Datenanalyse (PHY231) — Herbstsemester 2019

A short scientific Python repetition 17/09/2019

An important part of the exercises for this course involves programming in `python` using the scientific Python stack (numpy, scipy, pandas, etc.). We assume that you have completed the course "Informatik für Physikstudierende" (PHY114) successfully and that you are comfortable with programming in `python` at least at the level of that course.

The purpose of this short repetition is to help you refresh your memory and get you back up to speed with some basic python commands. If you have any problems going through this repetition, or solving the first exercise sheet that we hand out this week, we can recommend that you carefully work your way through the `python` tutorials at

http://www.physik.uzh.ch/local/teaching/PHY114/python/.

Before we dive into Python, a note on the coding style: as with any language, Python has (grammar) rules. While the interpreter is quite flexible (such as we humans are with incorrectly written texts) regarding certain aspects, there is a standard (as grammar rules in a normal language) called PEP8, which defines how variable names should be written, where spaces are placed and more. It is not required to read through the document, but most coding tools come with a built-in checker and the exercises here also adhere to this standard. We encourage you to follow the same style as it is an essential part of writing Python code.

1 Operators and variables

`python` has the usual arithmetic operators, `+ - * /` and `**` (power), as well as logical operators and comparisons:

`and`	and	`<`	less than	`>`	greater than
`or`	or	`<=`	less or equal	`>=`	greater or equal
`not`	not	`==`	equal	`!=`	not equal

'and' and 'or' are in the context of some mathematical libraries (such as Pandas) something different, namely '&' and '—' [1]

The result of the operation is returned and can be assigned to a variable or be used in a larger expression. In the IPython command line, if the returned value is not assigned, it is automatically printed. This is a feature of IPython and not general Python behaviour.

```
In [1]: 2 ** 5
Out[1]: 32

In [2]: 3 + 7;
```

Some common variables, such as `pi` = 3.14... or `e` = 2.71... are predefined in `numpy`. Additional variables can be defined simply by assigning them a value:

```
In [1]: radius = 3.0
```

[1]These can (only in the context of these libraries) perform logical operations `elementwise` on an array instead of just on a single value (as 'and' and 'or' do). Notice that the difference here between Python, the programming language and a scientific math library arises due to different goals.

```
In [2]: phi = 30 * np.pi / 180

In [3]: x = radius * np.cos(phi)

In [4]: x
Out[4]: 2.5981

In [5]: y = radius * np.sin(phi)

In [6]: y
Out[6]: 1.5

In [7]: np.sqrt(x ** 2 + y ** 2)
Out[7]: 3.0
```

Elementary mathematical functions are defined in `numpy` and have therefore to be accessed through this "namespace". Since we imported the module `numpy` with the abbreviation `np`, we can use the following functions by calling `np.sin`, `np.cos` etc. The following functions are a selection of the available mathematical functions.

```
sin,  cos,  tan,   arcsin,  arccos,  arctan,   exp,  log,   log10,
abs,  sqrt, round
```

The command `np.exp(x)` is equivalent to `np.e ** x` (but the first is to be preferred).

2 Arrays and matrices

The most important numpy data types for us are arrays (which also serve as matrices). While these objects store several values and *at first glance* seem similar to objects such as lists or tuples, there is a fundamental difference between the two: lists and tuples are objects of the general purpose programming language Python and are highly flexible. They can store any kind of object (such as lists of tuples of dicts...), usually not just numbers, can grow, shrink and be combined. They are used by scientists, web-developers and application developers alike. Arrays and matrices on the other hand are specialized objects ("classes", to be precise) that serve the purpose of mathematical computation. They are meant to store numbers (no lists or similar) and behave as their mathematical equivalent, hence they are mostly useful for scientific computing. Due to this specialization, operations such as the `+` operator differ significantly: `list + list` will simply combine the two lists while `array + array` will add the entries element-wise (the behaviour we know from vectors).

