Output:

	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6
time						
06:00:00	14.3	13.7	14.2	14.3	13.5	13.6
07:00:00	15.0	14.9	15.7	15.6	14.0	15.3
07:45:00	15.5	14.8	15.4	15.5	14.6	14.9
08:00:00	15.7	15.6	15.9	16.2	15.4	15.4
09:45:00	18.4	19.0	19.0	19.4	18.4	18.3
10:00:00	19.0	19.7	18.8	18.9	17.5	18.9
11:45:00	24.2	23.1	25.3	23.7	24.5	24.8
12:15:00	23.8	23.7	24.8	25.1	22.2	22.4
12:45:00	23.4	22.6	23.7	24.4	21.8	23.8
15:00:00	21.8	22.9	22.2	22.8	21.0	21.9
16:00:00	21.1	21.6	20.7	20.6	19.9	21.4
17:00:00	20.4	19.5	20.3	21.8	19.9	19.2
17:15:00	20.3	20.7	19.6	21.3	19.8	19.0
17:45:00	19.9	20.4	19.4	21.1	20.0	20.5
18:15:00	19.6	19.9	19.2	19.9	20.0	18.6
18:30:00	19.5	19.1	19.2	19.7	18.3	18.3

'dropna' can also be used to drop all columns in which some values are NaN. This can be achieved by assigning 1 to the axis parameter. The default value is False, as we have seen in our previous example. As every column from our sensors contain NaN values, they will all disappear:

```
df = disturbed_data.dropna(axis=1)
```

```
df[:5]
```

Output:

time
06:00:00
06:15:00
06:30:00
06:45:00
07:00:00

Let us change our task: We only want to get rid of all the rows, which contain more than one NaN value. The parameter 'thresh' is ideal for this task. It can be set to the minimum number. 'thresh' is set to an integer value, which defines the minimum number of non-NaN values. We have six temperature values in every row. Setting 'thresh' to 5 makes sure that we will have at least 5 valid floats in every remaining row:

```
cleansed_df = disturbed_data.dropna(thresh=5, axis=0)
cleansed_df[:7]
```

Output:

	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6
time						
06:00:00	14.3	13.7	14.2	14.3	13.5	13.6
06:15:00	14.5	14.5	14.0	NaN	14.5	14.7
06:30:00	14.6	15.1	14.8	15.3	14.0	NaN
06:45:00	14.8	14.5	15.6	15.2	NaN	14.6
07:00:00	15.0	14.9	15.7	15.6	14.0	15.3
07:15:00	NaN	15.2	14.6	15.3	15.5	14.9
07:30:00	15.4	15.3	15.6	15.6	14.7	NaN

Now we will calculate the mean values again, but this time on the DataFrame 'cleansed_df', i.e. where we have taken out all the rows, where more than one NaN value occurred.

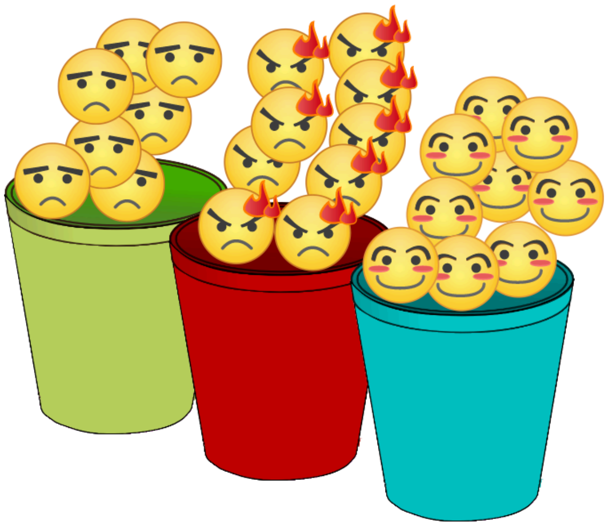
```
average_temp_series = cleansed_df.mean(axis=1)
sensors = cleansed_df.columns.values
df = cleansed_df.drop(sensors, axis=1)

# best practice:
df = df.assign(temperature=average_temp_series)  # inplace option
not available
df[:6]
```

Output:

	temperature
time	
06:00:00	13.933333
06:15:00	14.440000
06:30:00	14.760000
06:45:00	14.940000
07:00:00	15.083333
07:15:00	15.100000

BINNING IN PYTHON AND PANDAS

INTRODUCTION



Data binning, which is also known as bucketing or discretization, is a technique used in data processing and statistics. Binning can be used for example, if there are more possible data points than observed data points. An example is to bin the body heights of people into intervals or categories. Let us assume, we take the heights of 30 people. The length values can be between - roughly guessing - 1.30 metres to 2.50 metres. Theoretically, there are 120 different cm values possible, but we can have at most 30 different values from our sample group. One way to group them could be to put the measured values into bins ranging from 1.30 - 1.50 metres, 1.50 - 1.70 metres, 1.70 - 1.90 metres and so on. This means that the original data values, will be assigned to a bin into wich they fit according to their size. The original values will be replaced by values representing the corresponding intervals. Binning is a form of quantization.

Bins do not necessarily have to be numerical, they can be categorical values of any kind, like "dogs", "cats", "hamsters", and so on.

Binning is also used in image processing, binning. It can be used to reduce the amount of data, by combining neighboring pixel into single pixels. kxk binning reduces areas of k x k pixels into single pixel.

Pandas provides easy ways to create bins and to bin data. Before we describe these Pandas functionalities, we will introduce basic Python functions, working on Python lists and tuples.

BINNING IN PYTHON

The following Python function can be used to create bins.

```
def create_bins(lower_bound, width, quantity):
    """ create_bins returns an equal-width (distance) partitionin
g.
```

```
        It returns an ascending list of tuples, representing the i
ntervals.
        A tuple bins[i], i.e. (bins[i][0], bins[i][1])  with i >
0
        and i < quantity, satisfies the following conditions:
            (1) bins[i][0] + width == bins[i][1]
            (2) bins[i-1][0] + width == bins[i][0] and
                bins[i-1][1] + width == bins[i][1]
    """


    bins = []
    for low in range(lower_bound,
                     lower_bound + quantity*width + 1, width):
        bins.append((low, low+width))
    return bins
```

We will create now five bins (quantity=5) with a width of 10 (width=10) starting from 10 (lower_bound=10):

```
bins = create_bins(lower_bound=10,
                   width=10,
                   quantity=5)

bins
```

Output:
```
[(10, 20), (20, 30), (30, 40), (40, 50), (50, 60), (60, 70)]
```

The next function 'find_bin' is called with a list or tuple of bin 'bins', which have to be two-tuples or lists of two elements. The function finds the index of the interval, where the value 'value' is contained:

```
def find_bin(value, bins):
    """ bins is a list of tuples, like [(0,20), (20, 40), (40, 6
0)],
        binning returns the smallest index i of bins so that
        bin[i][0] <= value < bin[i][1]
    """

    for i in range(0, len(bins)):
        if bins[i][0] <= value < bins[i][1]:
            return i
    return -1
```

```
from collections import Counter
```

```
bins = create_bins(lower_bound=50,
                   width=4,
                   quantity=10)

print(bins)

weights_of_persons = [73.4, 69.3, 64.9, 75.6, 74.9, 80.3,
                      78.6, 84.1, 88.9, 90.3, 83.4, 69.3,
                      52.4, 58.3, 67.4, 74.0, 89.3, 63.4]

binned_weights = []

for value in weights_of_persons:
    bin_index = find_bin(value, bins)
    print(value, bin_index, bins[bin_index])
    binned_weights.append(bin_index)

frequencies = Counter(binned_weights)
print(frequencies)
```

```
[(50, 54), (54, 58), (58, 62), (62, 66), (66, 70), (70, 74), (74,
78), (78, 82), (82, 86), (86, 90), (90, 94)]
73.4 5 (70, 74)
69.3 4 (66, 70)
64.9 3 (62, 66)
75.6 6 (74, 78)
74.9 6 (74, 78)
80.3 7 (78, 82)
78.6 7 (78, 82)
84.1 8 (82, 86)
88.9 9 (86, 90)
90.3 10 (90, 94)
83.4 8 (82, 86)
69.3 4 (66, 70)
52.4 0 (50, 54)
58.3 2 (58, 62)
67.4 4 (66, 70)
74.0 6 (74, 78)
89.3 9 (86, 90)
63.4 3 (62, 66)
Counter({4: 3, 6: 3, 3: 2, 7: 2, 8: 2, 9: 2, 5: 1, 10: 1, 0: 1,
2: 1})
```

BINNING WITH PANDAS

The module Pandas of Python provides powerful functionalities for the binning of data. We will demonstrate this by using our previous data.

BINS USED BY PANDAS

We used a list of tuples as bins in our previous example. We have to turn this list into a usable data structure for the pandas function "cut". This data structure is an IntervalIndex. We can do this with pd.IntervalIndex.from_tuples:

```
import pandas as pd

bins2 = pd.IntervalIndex.from_tuples(bins)
```

"cut" is the name of the Pandas function, which is needed to bin values into bins. "cut" takes many parameters but the most important ones are "x" for the actual values und "bins", defining the IntervalIndex. "x" can be any 1-dimensional array-like structure, e.g. tuples, lists, nd-arrays and so on:

```
categorical_object = pd.cut(weights_of_persons, bins2)
print(categorical_object)
```

```
[(70, 74], (66, 70], (62, 66], (74, 78], (74, 78], ..., (58, 62],
(66, 70], (70, 74], (86, 90], (62, 66]]
Length: 18
Categories (11, interval[int64]): [(50, 54] < (54, 58] < (58, 62]
< (62, 66] ... (78, 82] < (82, 86] < (86, 90] < (90, 94]]
```

The result of the Pandas function "cut" is a so-called "Categorical object". Each bin is a category. The categories are described in a mathematical notation. "(70, 74]" means that this bins contains values from 70 to 74 whereas 70 is not included but 74 is included. Mathematically, this is a half-open interval, i.e. nn interval in which one endpoint is included but not the other. Sometimes it is also called an half-closed interval.

We had also defined the bins in our previous chapter as half-open intervals, but the other way round, i.e. left side closed and the right side open. When we used pd.IntervalIndex.from_tuples, we could have defined the "openness" of this bins by setting the parameter "closed" to one of the values:

- 'left': closed on the left side and open on the right
- 'right': (The default) open on the left side and closed on the right
- 'both': closed on both sides
- 'neither': open on both sides

To have the same behaviour as in our previous chapter, we will set the parameter closed to "left":

```
bins2 = pd.IntervalIndex.from_tuples(bins, closed="left")
```

```
categorical_object = pd.cut(weights_of_persons, bins2)
print(categorical_object)
```

```
[[70, 74), [66, 70), [62, 66), [74, 78), [74, 78), ..., [58, 62),
[66, 70), [74, 78), [86, 90), [62, 66)]
Length: 18
Categories (11, interval[int64]): [[50, 54) < [54, 58) < [58, 62)
< [62, 66) ... [78, 82) < [82, 86) < [86, 90) < [90, 94)]
```

OTHER WAYS TO DEFINE BINS

We used an IntervalIndex as a bin for binning the weight data. The function "cut" can also cope with two other kinds of bin representations:

- an integer:
  defining the number of equal-width bins in the range of the values "x". The

```
        range of "x" is extended by .1% on each side to includ
    e the minimum
        and maximum values of "x".
```

- sequence of scalars:
  Defines the bin edges allowing for non-uniform

```
        width. No extension of the range of "x" is done.
```

```
categorical_object = pd.cut(weights_of_persons, 18)

print(categorical_object)
```

```
[(71.35, 73.456], (69.244, 71.35], (62.928, 65.033], (75.561, 77.6
67], (73.456, 75.561], ..., (56.611, 58.717], (67.139, 69.244], (7
3.456, 75.561], (88.194, 90.3], (62.928, 65.033]]
Length: 18
Categories (18, interval[float64]): [(52.362, 54.506] < (54.506, 5
6.611] < (56.611, 58.717] < (58.717, 60.822] ... (81.878, 83.983]
< (83.983, 86.089] < (86.089, 88.194] < (88.194, 90.3]]
```

```
sequence_of_scalars = [ x[0] for x in bins]
sequence_of_scalars.append(bins[-1][1])
print(sequence_of_scalars)
categorical_object = pd.cut(weights_of_persons,
                            sequence_of_scalars,
                            right=False)
print(categorical_object)
```

```
[50, 54, 58, 62, 66, 70, 74, 78, 82, 86, 90, 94]
[[70, 74), [66, 70), [62, 66), [74, 78), [74, 78), ..., [58, 62),
[66, 70), [74, 78), [86, 90), [62, 66)]
Length: 18
Categories (11, interval[int64]): [[50, 54) < [54, 58) < [58, 62)
< [62, 66) ... [78, 82) < [82, 86) < [86, 90) < [90, 94)]
```

BIN COUNTS AND VALUE COUNTS

The next and most interesting question is now how we can see the actual bin counts. This can be accomplished with the function "value_counts":

```
pd.value_counts(categorical_object)
```

Output:

```
[74, 78)     3
[66, 70)     3
[86, 90)     2
[82, 86)     2
[78, 82)     2
[62, 66)     2
[90, 94)     1
[70, 74)     1
[58, 62)     1
[50, 54)     1
[54, 58)     0
dtype: int64
```

"categorical_object.codes" provides you with a labelling of the input values into the binning categories:

```
labels = categorical_object.codes
labels
```

Output:

```
array([ 5,  4,  3,  6,  6,  7,  7,  8,  9, 10,  8,  4,  0,
2,  4,  6,  9,
        3], dtype=int8)
```

categories is the IntervalIndex of the categories of the label indices:

```
categories = categorical_object.categories
categories
```

Output:

```
IntervalIndex([[50, 54), [54, 58), [58, 62), [62, 66), [66, 7
0) ... [74, 78), [78, 82), [82, 86), [86, 90), [90, 94)],
              closed='left',
              dtype='interval[int64]')
```

Correspondence from weights data to bins:

```
for index in range(len(weights_of_persons)):
    label_index = labels[index]
    print(weights_of_persons[index], label_index, categories[labe
l_index] )
```

```
73.4 5 [70, 74)
69.3 4 [66, 70)
64.9 3 [62, 66)
75.6 6 [74, 78)
74.9 6 [74, 78)
80.3 7 [78, 82)
78.6 7 [78, 82)
84.1 8 [82, 86)
88.9 9 [86, 90)
90.3 10 [90, 94)
83.4 8 [82, 86)
69.3 4 [66, 70)
52.4 0 [50, 54)
58.3 2 [58, 62)
67.4 4 [66, 70)
74.0 6 [74, 78)
89.3 9 [86, 90)
63.4 3 [62, 66)
```

```
categorical_object.categories
```

Output:
```
IntervalIndex([[50, 54), [54, 58), [58, 62), [62, 66), [66, 7
0) ... [74, 78), [78, 82), [82, 86), [86, 90), [90, 94)],
              closed='left',
              dtype='interval[int64]')
```

NAMING BINS

Let's imagine, we have an University, which confers three levels of Latin honors depending on the grade point average (GPA):

- "summa cum laude" requires a GPA above 3.9
- "magna cum laude" if the GPA is above 3.8
- "cum laude" if the GPA of 3.6 or above

```
degrees = ["none", "cum laude", "magna cum laude", "summa cum laud
e"]
student_results = [3.93, 3.24, 2.80, 2.83, 3.91, 3.698, 3.731, 3.2
5, 3.24, 3.82, 3.22]

student_results_degrees = pd.cut(student_results, [0, 3.6, 3.8,
3.9, 4.0], labels=degrees)
pd.value_counts(student_results_degrees)
```

Output:

```
none                6
summa cum laude     2
cum laude           2
magna cum laude     1
dtype: int64
```

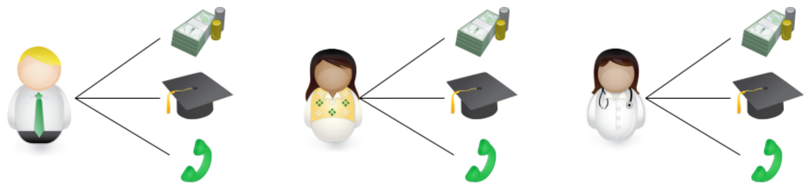
Let's have a look at the individual degrees of each student:

```
labels = student_results_degrees.codes
categories = student_results_degrees.categories

for index in range(len(student_results)):
    label_index = labels[index]
    print(student_results[index], label_index, categories[label_in
dex] )
```

```
3.93 3 summa cum laude
3.24 0 none
2.8 0 none
2.83 0 none
3.91 3 summa cum laude
3.698 1 cum laude
3.731 1 cum laude
3.25 0 none
3.24 0 none
3.82 2 magna cum laude
3.22 0 none
```

PANDAS TUTORIAL: MULTI-LEVEL INDEXING

INTRODUCTION



We learned the basic concepts of Pandas in our previous chapter of our tutorial on Pandas. We introduced the data structures

- Series and
- DataFrame

We also learned how to create and manipulate the Series and DataFrame objects in numerous Python programs.

