Introduction to Python

python™

From the very basics to little advanced
Part 2

Gabor Cseh – Lectures on Modern Scientific Computing (16. Nov. 2016)

WHAT NOW?

- Using Python for numerical calculations.
- Getting to know the basics of the numpy module.
- Exploring different visualizing (plotting options).
- Getting to know the matplotlib module.
- A hint of ODE solving with the scipy module.
- A hint of compiled Python code.
- Interfacing with C++ for a little bit.

WHAT IS NUMPY?

- Numerical Python.
- A fast library for numerical calculations.
- Support large, multidimensional arrays and matrices.
- And mathematical functions to operate on these arrays.
- Uses many tricks and clever solutions.

MAIN PARTS OF NUMPY

- Random – random number generation package
- Linalg – basic linear algebra package
- FFT – Fast Fourier Transform toolkit
- Polynomial – polynomial tools
- F2py – Fortran to Python Interfacd Generator (see at the end of the presentation)
- Doc – Built-in documentation about the features (easiest to use with Ipython)
- Lib – Basic common functions used by several packages
- Distutils – An improved version of distutils (a Python module for packaging)

THE VERY BASICS

- The main object of numpy is the homogeneous n-dimensional array (called ndarray).
- It is a table of elements.
- All the same type (different from Python lists!)
- Indexed with a tuple of positive integers (but negative indices can be used too).
- Dimensions are called axes.
- Number of dimensions are called rank (1-dim: rank 1, 2-dims: rank 2 etc.).

THE VERY BASICS

shell> pip install numpy

shell> python (or python3 in Linux)

>>> import numpy as np

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

NDARRAY FUNCTIONS

```
>>> a = np.array([[1,2,3],[4,5,6]])
array([[1, 2, 3],
       [4, 5, 6]])
```

Array creation.

```
>>> a.ndim
2
```

The number of dimensions (axes) Also called as *rank*.

```
>>> len(a)
2 → only the elements of the 1st dim
```

The standard Python length function gives only the elements of the 1st axis.

```
>>> a.shape
(2,3)
```

Dimensions of the array. (E.g. number of rows and cols at a 2-dimensional matrix.

NDARRAY FUNCTIONS

```
>>> a.size
6
```

Number of elements. (Product of the numbers in the shape.)

```
>>> a.dtype
dtype(‚int32’)
```

Type of the elements. (Many other choices than standard Python.)

```
>>> a.itemsize
4
```

The size of each elements in bytes.

```
>>> a.data
<memory at 0x03D4AD50>
```

The buffer containing the actual elements. We don’t use this normally, we use indexing facilities.

```
>>> type(a)
<class 'numpy.ndarray'>
```

The standard Python type function. Does not know the type of the elements!

EXAMPLES

```
>>> import numpy as np

>>> np.array(1,2,3,4)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in
<module>
    np.array(1,2,3,4)
ValueError: only 2 non-keyword
arguments accepted
```

WRONG!

RIGHT!

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

>>> np.array([x*x for x in range(1,10,2)])
array([ 1,  9, 25, 49, 81])

>>> np.arange(15).reshape(3, 5)
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

EXAMPLES

```
>>> a.dtype
dtype('int32')

>>> a[2, 3] = 13.4

>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

No change! (No dynamic casting here.)

```
>>> b = np.array([[1,2,3.4],[4,5,6]]
array([[ 1. ,  2. ,  3.4],
       [ 4. ,  5. ,  6. ]])

>>> b.dtype
dtype('float64')
```

However, at creation, always the higher type wins.

```
>>> c = np.array([[1,2],[3,4]],dtype=‚complex’)

>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

ARRAY CREATION

Creates an array with the desired shape containing only zeros (the base type is float64).

```
>>> np.zeros([3,4])
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

```
>>> np.empty([2,3])
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Creates an array with memory garbage. Faster than zeros or ones, but *has* to be filled with meaningful values.

Creates an array filled with ones (we can change the type if we want).

```
>>> np.ones([2,3,4], dtype=‚int16’)
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]]], dtype=int16)
```

ARRAY CREATION

```
>>> np.arange(1, 10, 2)
array([1, 3, 5, 7, 9])

>>> np.arange(0, 2, 0.3)
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])

>>> np.linspace(0, 2, 7)
array([ 0.,  0.33333333,  0.66666667,
        1.,  1.33333333, 1.66666667,  2.])

>>> x = np.linspace(0, 2*np.pi, 100)

>>> np.sin(x)
```

Similar to Python built-in range()

BUT! It accepts float arguments!

With arange, the resulting number of arguments are not predictable (due to finite floating point precision), so, many times, linspace is recommended.

From 0 to 2, 7 numbers requested

Useful if you want many points between two values

PRINTING

```
>>> print(np.arange(100000))
[    0     1     2 ..., 99997 99998 99999]

>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...,   97   98   99]
 [ 100  101  102 ...,  197  198  199]
 [ 200  201  202 ...,  297  298  299]
 ...,
 [9700 9701 9702 ..., 9797 9798 9799]
 [9800 9801 9802 ..., 9897 9898 9899]
 [9900 9901 9902 ..., 9997 9998 9999]]
```

By default, numpy does not let you print huge arrays on the screen, it will hide parts of it.

PRINTING

```
>>> np.set_printoptions(threshold=,nan’)
```

This can be set with the set_printoptions (amongst many others).

```
>>> np.get_printoptions()
{'edgeitems': 3, 'infstr': 'inf', 'threshold': 1000,
'formatter': None, 'linewidth': 75, 'nanstr':
'nan', 'suppress': False, 'precision': 8}
```

Edgeitems: items to show, when hide.
Infstr: string that represents infinite values.
Threshold: when to hide.
Formatter: special formatting at print.
Linewidth: printing line width.
Nanstr: string that represents nan values.
Suppress: suppress small values.
Precision: number of digits to show.

BASIC OPERATIONS

By default, all operators are elementwise!

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

>>> b = np.array(a[::-1])

>>> a-b
array([-3, -1,  1,  3])

>>> b**2
array([16,  9,  4,  1])

>>> 10*np.sin(a)
array([ 8.41470985,  9.09297427,
1.41120008, -7.56802495])
```

```
>>> a<2
array([ True, False, False, False],
dtype=bool)

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

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

>>> a*b
array([[ 1,  4],
       [ 9, 16]])

>>> np.dot(a,b)
array([[ 7, 10],
       [15, 22]])
```

TYPE CONVERSIONS

```
>>> a = np.ones([2,3],dtype=int)

>>> b = np.random.random([2,3])  # float64

>>> b += a
array([[ 3.35939571,  3.18736589, 3.75636479],
       [ 3.40100369,  3.60934271, 3.38729103]])

>>> a *= 3
array([[3, 3, 3],
       [3, 3, 3]])
```

```
>>> a *= 3.
Traceback (most recent call last):
  File "<pyshell#97>", line 1, in <module>
    a *= 3.
TypeError: Cannot cast ufunc multiply output from dtype('float64') to dtype('int32') with casting rule 'same_kind,
```

TYPE CONVERSIONS

```
>>> a += b
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    a += b
TypeError: Cannot cast ufunc add output from
dtype('float64') to dtype('int32') with casting
rule 'same_kind,

>>> a += b.astype(int)
array([[6, 6, 6],
       [6, 6, 6]])
```

The type conversion works automatically towards the more complex types, but requires the user's attention towards the simpler types. This is called upcasting.

SIMPLE OPERATIONS ON ARRAYS

```
>>> a = np.random.random([2,3])
array([[ 0.15797706,  0.57102237,
0.86529344],
       [ 0.8574179 ,  0.93736791,
0.05694467]])

>>> a.sum()
3.446023524367316

>>> a.min()
0.05694466828773343

>>> a.max()
0.9373679098546249 3

>>> a.mean()
0.5743372540612198

>>> a.cumsum()  # cumulative sum
array([ 0.15797706,  0.72899943,
1.59429288,
                2.45171077,  3.38907868,
3.44602335])
```

SIMPLE OPERATIONS ON ARRAYS

```
>>> a.diagonal()
array([ 0.15797706,  0.93736791])

>>> a.astype(‚complex’).imag
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

>>> a.astype(‚complex’).real
array([[ 0.15797706,  0.57102237,  0.86529344],
       [ 0.8574179 ,  0.93736791, 0.05694467]])
```

```
>>> a.round(decimals=0)
array([[ 0.,  1.,  1.],
       [ 1.,  1.,  0.]])

>>> a.transpose()
array([[ 0.15797706,  0.8574179 ],
       [ 0.57102237,  0.93736791],
       [ 0.86529344,  0.05694467]])
```

AXIS PARAMETER

```
>>> b = arange(12).reshape(3,4)
 array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])

>>> b.sum(axis=0)  # sum of each column
array([12, 15, 18, 21])

>>> b.min(axis=1)  # min of each row
array([0, 4, 8])

>>> b.cumsum(axis=1)  # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Using the axis parameter, the basic operations can be called on just the rows or the columns (in a 2-dimensional case, but the parameter is universal).

UNIVERSAL FUNCTIONS

```
>>> a = np.arange(3)
>>> np.exp(a)
array([ 2.,  3.,  4.])
>>> import math
>>> math.exp(a)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in
<module>
    math.exp(a)
TypeError: only length-1 arrays can be
converted to Python scalars
```

Numpy has a handful of mathematical functions, which can be called for any array with the proper type. These will be applied elementwise (not like the similar functions in Python's math module, which can only be applied for one element.