There are several ways in which to define an array:

```
In [1]: array1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

In [2]: array1.size
Out[2]: 8
```

Just to remind you: the 8 that was printed is a *special feature* of IPython. It could also be assigned to a new variable (as in line 1).

```
In [3]: array1.shape
Out[3]: (2, 4)

In [4]: array1[1, 2]
Out[4]: 7

In [5]: array2 = np.array([1.1, 1.2, 1.3, 1.4])

In [6]: array2
Out[6]: array([ 1.1,  1.2,  1.3,  1.4])

In [7]: array3 = np.array(range(10, 14))

In [8]: print(array3)  # equivalent to line 6
Out[8]: array([10, 11, 12, 13])

In [9]: array_angle = np.array(np.arange(0, 10, np.pi))

In [10]: array_angle
Out[10]: array([ 0.        ,  3.14159265,  6.28318531,  9.42477796])
```

An array can be composed from sub-arrays:

```
In [11]: stacked = np.vstack([array1, array_angle])
In [12]: stacked
Out[12]:
array([[ 1.        ,  2.        ,  3.        ,  4.        ],
        [ 5.        ,  6.        ,  7.        ,  8.        ],
        [ 0.        ,  3.14159265,  6.28318531,  9.42477796]])
```

Rows, columns or sub-arrays can be selected by specifying ranges of indices:

```
In [14]: stacked[0,:]
Out[14]: array([[ 1.,  2.,  3.,  4.]])

In [15]: stacked[0:2][:,1:3]
Out[15]:
array([[ 2.,  3.],
       [ 6.,  7.]])
```

Arrays can be transposed, inverted (if appropriate), and multiplied with each other:

```
In [16]: array5 = np.array([[1,2],[3,4]])

In [17]: array5.T
Out[17]:
array([[1, 3],
       [2, 4]])
```

Note here that array5 is *still the same* as before (print it!). What we see here is the *returned* value of the transpose method (called `T`), a view on the array.

```
In [18]: np.linalg.inv(array5)
Out[18]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

In [19]: array5 @ np.linalg.inv(array5)
Out[19]:
array([[  1.00000000e+00,   1.11022302e-16],
       [ -2.22044605e-16,   1.00000000e+00]])
```

The functions `multiply(...,...)` and `power(...,...)` are applied element by element. The operator '@' corresponds to a matrix multiplication, as shown above.

3 Reading data from an ASCII text file

The command `x = np.loadtxt('meine_datei.txt')` reads values from an existing text file `meine_datei.txt` and fills them into the array `x`. The array `x` has as many rows as the text file has lines and as many columns as each line of the text file has values. For this to work, all lines of the text file must contain the same number of values.

The command 'more' as used below is bash (the language used in the Terminal) and not Python, it can though be used within IPython; again a feature of the IPython console. While useful in some cases, do not mix this up: never use 'more' in a normal Python script, it won't work.

```
In [1]: more my_data.txt
 1.  2.  3.
 4.  5.  6.
 7.  8.  9.
10. 11. 12.

In [2]: x = np.loadtxt('my_data.txt')

In [3]: x
Out[3]:
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
```

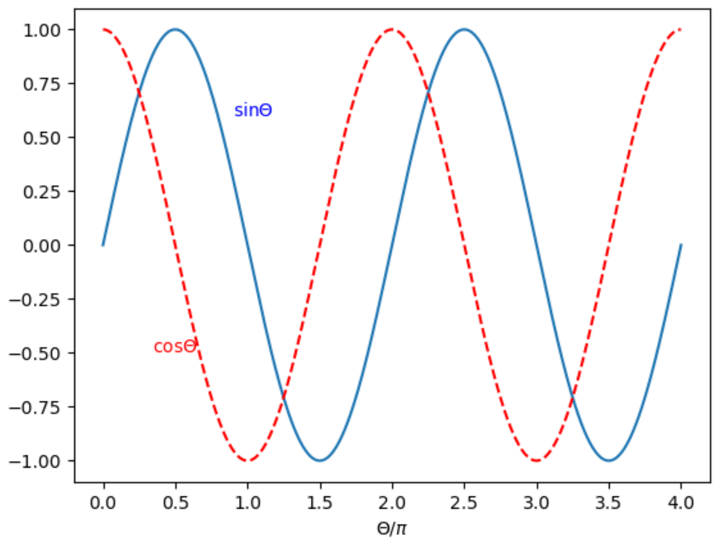
4 Plotting data or a function

The command `plt.plot(x,y)` plots the (one-dimensional) array `y` as a function of the (one-dimensional) array `x`. For this to work, the two arrays `x` and `y` must have the same number of elements (as for a vector). The type (`'-'`, `'--'`, `':'`, `'+'`, `'.'`, `'*'`) and colour (`'k'`, `'g'`, `'r'`, `'b'`) of the line connecting the points can be specified as optional arguments. The commands `xlabel` and `ylabel` can be used to add axis labels. The plot range can be defined with the command `plt.axis([xmin, xmax, ymin, ymax])`. Without this command, the plot range is set automatically.