Now it is time to learn some further aspects of theses data structures in this chapter of our tutorial.

We will start with advanced indexing possibilities in Pandas.

ADVANCED OR MULTI-LEVEL INDEXING

Advanced or multi-level indexing is available both for Series and for DataFrames. It is a fascinating way of working with higher dimensional data, using Pandas data structures. It's an efficient way to store and manipulate arbitrarily high dimension data in 1-dimensional (Series) and 2-dimensional tabular (DataFrame) structures. In other words, we can work with higher dimensional data in lower dimensions. It's time to present an example in Python:

```
import pandas as pd

cities = ["Vienna", "Vienna", "Vienna",
          "Hamburg", "Hamburg", "Hamburg",
          "Berlin", "Berlin", "Berlin",
          "Zürich", "Zürich", "Zürich"]
index = [cities, ["country", "area", "population",
                  "country", "area", "population",
                  "country", "area", "population",
                  "country", "area", "population"]]
print(index)
```

```
[['Vienna', 'Vienna', 'Vienna', 'Hamburg', 'Hamburg', 'Hamburg',
'Berlin', 'Berlin', 'Berlin', 'Zürich', 'Zürich', 'Zürich'], ['cou
ntry', 'area', 'population', 'country', 'area', 'population', 'cou
ntry', 'area', 'population', 'country', 'area', 'population']]
```

```
data = ["Austria", 414.60,    1805681,
        "Germany", 755.00,    1760433,
        "Germany", 891.85,    3562166,
        "Switzerland", 87.88, 378884]

city_series = pd.Series(data, index=index)
print(city_series)
```

```
Vienna   country             Austria
         area                  414.6
         population          1805681
Hamburg  country             Germany
         area                    755
         population          1760433
Berlin   country             Germany
         area                 891.85
         population          3562166
Zürich   country         Switzerland
         area                  87.88
         population           378884
dtype: object
```

We can access the data of a city in the following way:

```
print(city_series["Vienna"])
```

```
country       Austria
area            414.6
population    1805681
dtype: object
```

We can also access the information about the country, area or population of a city. We can do this in two ways:

```
print(city_series["Vienna"]["area"])
```

```
414.6
```

The other way to accomplish it:

```
print(city_series["Vienna", "area"])
```

```
414.6
```

We can also get the content of multiple cities at the same time by using a list of city names as the key:

```
city_series["Hamburg",:]
```

Output:
```
country        Germany
area               755
population     1760433
dtype: object
```

If the index is sorted, we can also apply a slicing operation:

```
city_series = city_series.sort_index()
print("city_series with sorted index:")
print(city_series)

print("\n\nSlicing the city_series:")
city_series["Berlin":"Vienna"]
```

```
city_series with sorted index:
Berlin   area                891.85
         country            Germany
         population         3562166
Hamburg  area                   755
         country            Germany
         population         1760433
Vienna   area                 414.6
         country            Austria
         population         1805681
Zürich   area                 87.88
         country        Switzerland
         population          378884
dtype: object


Slicing the city_series:
```

Output:
```
Berlin   area              891.85
         country          Germany
         population       3562166
Hamburg  area                 755
         country          Germany
         population       1760433
Vienna   area               414.6
         country          Austria
         population       1805681
dtype: object
```

In the next example, we show that it is possible to access the inner keys as well:

```
print(city_series[:, "area"])
```

```
Berlin      891.85
Hamburg        755
Vienna       414.6
Zürich       87.88
dtype: object
```

SWAPPING MULTIINDEX LEVELS

It is possible to swap the levels of a MultiIndex with the method swaplevel:

```
swaplevel(self, i=-2, j=-1, copy=True)
    Swap levels i and j in a MultiIndex


    Parameters
    ----------
    i, j : int, string (can be mixed)
           Level of index to be swapped. Can pass level name as string.
           The indexes 'i' and 'j' are optional, and default to
           the two innermost levels of the index

    Returns
    -------
    swapped : Series
```

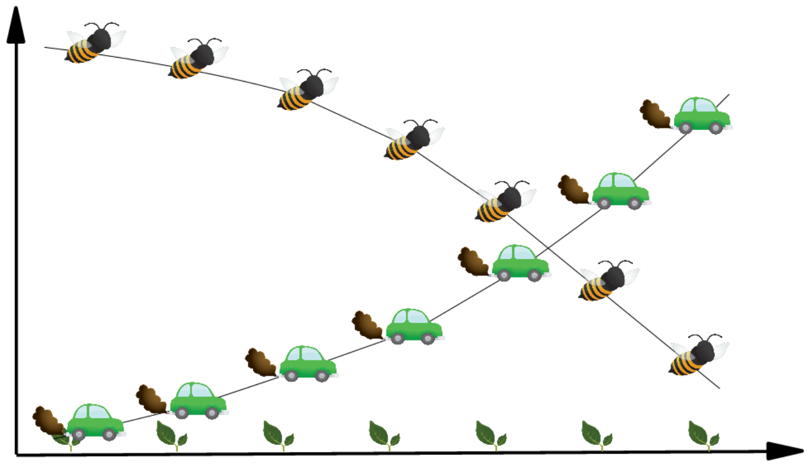
```
city_series = city_series.swaplevel()
city_series.sort_index(inplace=True)
city_series
```

Output:

```
area        Berlin          891.85
            Hamburg            755
            Vienna           414.6
            Zürich           87.88
country     Berlin         Germany
            Hamburg        Germany
            Vienna         Austria
            Zürich     Switzerland
population  Berlin         3562166
            Hamburg        1760433
            Vienna         1805681
            Zürich          378884
dtype: object
```

DATA VISUALIZATION WITH PANDAS

INTRODUCTION



It is seldom a good idea to present your scientific or business data solely in rows and columns of numbers. We rather use various kinds of diagrams to visualize our data. This makes the communication of information more efficiently and easy to grasp. In other words, it makes complex data more accessible and understandable. The numerical data can be graphically encoded with line charts, bar charts, pie charts, histograms, scatterplots and others.

We have already seen the powerful capabilities of for creating publication-quality plots. Matplotlib is a low-level tool to achieve this goal, because you have to construe your plots by adding up basic components, like legends, tick labels, contours and so on.
Pandas provides various plotting possibilities, which make like a lot easier.

We will start with an example for a line plot.

LINE PLOT IN PANDAS

SERIES

Both the Pandas Series and DataFrame objects support a plot method.

You can see a simple example of a line plot with for a Series object. We use a simple Python list "data" as the data for the range. The index will be used for the x values, or the domain.

```
import pandas as pd

data = [100, 120, 140, 180, 200, 210, 214]
s = pd.Series(data, index=range(len(data)))

s.plot()
```

```
Output: <matplotlib.axes._subplots.AxesSubplot at 0x7fe98be8a710>
```

It is possible to suppress the usage of the index by setting the keyword parameter "use_index" to False. In our example this will give us the same result:

```
s.plot(use_index=False)
```

```
Output: <matplotlib.axes._subplots.AxesSubplot at 0x7fe98be8a710>
```

We will experiment now with a Series which has an index consisting of alphabetical values.

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
S.plot()
```

```
Output: <matplotlib.axes._subplots.AxesSubplot at 0x7fe98be8a710>
```

LINE PLOTS IN DATAFRAMES

We will introduce now the plot method of a DataFrame. We define a dcitionary with the population and area figures. This dictionary can be used to create the DataFrame, which we want to use for plotting:

```
import pandas as pd

cities = {"name": ["London", "Berlin", "Madrid", "Rome",
                   "Paris", "Vienna", "Bucharest", "Hamburg",
                   "Budapest", "Warsaw", "Barcelona",
                   "Munich", "Milan"],
          "population": [8615246, 3562166, 3165235, 2874038,
                         2273305, 1805681, 1803425, 1760433,
                         1754000, 1740119, 1602386, 1493900,
                         1350680],
          "area" : [1572, 891.85, 605.77, 1285,
                    105.4, 414.6, 228, 755,
                    525.2, 517, 101.9, 310.4,
                    181.8]
}

city_frame = pd.DataFrame(cities,
                          columns=["population", "area"],
                          index=cities["name"])
print(city_frame)
```

```
           population     area
London        8615246  1572.00
Berlin        3562166   891.85
Madrid        3165235   605.77
Rome          2874038  1285.00
Paris         2273305   105.40
Vienna        1805681   414.60
Bucharest     1803425   228.00
Hamburg       1760433   755.00
Budapest      1754000   525.20
Warsaw        1740119   517.00
Barcelona     1602386   101.90
Munich        1493900   310.40
Milan         1350680   181.80
```

The following code plots our DataFrame city_frame. We will multiply the area column by 1000, because otherwise the "area" line would not be visible or in other words would be overlapping with the x axis:

```
city_frame["area"] *= 1000
city_frame.plot()
```

Output: <matplotlib.axes._subplots.AxesSubplot at 0x7fe9b13f2c50>

This plot is not coming up to our expectations, because not all the city names appear on the x axis. We can change this by defining the xticks explicitly with "range(len((city_frame.index))". Furthermore, we have to set use_index to True, so that we get city names and not numbers from 0 to len((city_frame.index):

```
city_frame.plot(xticks=range(len(city_frame.index)),
                use_index=True)
```

Output: <matplotlib.axes._subplots.AxesSubplot at 0x7fe983cd72e8>

Now, we have a new problem. The city names are overlapping. There is remedy at hand for this problem as well. We can rotate the strings by 90 degrees. The names will be printed vertically afterwards:

```
city_frame.plot(xticks=range(len(city_frame.index)),
                use_index=True,
                rot=90)
```

Output: <matplotlib.axes._subplots.AxesSubplot at 0x7fe98bbf6c88>

USING TWIN AXES

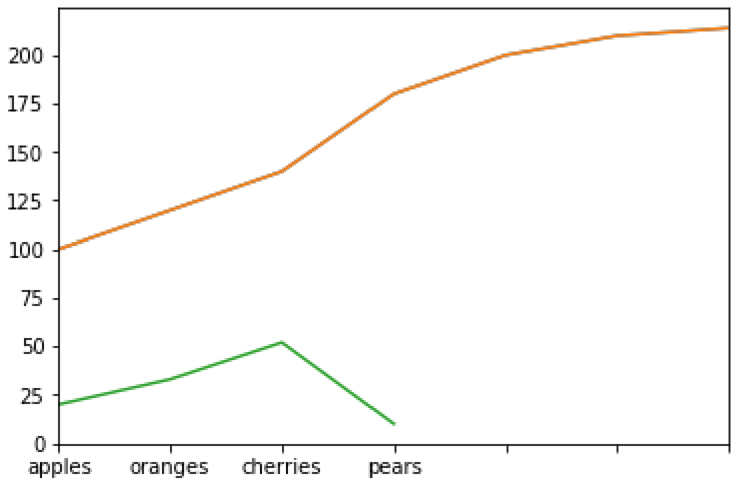
We multiplied the area column by 1000 to get a proper output. Instead of this, we could have used twin axes. We will demonstrate this in the following example. We will recreate the city_frame DataFrame to get the original area column:

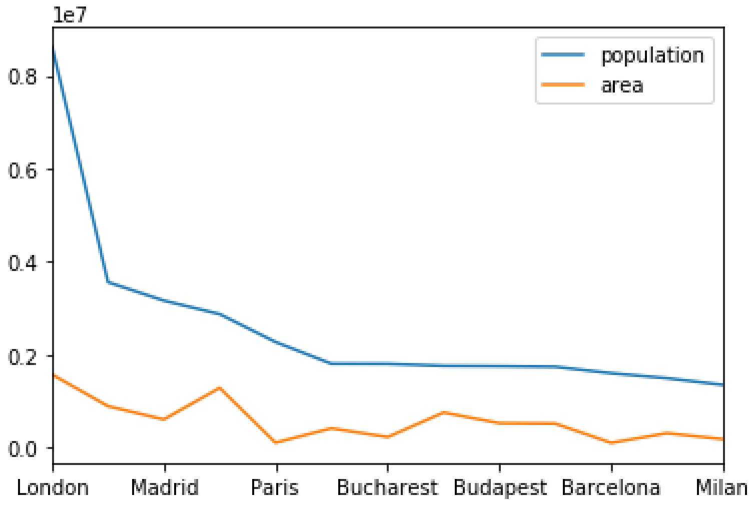
```
city_frame = pd.DataFrame(cities,
                          columns=["population", "area"],
                          index=cities["name"])
print(city_frame)
```

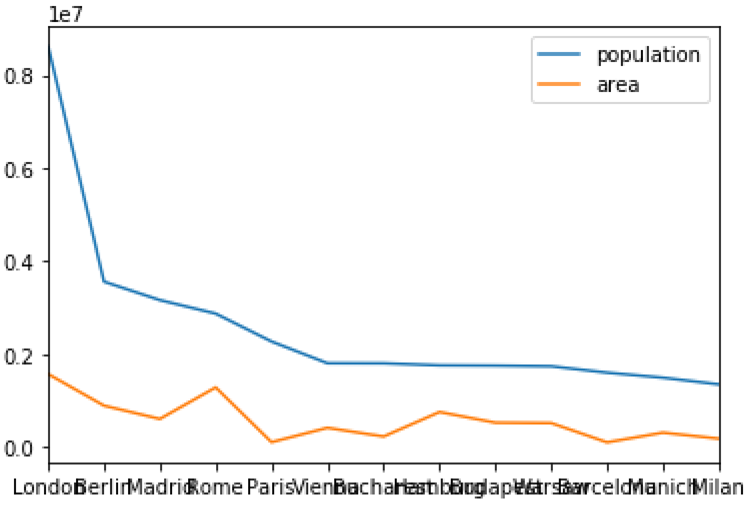
```
           population     area
London        8615246  1572.00
Berlin        3562166   891.85
Madrid        3165235   605.77
Rome          2874038  1285.00
Paris         2273305   105.40
Vienna        1805681   414.60
Bucharest     1803425   228.00
Hamburg       1760433   755.00
Budapest      1754000   525.20
Warsaw        1740119   517.00
Barcelona     1602386   101.90
Munich        1493900   310.40
Milan         1350680   181.80
```

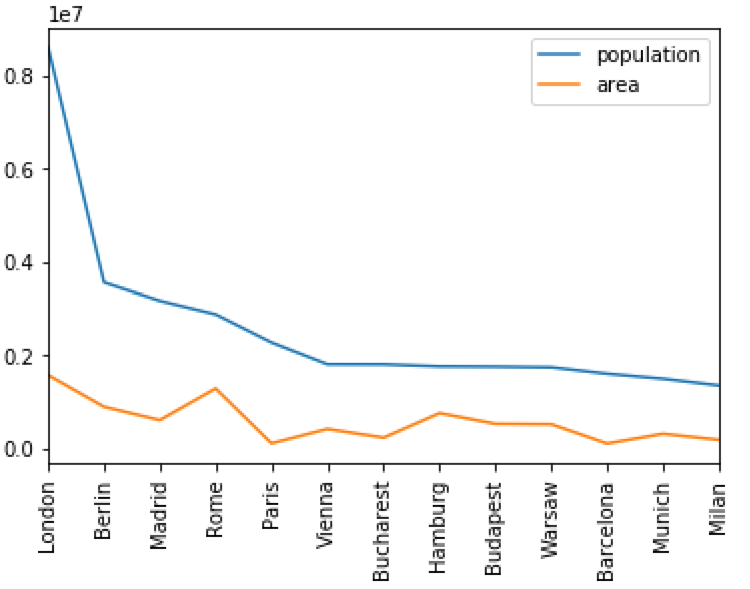
To get a twin axes represenation of our diagram, we need subplots from the module matplotlib and the function "twinx":

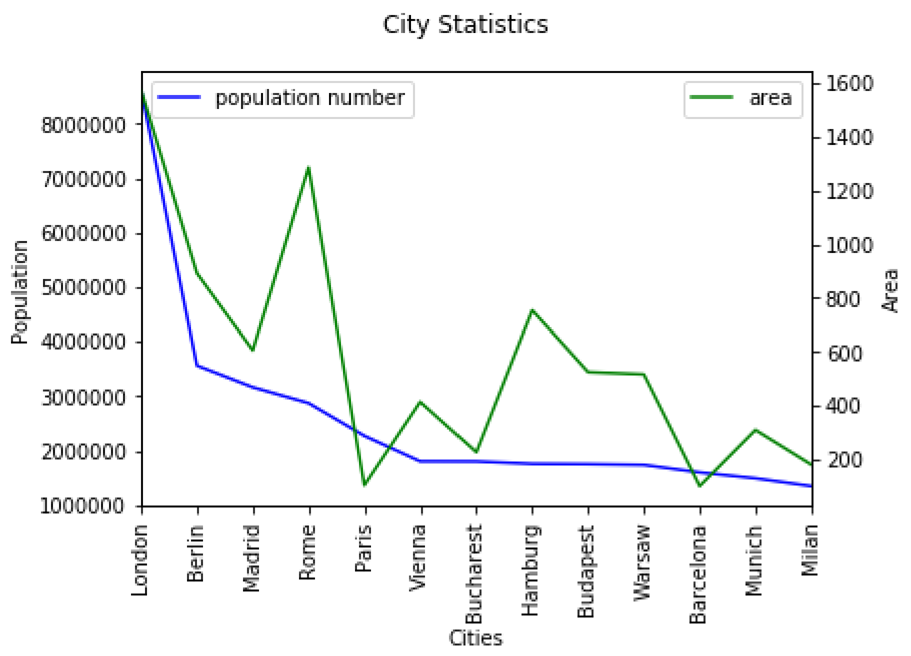
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
fig.suptitle("City Statistics")
ax.set_ylabel("Population")
ax.set_xlabel("Cities")

ax2 = ax.twinx()
ax2.set_ylabel("Area")

city_frame["population"].plot(ax=ax,
                              style="b-",
                              xticks=range(len(city_frame.index)),
                              use_index=True,
                              rot=90)

city_frame["area"].plot(ax=ax2,
                        style="g-",
                        use_index=True,
                        rot=90)

ax.legend(["population number"], loc=2)
ax2.legend(loc=1)

plt.show()
```