The (not exhaustive list) of these functions is: all, any, argmax, argmin, argsort, average, bincount, ceil, conjugate, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inv, max, mean, median, min, nonzero, round, sort, std, sum, trace, transpose, where

INDEXING

- Numpy arrays are very flexible in terms of indexing. Slices can be applied here and there are several advanced indexing options to look at.
- 1-dimensional arrays indexed much like Python lists:

```
>>> a = np.arange(10)**3
array([  0,   1,   8,  27,  64, 125,
216, 343, 512, 729])


>>> a[2]
8
```

```
>>> a[2:5]
array([ 8, 27, 64])

>>> a[:6:2] = -1000
array([-1000,     1, -1000,    27, -1000,
125,   216,   343,   512,   729])
```

INDEXING

```
>>> a[ : :-1]  # reversed a
array([  729,   512,   343,   216,   125, -
1000,    27, -1000,     1, -1000])
```

```
>>> for i in a:
...         print i**(1/3.)

Warning (from warnings module):
  File "__main__", line 2
RuntimeWarning: invalid value
encountered in power
 nan 1.0 nan 3.0 nan 5.0 6.0 7.0 8.0 9.0
```

INDEXING (MULTIDIMENSIONAL)

```
>>> def f(x,y):
...         return 10*x+y

>>> a = fromfunction(f,[5,4],dtype=int)
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])

>>> a[2,3]
23
```

With fromfunction, one can use a user-defined function to fill up an array (the first index is the row number, the second is the column).

```
>>> a[0:5, 1]  # each row in the second column
array([ 1, 11, 21, 31, 41])

>>> a[ : 1]  # equivalent to the previous example
array([ 1, 11, 21, 31, 41])

>>> a[1:3, : ]  # each column in the 3nd and 3rd row
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

INDEXING (MULTIDIMENSIONAL)

```
>>> a[-1]  # the last row = a[-1,:]
array([40, 41, 42, 43])

>>> a[0:2,0:2]  # subarray
array([[ 0,  1],
       [10, 11]])

>>> a[0:4:2,0:4:2]  # another subarray
array([[ 0,  2],
       [20, 22]])
```

```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```

DOTS

If at a multidimensional array there would be more „:” after eachother, it can be replaced with three dots.

```
>>> c =  np.array( [[[  0,  1,  2],
[ 10, 12, 13]], [[100,101,102],
[110,112,113]]] ) # a 3D array

>>> c.shape
(2, 2, 3)
```

```
>>> c[1,...]  # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])

>>> c[...,2]  # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

ITERATION

Iteration is row-wise:

```
>>> a = np.arange(12).reshape(4,3)

>>> for row in a:
        print(row)

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

Elementwise iteration:

```
>>> for i in a.flat:
        print(i)

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

INDEXING WITH ARRAYS

Additionally to the Python list indexing rules (integers and slicing), numpy arrays can be indexed with arrays of integers and booleans.

An array of indices.

```
>>> a = arange(12)**2

>>> i = array([1,1,3,8,5 ])

>>> a[i]
array([ 1,  1,  9, 64, 25])
```

The elements of a at the positions i.

A bidimensional array of indices.

```
>>> j = array([[ 3, 4], [ 9, 7 ]])

>>> a[j]
array([[ 9, 16],
       [81, 49]])
```

The same shape as j.

INDEXING WITH ARRAYS

```
>>> a = arange(12).reshape(3,4)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> i = array( [ [0,1],[1,2] ] )  #for the 1st dim

>>> j = array( [ [2,1],[3,3] ] )  #for the 2nd dim

>>> a[i,j]  # i and j must have equal shape
array([[ 2,  5],[ 7, 11]])
```

The same as:

```
>>> l = [i,j]
>>> a[l]
array([[ 2,  5],
       [ 7, 11]])
```

```
>>> a[i,2]
array([[ 2,  6],[ 6, 10]])

>>> a[:,j]
array([[[ 2,  1],
        [ 3,  3]],
       [[ 6,  5],
        [ 7,  7]],
       [[10,  9],
        [11, 11]]])
```

INDEXING WITH BOOLEANS

Indexing an array with integer indices specify what elements to pick. Indexing with boolean values we choose what elements we want and what not.

```
>>> a = arange(12).reshape(3,4)

>>> b = a > 4

>>> b  # b is a boolean with a's shape
array([[False, False, False, False],
       [False, True, True, True],
       [True, True, True, True]], dtype=bool)
```

```
>>> a[b]  # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])

>>> a[b] = 0  # All elements of 'a' higher than
4 become 0

>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

INDEXING WITH BOOLEANS

```
>>> a = arange(12).reshape(3,4)

>>> b1 = array([False,True,True])  # 1st dim

>>> b2 = array([True,False,True,False]) # 2nd dim

>>> a[b1,:] # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
>>> a[b1]  # same thing
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> a[:,b2]  # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])

>>> a[b1,b2]  # a weird thing to do
array([ 4, 10])
```

SHAPE MANIPULATION

```
>>> a =
np.floor(10*np.random.random((3,4)))
array([[ 7.,  5.,  9.,  3.],
       [ 7.,  2.,  7.,  8.],
       [ 6.,  8.,  3.,  2.]])

>>> a.shape
(3, 4)
```

```
>>> a.ravel() # flatten the array
array([ 7.,  5.,  9.,  3.,  7.,  2.,  7.,  8.,  6.,  8.,
3.,  2.])

>>> a.shape = (6, 2)  # only this change the
shape!

>>> a.transpose()
array([[ 7.,  9.,  7.,  7.,  6.,  3.],
          [ 5.,  3.,  2.,  8.,  8.,  2.]])
```

SHAPE MANIPULATION

```
>>> a
array([[ 7.,  5.],
       [ 9.,  3.],
       [ 7.,  2.],
       [ 7.,  8.],
       [ 6.,  8.],
       [ 3.,  2.]])

>>> a.resize((2,6))  # this
also changes the shape!
```

```
>>> a
array([[ 7.,  5.,  9.,  3.,  7.,  2.],
       [ 7.,  8.,  6.,  8.,  3.,  2.]])

>>> a.reshape(3,-1)  # this case, the missing
info (-1) will be calculated
array([[ 7.,  5.,  9.,  3.],
       [ 7.,  2.,  7.,  8.],
       [ 6.,  8.,  3.,  2.]])
```

VIEWS AND SHALLOW COPYING

```
>>> a = arange(12)

>>> b = a # no new object is created

>>> b is a # a and b are two names for
the same object
True

>>> b.shape = 3,4    # changes the
shape of a

>>> a.shape
(3, 4)

>>> c = a.view()

>>> c is a
False

>>> c.base is a # c is a view of the
data owned by a
True
```

VIEWS AND SHALLOW COPYING

```
>>> c.flags.owndata
False

>>> c.shape = 2,6 # a's shape not change

>>> a.shape
(3, 4)

>>> c[0,4] = 1234  # a's data changes
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

```
>>> s = a[:,1:3]

>>> s[:] = 10 # s[:] is a view of s. Note the difference between s=10 and s[:]=10

>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

DEEP COPYING

```
>>> d = a.copy()  # a new array object with new data is created

>>> d is a
False

>>> d.base is a  # d doesn't share anything with a
False

>>> d[0,0] = 9999

>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

A BIT OF LINEAR ALGEBRA

```
>>> import numpy as np

>>> from numpy.linalg import linalg

>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
[[ 1.  2.]
 [ 3.  4.]]

>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])

>>> linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])

>>> np.dot(a,linalg.inv(a))
array([[  1.00000000e+00,   1.11022302e-16],
       [  0.00000000e+00,   1.00000000e+00]]

>>> u = np.eye(2) # unit 2x2 matrix; "eye"
represents "I„
array([[ 1.,  0.],
       [ 0.,  1.]])
```

A BIT OF LINEAR ALGEBRA

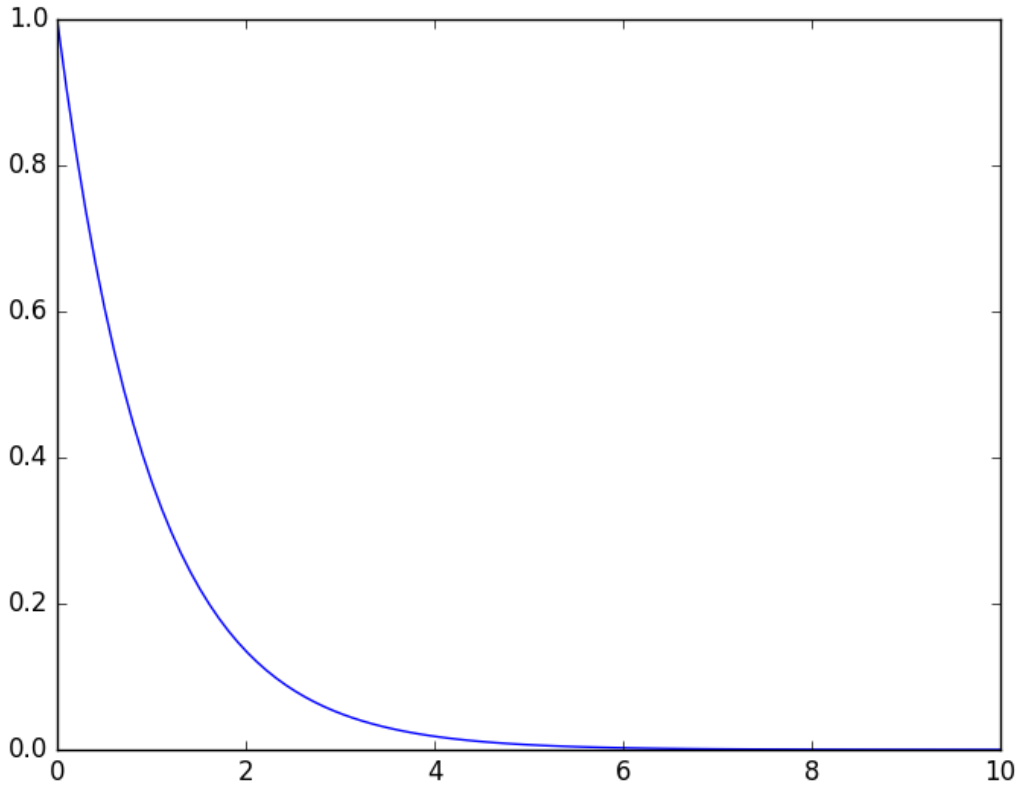
```
>>> j = np.array([[0.0, -1.0], [1.0, 0.0]])

>>> np.dot (j, j) # matrix product
array([[-1.,  0.],
       [ 0., -1.]])

