# basic-numpy

March 25, 2015

Run using `ipython3 nbconvert --to slides --post serve basic-numpy.ipynb`

In [1]:
```
# For easier python 2 compatibility:
from __future__ import division, print_function, absolute_import

# Normal imports:
import numpy as np
```

# 1 Introduction to the scientific python stack

# 2 Basic module NumPy

- Basis for scientific computing with Python
- A powerfull N-dimension array object
- Basic and not so basic math operations
- Linear algebra operations
- *Normally* homogenous data (i.e. numbers)
- random numbers, FFT, sorting, ...

## 2.1 Convention

In [2]:
```
import numpy as np
```

# 3 Numpy is useful

- Homogenous data:
- Experimental data sets
- Simulations
- ...

**For example**:

In [3]:
```
x = np.linspace(0, 9, 1001)
print(x[::200])
```

```
[ 0.   1.8  3.6  5.4  7.2  9. ]
```

In [4]:
```
print(np.sqrt(x)[::200])
```

```
[ 0.          1.34164079  1.8973666   2.32379001  2.68328157  3.        ]
```

NumPy can be much faster (typically $\approx 50\times$):

```
In [5]: %%timeit
        numpy_result = np.sqrt(x)
```

100000 loops, best of 3: 4.52 $\mu$s per loop

```
In [6]: from math import sqrt
        x = list(x)
```

```
In [7]: %%timeit
        python_result = [sqrt(value) for value in x]
```

10000 loops, best of 3: 74.9 $\mu$s per loop

(The actual speedup depends, but it can be much more for simple operations like $\times, +$)

# 4 Arrays are N-Dimensional (ndarray)

A bit like nested lists:

## 4.1 0 Dimensional

```
In [8]: arr = np.array(5)
        # although 0-d arrays are sometimes a bit special :(
        print(arr)
        print('The dimension is:', arr.ndim)
        print('The shape is:', arr.shape)
```

```
5
The dimension is: 0
The shape is: ()
```

## 4.2 1 Dimensional

```
In [9]: arr = np.array([4, 5, 6])
        print(arr)
        print('The dimension is:', arr.ndim)
        print('The shape is:', arr.shape)
```

```
[4 5 6]
The dimension is: 1
The shape is: (3,)
```

## 4.3 2 Dimensional

```
In [10]: arr = np.array([[1, 2],
                         [3, 4],
                         [5, 6]])
         print(arr)
         print('The dimension is:', arr.ndim)
         print('The shape is:', arr.shape)
```

```
[[1 2]
 [3 4]
 [5 6]]
The dimension is: 2
The shape is: (3, 2)
```

### 4.4 3 Dimensional

```
In [11]: arr = np.array([[[1, 2],
                           [2, 3]],
                          [[4, 5],
                           [8, 9]]])
         print('The dimension is:', arr.ndim)
         print('The shape is:', arr.shape)
```

```
The dimension is: 3
The shape is: (2, 2, 2)
```

# 5 Array creation

## 5.1 Homogeneous arrays

```
In [12]: arr = np.zeros((3, 4))
         print(repr(arr))
         print('Shape:', arr.shape)
```

```
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
Shape: (3, 4)
```

```
In [13]: np.zeros(3, dtype=np.int64)
```

```
Out[13]: array([0, 0, 0])
```

```
In [14]: np.full((1,3), 5)
```

```
Out[14]: array([[ 5.,  5.,  5.]])
```

Check `help(np.zeros)`, etc. for other handy variations.

## 5.2 Special one dimensional arrays

Ranges for float and integer arrays

```
In [15]: np.linspace(0, 3, 7)
```

```
Out[15]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ])
```

```
In [16]: np.arange(10)  # much like pythons range
```

```
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Note: Please do not use `arange` for floats!

### 5.2.1 N-Dimensional versions:

- `np.meshgrid; np.ogrid; np.mgrid`

### 5.3 Random arrays

Draw from the interval $[0, 1[$:

```
In [17]: np.random.random((1, 5))
```

```
Out[17]: array([[ 0.63482226,  0.74569267,  0.71460822,  0.01426573,  0.0842493 ]])
```

Draw from a normal distribution with mean 2 and standard deviation 3:

```
In [18]: np.random.normal(loc=2, scale=3, size=(1, 5))
```

```
Out[18]: array([[ 2.96881514,  3.44006544, -1.08167156, -0.0345131 , -2.15090254]])
```

- And many more, check `np.random.<tab>`!

Note: `np.random.seed()` makes sure you get the same numbers. Using `np.random.RandomState` is often even better.

### 5.4 Special arrays

```
In [19]: np.eye(3)
```

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

```
In [20]: np.diag([1, 5, 2], k=1)
```

```
Out[20]: array([[0, 1, 0, 0],
                [0, 0, 5, 0],
                [0, 0, 0, 2],
                [0, 0, 0, 0]])
```

And many many more...

## 6 Lets do math!

- Math is **elementwise** (unlike lists)