200
175
150
125
100
75
50
25
0
apples
oranges
cherries
pears


1e7
0.8
0.6
0.4
0.2
0.0
population
area
London
Madrid
Paris
Bucharest
Budapest
Barcelona
Milan

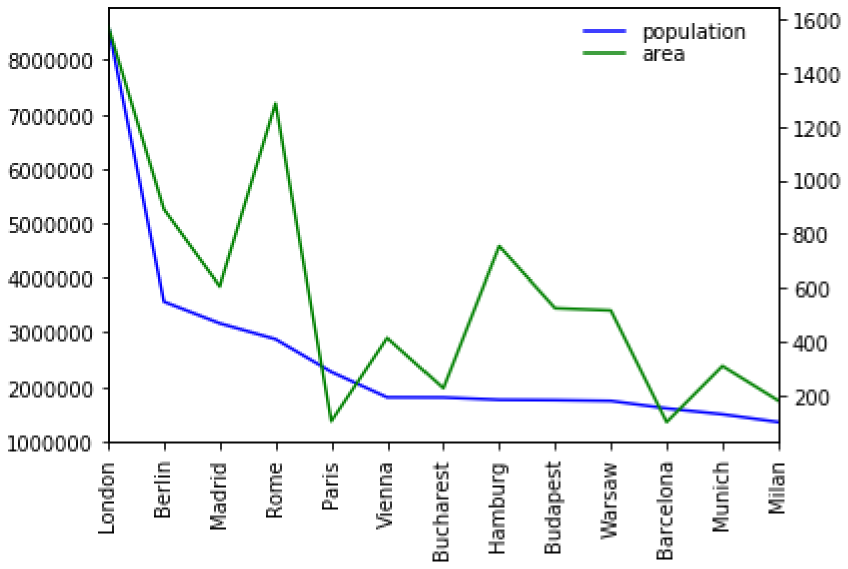

1e7
0.8
0.6
0.4
0.2
0.0
population
area
London
Berlin
Madrid
Rome
Paris
Vienna
Bucharest
Hamburg
Budapest
Warsaw
Barcelona
Munich
Milan


1e7
0.8
0.6
0.4
0.2
0.0
population
area
London
Berlin
Madrid
Rome
Paris
Vienna
Bucharest
Hamburg
Budapest
Warsaw
Barcelona
Munich
Milan



We can also create twin axis directly in Pandas without the aid of Matplotlib. We demonstrate this in the code of the following program:

```
import matplotlib.pyplot as plt

ax1= city_frame["population"].plot(style="b-",
                                   xticks=range(len(city_frame.index)),
                                   use_index=True,
                                   rot=90)
ax2 = ax1.twinx()

city_frame["area"].plot(ax=ax2,
                        style="g-",
                        use_index=True,
                        #secondary_y=True,
                        rot=90)



ax1.legend(loc = (.7,.9), frameon = False)
ax2.legend( loc = (.7, .85), frameon = False)

plt.show()
```



MULTIPLE Y AXES

Let's add another axes to our city_frame. We will add a column with the population density, i.e. the number of people per square kilometre:

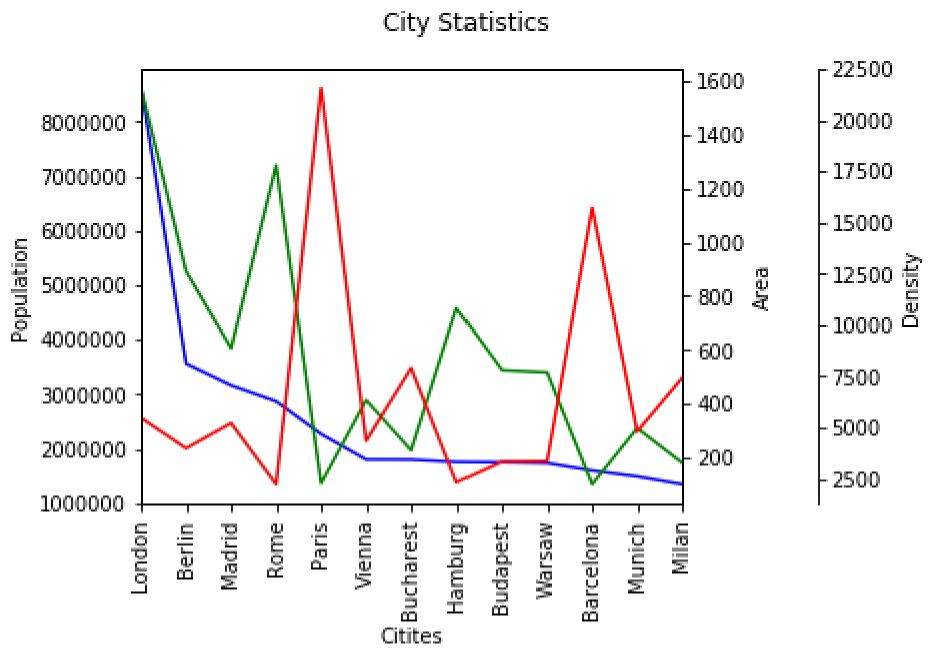
```
city_frame["density"] = city_frame["population"] / city_frame["area"]

city_frame
```

Output:

	population	area	density
London	8615246	1572.00	5480.436387
Berlin	3562166	891.85	3994.131300
Madrid	3165235	605.77	5225.143206
Rome	2874038	1285.00	2236.605447
Paris	2273305	105.40	21568.358634
Vienna	1805681	414.60	4355.236372
Bucharest	1803425	228.00	7909.758772
Hamburg	1760433	755.00	2331.699338
Budapest	1754000	525.20	3339.680122
Warsaw	1740119	517.00	3365.800774
Barcelona	1602386	101.90	15725.083415
Munich	1493900	310.40	4812.822165
Milan	1350680	181.80	7429.482948

Now we have three columns to plot. For this purpose, we will create three axes for our values:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
fig.suptitle("City Statistics")
ax.set_ylabel("Population")
ax.set_xlabel("Citites")

ax_area, ax_density = ax.twinx(), ax.twinx()
ax_area.set_ylabel("Area")
ax_density.set_ylabel("Density")

rspine = ax_density.spines['right']
rspine.set_position(('axes', 1.25))
ax_density.set_frame_on(True)
ax_density.patch.set_visible(False)
fig.subplots_adjust(right=0.75)

city_frame["population"].plot(ax=ax,
                              style="b-",
                              xticks=range(len(city_frame.index)),
                              use_index=True,
                              rot=90)

city_frame["area"].plot(ax=ax_area,
                        style="g-",
                        use_index=True,
                        rot=90)

city_frame["density"].plot(ax=ax_density,
                           style="r-",
                           use_index=True,
                           rot=90)

plt.show()
```



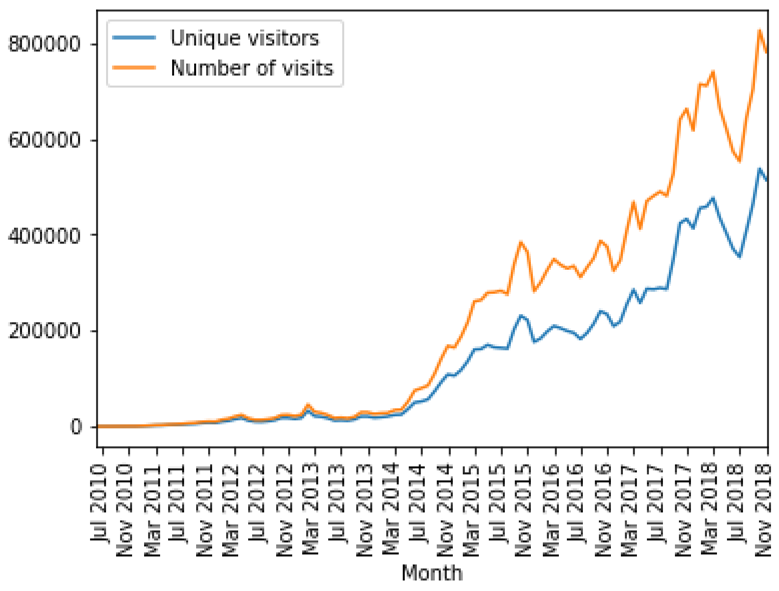
A MORE COMPLEX EXAMPLE

We use the previously gained knowledge in the following example. We use a file with visitor statistics from our website python-course.eu. The content of the file looks like this:

```
Month Year   "Unique visitors"   "Number of visits"  Pages   Hits    Bandwid
th Unit
Jun 2010     11  13  42  290 2.63 MB
Jul 2010     27  39  232 939 9.42 MB
Aug 2010     75  87  207 1,096   17.37 MB
Sep 2010     171 221 480 2,373   39.63 MB

...

Apr 2018     434,346 663,327 1,143,762   7,723,268   377.56 GB
May 2018     402,390 619,993 1,124,517   7,307,779   394.72 GB
Jun 2018     369,739 573,102 1,034,335   6,773,820   386.60 GB
Jul 2018     352,670 552,519   967,778   6,551,347   375.86 GB
Aug 2018     407,512 642,542 1,223,319   7,829,987   472.37 GB
Sep 2018     463,937 703,327 1,187,224   8,468,723   514.46 GB
Oct 2018     537,343 826,290 1,403,176  10,013,025   620.55 GB
Nov 2018     514,072 781,335 1,295,594   9,487,834   642.16 GB
```

```
%matplotlib inline

import pandas as pd

data_path = "data1/"
data = pd.read_csv(data_path + "python_course_monthly_history.txt",
                   quotechar='"',
                   thousands=",",
                   delimiter=r"\s+")

def unit_convert(x):
    value, unit = x
    if unit == "MB":
        value *= 1024
    elif unit == "GB":
        value *= 1048576 # i.e. 1024 **2
    return value

b_and_u= data[["Bandwidth", "Unit"]]
bandwidth = b_and_u.apply(unit_convert, axis=1)

del data["Unit"]
data["Bandwidth"] = bandwidth

month_year =  data[["Month", "Year"]]
month_year = month_year.apply(lambda x: x[0] + " " + str(x[1]),
                              axis=1)
data["Month"] = month_year
del data["Year"]

data.set_index("Month", inplace=True)
del data["Bandwidth"]

data[["Unique visitors", "Number of visits"]].plot(use_index=True,
                                                   rot=90,
                                                   xticks=range(1, len(da
ta.index),4))
```

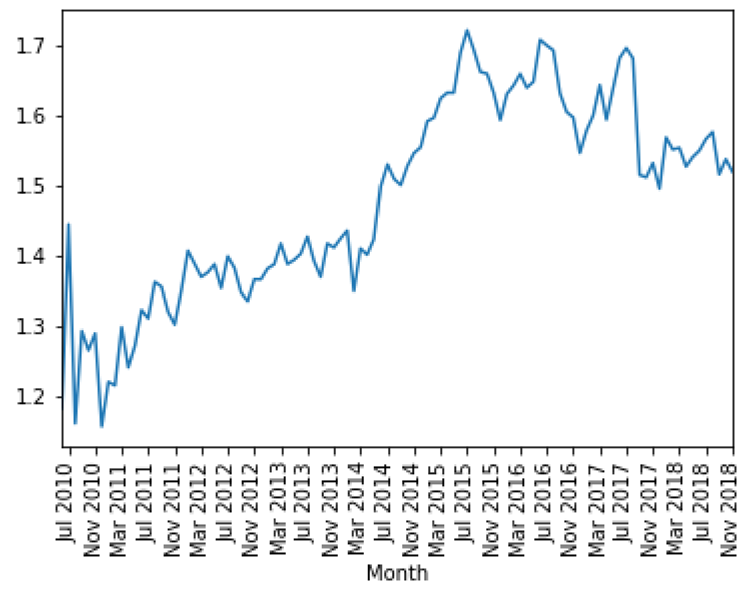
Output:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe9836b27b8>
```



```
ratio = pd.Series(data["Number of visits"] / data["Unique visitors"],
                  index=data.index)
ratio.plot(use_index=True,
           xticks=range(1, len(ratio.index),4),
           rot=90)
```

Output:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe9838e5860>
```



CONVERTING STRING COLUMNS TO FLOATS

In the folder "data1", we have a file called

"tiobe_programming_language_usage_nov2018.txt"

with a ranking of programming languages by usage. The data has been collected and created by TIOBE in November 2018.

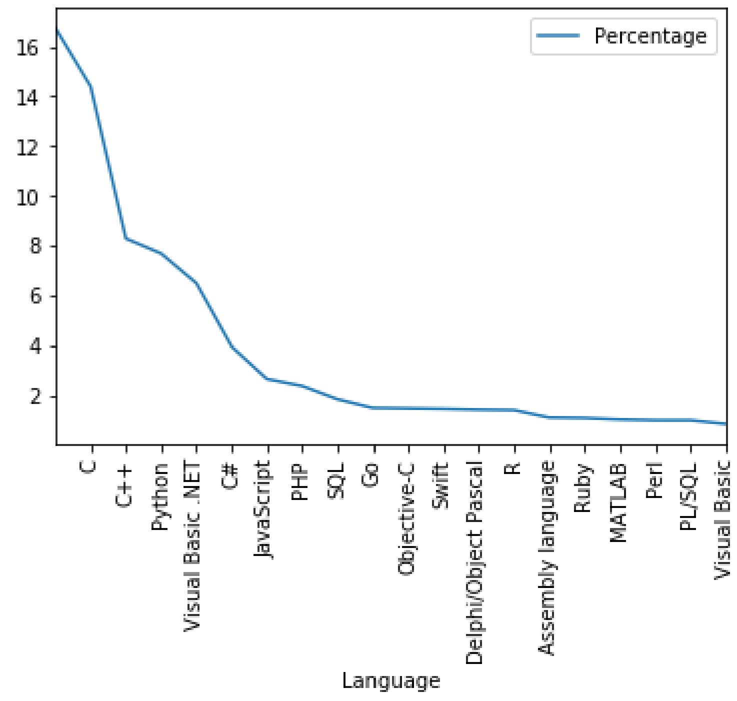
The file looks like this:

```
    Position    "Language"  Percentage
    1       Java    16.748%
    2       C   14.396%
    3       C++ 8.282%
    4       Python  7.683%
    5       "Visual Basic .NET" 6.490%
    6       C#  3.952%
    7       JavaScript  2.655%
    8       PHP 2.376%
    9       SQL 1.844%
```

The percentage column contains strings with a percentage sign. We can get rid of this when we read in the data with read_csv. All we have to do is define a converter function, which we to read_csv via the converters dictionary, which contains column names as keys and references to functions as values.