>>> np.trace(u)  # trace
2.0

>>> y = np.array([[5.], [7.]])

>>> linalg.solve(a, y)
array([[-3.],
       [ 4.]])

>>> np.dot(a, linalg.solve(a,y))
array([[ 5.],
       [ 7.]])

>>> linalg.eig(j)
(array([ 0.+1.j,  0.-1.j]),
array([[ 0.70710678+0.j,  0.7071067+0.j],
       [ 0.00-0.707106j,  0.00+0.707106j]]))
```

VISUALIZING DATA

- We will use Matplotlib here.
- Widely used module in the Python community.
- It is a good way to visualize data.
- Object-oriented plotting library → easy to integrate into GUIs.
- Publication-quality figures can also be easily created with it.
- Together with Numpy, it can be used in a common namespace called pylab, an easy-to-learn environment for MATLAB programmers.
- Pylab is a viable alternative to MATLAB as a (teaching) tool for numerical mathematics and signal processing.

PLOTTING – THE FIRST STEPS

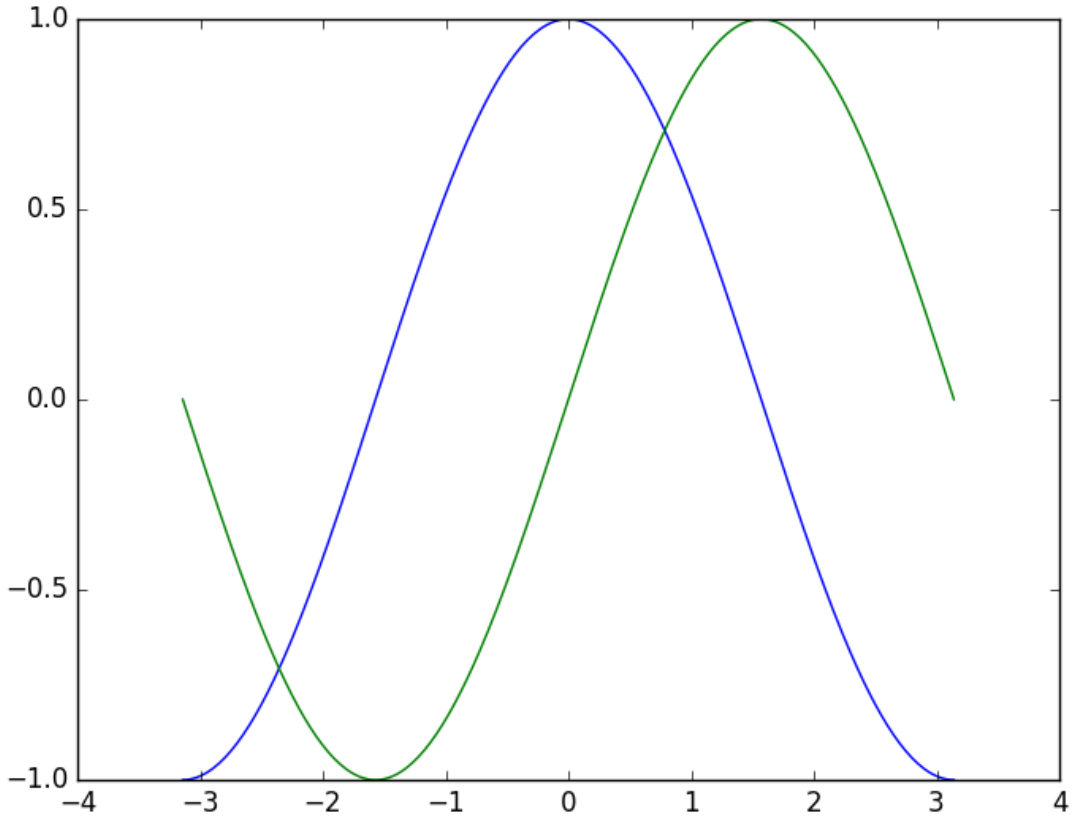
```
>>> import matplotlib.pyplot as plt

>>> import numpy as np

>>> a = np.linspace(0,10,100)

>>> b = np.exp(-a)

>>> plt.plot(a,b)

>>> plt.show()
```



PLOTTING – TWO LINES

```
>>> a = np.linspace(-np.pi, np.pi, 256, endpoint=True)

>>> c,s = np.cos(a), np.sin(a)

>>> plt.plot(a,c)
>>> plt.plot(a,s)

>>> plt.show()
```

We used many default settings now (which are very good in most cases). But how can we set them manually?



PLOTTING – TWO LINES

```
>>> plt.figure(figsize=(8,6), dpi=100)

>>> plt.subplot(111)

>>> a = np.linspace(-np.pi, np.pi,
256,endpoint=True)

>>> c,s = np.cos(a), np.sin(a)

>>> plt.plot(a, c, color="blue",
linewidth=1.0, linestyle="-")

>>> plt.plot(a, s, color="green",
linewidth=1.0, linestyle="-")
```

Create a new figure of size 8x6 inches, using 100 dots per inch

Create a new subplot from a grid of 1x1

Plot cosine using blue color with a continuous line of width 1 (pixels)

Plot sine using green color with a continuous line of width 1 (pixels)

PLOTTING – TWO LINES

```
>>> plt.xlim(-3.0,3.0)
```
Set x limits

```
>>> plt.xticks(np.linspace(-3,3,9,endpoint=True))
```
Set x ticks

```
>>> plt.ylim(-1.2,1.2)
```
Set y limits

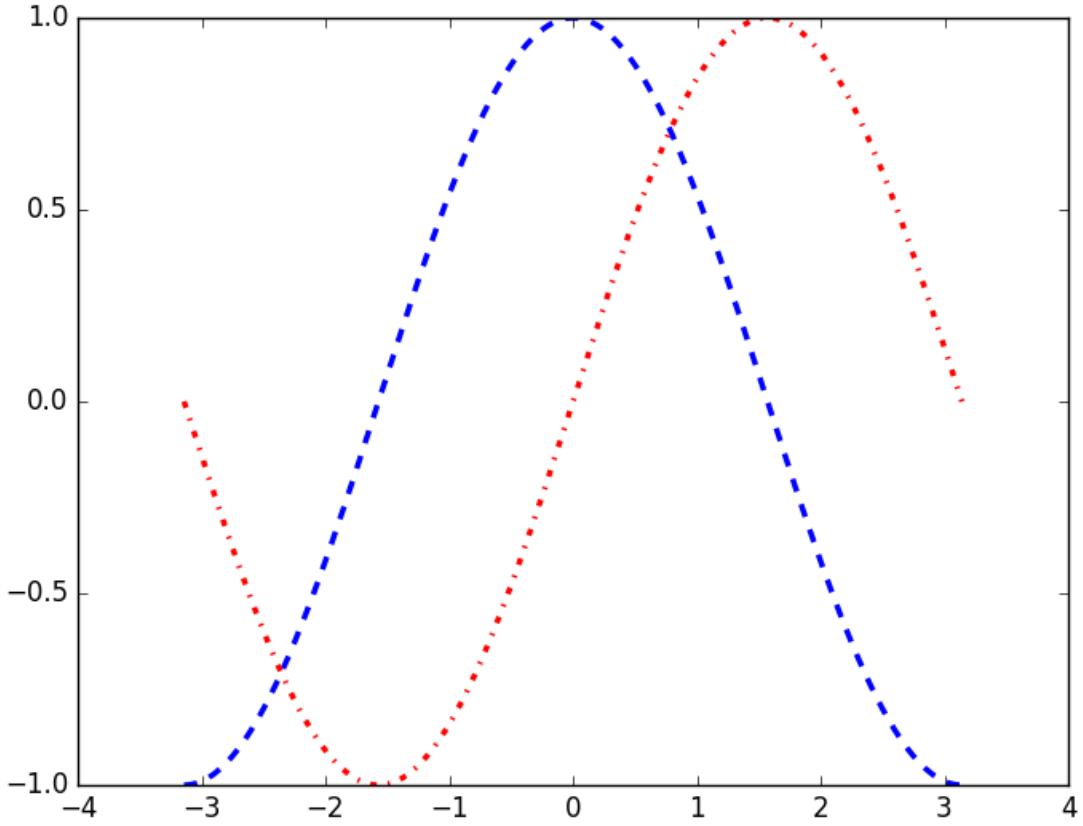
```
>>> plt.yticks(np.linspace(-1.2, 1.2, 12,
endpoint=True))
```
Set y ticks