The generated figure is displayed in a separate window.

(In the following lines, as seen before, IPython will maybe print some return value. That's fine for us, look at it out of interest, but it is not displayed here since it is not relevant for the example)

```
In [1]: x = np.linspace(0, 4, 400)

In [2]: y = np.sin(x * np.pi)

In [3]: plt.plot(x, y)

In [4]: plt.plot(x, np.cos(x * np.pi), 'r--')

In [5]: plt.xlabel(r'$\Theta/\pi$')

In [6]: plt.text(0.9, 0.6, r'$\sin\Theta$', color='b')

In [7]: plt.text(0.34, -0.5, r'$\cos\Theta$', color='r')
```



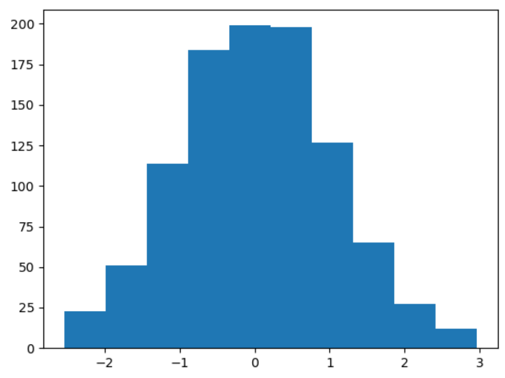
5 Histograms

The command `plt.hist(x)` fills the elements of a (one-dimensional) array `x` into a one-dimensional histogram **and** displays this histogram in a separate window[2]. If no further parameters are given, the histogram is generated with ten bins of equal width. The bin boundaries are set automatically such that the range of the histogram covers the full range of values of the array `x`. The command `plt.hist(x, nbins)`, where `nbins` is a natural number, generates a histogram with `nbins` bins of equal width. The bin boundaries and the range of the histogram are again set automatically. The command `plt.hist(x, xbins)`, where `xbins` is a one-dimensional array with `n` real numbers, generates a histogram with `n-1` bins and bin boundaries that are defined by the elements of the array `xbins`.

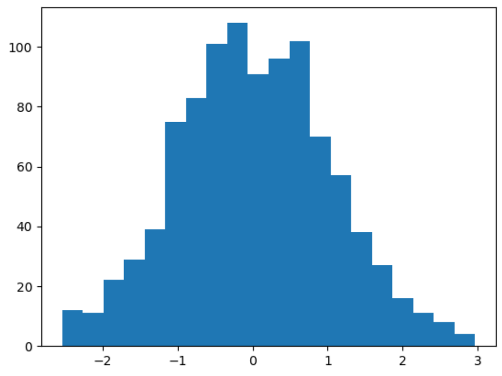
In the following examples, the command `np.random.normal(size=1000)` is used to generate 1000 Gaussian distributed random numbers and these are then filled into histograms:

```
In [1]: x = np.random.normal(size=1000)(1000)

In [2]: plt.hist(x)
```

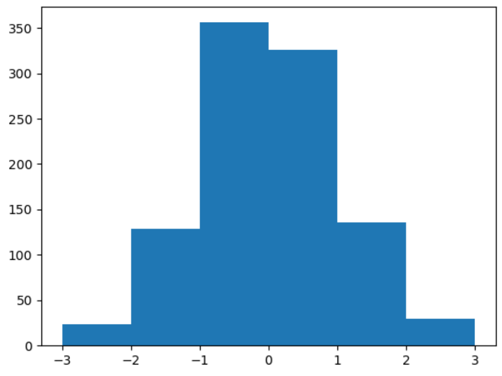
[2]In case you ever want to create a histogram *without* plotting, there is a function in numpy, `np.histogram`, that can be used


200
175
150
125
100
75
50
25
0
−2
−1
0
1
2
3

```
In [3]: plt.hist(x, 20)
```



```
In [4]: xbins = np.linspace(-3, 3, 7)

In [5]: plt.hist(x, xbins)
```