```
def strip_percentage_sign(x):
    return float(x.strip('%'))

data_path = "data1/"
progs = pd.read_csv(data_path + "tiobe_programming_language_usage_nov201
8.txt",
                   quotechar='"',
                   thousands=",",
                   index_col=1,
                   converters={'Percentage':strip_percentage_sign},
                   delimiter=r"\s+")

del progs["Position"]

print(progs.head(6))
progs.plot(xticks=range(1, len(progs.index)),
           use_index=True, rot=90)

plt.show()
```

```
                          Percentage
Language
Java                          16.748
C                             14.396
C++                            8.282
Python                         7.683
Visual Basic .NET              6.490
C#                             3.952
```



BAR PLOTS IN PANDAS

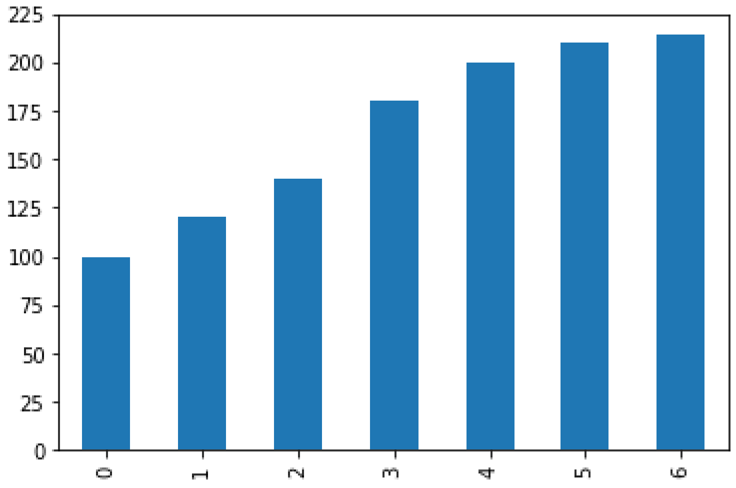
To create bar plots with Pandas is as easy as plotting line plots. All we have to do is add the keyword parameter "kind" to the plot method and set it to "bar".

A SIMPLE EXAMPLE

```
import pandas as pd

data = [100, 120, 140, 180, 200, 210, 214]
s = pd.Series(data, index=range(len(data)))
```

```
s.plot(kind="bar")
```

Output:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe98349a5c0>
```

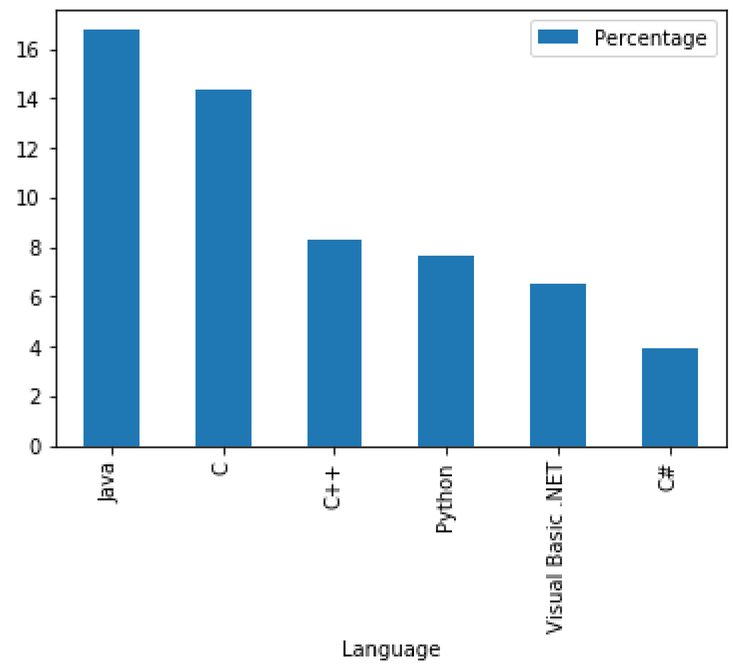


BAR PLOT FOR PROGRAMMING LANGUAGE USAGE

Let's get back to our programming language ranking. We will printout now a bar plot of the six most used programming languages:

```
progs[:6].plot(kind="bar")
```

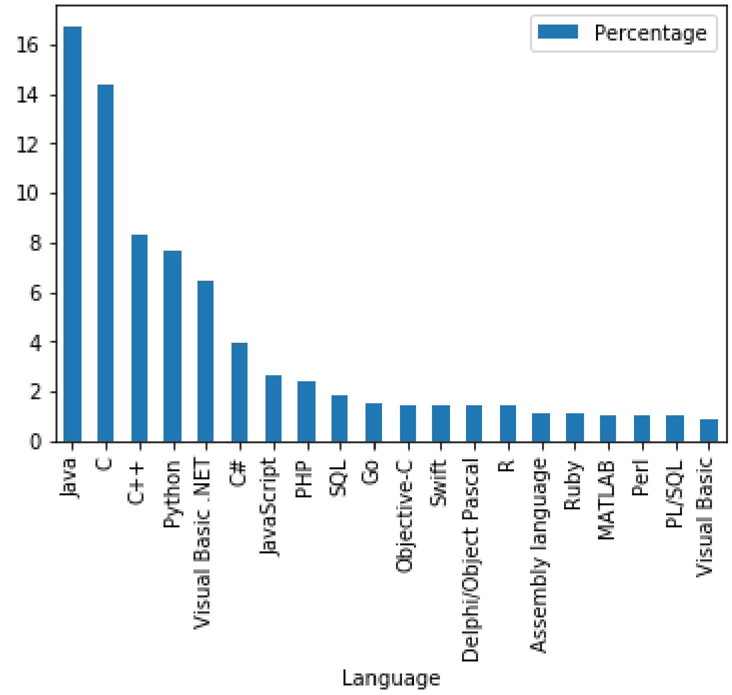
Output:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe983ac05c0>
```



Now the whole chart with all programming languages:

```
progs.plot(kind="bar")
```

Output:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe983a84208>
```

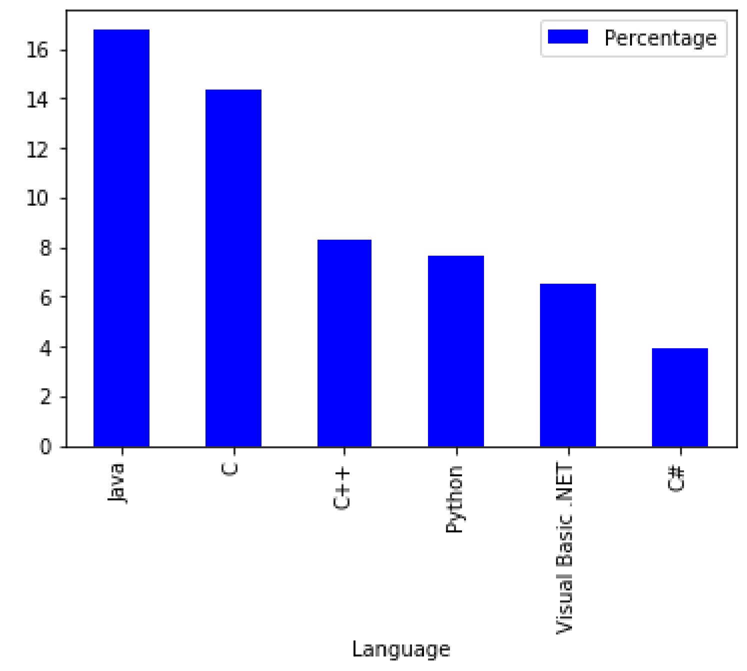


COLORIZE A BAR PLOT

It is possible to colorize the bars indivually by assigning a list to the keyword parameter color:

```
my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs[:6].plot(kind="bar",
               color=my_colors)
```

Output: `<matplotlib.axes._subplots.AxesSubplot at 0x7fe9834509e8>`

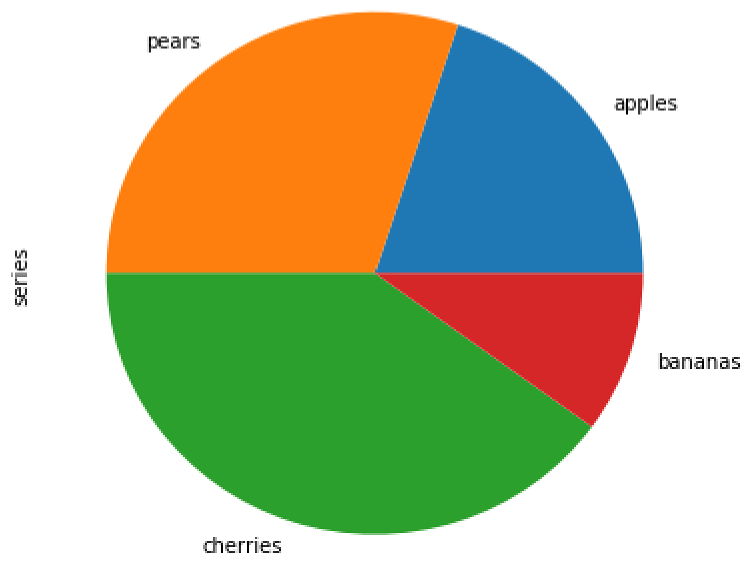


PIE CHART DIAGRAMS IN PANDAS

A simple example:

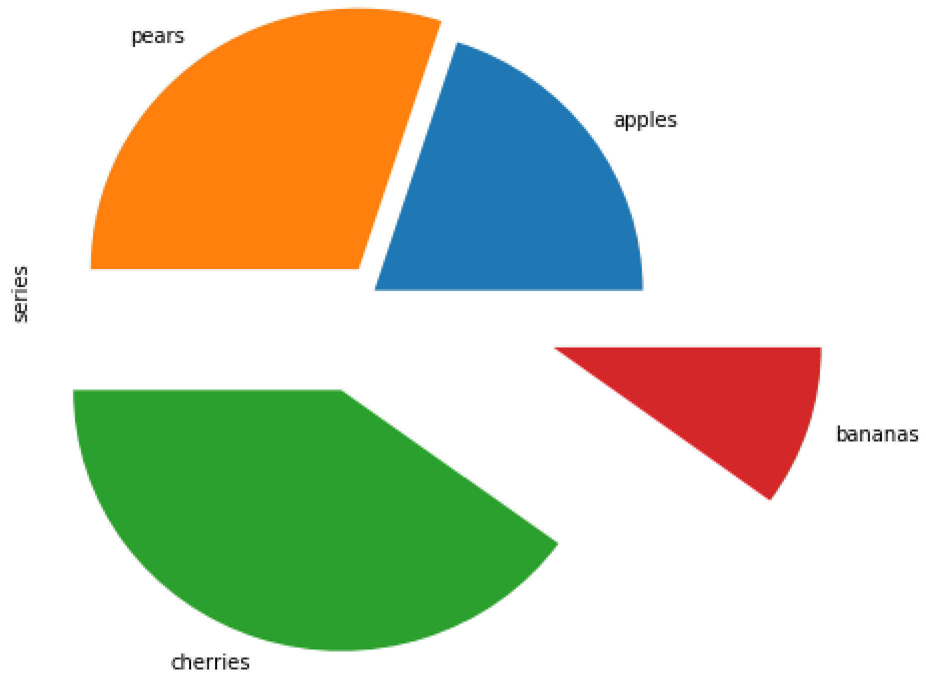
```
import pandas as pd

fruits = ['apples', 'pears', 'cherries', 'bananas']
series = pd.Series([20, 30, 40, 10],
                   index=fruits,
                   name='series')

series.plot.pie(figsize=(6, 6))
```

```
Output: <matplotlib.axes._subplots.AxesSubplot at 0x7fe9832fe048>
```



```
fruits = ['apples', 'pears', 'cherries', 'bananas']

series = pd.Series([20, 30, 40, 10],
                   index=fruits,
                   name='series')
explode = [0, 0.10, 0.40, 0.7]
series.plot.pie(figsize=(6, 6),
                explode=explode)
```

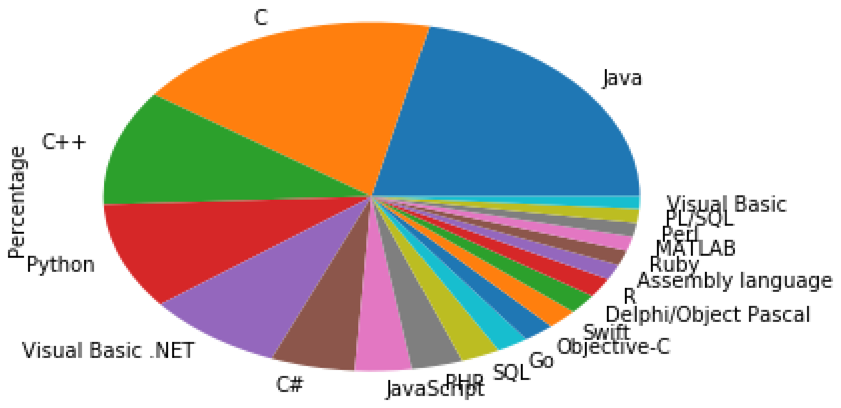
Output: `<matplotlib.axes._subplots.AxesSubplot at 0x7fe983370198>`



We will replot the previous bar plot as a pie chart plot:

```
import matplotlib.pyplot as plt

my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs.plot.pie(subplots=True,
               legend=False)
```

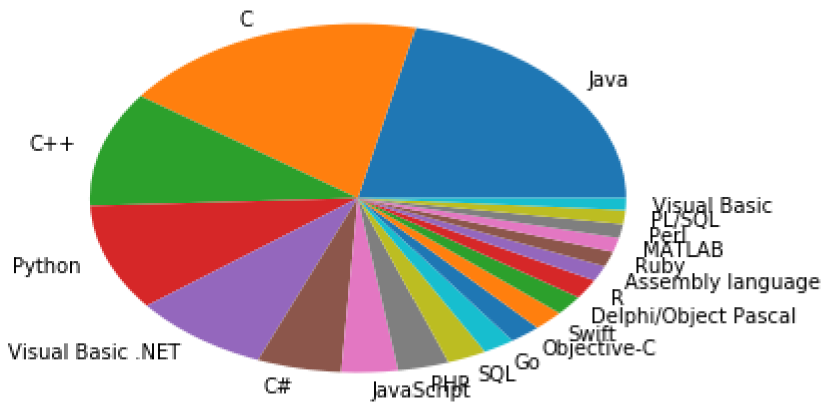
Output:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7fe983244
0b8>], dtype=object)
```



It looks ugly that we see the y label "Percentage" inside our pie plot. We can remove it by calling "plt.ylabel('')"

```
import matplotlib.pyplot as plt

my_colors = ['b', 'r', 'c', 'y', 'g', 'm']
progs.plot.pie(subplots=True,
               legend=False)

plt.ylabel('')
```

Output:
```
<matplotlib.text.Text at 0x7fe983117b70>
```


C
Java
C++
Visual Basic
PL/SQL
Perl
MATLAB
Ruby
Assembly language
Python
R
Delphi/Object Pascal
Swift
Objective-C
Visual Basic .NET
Go
SQL
PHP
C#
JavaScript

PYTHON, DATE AND TIME

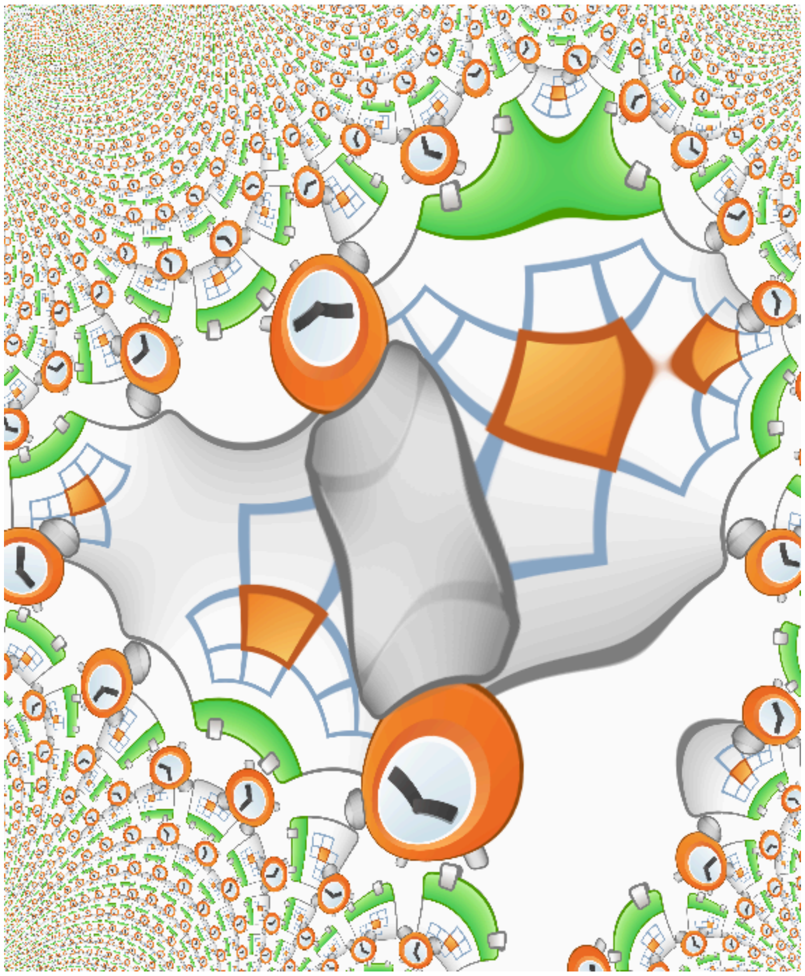
INTRODUCTION

Python provides rich functionalities for dealing with date and time data. The standard libraries contains the modules

- time
- calendar
- datetime

These modules supply classes for manipulating dates and times in both simple and complex ways.

Especially, the datetime class will be very important for the timeseries of Pandas.



PYTHON STANDARD MODULES FOR TIME DATA

The most important modules of Python dealing with time are the modules `time`, `calendar` and `datetime`.

The datetime module provides various classes, methods and functions to deal with dates, times, and time intervals.

The datetime module provides the following classes:

- The instances of the date class represent dates, whereas the year can range between 1 and 9999.
- The instances of the datetime class are made up both by a date and a time.
- The time class implements time objects.
- The timedelta class is used to hold the differences between two times or two date objects.
- The tzinfo class is used to implement timezone support for time and datetime objects.

Let's start with a date object.

THE DATE CLASS

```python
from datetime import date

x = date(1993, 12, 14)
print(x)
```

```
1993-12-14
```

We can instantiate dates in the range from January 1, 1 to December 31, 9999. This can be inquired from the attributes `min` and `max`:

```python
from datetime import date

print(date.min)
print(date.max)
```

```
0001-01-01
9999-12-31
```

We can apply various methods to the date instance above.
The method toordinal returns the proleptic Gregorian ordinal. The proleptic Gregorian calendar is produced by extending the Gregorian calendar backward to dates preceding its official introduction in 1582. January 1 of year 1 is day 1.