```
#plt.savefig("../figures/exercice_2.png",dpi=72)
```
Save figure

```
>>> plt.show()
```
Show result on screen

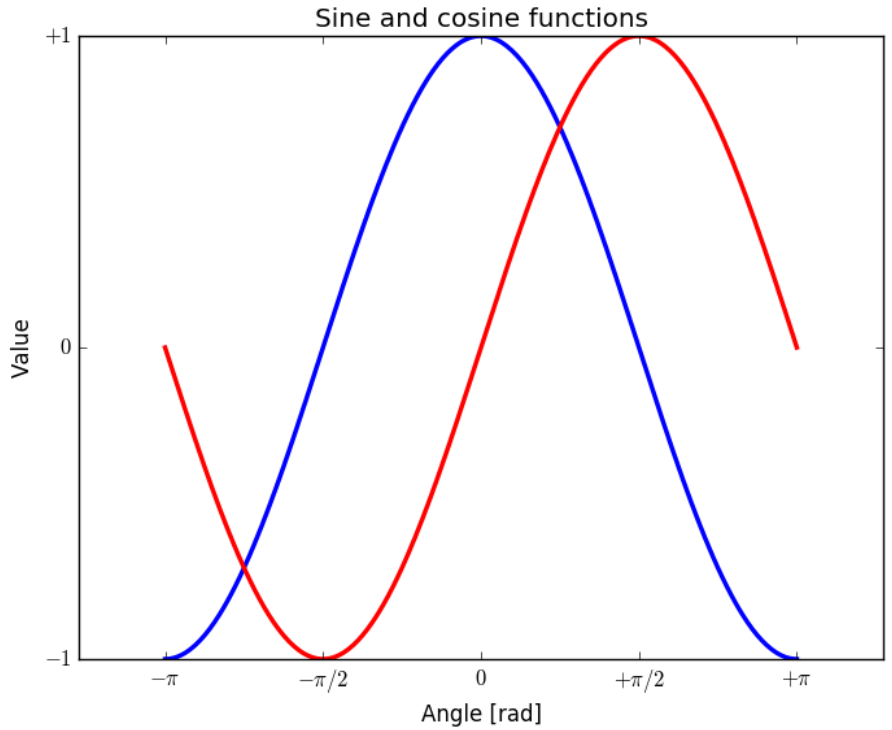
COLOR, LINEWIDTH AND LINE STYLE

```
>>> plt.figure(figsize=(10,6), dpi=80)

>>> plt.plot(a, c, color="blue",
linewidth=2.5, linestyle=„dashed")

>>> plt.plot(a, s, color="red",
linewidth=2.5, linestyle=„dashdot")

>>> plt.show()
```



TITLE, LABELS AND TICK LABELS

```
>>> plt.plot(a, c, color="blue", linewidth=2.5)
>>> plt.plot(a, s, color="red",  linewidth=2.5)
>>> plt.xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
  [r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
>>> plt.yticks([-1, 0, +1],
      [r'$-1$', r'$0$', r'$+1$'])
>>> plt.title(„Sine and cosine functions”)
>>> plt.xlabel(„Angle [rad]”)
>>> plt.ylabel(„Value”)
>>> plt.show()
```



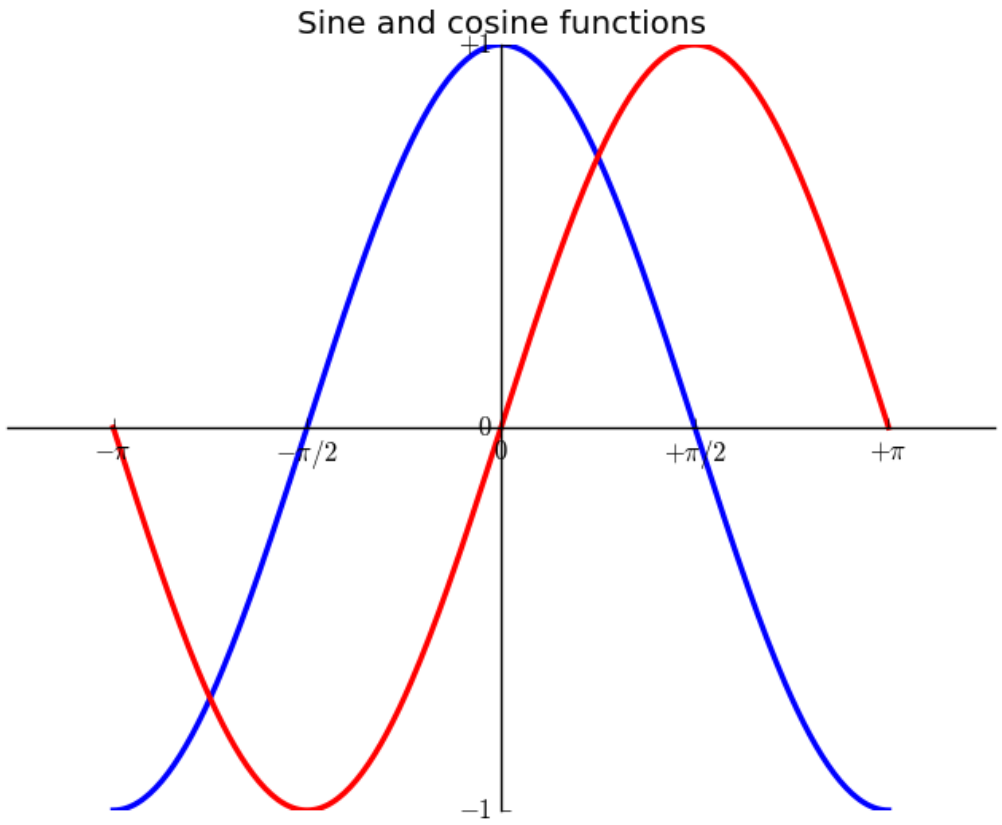
Raw string, escape sequences are not parsed!

MOVING THE SPINES

- Spines are the lines connecting the axis tick marks.
- They can be freely moved, however, the default position is at the boundaries of the plot.
- Let’s move them!

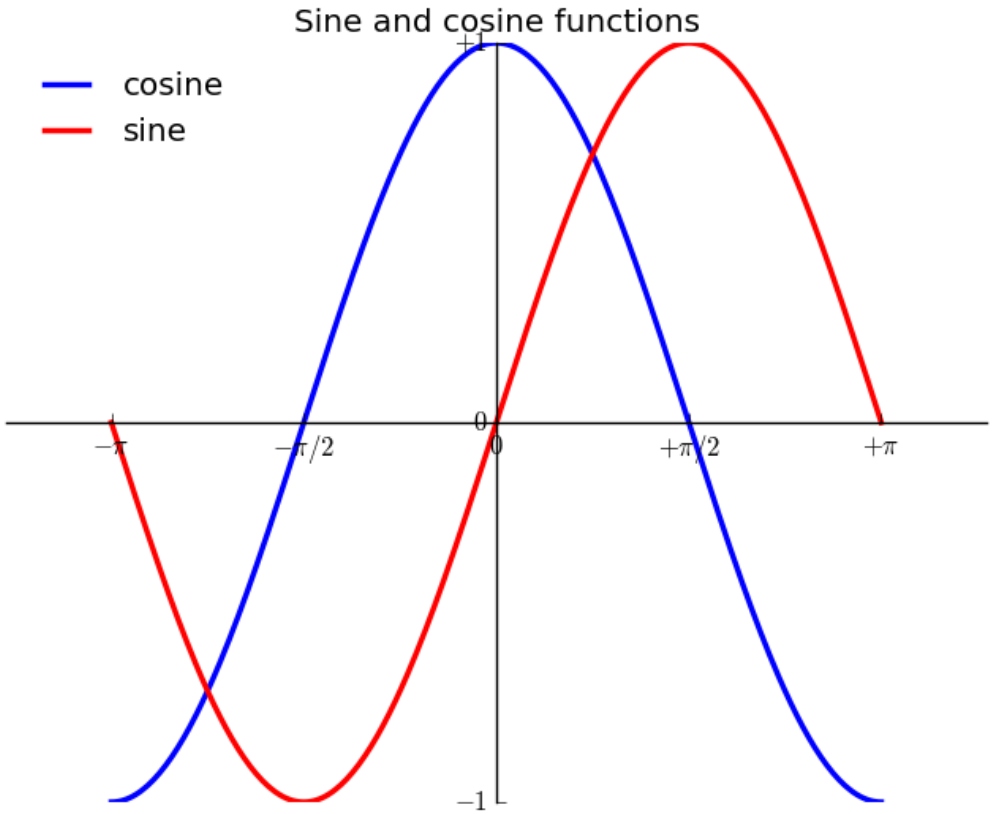
MOVING THE SPINES

```
>>> ax = plt.gca()
>>> ax.spines['right'].set_color('none')
>>> ax.spines['top'].set_color('none')
>>> ax.xaxis.set_ticks_position('bottom')
>>> ax.spines['bottom'].set_position(('data',0))
>>> ax.yaxis.set_ticks_position('left')
>>> ax.spines['left'].set_position(('data',0))
>>> plt.show()
```



ADDING LEGEND

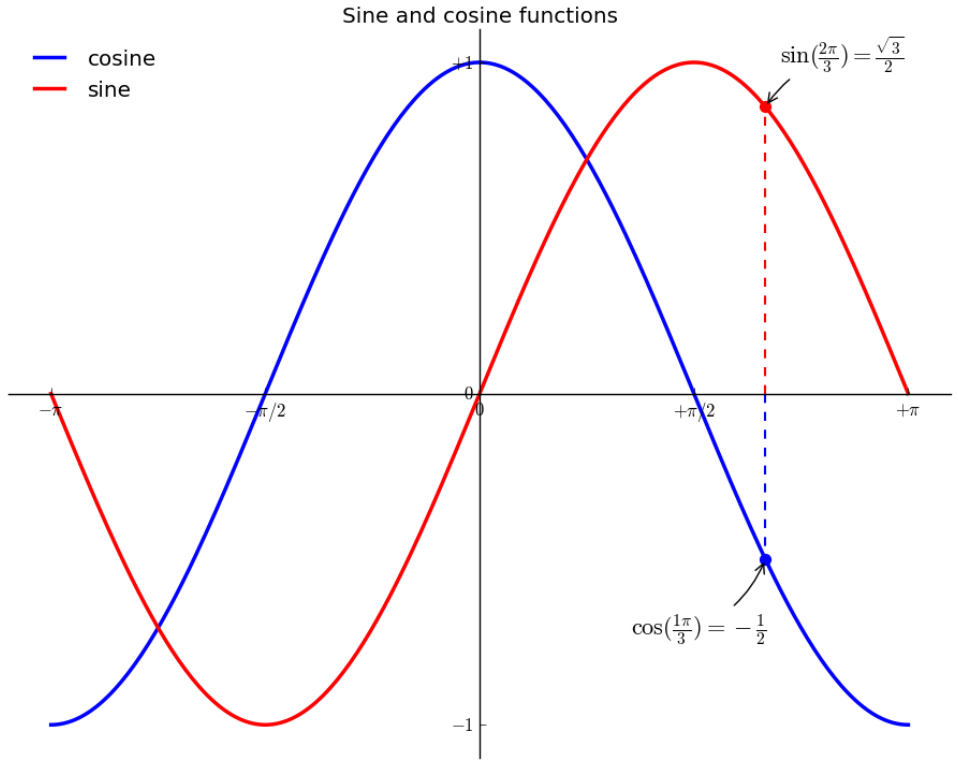
```
[…]
>>> plt.plot(a, c, color="blue",
linewidth=2.5, linestyle="-",
label="cosine")
>>> plt.plot(a, s, color="red",
linewidth=2.5, linestyle="-",
label="sine")
>>> plt.legend(loc='upper left',
frameon=False)
>>> plt.show()
```



ANNOTATING POINTS