The command `y = plt.hist(x, xbins)` creates a 3-tuple `y`, that consists of two arrays, `y[0]` and `y[1]`, and a list, `y[2]`. The array `y[0]` contains the number of entries in each bin of the histogram. The array `y[1]` contains the bin boundaries. The list `y[2]` contains the graphics objects that are needed for displaying the histogram (line type, colours, etc.) and is usually not of interest for us. Note, that for a histogram with `n` bins, the array `y[0]` has `n` elements, while the array `y[1]` has `n+1` elements:

```
In [6]: y = plt.hist(x, xbins)

In [7]: y[0]
Out[7]:
array([   5.,   23.,   40.,   72.,  135.,  207.,  192.,  150.,   93.,
         49.,   23.,    7.])

In [8]: len(y[0])
Out[8]: 12
```

```
In [9]: y[1]
Out[9]:
array([-3. , -2.5, -2. , -1.5, -1. , -0.5,  0. ,  0.5,  1. ,  1.5,  2. ,
        2.5,  3. ])

In [10]: len(y[1])
Out[10]: 13
```

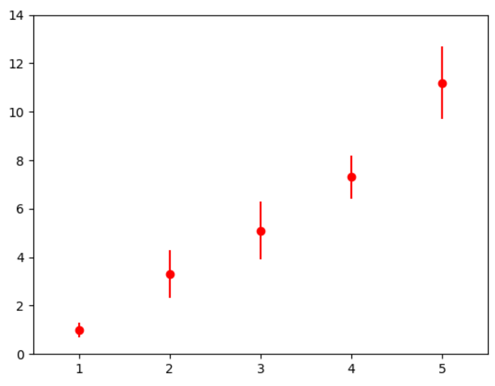
6 Plotting data with error bars

The command `plt.errorbars(x, y, dy,fmt='ro')`, with one-dimensional arrays `x`, `y` and `dy`, plots the data points (x,y) with vertical error bars of length +/− `dy`. The parameter `fmt=''` defines the marker type (`'o'`,`'x'`,`'.'`) used to plot the data points (x,y) and the colour (`'k'`, `'g'`, `'r'`, `'b'`) used to plot the marker and the error bars. The plot range can be defined with the command `plt.axis([xmin, xmax, ymin, ymax])`. Without this command, the plot range is set automatically.

The generated figure is displayed in a separate window.

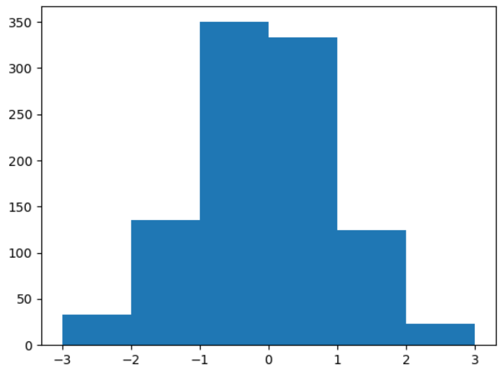
In the following example, the data to be plotted are read from an existing text file `my_measurements.txt`, where the first column contains the x values, the second column contains the y values and the third column contains the uncertainties on the y values:

```
In [1]: more my_measurements.txt
 1.   1.0  0.3
 2.   3.3  1.0
 3.   5.1  1.2
 4.   7.3  0.9
 5. 11.2  1.5

In [2]: xydy = np.loadtxt('my_measurements.txt')

In [3]: plt.errorbar(xydy[:, 0], xydy[:, 1], xydy[:, 2], fmt='ro')

In [4]: plt.axis([0.5, 5.5, 0, 14])
```



In the following example, data are filled into a histogram and this is then displayed with error bars. The uncertainty on the number of entries in each bin of the histogram is calculated as the square root of the number of entries. We will discuss later in the course of the lecture why the errors are (in this case) like this. The markers and error bars should be displayed in the centre of each bin and these have to be calculated from the bin boundaries of the histogram:

```
In [1]: x = np.random.normal(size=1000)

In [2]: xbins = np.linspace(-3, 3, 7)

In [3]: y = plt.hist(x, xbins)

In [4]: binmitten = (y[1][1:] + y[1][:-1]) / 2.

In [5]: errorbar(binmitten,y[0],sqrt(y[0]),fmt='ro')
```



7 Help

To get help in iPython, simply type in the command followed by a question mark. For example,

```
In [1]: np.loadtxt?
```

gives rather detailed information on the definition, parameters and options for the command `nploadtxt`. Help in IPython is generally good and useful and we encourage you to make extensive use of it.