```python
x.toordinal()
```

Output: 727911

It is possible to calculate a date from a ordinal by using the class method "fromordinal":

```
date.fromordinal(727911)
```

Output: datetime.date(1993, 12, 14)

If you want to know the weekday of a certain date, you can calculate it by using the method weekday:

```
x.weekday()
```

Output: 1

```
date.today()
```

Output: datetime.date(2017, 4, 12)

We can access the day, month and year with attributes:

```
print(x.day)
print(x.month)
print(x.year)
```

```
14
12
1993
```

THE TIME CLASS

The time class is similarly organized than the date class.

```
from datetime import time

t = time(15, 6, 23)
print(t)
```

```
15:06:23
```

The possible times range between:

```
print(time.min)
print(time.max)
```

```
00:00:00
23:59:59.999999
```

Accessing 'hour', 'minute' and 'second':

```
t.hour, t.minute, t.second
```

Output:
```
(15, 6, 23)
```

Each component of a time instance can be changed by using 'replace':

```
t = t.replace(hour=11, minute=59)
t
```

Output:
```
datetime.time(11, 59, 23)
```

We can render a date as a C-style like string, corresponding to the C ctime function:

```
x.ctime()
```

Output:
```
'Tue Dec 14 00:00:00 1993'
```

THE DATETIME CLASS

The datetime module provides us with functions and methods for manipulating dates and times. It supplies functionalities for date and time arithmetic, i.e.

addition and subtraction. Another focus of the implementation is on attribute extraction for output manipulation and formatting.

There are two kinds of date and time objects:

- naive
- aware

If a time or date object is naive it doesn't contain information to compare or locate itself relative to other date or time objects. The semantics, if such a naive object belongs to a certain time zone, e.g. Coordinated Universal Time (UTC), local time, or some other timezone is contained in the logic of the program.

An aware object on the other hand possesses knowledge of the time zone it belongs to or the daylight saving time information. This way it can locate itself relative to other aware objects.

How can you tell if a datetime object t is aware?

t is aware if t.tzinfo is not None and t.tzinfo.utcoffset(t) is not None. Both conditions have to be fulfilled

On the other hand an object t is naive if t.tzinfo is None or t.tzinfo.utcoffset(t) is None

Let's create a datetime object:

```python
from datetime import datetime
t = datetime(2017, 4, 19, 16, 31, 0)
t
```

Output:
```
datetime.datetime(2017, 4, 19, 16, 31)
```

t is naive, because the following is True:

```python
t.tzinfo == None
```

Output:
```
True
```

We will create an aware datetime object from the current date. For this purpose we need the module pytz. pytz is a module, which brings the Olson tz database into Python. The Olson timezones are nearly completely supported by this module.

```
from datetime import datetime
import pytz
t = datetime.now(pytz.utc)
```

We can see that both t.tzinfo and t.tzinfo.utcoffset(t) are different from None, so t is an aware object:

```
t.tzinfo, t.tzinfo.utcoffset(t)
```

Output:
```
(<UTC>, datetime.timedelta(0))
```

```
from datetime import datetime, timedelta as delta
ndays = 15
start = datetime(1991, 4, 30)
dates = [start - delta(days=x) for x in range(0, ndays)]
dates
```

Output:
```
[datetime.datetime(1991, 4, 30, 0, 0),
 datetime.datetime(1991, 4, 29, 0, 0),
 datetime.datetime(1991, 4, 28, 0, 0),
 datetime.datetime(1991, 4, 27, 0, 0),
 datetime.datetime(1991, 4, 26, 0, 0),
 datetime.datetime(1991, 4, 25, 0, 0),
 datetime.datetime(1991, 4, 24, 0, 0),
 datetime.datetime(1991, 4, 23, 0, 0),
 datetime.datetime(1991, 4, 22, 0, 0),
 datetime.datetime(1991, 4, 21, 0, 0),
 datetime.datetime(1991, 4, 20, 0, 0),
 datetime.datetime(1991, 4, 19, 0, 0),
 datetime.datetime(1991, 4, 18, 0, 0),
 datetime.datetime(1991, 4, 17, 0, 0),
 datetime.datetime(1991, 4, 16, 0, 0)]
```

DIFFERENCES BETWEEN TIMES

Let's see what happens, if we subtract to datetime objects:

```
from datetime import datetime

delta = datetime(1993, 12, 14) - datetime(1991, 4, 30)
delta, type(delta)
```

Output:
```
(datetime.timedelta(959), datetime.timedelta)
```

The result of the subtraction of the two datetime objects is a timedelta object, as we can see from the example above.

We can get information about the number of days elapsed by using the attribute 'days':

```
delta.days
```

Output: 959

```
t1 = datetime(2017, 1, 31, 14, 17)
t2 = datetime(2015, 12, 15, 16, 59)
delta = t1 - t2
delta.days, delta.seconds
```

Output: (412, 76680)

It is possible to add or subtract a timedelta to a
datetime object to calculate a new datetime object by adding or subtracting the
delta in days:

```
from datetime import datetime, timedelta
d1 = datetime(1991, 4, 30)
d2 = d1 + timedelta(10)
print(d2)
print(d2 - d1)

d3 = d1 - timedelta(100)
print(d3)
d4 = d1 - 2 * timedelta(50)
print(d4)
```

```
1991-05-10 00:00:00
10 days, 0:00:00
1991-01-20 00:00:00
1991-01-20 00:00:00
```

It is also possible to add days and minutes to t datetime object:

```
from datetime import datetime, timedelta
d1 = datetime(1991, 4, 30)
d2 = d1 + timedelta(10,100)
print(d2)
print(d2 - d1)
```

```
1991-05-10 00:01:40
10 days, 0:01:40
```

CONVERT DATETIME OBJECTS TO STRINGS

The easiest way to convert a datetime object into a string consists in using str.

```
s = str(d1)
s
```

Output: '1991-04-30 00:00:00'

CONVERSION WITH STRFTIME

The method call datetime.strftime(format) return a string representing the date and time, controlled by an explicit format string. A complete list of formatting directives can be found at strftime:

```
print(d1.strftime('%Y-%m-%d'))
print("weekday: " + d1.strftime('%a'))
print("weekday as a full name: " + d1.strftime('%A'))
```

```
# Weekday as a decimal number, where 0 is Sunday
# and 6 is Saturday
print("weekday as a decimal number: " + d1.strftime('%w'))
```

```
1991-04-30
weekday: Tue
weekday as a full name: Tuesday
weekday as a decimal number: 2
```

Formatting months:

```
# Day of the month as a zero-padded decimal number.
# 01, 02, ..., 31
print(d1.strftime('%d'))

# Month as locale’s abbreviated name.
# Jan, Feb, ..., Dec (en_US);
# Jan, Feb, ..., Dez (de_DE)
print(d1.strftime('%b'))

# Month as locale’s full name.
# January, February, ..., December (en_US);
# Januar, Februar, ..., Dezember (de_DE)
print(d1.strftime('%B'))

# Month as a zero-padded decimal number.
# 01, 02, ..., 12
print(d1.strftime('%m'))
```

```
30
Apr
April
04
```

CREATING DATETIME OBJECTS FROM STRINGS

We can use strptime to create new datetime object by parsing a string containing a data and time. The arguments of strptime are the string to be parsed and a format specification.

```
from datetime import datetime
t = datetime.strptime("30 Nov 00", "%d %b %y")
print(t)
```

```
2000-11-30 00:00:00
```

```
dt = "2007-03-04T21:08:12"
datetime.strptime( dt, "%Y-%m-%dT%H:%M:%S" )
```

Output:
```
datetime.datetime(2007, 3, 4, 21, 8, 12)
```

```
dt = '12/24/1957 4:03:29 AM'
dt = datetime.strptime(dt, '%m/%d/%Y %I:%M:%S %p')
dt
```

Output:
```
datetime.datetime(1957, 12, 24, 4, 3, 29)
```

We can create an English date string on a Linux machine with the Shell command

```
LC_ALL=en_EN.utf8 date
```

```
dt = 'Wed Apr 12 20:29:53 CEST 2017'
dt = datetime.strptime(dt, '%a %b %d %H:%M:%S %Z %Y')
print(dt)
```

```
2017-04-12 20:29:53
```

Though datetime.strptime is an easy way to parse a date with a known format, it can be quote complicated and cumbersome to write every time a new specification string for new date formats.

Using the parse method from dateutil.parser:

```
from dateutil.parser import parse

parse('2011-01-03')
```

Output:
```
datetime.datetime(2011, 1, 3, 0, 0)
```

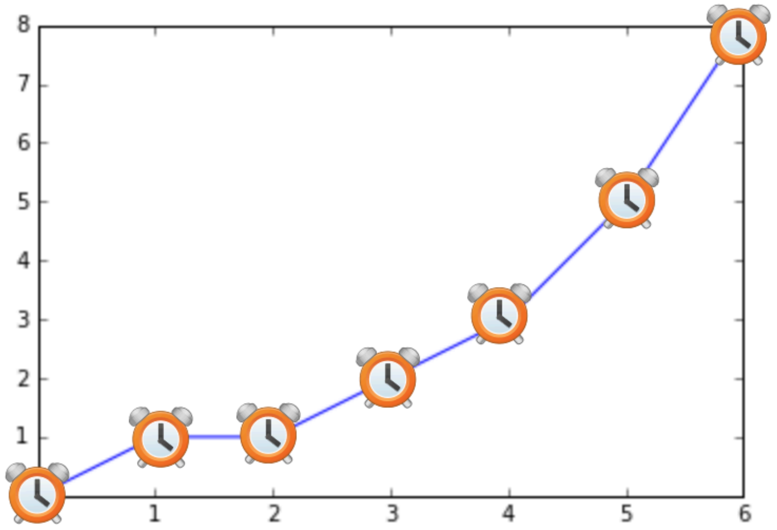
```
parse('Wed Apr 12 20:29:53 CEST 2017')
```

Output:
```
datetime.datetime(2017, 4, 12, 20, 29, 53, tzinfo=tzlocal())
```

PYTHON, PANDAS AND TIME SERIES

INTRODUCTION

Our next chapter of our Pandas Tutorial deals with time series. A time series is a series of data points, which are listed (or indexed) in time order. Usually, a time series is a sequence of values, which are equally spaced points in time. Everything which consists of measured data connected with the corresponding time can be seen as a time series. Measurements can be taken irregularly, but in most cases time series consist of fixed frequencies. This means that data is measured or taken in a regular pattern, i.e. for example every 5 milliseconds, every 10 seconds, or very hour. Often time series are plotted as line charts.



In this chapter of our tutorial on Python with Pandas, we will introduce the tools from Pandas dealing with time series. You will learn how to cope with large time series and how modify time series.

Before you continue reading it might be useful to go through our tutorial on the standard Python modules dealing with time processing, i.e. datetime, time and calendar:

TIME SERIES IN PANDAS AND PYTHON

We could define a Pandas Series, which is built with an index consisting of time stamps.

```
import numpy as np
import pandas as pd

from datetime import datetime, timedelta as delta
ndays = 10
start = datetime(2017, 3, 31)
dates = [start - delta(days=x) for x in range(0, ndays)]
```

```
values = [25, 50, 15, 67, 70, 9, 28, 30, 32, 12]

ts = pd.Series(values, index=dates)
ts
```

Output:

```
2017-03-31    25
2017-03-30    50
2017-03-29    15
2017-03-28    67
2017-03-27    70
2017-03-26     9
2017-03-25    28
2017-03-24    30
2017-03-23    32
2017-03-22    12
dtype: int64
```

Let's check the type of the newly created time series:

```
type(ts)
```

Output:

```
pandas.core.series.Series
```

What does the index of a time series look like? Let's see:

```
ts.index
```

Output:
```
DatetimeIndex(['2017-03-31', '2017-03-30', '2017-03-29', '2017-03-2
8',
               '2017-03-27', '2017-03-26', '2017-03-25', '2017-03-2
4',
               '2017-03-23', '2017-03-22'],
              dtype='datetime64[ns]', freq=None)
```

We will create now another time series:

```
values2 = [32, 54, 18, 61, 72, 19, 21, 33, 29, 17]

ts2 = pd.Series(values2, index=dates)
```

It is possible to use arithmetic operations on time series like we did with other series. We can for example add the two previously created time series:

```
ts + ts2
```

Output:
```
2017-03-31     57
2017-03-30    104
2017-03-29     33
2017-03-28    128
2017-03-27    142
2017-03-26     28
2017-03-25     49
2017-03-24     63
2017-03-23     61
2017-03-22     29
dtype: int64
```

Arithmetic mean between both Series, i.e. the values of the series:

```
(ts + ts2) / 2
```

Output:
```
2017-03-31    28.5
2017-03-30    52.0
2017-03-29    16.5
2017-03-28    64.0
2017-03-27    71.0
2017-03-26    14.0
2017-03-25    24.5
2017-03-24    31.5
2017-03-23    30.5
2017-03-22    14.5
dtype: float64
```

As with other series the indices don't have to be the same.

```
import pandas as pd

from datetime import datetime, timedelta as delta

ndays = 10

start = datetime(2017, 3, 31)
dates = [start - delta(days=x) for x in range(0, ndays)]

start2 = datetime(2017, 3, 26)
dates2 = [start2 - delta(days=x) for x in range(0, ndays)]

values = [25, 50, 15, 67, 70, 9, 28, 30, 32, 12]
values2 = [32, 54, 18, 61, 72, 19, 21, 33, 29, 17]

ts = pd.Series(values, index=dates)
ts2 = pd.Series(values2, index=dates2)

ts + ts2
```

Output:
```
2017-03-17     NaN
2017-03-18     NaN
2017-03-19     NaN
2017-03-20     NaN
2017-03-21     NaN
2017-03-22    84.0
2017-03-23    93.0
2017-03-24    48.0
2017-03-25    82.0
2017-03-26    41.0
2017-03-27     NaN
2017-03-28     NaN
2017-03-29     NaN
2017-03-30     NaN
2017-03-31     NaN
dtype: float64
```

CREATE DATE RANGES

The date_range method of the pandas module can be used to generate a DatetimeIndex:

```
import pandas as pd

index = pd.date_range('12/24/1970', '01/03/1971')
index
```

Output:
```
DatetimeIndex(['1970-12-24', '1970-12-25', '1970-12-26', '1970-12-2
7',
                '1970-12-28', '1970-12-29', '1970-12-30', '1970-12-3
1',
                '1971-01-01', '1971-01-02', '1971-01-03'],
               dtype='datetime64[ns]', freq='D')
```

We have passed a start and an end date to date_range in our previous example. It is also possible to pass only a start or an end date to the function. In this case, we have to determine the number of periods to generate by setting the keyword parameter 'periods':

```
index = pd.date_range(start='12/24/1970', periods=4)
print(index)
```

```
DatetimeIndex(['1970-12-24', '1970-12-25', '1970-12-26', '1970-12-27'], d
type='datetime64[ns]', freq='D')
```

```
index = pd.date_range(end='12/24/1970', periods=3)
print(index)
```

```
DatetimeIndex(['1970-12-22', '1970-12-23', '1970-12-24'], dtype='datetime
64[ns]', freq='D')
```

We can also create time frequencies, which consists only of business days for example by setting the keyword parameter 'freq' to the string 'B':

```
index = pd.date_range('2017-04-07', '2017-04-13', freq="B")
print(index)
```

```
DatetimeIndex(['2017-04-07', '2017-04-10', '2017-04-11', '2017-04-12',
               '2017-04-13'],
              dtype='datetime64[ns]', freq='B')
```

In the following example, we create a time frequency which contains the month ends between two dates. We can see that the year 2016 contained the 29th of February, because it was a leap year:

```
index = pd.date_range('2016-02-25', '2016-07-02', freq="M")
index
```

Output:
```
DatetimeIndex(['2016-02-29', '2016-03-31', '2016-04-30', '2016-05-3
1',
                   '2016-06-30'],
                  dtype='datetime64[ns]', freq='M')
```

Other aliases:

Alias	Description
B	business day frequency
C	custom business day frequency (experimental)
D	calendar day frequency
W	weekly frequency
M	month end frequency
BM	business month end frequency
MS	month start frequency
BMS	business month start frequency
Q	quarter end frequency
BQ	business quarter endfrequency
QS	quarter start frequency
BQS	business quarter start frequency
A	year end frequency

Alias	Description
BA	business year end frequency
AS	year start frequency
BAS	business year start frequency
H	hourly frequency
T	minutely frequency
S	secondly frequency
L	milliseonds
U	microseconds