```
In [21]: arr = np.linspace(-1, 1, 5)
         # 5 steps from -1 to 1 (including both)
         print(arr)
```

```
[-1.  -0.5  0.   0.5  1. ]
```

```
In [22]: print(arr + 1)
```

```
[ 0.   0.5  1.   1.5  2. ]
```

### 6.1 Unary functions

```
In [23]: np.sin(arr)
```

```
Out[23]: array([-0.84147098, -0.47942554,  0.        ,  0.47942554,  0.84147098])
```

```
In [24]: arr**2
```

```
Out[24]: array([ 1.  ,  0.25,  0.  ,  0.25,  1.  ])
```

```
In [25]: abs(arr)
```

```
Out[25]: array([ 1. ,  0.5,  0. ,  0.5,  1. ])
```

```
In [26]: -arr
```

```
Out[26]: array([ 1. ,  0.5, -0. , -0.5, -1. ])
```

## 6.2 Binary functions

```
In [27]: arr + arr  # also np.add(arr, arr)

Out[27]: array([-2., -1.,  0.,  1.,  2.])

In [28]: arr * 2

Out[28]: array([-2., -1.,  0.,  1.,  2.])

In [29]: arr + 3

Out[29]: array([ 2. ,  2.5,  3. ,  3.5,  4. ])

In [30]: arr == -1

Out[30]: array([ True, False, False, False, False], dtype=bool)
```

## 6.3 Reductions

Many reductions are available either through `ndarray` attributes or as numpy functions:

```
In [31]: np.sum(arr)  # or
         arr.sum()

Out[31]: 0.0
```

*Warning:* Usual `sum(arr)` is *not* correct since it is a python function and not an operator.

## 6.4 Other reductions

```
In [32]: np.mean(arr)  # or
         arr.mean()

Out[32]: 0.0

In [33]: # Logical:
         arr.all(); arr.any()
         # Minimum/Maximum:
         arr.min(); arr.max(); arr.argmin(); arr.argmax()
         # Standard deviation:
         arr.std(ddof=1); arr.var()
         np.median(arr);  # not arr.median here

Out[33]: 0.0
```

## 6.5 Recutions – axis

Most reduction (-like) functions accept an `axis` argument:

```
In [34]: arr = np.random.random((50, 3))
         arr.sum(0)

Out[34]: array([ 26.99593018,  26.52235909,  23.80457886])

In [35]: arr.sum(0, keepdims=True)

Out[35]: array([[ 26.99593018,  26.52235909,  23.80457886]])
```

You can specify multiple axes (its still called "axis"):

```
In [36]: arr.sum(axis=(0, 1))

Out[36]: 77.322868130310965
```

# 7 *(Master)* Reductions

All binary ufuncs (simple elementwise math functions from above) allow reductions. For example `arr.sum()` is actually a thin wrapper around `np.add.reduce(arr)`!

# 8 All Available functions

All available unary (math/ufunc) functions:

```
In [37]: for obj_string in dir(np):
             obj = getattr(np, obj_string)
             if (isinstance(obj, np.ufunc)
                     and obj.nin == 1 and obj.nout == 1):
                 print(obj_string, end=', ')
```

```
abs, absolute, arccos, arccosh, arcsin, arcsinh, arctan, arctanh, bitwise_not, ceil, conj, conjugate, co
```

All available binary (math/ufunc) functions:

```
In [38]: for obj_string in dir(np):
             obj = getattr(np, obj_string)
             if (isinstance(obj, np.ufunc)
                     and obj.nin == 2 and obj.nout == 1):
                 print(obj_string, end=', ')
```

```
add, arctan2, bitwise_and, bitwise_or, bitwise_xor, copysign, divide, equal, floor_divide, fmax, fmin, fr
```

Other (math/ufunc) functions:

```
In [39]: for obj_string in dir(np):
             obj = getattr(np, obj_string)
             if (isinstance(obj, np.ufunc)
                     and (obj.nin > 2 or obj.nout != 1)):
                 print(obj_string, end=', ')
```

```
frexp, modf,
```

# 9 Broadcasting

Implicite repetition of arrays.

We have already done this:

```
In [40]: arr = np.array([1, 2, 3, 4])
         arr + 3
```

```
Out[40]: array([4, 5, 6, 7])
```

Is actually:

```
In [41]: threes = np.array([3, 3, 3, 3])
         arr + threes
```

```
Out[41]: array([4, 5, 6, 7])
```

This happens (more efficiently):

```
In [42]: three = np.array(3)
         three = three.reshape(1)
         print('Shape:', three.shape)
         threes = three.repeat(len(arr))
         print('Threes shape:', threes.shape)
```