```
>>> t = 2*np.pi/3
>>> plt.plot([t,t],[0,np.cos(t)], color ='blue', linewidth=1.5, linestyle="--")
>>> plt.scatter([t,],[np.cos(t),], 50, color ='blue')
>>> plt.annotate(r'$\cos(\frac{2\pi}{3})=-\frac{1}{2}$',
        xy=(t, np.cos(t)), xycoords='data',
        xytext=(-90, -50), textcoords='offset points', fontsize=16,
        arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
>>> plt.plot([t,t],[0,np.sin(t)], color ='red', linewidth=1.5, linestyle="--")
>>> plt.scatter([t,],[np.sin(t),], 50, color ='red')
>>> plt.annotate(r'$\sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$',
        xy=(t, np.sin(t)), xycoords='data',
        xytext=(+10, +30), textcoords='offset points', fontsize=16,
        arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
```

ANNOTATING POINTS



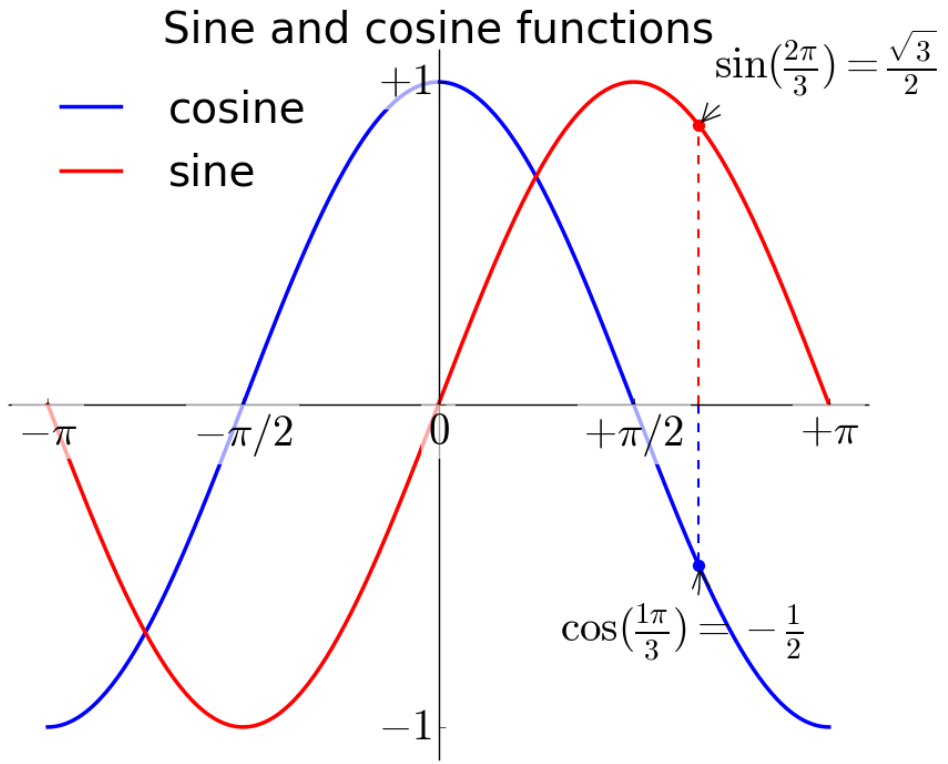
Look at this plot!
It's nice.

BUT not THAT nice…

We can make a little better…

ANNOTATING POINTS

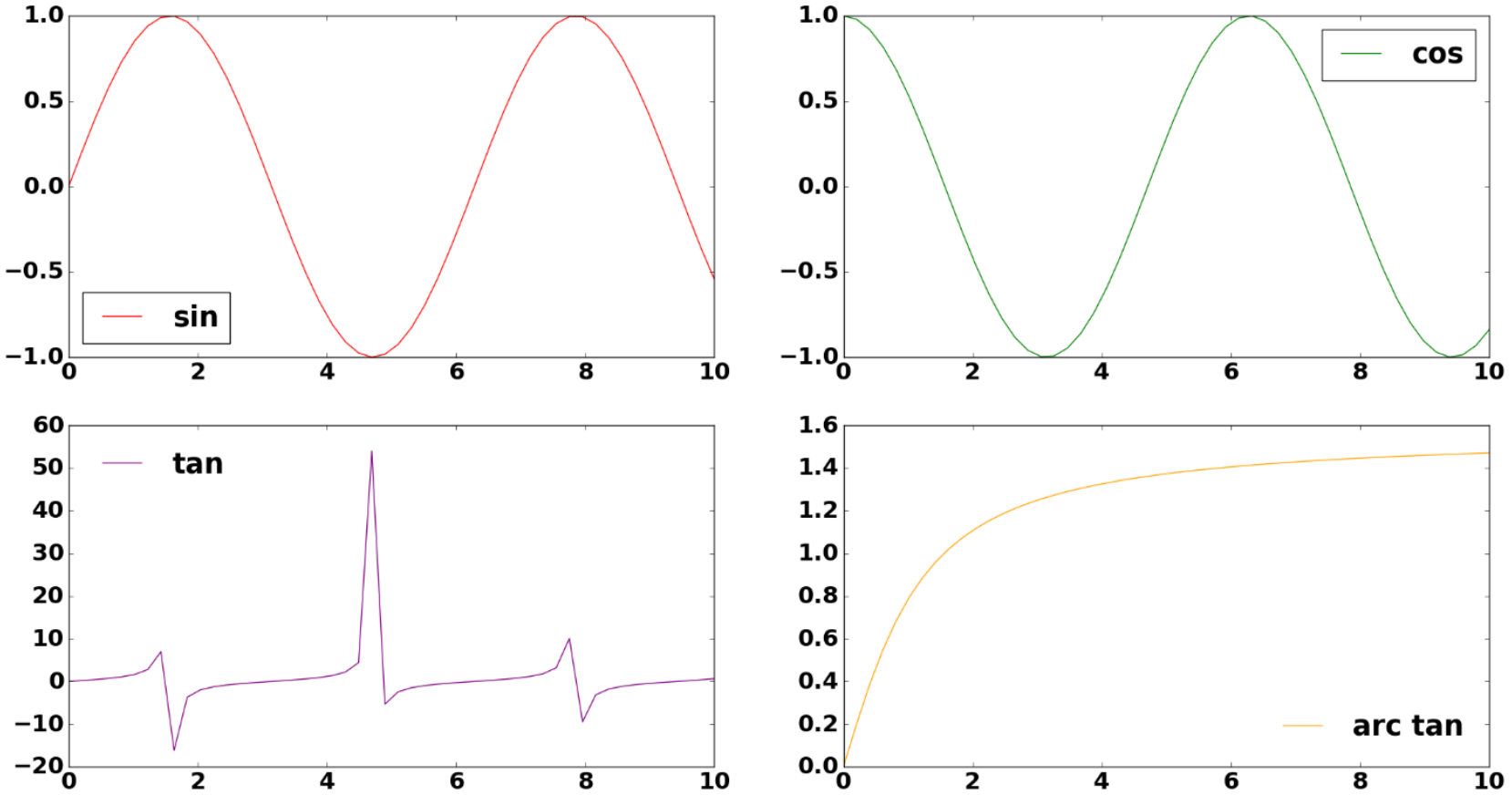
```
[…]
>>> axes = ax.get_xticklabels() +
ax.get_yticklabels()
>>> for label in axes:
        label.set_fontsize(16)
        label.set_bbox(dict(facecolor='white',
edgecolor='None', alpha=0.65))
[…]
```



SUBPLOTS AND FONT STYLE