```
index = pd.date_range('2017-02-05', '2017-04-13', freq="W-Mon")
index
```

Output:
```
DatetimeIndex(['2017-02-06', '2017-02-13', '2017-02-20', '2017-02-2
7',
               '2017-03-06', '2017-03-13', '2017-03-20', '2017-03-2
7',
               '2017-04-03', '2017-04-10'],
              dtype='datetime64[ns]', freq='W-MON')
```

In [ ]:

EXPENSES AND INCOME EXAMPLE

In this chapter of our Pandas tutorial we will deal with simple Expense and Income tables for private usage. Some say that if you want to manage your money successfully you will have to track your income and expenses. Monitoring the flow of money is important to understand your financial situation. You need to know exactly how much is coming in and going out. This article is not meant to convince you of the necessity of doing this. The main focus is rather on the possibilities offered by Python and Pandas to program the necessary tools.

We will start with a small example, suitable for private purposes and the following chapter of our tutorial continues with a more extensive example suitable for small businesses.

PRIVATE BUDGETING WITH PYTHON AND PANDAS

Let us assume that you have already a csv file containing your expenses and income over a certain period of time. This journal may look like this:

```
Date;Description;Category;Out;In
2020-06-02;Salary Frank;Income;0;4896.44
2020-06-03;supermarket;food and beverages;132.40;0
2020-06-04;Salary Laura;Income;0;4910.14
2929-06-04;GreenEnergy Corp., (electricity);utility;87.34;0
2020-06-09;water and sewage;utility;60.56;0
2020-06-10;Fitness studio, Jane;health and sports;19.00;0
2020-06-11;payment to bank;monthly redemption payment;1287.43;0
2020-06-12;LeGourmet Restaurant;restaurants and hotels;145.00;0
2020-06-13;supermarket;food and beverages;197.42;0
2020-06-13;Pizzeria da Pulcinella;restaurants and hotels;60.00;0
2020-06-26;supermarket;food and beverages;155.42;0
2020-06-27;theatre tickets;education and culture;125;0
```

```
2020-07-02;Salary Frank;Income;0;4896.44
2020-07-03;supermarket;food and beverages;147.90;0
2020-07-05;Salary Laura;Income;0;4910.14
2020-07-08;Golf Club, yearly payment;health and sports;612.18;0
2020-07-09;house insurance;insurances and taxes;167.89;0
2020-07-10;Fitness studio, Jane;health and sports;19.00;0
2020-07-10;supermarket;food and beverages;144.12;0
2020-07-11;payment to bank;monthly redemption payment;1287.43;0
2020-07-18;supermarket;food and beverages;211.24;0
2020-07-13;Pizzeria da Pulcinella;restaurants and hotels;33.00;0
2020-07-23;Cinema;education and culture;19;0
2020-07-25;supermarket;food and beverages;186.11;0
```

The above mentioned csv file is saved under the name `expenses_and_income.csv` in the folder `data`. It is easy to read it in with Pandas as we can see in our chapter Pandas Data Files:

```python
import pandas as pd

exp_inc = pd.read_csv("data1/expenses_and_income.csv", sep=";")
exp_inc
```

Output:

	Date	Description	Category	Out	In
0	2020-06-02	Salary Frank	Income	0.00	4896.44
1	2020-06-03	supermarket	food and beverages	132.40	0.00
2	2020-06-04	Salary Laura	Income	0.00	4910.14
3	2929-06-04	GreenEnergy Corp., (electricity)	utility	87.34	0.00
4	2020-06-09	water and sewage	utility	60.56	0.00
5	2020-06-10	Fitness studio, Jane	health and sports	19.00	0.00
6	2020-06-11	payment to bank	monthly redemption payment	1287.43	0.00
7	2020-06-12	LeGourmet Restaurant	restaurants and hotels	145.00	0.00
8	2020-06-13	supermarket	food and beverages	197.42	0.00
9	2020-06-13	Pizzeria da Pulcinella	restaurants and hotels	60.00	0.00
10	2020-06-26	supermarket	food and beverages	155.42	0.00

	Date	Description	Category	Out	In
11	2020-06-27	theatre tickets	education and culture	125.00	0.00
12	2020-07-02	Salary Frank	Income	0.00	4896.44
13	2020-07-03	supermarket	food and beverages	147.90	0.00
14	2020-07-05	Salary Laura	Income	0.00	4910.14
15	2020-07-08	Golf Club, yearly payment	health and sports	612.18	0.00
16	2020-07-09	house insurance	insurances and taxes	167.89	0.00
17	2020-07-10	Fitness studio, Jane	health and sports	19.00	0.00
18	2020-07-10	supermarket	food and beverages	144.12	0.00
19	2020-07-11	payment to bank	monthly redemption payment	1287.43	0.00
20	2020-07-18	supermarket	food and beverages	211.24	0.00
21	2020-07-13	Pizzeria da Pulcinella	restaurants and hotels	33.00	0.00
22	2020-07-23	Cinema	education and culture	19.00	0.00

	Date	Description	Category	Out	In
23	2020-07-25	supermarket	food and beverages	186.11	0.00

By reading the CSV file, we created a DataFrame object. What can we do with it, or in other words: what information interests Frank and Laura? Of course they are interested in the account balance. They want to know what the total income was and they want to see the total of all expenses.

The balances of their expenses and incomes can be easily calculated by applying the sum on the DataFrame `exp_inc[['Out', 'In']]`:

```
exp_inc[['Out', 'In']].sum()
```

Output:
```
Out       5097.44
In       19613.16
dtype:  float64
```

What other information do they want to gain from the data? They might be interested in seeing the expenses summed up according to the different categories. This can be done using groupby and sum:

```
category_sums = exp_inc.groupby("Category").sum()
category_sums
```

Output:

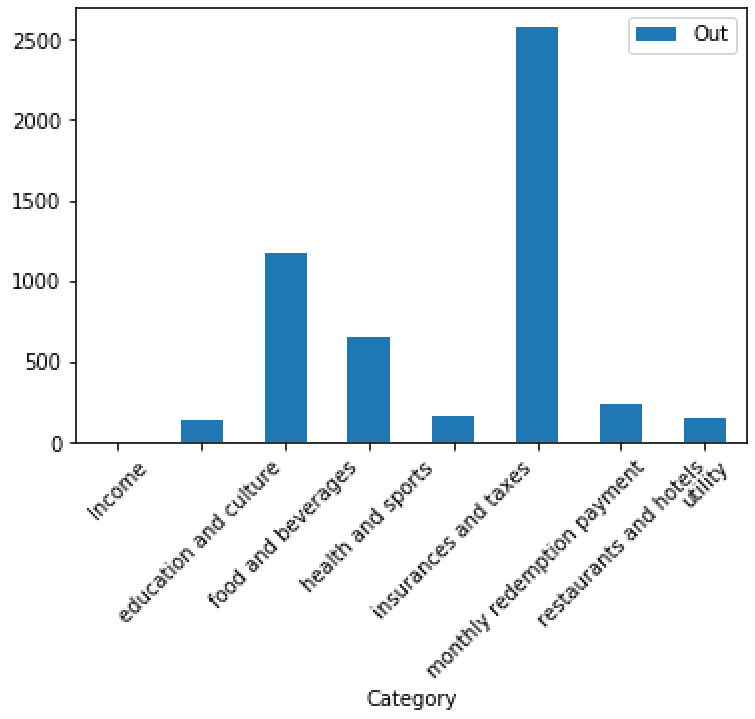
```
                                 Out          In
                    Category
                      Income     0.00   19613.16
       education and culture   144.00       0.00
         food and beverages   1174.61       0.00
           health and sports   650.18       0.00
       insurances and taxes    167.89       0.00
 monthly redemption payment   2574.86       0.00
       restaurants and hotels  238.00       0.00
                     utility   147.90       0.00
```

```
category_sums.index
```

Output:
```
Index(['Income', 'education and culture', 'food and beverages',
       'health and sports', 'insurances and taxes',
       'monthly redemption payment', 'restaurants and hotels', 'uti
lity'],
      dtype='object', name='Category')
```

```
import matplotlib.pyplot as plt

ax = category_sums.plot.bar(y="Out")
plt.xticks(rotation=45)
```

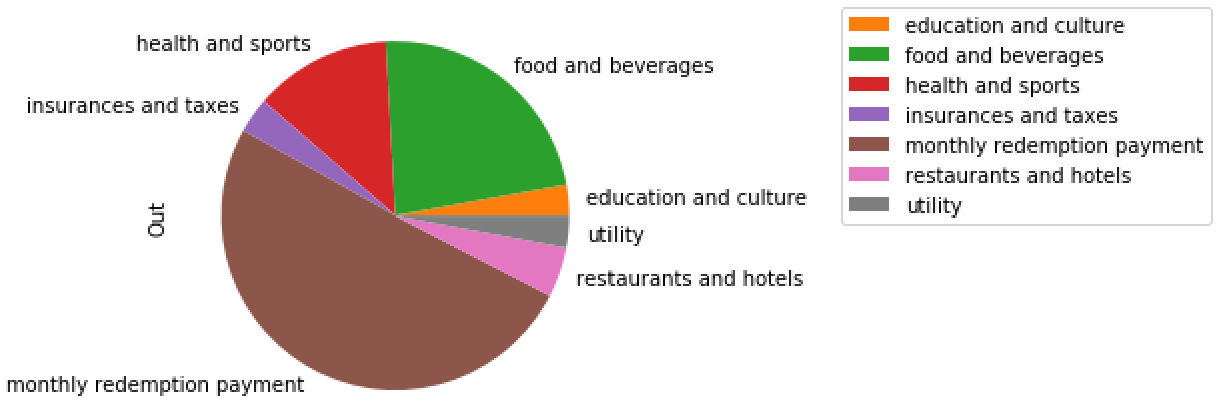
Output:
```
(array([0, 1, 2, 3, 4, 5, 6, 7]), <a list of 8 Text xticklabel obje
cts>)
```



We can also display this as a pie chart:

```
ax = category_sums.plot.pie(y="Out")
ax.legend(loc="upper left", bbox_to_anchor=(1.5, 1))
```

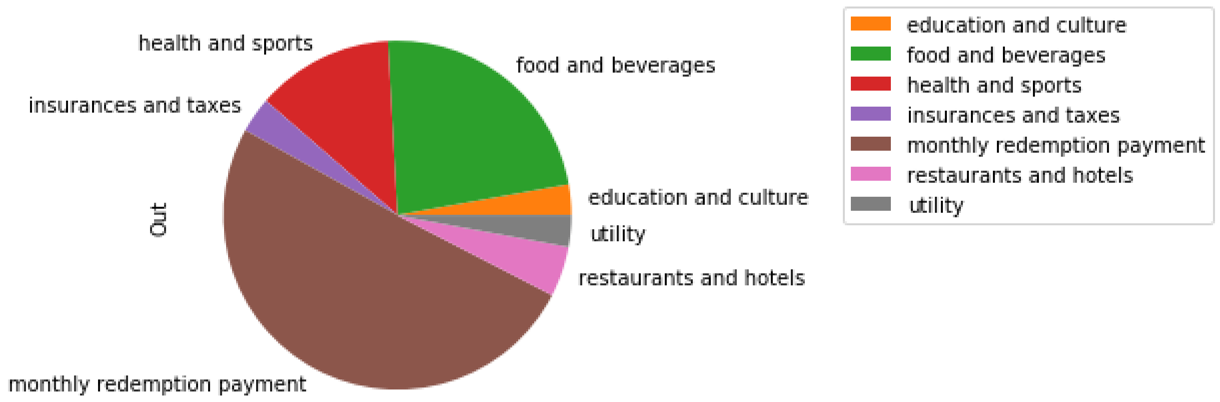
Output:
```
<matplotlib.legend.Legend at 0x7fe20d953fd0>
```



Alternatively, we can create the same pie plot with the following code:

```
ax = category_sums["Out"].plot.pie()
ax.legend(loc="upper left", bbox_to_anchor=(1.5, 1))
```

Output:
```
<matplotlib.legend.Legend at 0x7fe20d90c810>
```



If you imagine that you will have to type in all the time category names like "household goods and service" or "rent and mortgage interest", you will agree that it is very likely to have typos in your journal of expenses and income.

So it will be a good idea to use numbers (account numbers) for your categories. The following categories are available in our example.

The following categories are provided:

Category	Account Number
rent and mortgage interest	200
insurances and taxes	201
food and beverages	202
education and culture	203
transport	204
health and sports	205
household goods and services	206

	Category	**Account Number**
	clothing	207
	communications	208
	restaurants and hotels	209
	utility (heating, electricity, water, and garbage)	210
	other expenses	211
	income	400

We can implement this as a dictionary mapping categories into account numbers:

```
category2account = {'monthly redemption payment': '200',
                    'insurances and taxes': '201',
                    'food and beverages': '202',
                    'education and culture': '203',
                    'transport': '204',
                    'health and sports': '205',
                    'household goods and services': '206',
                    'clothing': '207',
                    'communications': '208',
                    'restaurants and hotels': '209',
                    'utility': '210',
                    'other expenses': '211',
                    'Income': '400'}
```

The next step is to replace our "clumsy" category names with the account numbers. The `replace` method of `DataFrame` is ideal for this purpose. We can replace all the occurrences of the category names in our DataFrame by the corresponding account names:

```
exp_inc.replace(category2account, inplace=True)
exp_inc.rename(columns={"Category": "Accounts"}, inplace=True)
exp_inc[:5]
```

Output:

	Date	Description	Accounts	Out	In
0	2020-06-02	Salary Frank	400	0.00	4896.44
1	2020-06-03	supermarket	202	132.40	0.00
2	2020-06-04	Salary Laura	400	0.00	4910.14
3	2929-06-04	GreenEnergy Corp., (electricity)	210	87.34	0.00
4	2020-06-09	water and sewage	210	60.56	0.00

We will save this `DataFrame` object now in an excel file. This excel file will have two sheets: One with the "expenses and income" journal and the other one with the mapping of account numbers to category names.
We will turn the category2account dictionary into a Series object for this purpose. The account numbers serve as the index:

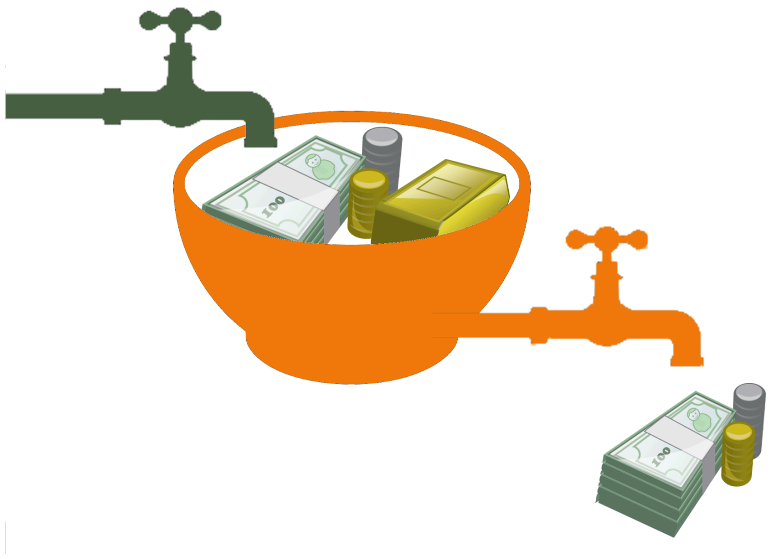
```
account_numbers = pd.Series(list(category2account.keys()), index=category
2account.values())
account_numbers.name = "Description"
account_numbers.rename("Accounts")
```

Output:
```
200      monthly redemption payment
201             insurances and taxes
202               food and beverages
203            education and culture
204                        transport
205                health and sports
206    household goods and services
207                         clothing
208                   communications
209             restaurants and hotels
210                          utility
211                   other expenses
400                           Income
Name: Accounts, dtype: object
```