```
Shape: (1,)
Threes shape: (4,)
```

```
In [43]: result = arr + threes
```

```
In [44]: red = np.arange(8)
         green = np.ones((4, 3, 1))

         blue = red * green
         print(blue.shape)
```

```
(4, 3, 8)
```

### 9.1 Example:

```
In [45]: arr1 = np.arange(5)
         arr2 = np.arange(10, 15)
```

Create `(5, 5)` array, which combines all numbers:

```
In [46]: arr2 = arr2[:, np.newaxis]  # later
         print(arr2.shape)
```

```
(5, 1)
```

```
In [47]: print(arr1 +  arr2)
```

```
[[10 11 12 13 14]
 [11 12 13 14 15]
 [12 13 14 15 16]
 [13 14 15 16 17]
 [14 15 16 17 18]]
```

## 10 Container manipulation

### 10.1 Reshaping

You can reshape arrays (more in the exercizes)

```
In [48]: np.arange(10).reshape(2, 5)
```

```
Out[48]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

```
In [49]: np.arange(10).reshape(2, -1)  # -1 is a "joker"
```

```
Out[49]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

## 10.2 Other manipulations

In [50]:
```
np.arange(3).repeat(2)
```

Out[50]:
```
array([0, 0, 1, 1, 2, 2])
```

In [51]:
```
np.concatenate((np.arange(2), np.arange(3)))
```

Out[51]:
```
array([0, 1, 0, 1, 2])
```

- Concatenate has many friends (hstack, see "See Also")

There are more to explorer for specific tasks.

# 11 Indexing

- Indexing is very powerfull in NumPy
- There are different types of indexing:

1. Picking a single element
2. Slicing
3. Advanced indexing:

- picking many elements at once

4. (Advanced) Boolean indexing:

- Selecting based on logical expressions

Note: *Advanced* indexing is often also called *fancy* indexing

## 11.1 Picking an element

Picking a single element from an array requires an integer along each dimension

In [52]:
```
arr = np.arange(10).reshape(2, 5)
print(arr)
```

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

In [53]:
```
print(arr[0, 3])
print(arr[1, 2])
```

```
3
7
```

In [54]:
```
indx = (0, 3); print(arr[indx])  # This is identical!
```

```
3
```

# 12 Incomplete index

```
In [55]: arr
```

```
Out[55]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

```
In [56]: arr[0]
```

```
Out[56]: array([0, 1, 2, 3, 4])
```

This is much like a list of lists. But **never** do this:

```
In [57]: arr[0][1]
```

```
Out[57]: 1
```

## 12.1 Slicing

- You already know `:` (`slice`) from lists
- We can do it in arbitrary dimensions
- NumPy also has `...` (`Ellipsis`)
- NumPy also has `np.newaxis` (identical to `None`)

### 12.1.1 Simple slicing

- Just like slicing of lists `list[start:stop:step]`
- For many dimensions each is sliced seperatly

```
In [58]: arr
```

```
Out[58]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

```
In [59]: arr[1:, 1::2]
```

```
Out[59]: array([[6, 8]])
```

```
In [60]: # Maybe it helps to realize this:
         arr[1:][:, 1::2]  # note the single : is to "skip" that dimension
```

```
Out[60]: array([[6, 8]])
```

### 12.1.2 Ellipsis

`...` replaces an arbitray number of `:`

```
In [61]: print(arr[1, ...])  # identical to arr[1]
         print(repr(arr[1, 0, ...]))  # never a scalar
         print(repr(arr[1, 0]))  # is a scalar (immutable)
```

```
[5 6 7 8 9]
array(5)
5
```

```
In [62]: high_dimensional = np.ones((5, 4, 3, 2))
         high_dimensional[..., :1].shape
```

```
Out[62]: (5, 4, 3, 1)
```

```
In [63]: high_dimensional[0, ..., 1].shape
```

```
Out[63]: (4, 3)
```

### 12.1.3 Newaxis

Inserts a new axis of size 1 (increases dimension)

```
In [64]: arr.shape

Out[64]: (2, 5)

In [65]: arr[:, np.newaxis, :, np.newaxis].shape

Out[65]: (2, 1, 5, 1)

In [66]: arr[..., None].shape

Out[66]: (2, 5, 1)

In [67]: np.newaxis is None

Out[67]: True
```

## 12.2 Advanced (integer) Indexing

Create a new array selecting many elements

```
In [68]: arr = np.arange(1, 6)

In [69]: indx = np.array([0, 4, 2], dtype=np.intp)
         arr[indx]

Out[69]: array([1, 5, 3])

In [70]: arr[0], arr[4], arr[2]

Out[70]: (1, 5, 3)

In [71]: arr[[0, 4, 2]]  # But nested lists do not!