```
>>> import matplotlib
>>> font = {'family': 'normal',
        'weight': 'bold',
        'size': 22}
>>> matplotlib.rc('font', **font)
>>> plt.figure()
>>> plt.subplot(221)
>>> plt.plot(x, np.sin(x), label="sin",
color="red")
>>> plt.legend(loc='lower left',
frameon=True)
>>> plt.subplot(222)
>>> plt.plot(x, np.cos(x), label="cos",
color="green")
>>> plt.legend(loc='upper right',
frameon=True)
>>> plt.subplot(223)
>>> plt.plot(x, np.tan(x), label="tan",
color="purple")
>>> plt.legend(loc='upper left',
frameon=False)
>>> plt.subplot(224)
>>> plt.plot(x, np.arctan(x), label="arc tan",
color="orange")
>>> plt.legend(loc='lower right',
frameon=False)
>>> plt.show()
```

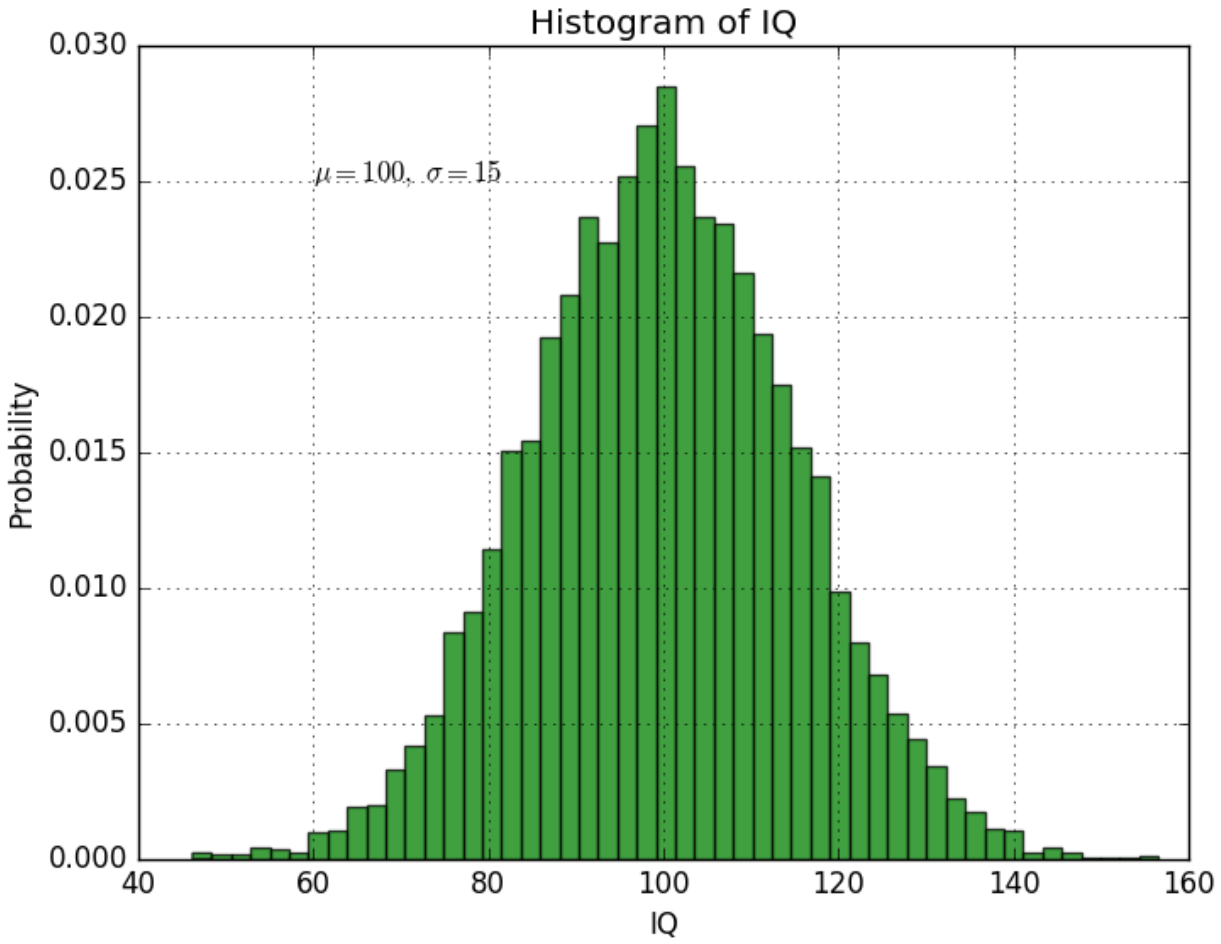
SUBPLOTS AND FONT STYLE



HISTOGRAMS

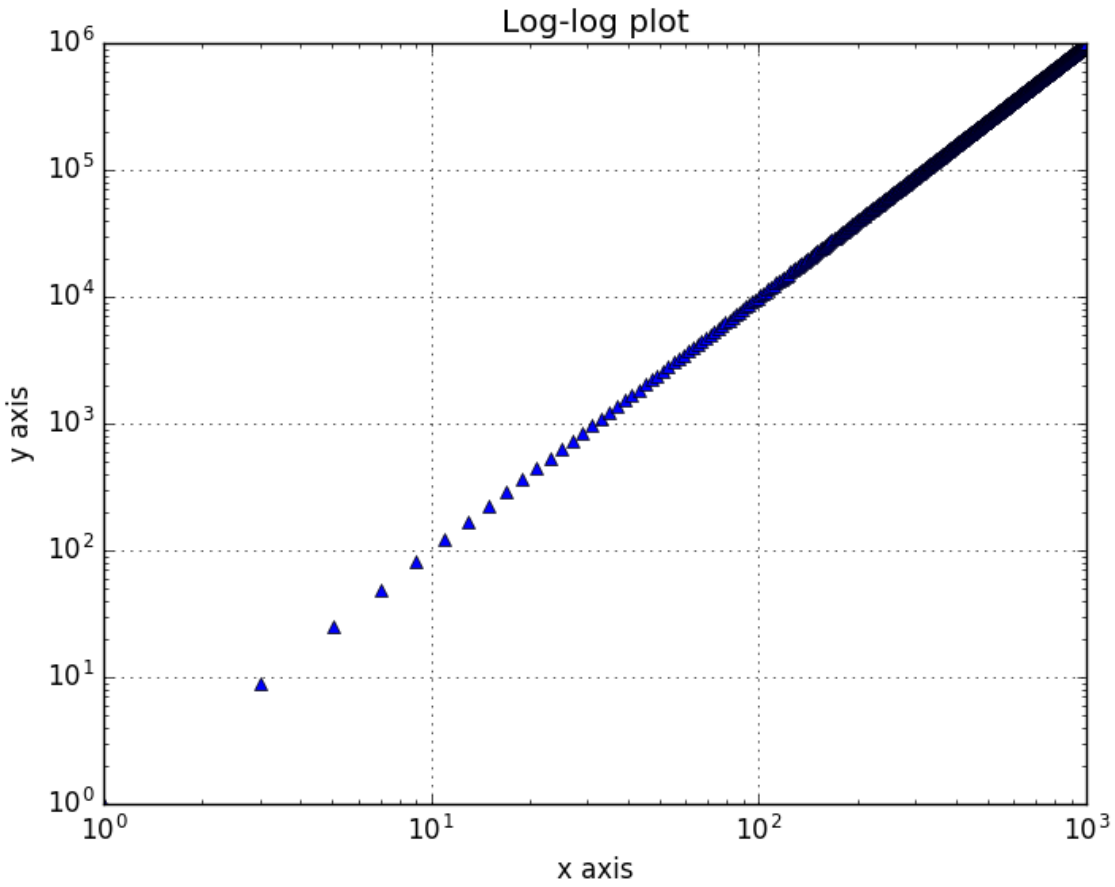
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> mu, sigma = 100, 15
>>> x = mu + sigma * np.random.randn(10000)
>>> # the histogram of the data
>>> n, bins, patches = plt.hist(x, 50, normed=1, facecolor='green', alpha=0.75)
>>> plt.xlabel(‚IQ')
>>> plt.ylabel('Probability')
>>> plt.title('Histogram of IQ')
>>> plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
>>> plt.axis([40, 160, 0, 0.03])
>>> plt.grid(True)
>>> plt.show()
```

HISTOGRAMS


Histogram of IQ
$\mu = 100,\ \sigma = 15$
Probability
0.030
0.025
0.020
0.015
0.010
0.005
0.000
40
60
80
100
120
140
160
IQ

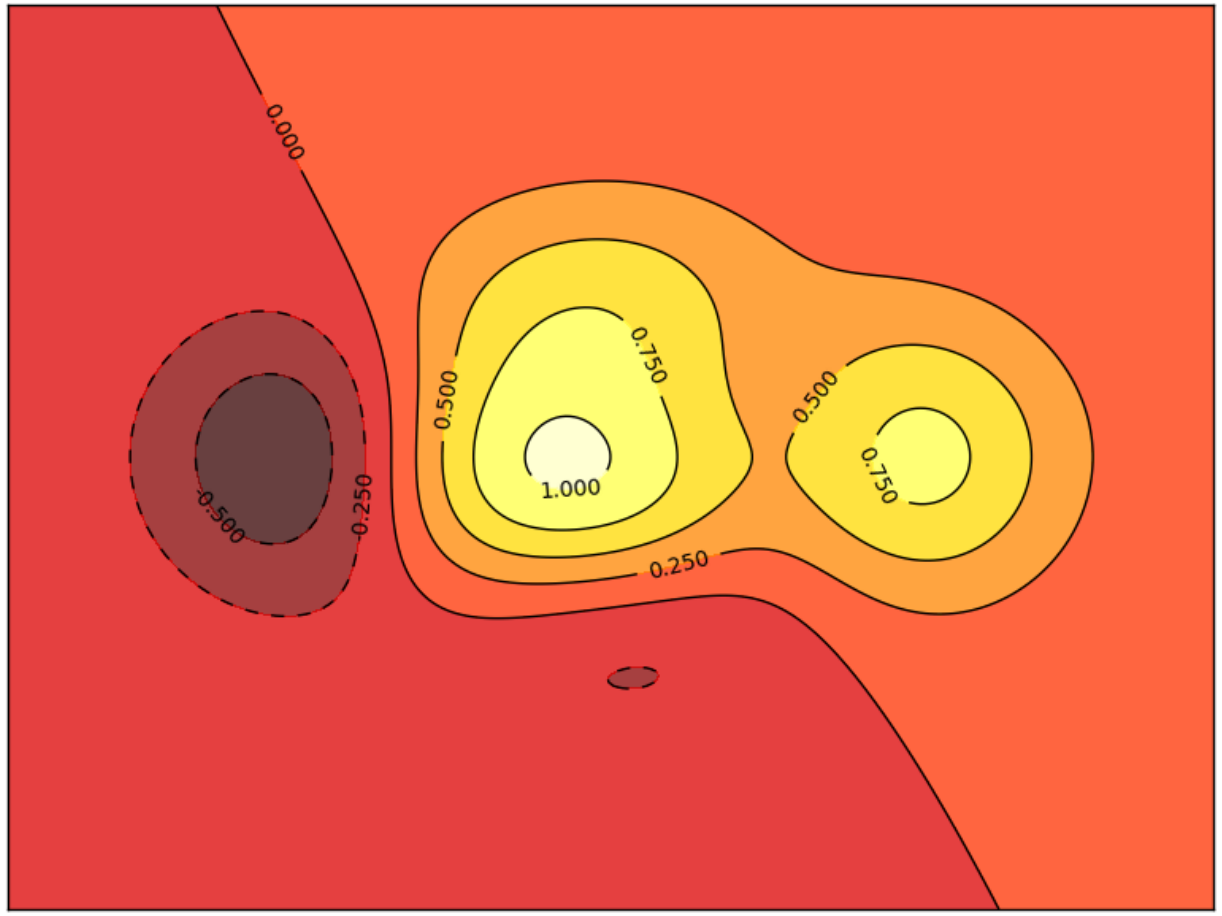
LOGARITHMIC AXES

```
>>> x = np.linspace(1, 10, 50)
>>> plt.plot(x, x**2, ‚b^’)
>>> plt.yscale(‚log’)
>>> plt.xscale(‚log’)
>>> plt.title(„Log-log plot”)
>>> plt.xlabel(„x axis)
>>> plt.ylabel(„y axis)
>>> plt.grid(True)
```