```python
exp_inc.insert(1, "accounts", account_numbers)
```

```python
with pd.ExcelWriter('data1/expenses_and_income_2020.xlsx') as writer:
    account_numbers.to_excel(writer, "account numbers")
    exp_inc.to_excel(writer, "journal")
    writer.save()
```

NET INCOME METHOD EXAMPLE



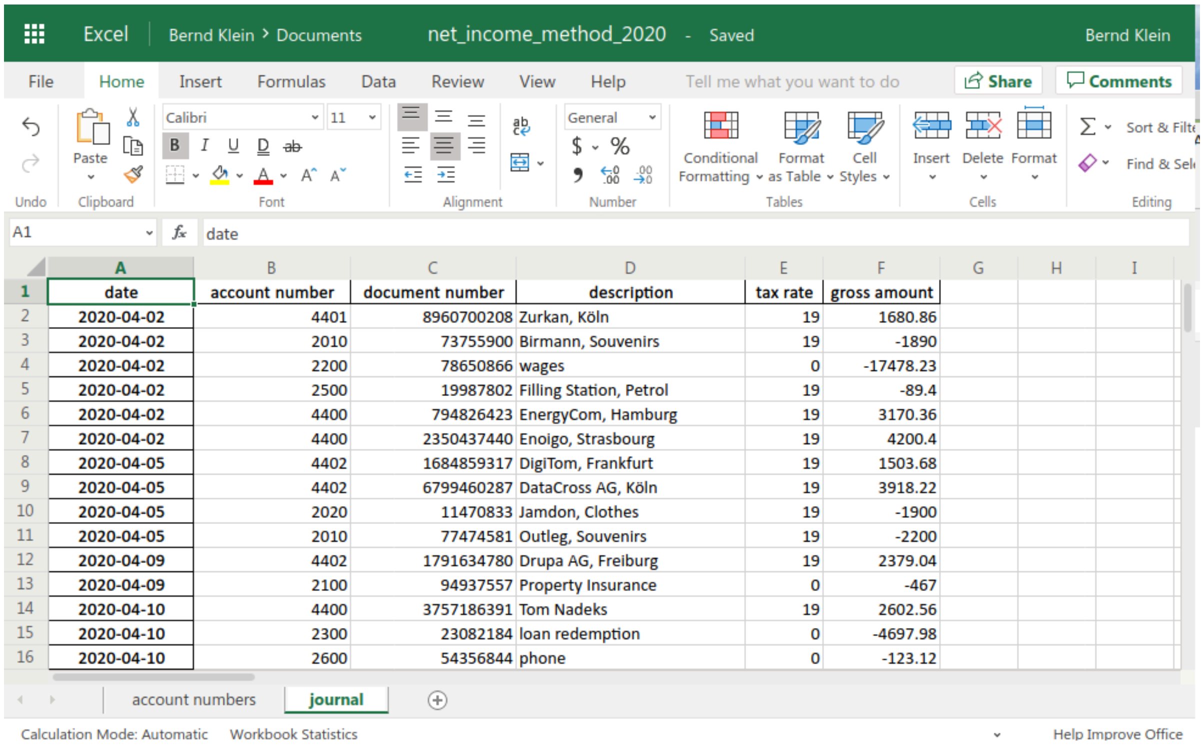
The Net Income Method (in Germany known as Einnahmeüberschussrechnung, EÜR) is a simplified profit determination method. Under German law, self-employed persons such as doctors, lawyers, architects and others have the choice of submitting annual accounts or using the simple net income method. What we present in this chapter of our Pandas tutorial will not only be interesting for small companies, especially in Germany and EU countries, but also for other countries in the world as well.

We are mainly interested in introducing the different ways in which pandas and python can solve such problems. We show what it is all about to monitor the flow of money and in this way to better understand the financial situation. The algorithms can also be used for tax purposes. But be warned, this is a very general treatment of the matter and needs to be adjusted to the actual tax situation in your country. Although it is mainly based on German law, we cannot guarantee its accuracy! So, before you actually use it, you have to make sure that it is suitable for your situation and the tax laws of your country.

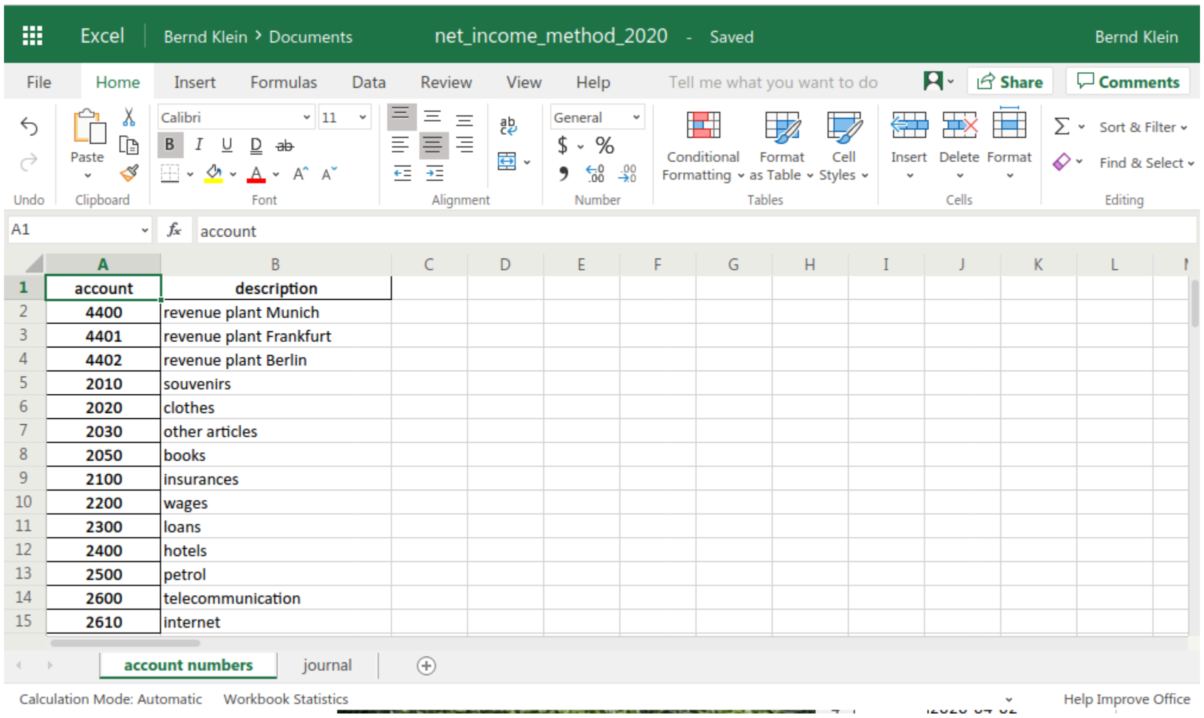
Very often a Microsoft Excel file is used for mainting a journal file of the financial transactions.
In the `data1` folder there is a file that contains an Excel file `net_income_method_2020.xlsx` with the accounting data of a fictitious company. We could have used simple texts files like csv as well.

JOURNAL FILE

This excel document contains two data sheets: One with the actual data "journal" and one with the name "account numbers", which contains the mapping from the account numbers to the description.



A	B	C	D	E	F
date	account number	document number	description	tax rate	gross amount
2020-04-02	4401	8960700208	Zurkan, Köln	19	1680.86
2020-04-02	2010	73755900	Birmann, Souvenirs	19	-1890
2020-04-02	2200	78650866	wages	0	-17478.23
2020-04-02	2500	19987802	Filling Station, Petrol	19	-89.4
2020-04-02	4400	794826423	EnergyCom, Hamburg	19	3170.36
2020-04-02	4400	2350437440	Enoigo, Strasbourg	19	4200.4
2020-04-05	4402	1684859317	DigiTom, Frankfurt	19	1503.68
2020-04-05	4402	6799460287	DataCross AG, Köln	19	3918.22
2020-04-05	2020	11470833	Jamdon, Clothes	19	-1900
2020-04-05	2010	77474581	Outleg, Souvenirs	19	-2200
2020-04-09	4402	1791634780	Drupa AG, Freiburg	19	2379.04
2020-04-09	2100	94937557	Property Insurance	0	-467
2020-04-10	4400	3757186391	Tom Nadeks	19	2602.56
2020-04-10	2300	23082184	loan redemption	0	-4697.98
2020-04-10	2600	54356844	phone	0	-123.12



A	B
account	description
4400	revenue plant Munich
4401	revenue plant Frankfurt
4402	revenue plant Berlin
2010	souvenirs
2020	clothes
2030	other articles
2050	books
2100	insurances
2200	wages
2300	loans
2400	hotels
2500	petrol
2600	telecommunication
2610	internet

We will read this excel file into two DataFrame objects:

```python
import pandas as pd

with pd.ExcelFile("data1/net_income_method_2020.xlsx") as xl:
    accounts2descr = xl.parse("account numbers",
                              index_col=0)
    journal = xl.parse("journal",
                       index_col=0,
                      )

journal.index = pd.to_datetime(journal.index)
journal.index
```

Output:
```
DatetimeIndex(['2020-04-02', '2020-04-02', '2020-04-02', '2020-04-0
2',
               '2020-04-02', '2020-04-02', '2020-04-05', '2020-04-0
5',
               '2020-04-05', '2020-04-05', '2020-04-09', '2020-04-0
9',
               '2020-04-10', '2020-04-10', '2020-04-10', '2020-04-1
0',
               '2020-04-10', '2020-04-10', '2020-04-13', '2020-04-1
3',
               '2020-04-13', '2020-04-26', '2020-04-26', '2020-04-2
6',
               '2020-04-26', '2020-04-27', '2020-05-03', '2020-05-0
3',
               '2020-05-03', '2020-05-03', '2020-05-05', '2020-05-0
5',
               '2020-05-08', '2020-05-09', '2020-05-10', '2020-05-1
1',
               '2020-05-11', '2020-05-11', '2020-05-11', '2020-05-1
1',
               '2020-05-13', '2020-05-18', '2020-05-25', '2020-05-2
5',
               '2020-06-01', '2020-06-02', '2020-06-03', '2020-06-0
3',
               '2020-06-04', '2020-06-04', '2020-06-09', '2020-06-1
0',
               '2020-06-10', '2020-06-11', '2020-06-11', '2020-06-1
1',
               '2020-06-11', '2020-06-11', '2020-06-12', '2020-06-1
3',
               '2020-06-13', '2020-06-26', '2020-06-26', '2020-06-2
7',
               '2020-07-02', '2020-07-03', '2020-07-05', '2020-07-0
5',
               '2020-07-08', '2020-07-09', '2020-07-10', '2020-07-1
0',
               '2020-07-10', '2020-07-10', '2020-07-10', '2020-07-1
0',
               '2020-07-11', '2020-07-11', '2020-07-13', '2020-07-1
8',
               '2020-07-23', '2020-07-23', '2020-07-25', '2020-07-2
5',
               '2020-07-27', '2020-07-26', '2020-07-28'],
              dtype='datetime64[ns]', name='date', freq=None)
```

The first one is the tab "account numbers" which contains the mapping from the account numbers to the description of the accounts:

```
accounts2descr
```

Output:

	description
account	
4400	revenue plant Munich
4401	revenue plant Frankfurt
4402	revenue plant Berlin
2010	souvenirs
2020	clothes
2030	other articles
2050	books
2100	insurances
2200	wages
2300	loans
2400	hotels
2500	petrol

	description
account	
2600	telecommunication
2610	internet

The second data sheet "journal" contains the actual journal entries:

```
journal[:10]
```

Output:

	account number	document number	description	tax rate	gross amount
date					
2020-04-02	4402	8983233038	Zurkan, Köln	19	4105.98
2020-04-02	2010	57550799	Birmann, Souvenirs	19	-1890.00
2020-04-02	2200	14989004	wages	0	-17478.23
2020-04-02	2500	12766279	Filling Station, Petrol	19	-89.40
2020-04-02	4400	3733462359	EnergyCom, Hamburg	19	4663.54
2020-04-02	4402	7526058231	Enoigo, Strasbourg	19	2412.82
2020-04-05	4402	1157284466	Qbooks, Frankfurt	7	2631.42
2020-04-05	4402	7009463592	Qbooks, Köln	7	3628.45
2020-04-05	2020	68433353	Jamdon, Clothes	19	-1900.00
2020-04-05	2010	53353169	Outleg, Souvenirs	19	-2200.00

There are many ways to analyze this data. We can for example sum up all the accounts:

```
account_sums = journal[["account number", "gross amount"]].groupby("accou
nt number").sum()
account_sums
```

Output:

```
                     gross amount
account number
          2010           -4090.00
          2020          -10500.80
          2030           -1350.00
          2050            -900.00
          2100            -612.00
          2200          -69912.92
          2300          -18791.92
          2400           -1597.10
          2500             -89.40
          2600            -492.48
          2610            -561.00
          4400           37771.84
```

	gross amount
account number	
4401	69610.35
4402	61593.99

ACCOUNT CHARTS

What about showing a pie chart of these sums? We encounter one problem: Pie charts cannot contain negative values. However this is not a real problem. We can split the accounts into income and expense accounts. Of course, this corresponds more to what we really want to see.

CHARTS FOR THE INCOME ACCOUNTS

We create a DataFrame with the income accounts:

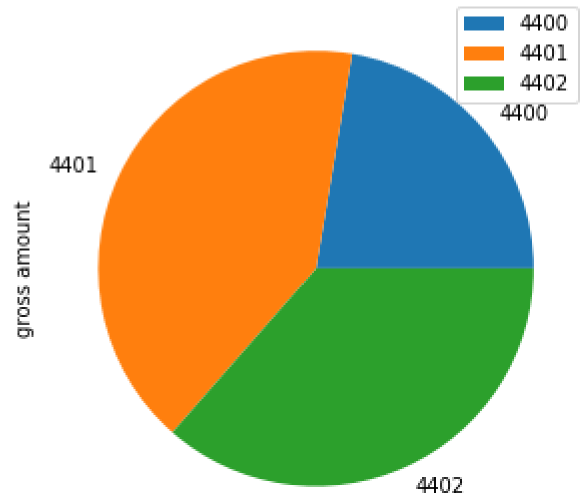
```
income_accounts = account_sums[account_sums["gross amount"] > 0]
income_accounts
```

Output:

	gross amount
account number	
4400	37771.84
4401	69610.35
4402	61593.99

We can now visualize these values in a pie chart.

```
plot = income_accounts.plot(y='gross amount', figsize=(5, 5), kind="pie")
```



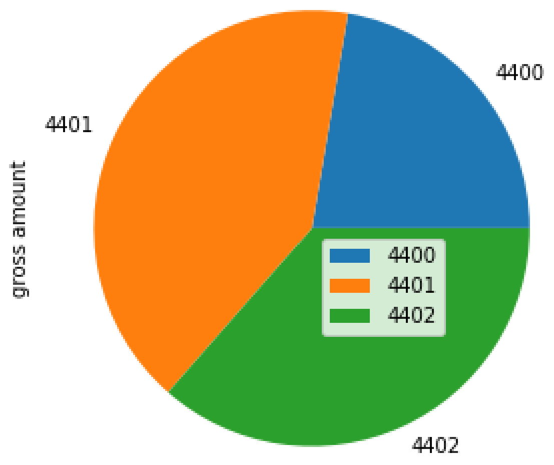
You probably don't like the position of the legend? With the parameter bbox_to_anchor we can position it on a desired position. So, if we want to, we can even move it outside of the plot. With the relative coordinate values `(0.5, 0.5)` we position the legend in the center of the plot.
The legend is a box structure. So the question is what does the postion (0.5, 0.5) mean? We can define this by using the parameter `loc` additionally:

loc value	meaning
upper left	bbox_to_anchor describes the position of the left upper corner of the legend box
upper right	correspondingly the upper right corner
lower left	the lower left corner
lower left	the lower right corner

We use this to position the legend with its left upper corner positioned in the middle of the plot:

```
plot = income_accounts.plot(y='gross amount',
                             figsize=(5, 5),
                               kind="pie")
plot.legend(bbox_to_anchor=(0.5, 0.5),
             loc="upper left")
```

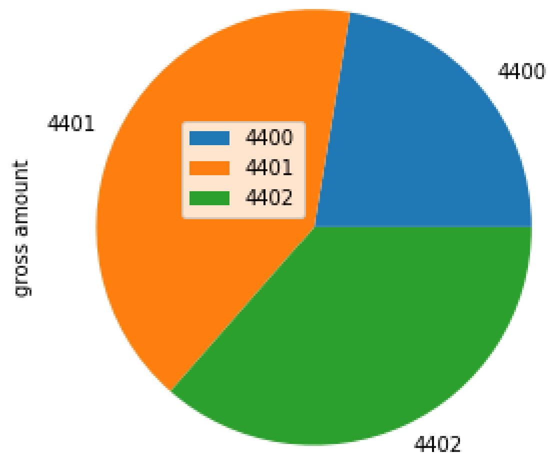
Output: <matplotlib.legend.Legend at 0x7f172ca03a90>



Now we position the lower right corner of the legend into the center of the plot:

```
plot = income_accounts.plot(y='gross amount', figsize=(5, 5), kind="pie")
plot.legend(bbox_to_anchor=(0.5, 0.5),
             loc="lower right")
```

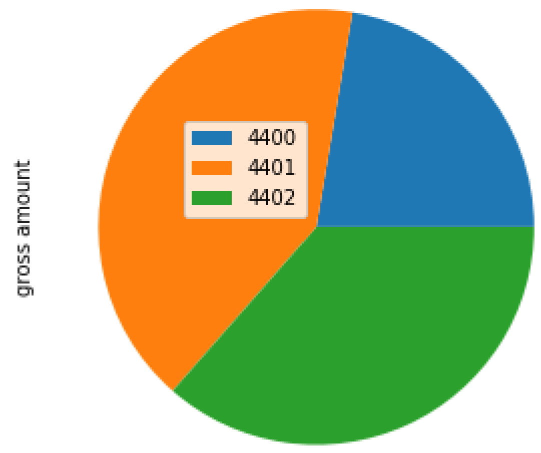
Output: `<matplotlib.legend.Legend at 0x7f172c9c5590>`



There is another thing we can improve. We see the labels 4400, 4401, and 4402 beside of each pie segment. In addition, we see them in the legend. This is ugly and redundant information. In the following we will turn the labels off, i.e. set them to an empty string, in the plot and we explicitly set them in the legend method:

```
plot = income_accounts.plot(y='gross amount',
                            figsize=(5, 5),
                            kind="pie",
                            labels=['', '', ''])
plot.legend(bbox_to_anchor=(0.5, 0.5),
            labels=income_accounts.index)
```

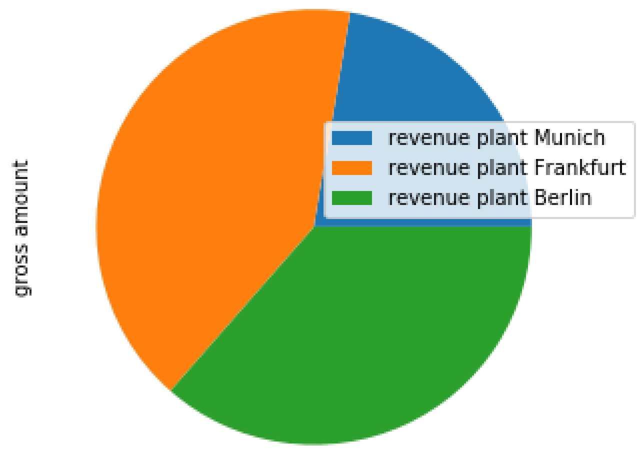
Output:
```
<matplotlib.legend.Legend at 0x7f172c956550>
```



Now, we are close to perfection. Just one more tiny thing. Some might prefer to see the actual description text rather than an account number. We will cut out this information from the DataFrame `accounts2descr` by using `loc` and the list of desired numbers `[4400, 4401, 4402]`. The result of this operation will be the argument of the `set_index` method. (Atention: `reindex` is not giving the wanted results!)

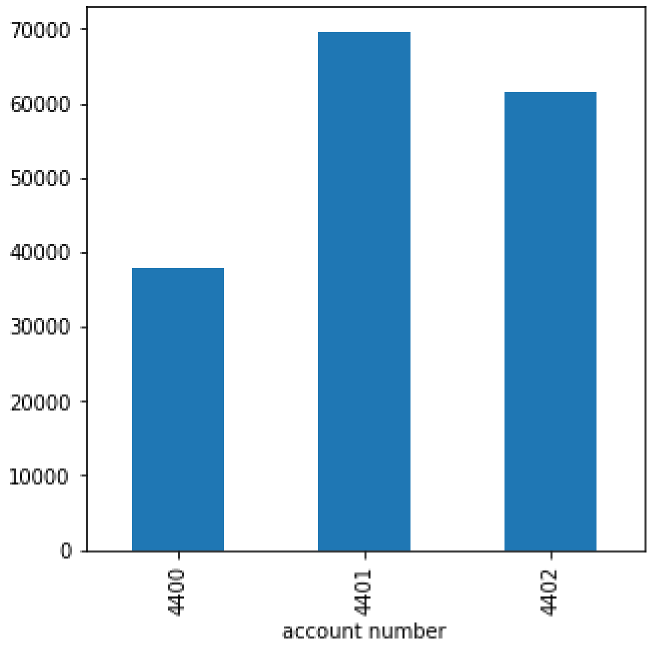
```
descriptions = accounts2descr["description"].loc[[4400, 4401, 4402]]
plot = income_accounts.plot(kind="pie",
                            y='gross amount',
                            figsize=(5, 5),
                            labels=['', '', ''])

plot.legend(bbox_to_anchor=(0.5, 0.5),
            loc="lower left",
            labels=descriptions)
```

Output: `<matplotlib.legend.Legend at 0x7f172c8ba8d0>`



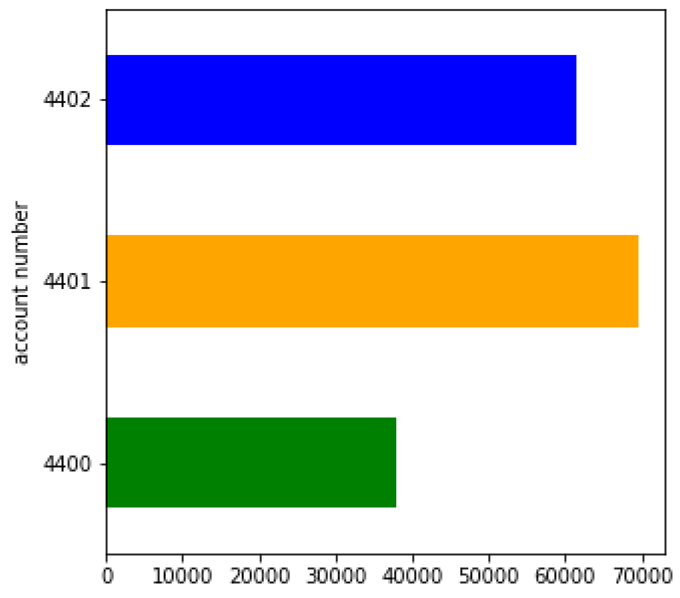
Would you prefer a bar chart? No problem, we just have to set the parameter `kind` to `bar` instead of `pie`:

```
plot = income_accounts.plot(y='gross amount',
                            figsize=(5, 5),
                            kind="bar",
                            legend=False)
```



For bar charts we have to set `kind` to `barh`. So that it doesn't get too boring, we can also color the bars by passing a list of colors to the color parameter:

```
plot = income_accounts.plot(y='gross amount',
                            figsize=(5, 5),
                            kind="barh",
                            legend=False,
                            color=['green', 'orange', 'blue'])
```



CHARTS FOR THE EXPENSES ACCOUNTS

we can do the same now with our debitors (expenses accounts):

```
expenses_accounts = account_sums[account_sums["gross amount"] < 0]
expenses_accounts
```

Output:

```
                gross amount
account number
          2010      -4090.00
          2020     -10500.80
          2030      -1350.00
          2050       -900.00
          2100       -612.00
          2200     -69912.92
          2300     -18791.92
          2400      -1597.10
          2500        -89.40
          2600       -492.48
          2610       -561.00
```

```
acc2descr_expenses = accounts2descr["description"].loc[expenses_account
s.index]
acc2descr_expenses
```

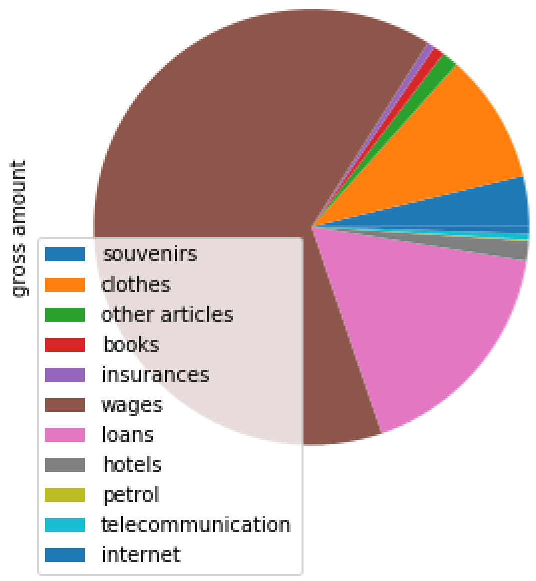
Output:

```
account number
2010             souvenirs
2020               clothes
2030        other articles
2050                 books
2100            insurances
2200                 wages
2300                 loans
2400                hotels
2500                petrol
2600     telecommunication
2610              internet
Name: description, dtype: object
```

```
expenses_accounts.set_index(acc2descr_expenses.values, inplace=True)

expenses_accounts *= -1
```

```
labels = [''] * len(expenses_accounts)
plot = expenses_accounts.plot(kind="pie",
                              y='gross amount',
                              figsize=(5, 5),
                              labels=labels)
plot.legend(bbox_to_anchor=(0.5, 0.5),
            labels=expenses_accounts.index)
```

Output:
```
<matplotlib.legend.Legend at 0x7f172868f0d0>
```



TAX SUMS

We will sum up the amount according to their tax rate.

```
journal.drop(columns=["account number"])
```

Output:

	document number	description	tax rate	gross amount
date				
2020-04-02	8983233038	Zurkan, Köln	19	4105.98
2020-04-02	57550799	Birmann, Souvenirs	19	-1890.00
2020-04-02	14989004	wages	0	-17478.23
2020-04-02	12766279	Filling Station, Petrol	19	-89.40
2020-04-02	3733462359	EnergyCom, Hamburg	19	4663.54
...	...	...	...	...
2020-07-25	5204418668	BoKoData, Bodensee, Konstanz	19	3678.38
2020-07-25	85241331	Hotel, Konstanz	7	-583.00
2020-07-27	26865618	Hotel, Franfurt	7	-450.00
2020-07-26	5892708524	Oscar Zopvar KG, Luxemburg	19	2589.80
2020-07-28	1633775505	Oscar Zopvar KG, Luxemburg	19	3578.46

```
87 rows × 4 columns
```

In the following we will define a function `tax_sums` that calculates the VAT sums according to tax rates from a journal DataFrame:

```
def tax_sums(journal_df, months=None):
    """ Returns a DataFrame with sales and tax rates -
        If a number or list is passed to 'months', only the sales
        of the corresponding months will be used.
        Example: tax_sums(df, months=[3, 6]) will only use the months
        3 (March) and 6 (June)"""
    if months:
        if isinstance(months, int):
            month_cond = journal_df.index.month == months
        elif isinstance(months, (list, tuple)):
            month_cond = journal_df.index.month.isin(months)
        positive = journal_df["gross amount"] > 0
        # sales_taxes eq. umsatzsteuer
        sales_taxes = journal_df[positive & month_cond]
        negative = journal_df["gross amount"] < 0
        # input_taxes equivalent to German Vorsteuer
        input_taxes = journal_df[negative & month_cond]
    else:
        sales_taxes = journal_df[journal_df["gross amount"] > 0]
        input_taxes = journal_df[journal_df["gross amount"] < 0]

    sales_taxes = sales_taxes[["tax rate", "gross amount"]].groupby("tax
rate").sum()
    sales_taxes.rename(columns={"gross amount": "Sales Gross"},
                       inplace=True)
    sales_taxes.index.name = 'Tax Rate'

    input_taxes = input_taxes[["tax rate", "gross amount"]].groupby("tax
rate").sum()
    input_taxes.rename(columns={"gross amount": "Expenses Gross"},
                      inplace=True)
    input_taxes.index.name = 'Tax Rate'

    taxes = pd.concat([input_taxes, sales_taxes], axis=1)
    taxes.insert(1,
                 column="Input Taxes",
                 value=(taxes["Sales Gross"] * taxes.index / 100).roun
d(2))
    taxes.insert(3,
                 column="Sales Taxes",
                      value=(taxes["Expenses Gross"] * taxes.index / 100).roun
d(2))

     return taxes.fillna(0)

tax_sums(journal)
```

Output:

	Expenses Gross	Input Taxes	Sales Gross	Sales Taxes
Tax Rate				
0	-90102.20	0.00	8334.43	-0.00
7	-3847.10	786.85	11240.71	-269.30
19	-14948.32	28386.20	149401.04	-2840.18

```
stsum_5 = tax_sums(journal, months=5)
stsum_6 = tax_sums(journal, months=6)
stsum_5
```

Output:

	Expenses Gross	Input Taxes	Sales Gross	Sales Taxes
Tax Rate				
0	-22411.53	0.00	0.00	-0.00
7	-900.00	0.00	0.00	-63.00
19	-145.00	5952.51	31328.98	-27.55

```
tax_sums(journal, months=[5, 6])
```

Output:

```
          Expenses Gross   Input Taxes   Sales Gross   Sales Taxes
Tax Rate
       0       -44812.14          0.00       6479.47         -0.00
       7         -900.00        348.66       4980.84        -63.00
      19         -268.12      12520.85      65899.23        -50.94
```

LINEAR COMBINATION

DEFINITIONS

A linear combination in mathematics is an expression constructed from a set of terms by multiplying
each term by a constant and adding the results.

Example of a linear combination:

$a \cdot x + b \cdot y$ is a linear combination of x and y with a and b constants.

Generally;

$p = \lambda_1 \cdot x_1 + \lambda_2 \cdot x_2 \ldots \lambda_n \cdot x_n$

p is the scalar product of the values $x_1, x_2 \ldots x_n$ and

$\lambda_1, \lambda_2 \ldots \lambda_n$ are called scalars.

In most applications $x_1, x_2 \ldots x_n$ are vectors and the lambdas are
integers or real numbers. (For those, who prefer it mor formally: $x_1, x_2 \ldots x_n \in V$ and V is a vector space, and

$\lambda_1, \lambda_2 \ldots \lambda_n \in K$ with K being a field)

</p>

LINEAR COMBINATIONS IN PYTHON

The vector y = (3.21, 1.77, 3.65) can be easily written as a linear combination of the unit vectors
(0,0,1), (0,1,0) and (1,0,0):

$(3.21, 1.77, 3.65) = 3.21 \cdot (1,0,0) + 1.77 \quad (0,1,0) + 3.65 \cdot (0,0,1)$

We can do the calculation with Python, using the module numpy:

```python
import numpy as np
x = np.array([[0, 0, 1],
              [0, 1, 0],
              [1, 0, 0]])
y = ([3.65, 1.55, 3.42])
scalars = np.linalg.solve(x, y)
scalars
```

Output:
```
array([3.42, 1.55, 3.65])
```

The previous example was very easy, because we could work out the result in our head. What about writing our vector y = (3.21, 1.77, 3.65) as a linear combination of the vectors (0,1,1), (1,1,0) and (1,0,1)? It looks like this in Python:

```python
import numpy as np
x = np.array([[0, 1, 1],
              [1, 1, 0],
              [1, 0, 1]])
y = ([3.65, 1.55, 3.42])
scalars = np.linalg.solve(x, y)
scalars
```

Output:
```
array([0.66, 0.89, 2.76])
```

ANOTHER EXAMPLE

Any integer between -40 and 40 can be written as a linear combination of 1, 3, 9,

27 with scalars being elements
of the set {-1, 0, 1}.

For example:

7 = 1 · 1 + (-1) · 3 + 1 · 9 + 0 · 27

We can calculate these scalars with Python. First we need a generator generating all the possible scalar
combinations. If you have problems in understanding the concept of a generator, we recommend the chapter
"Iterators and Generators" of our tutorial.

```
def factors_set():
    for i in [-1, 0, 1]:
        for j in [-1,0,1]:
            for k in [-1,0,1]:
                for l in [-1,0,1]:
                    yield (i, j, k, l)
```

We will use the memoize() technique (see chapter "Memoization and Decorators" of our tutorial) to memorize previous results:

```
def memoize(f):
    results = {}
    def helper(n):
        if n not in results:
            results[n] = f(n)
        return results[n]
    return helper
```

Finally, in our function linear_combination() we check every scalar tuple, if it can create the value n:

```
@memoize
```

```python
def linear_combination(n):
    """ returns the tuple (i,j,k,l) satisfying
        n = i*1 + j*3 + k*9 + l*27       """
    weighs = (1,3,9,27)

    for factors in factors_set():
       sum = 0
       for i in range(len(factors)):
          sum += factors[i] * weighs[i]
       if sum == n:
          return factors
```

Putting it all together results in the following script:

```python
def factors_set():
    for i in [-1, 0, 1]:
        for j in [-1, 0, 1]:
            for k in [-1, 0, 1]:
                for l in [-1, 0, 1]:
                    yield (i, j, k, l)


def memoize(f):
    results = {}
    def helper(n):
        if n not in results:
            results[n] = f(n)
        return results[n]
    return helper

@memoize
def linear_combination(n):
    """ returns the tuple (i,j,k,l) satisfying
        n = i*1 + j*3 + k*9 + l*27       """
    weighs = (1, 3, 9, 27)

    for factors in factors_set():
       sum = 0
       for i in range(len(factors)):
          sum += factors[i] * weighs[i]
       if sum == n:
          return factors

# calculate the linear combinations of the first 10 positive integers:
for i in range(1,11):
    print(linear_combination(i))
```

```
(1, 0, 0, 0)
(-1, 1, 0, 0)
(0, 1, 0, 0)
(1, 1, 0, 0)
(-1, -1, 1, 0)
(0, -1, 1, 0)
(1, -1, 1, 0)
(-1, 0, 1, 0)
(0, 0, 1, 0)
(1, 0, 1, 0)
```