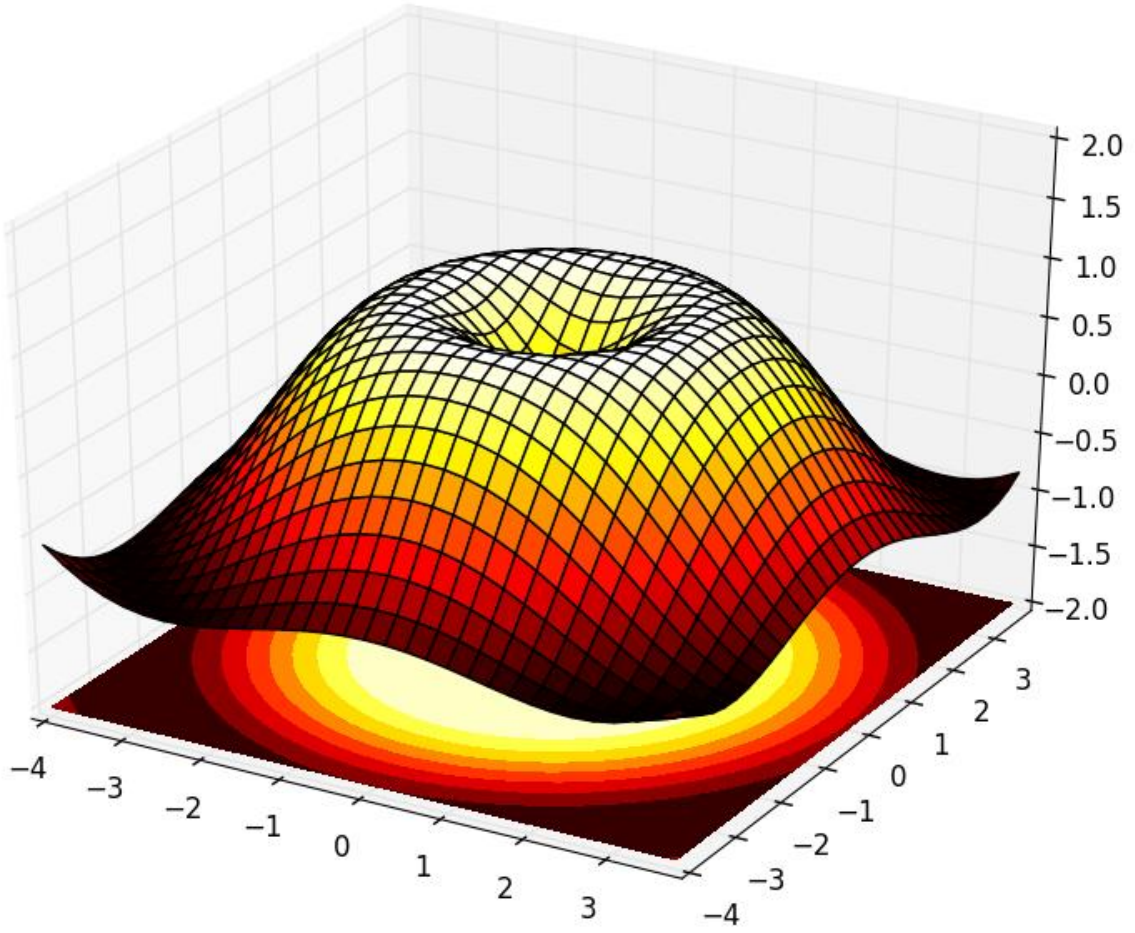
CONTOUR PLOT

```
>>> def f(x,y):
    return (1-x/2+x**5+y**3)*np.exp(-x**2-y**2)
>>> n = 256
>>> x = np.linspace(-3,3,n)
>>> y = np.linspace(-3,3,n)
>>> X,Y = np.meshgrid(x,y)
>>> plt.axes([0.025,0.025,0.95,0.95])
>>> plt.contourf(X, Y, f(X,Y), 8, alpha=.75,
cmap=plt.cm.hot)
>>> C = plt.contour(X, Y, f(X,Y), 8,
colors='black', linewidth=.5)
>>> plt.clabel(C, inline=1, fontsize=10)
>>> plt.xticks([]), plt.yticks([])
>>> plt.show()
```



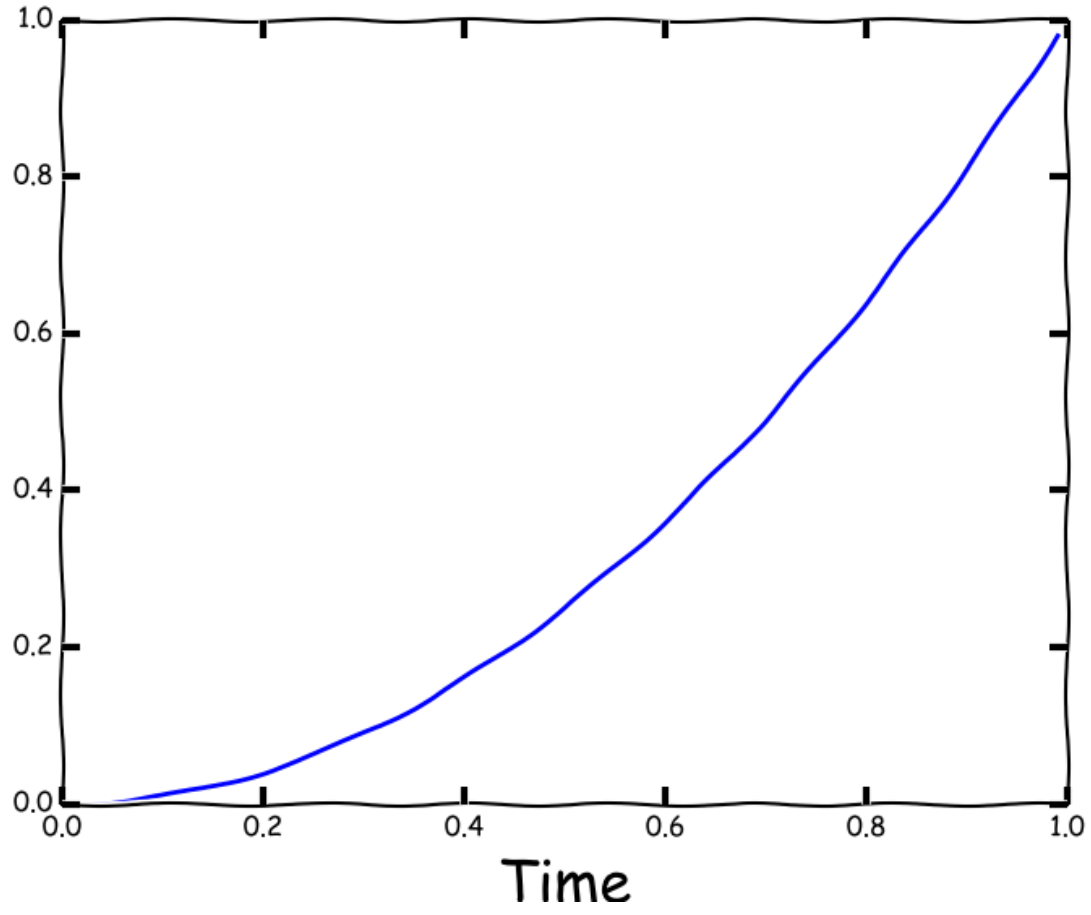
3D PLOT

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = Axes3D(fig)
>>> X = np.arange(-4, 4, 0.25)
>>> Y = np.arange(-4, 4, 0.25)
>>> X, Y = np.meshgrid(X, Y)
>>> R = np.sqrt(X**2 + Y**2)
>>> Z = np.sin(R)
>>> ax.plot_surface(X, Y, Z, rstride=1,
cstride=1, cmap=plt.cm.hot)
>>> ax.contourf(X, Y, Z, zdir='z', offset=-2,
cmap=plt.cm.hot)
>>> ax.set_zlim(-2,2)
```



A BIT OF FUN AND THE „WITH”

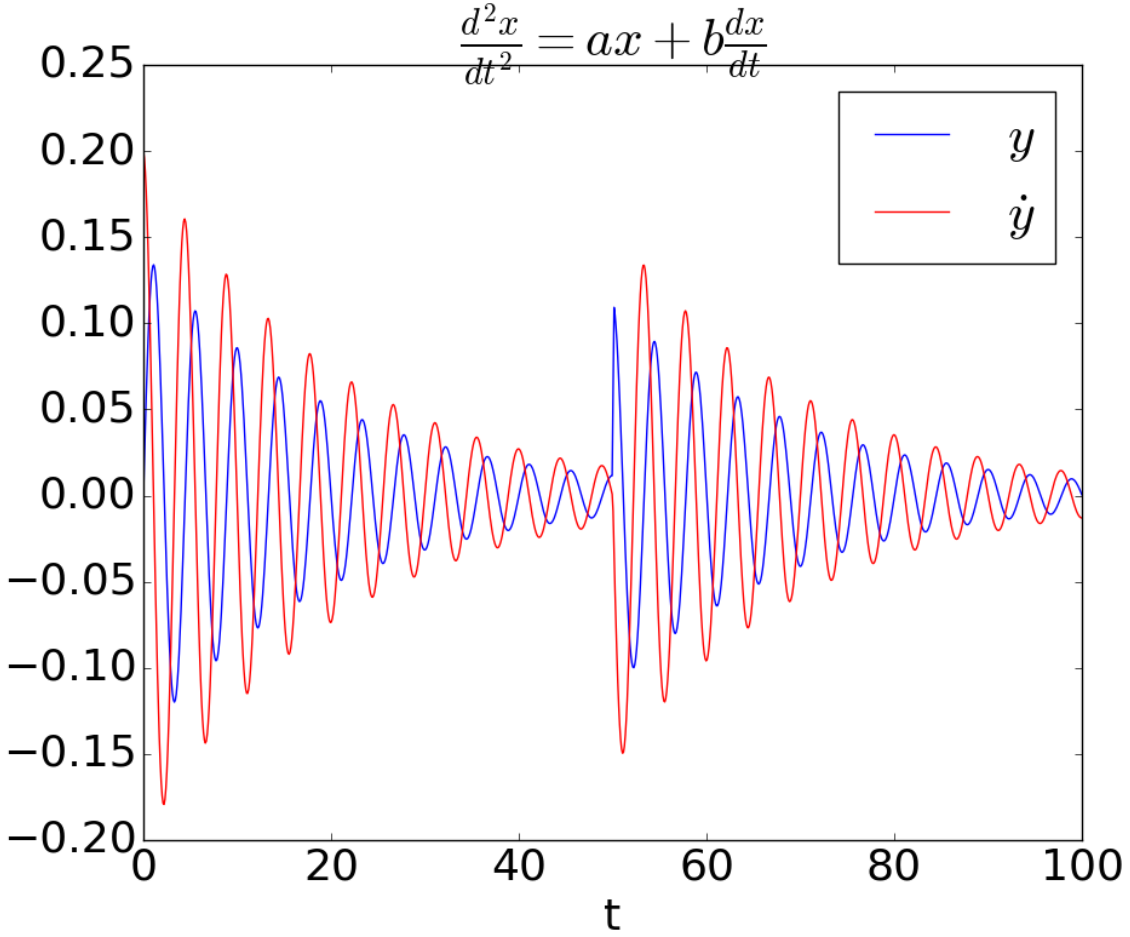
```
import numpy as np
import matplotlib.pylab as plt

t = np.arange(0,1,0.01)

with plt.xkcd():
    plt.plot(t,t*t)
    plt.xlabel('Time’,fontsize=26)
```



ORDINARY DIFFERENTIAL EQUATIONS

```
>>> from scipy.integrate import odeint
def deriv(y, t):  # return derivatives of the array
    a = -2.0
    b = -0.1
    if (t > 50 and (t % 50) < 0.1):
        y[1] += 1.0
    return np.array([y[1], a * y[0] + b * y[1]])


>>> time = np.linspace(0.0, 100.0, 1000)
>>> yinit = np.array([0.0005, 0.2])  # initial
values
>>> y = odeint(deriv, yinit, time)
```

It uses lsoda from the FORTRAN library odepack.



COMPILING PYTHON CODE

COMPILING PYTHON CODE

- We are using numba
- It has a just-in-time compiler
- Using the llvm compiler infrastructure
- Generating native machine instructions (import time, runtime or statically)
- Supports CUDA and HSA
- It is a simple, but useful way to speed up Python code without any hard work

COMPILING PYTHON CODE

```
from numba import jit
from numpy import arange
import timeit

def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result

a = arange(9).reshape(3, 3)

def withoutjit():
    sum2d(a)
```

```
@jit()
def sum2djit(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result


def withjit():
    sum2djit(a)
```

COMPILING PYTHON CODE

```
cycles = 1000000
print("The native Python running time for ",
cycles, " cycles is:")
native_time = timeit.timeit(stmt=withoutjit,
number=cycles)
print('%5.2f s' % native_time)
print("The numba JIT running time for ", cycles, "
cycles is:")
jit_time = timeit.timeit(stmt=withjit,
number=cycles)
print('%5.2f s' % jit_time)
print("The speed difference for the JIT is:")
print('%5.2fx' % (native_time / jit_time))
```

The output:

```
The native Python running time for
1000000  cycles is:
 6.25 s
The numba JIT running time for
1000000  cycles is:
 0.57 s
The speed difference for the JIT is:
```

10.95x

WRAPPING

WRAPPING

- „Wrap” the library to a form, which Python can understand
- Using a thin interface, we integrate the library into Python
- We can move data between the library and Python

How?

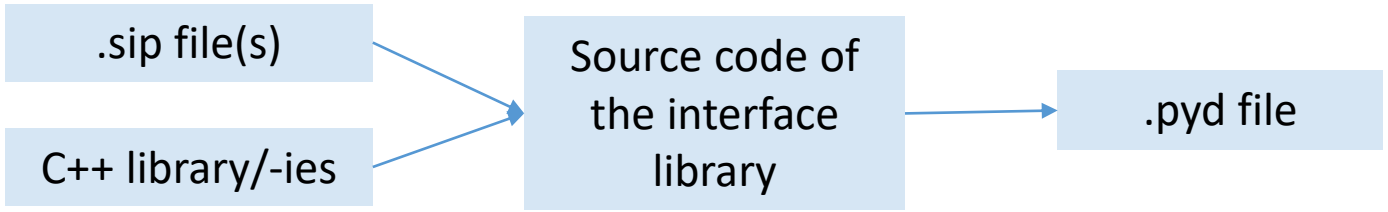
- There is a „factory” interface for C
- There are libraries for other languages:
- SWIG (multi), f2py (Fortran), **SIP** (C++), boost.python, shiboken...

WHAT IS SIP?

- A semi-automatic tool, with which we can easily wrap libraries to Python
- A program, which generates the wrapper code
- A library, which handles the C++ functionality
- Developer: Phil Thomson (developer of PyQt)
- Automatically generates the interface necessary for wrapping for C/C++
- Makes possible the automatic translation of simple types from C++ to Python
- For the complex data types, manual tuning is needed

WHAT IS THE STRUCTURE OF SIP?

- The input of the SIP program is a file with the extension .sip → generating .h and .cpp pairs → compiling → library, with which the Python can communicate
- The .sip files are written in a special script language, using only the essence of the original library headers
- Code generation is mostly automatic, sometimes things have to be adjusted (sometimes the source code does not document enough (for SIP) how to handle certain data structures).



A SIMPLE EXAMPLE

word.h

```
// Define the interface to the word library.
#include <iostream>

class Word {
    const char *the_word;

public:
    Word(){}

    char *reverse() const
          {
          std::cout << "Hello world!" << std::endl;
                  return 0;
          }
};
```

word.sip

```
// Define the SIP wrapper to the
word library.

%Module word

class Word {

%TypeHeaderCode
#include <word.h>
%End

public:
    Word();

    char *reverse() const;
};
```

A SIMPLE EXAMPLE

1. Generating the interface: sip –c . word.sip
2. Result: sipAPIword.h, sipwordcmodule.cpp, sipwordWord.cpp
3. This has to be compiled and linked with Python and the original C++ library. Simplest way: SIP build system. For this:
4. Python script: configure.py (we have to write it)
5. With the python configure.py we create the Makefile
6. With the make command we create the word.pyd file
7. In Python: import word
8. word.Word().reverse()
9. Hello World!

A SIMPLE EXAMPLE

1. Generating the interface: sip –c . word.sip
2. Result: sipAPIword.h, sipwordcmodule.cpp, sipwordWord.cpp
3. This has to be compiled and linked with Python and the original C++ library. Simplest way: SIP build system. For this:
4. Python script: configure.py (we have to write it)
5. With the python configure.py we create the Makefile
6. With the make command we create the word.pyd file
7. In Python: import word
8. word.Word().reverse()
9. Hello World!

A SIMPLE EXAMPLE

configure.py

```
import os
import sipconfig

# The name of the SIP build file generated by SIP and used by the build
# system.
build_file = "word.sbf"

# Get the SIP configuration information.
config = sipconfig.Configuration()

# Run SIP to generate the code.
os.system(" ".join([config.sip_bin, "-c", ".", "-b", build_file, "word.sip"]))

# Create the Makefile.
makefile = sipconfig.SIPModuleMakefile(config, build_file)

# Add the library we are wrapping.  The name doesn't include any platform
# specific prefixes or extensions (e.g. the "lib" prefix on UNIX, or the
# ".dll" extension on Windows).
#makefile.extra_libs = ["word"]
makefile.extra_lib_dirs = ["."]

# Generate the Makefile itself.
makefile.generate()
```

USEFUL STUFF TO READ

- David Beazley: Python Cookbook (3rd edition)
- Mark Lutz: Learning Python (5th edition)
- David Mertz: Functional Programming in Python
- Python tutorial
- Numpy tutorial
- Matplotlib tutorial
- Scipy tutorial
- There are many good tutorials on the web for Python and related stuff
- Use StackOverflow too!