         # intp is safest, but please do not worry about it generally:
         arr[np.array([0, 4, 2], dtype=np.uint16)]

Out[71]: array([1, 5, 3])
```

### 12.2.1 Can be combined with other indexing

```
In [72]: arr = np.arange(6).reshape(2, 3)
         arr

Out[72]: array([[0, 1, 2],
                [3, 4, 5]])

In [73]: arr[[1, 0], :2]

Out[73]: array([[3, 4],
                [0, 1]])
```

Slicing happens first, then the rest (actually it is more the other way around)

### 12.2.2 Can have arbitrary shapes

```
In [74]: arr
```

```
Out[74]: array([[0, 1, 2],
                [3, 4, 5]])
```

```
In [75]: indx = np.array([[0, 1],
                          [2, 1]])
         arr[0, indx]
```

```
Out[75]: array([[0, 1],
                [2, 1]])
```

### 12.2.3 Two advanced indexes are iterated together

This is *not* like slicing (or matlab)!

```
In [76]: arr
```

```
Out[76]: array([[0, 1, 2],
                [3, 4, 5]])
```

```
In [77]: indx1 = np.array([1, 0])
         indx2 = np.array([0, 2])
         arr[indx1, indx2]
```

```
Out[77]: array([3, 2])
```

The output has the same shapes as the (broadcasted) input!

### 12.2.4 Example: Make use of multiple advanced indices

Select the "corners" with advanced indexing

```
In [78]: arr
```

```
Out[78]: array([[0, 1, 2],
                [3, 4, 5]])
```

```
In [79]: indx1 = np.array([0, -1])
         indx2 = np.array([[[0],
                            [-1]]])
         arr[indx1, indx2]
```

```
Out[79]: array([[[0, 3],
                 [2, 5]]])
```

## 12.3 Boolean indexing

- Filter an array
- Can work on the full arrays or some axes
- Result is always one dimensional

```
In [80]: arr
```

```
Out[80]: array([[0, 1, 2],
                [3, 4, 5]])
```

```
In [81]: arr > 3
```

```
Out[81]: array([[False, False, False],
                [False,  True,  True]], dtype=bool)
```

```
In [82]: arr[arr > 3]
```

```
Out[82]: array([4, 5])
```

### 12.3.1 Use it on part of the array only

```
In [83]: arr
```

```
Out[83]: array([[0, 1, 2],
                [3, 4, 5]])
```

```
In [84]: print('Every column starting >=1:', repr(arr[0] >= 1))
         arr[:, arr[0] >= 1]  # Every column starting >= 1
```

```
Every column starting >=1: array([False,  True,  True], dtype=bool)
```

```
Out[84]: array([[1, 2],
                [4, 5]])
```

```
In [85]: print('Every row starting >1:', repr(arr[:, 0] > 1))
         arr[arr[:, 0] > 1]
```

```
Every row starting >1: array([False,  True], dtype=bool)
```

```
Out[85]: array([[3, 4, 5]])
```

## 12.4 Assignments

Indexing can also be used for *assignmnents*:

```
In [106]: arr = np.arange(10)
          arr[:4] = 0
          arr
```

```
Out[106]: array([0, 0, 0, 0, 4, 5, 6, 7, 8, 9])
```

```
In [107]: arr[[-1, -3]] = 999
          arr
```

```
Out[107]: array([  0,   0,   0,   0,   4,   5,   6, 999,   8, 999])
```

```
In [108]: arr[arr > 10] = 42
          arr
```

```
Out[108]: array([ 0,  0,  0,  0,  4,  5,  6, 42,  8, 42])
```

## 12.5 Some notes

- All of these can be combined
- But combination can be very complicating in some cases
  - for example `arr[index_array, :, index_array]`.
- Indexing is best with the `np.intp` type (others may be slower or in principle unsafe)

# 13 Memory and Views

1. Numpy arrays are of course mutable objects.
2. Slices create **views** into the same memory.

```
In [86]: arr = np.arange(10)
         # Take the first half:
         view = arr[:5]
         # Set it to 0:
         view[...] = 0
         print(arr)
```

```
[0 0 0 0 0 5 6 7 8 9]
```

*Note:* Advanced indexing *always* creates a copy, slicing *never* creates a copy.

## 13.1 Take care about views

- Some functions will return a view *if possible* i.e.:
  - `np.reshape`
  - `np.ravel`
- A functions *reads* → views are great.
- A function *writes* → no view unless `out` argument.

## 13.2 Optimizing memory usage

- Many numpy functions provide an `out` argument (all ufuncs): `arr += 1` is the same as `np.add(arr, 1, out=arr)`
- **Be careful** when using the `out`. *Only* use out when this is **not** the same data. A common pitfall for example is:

  `arr += arr.T`

  Since `arr.T` is a view, it is changed during the operation → **unpredictable results** (it might even work)

- A small view into a large array will keep the large array alive.

## 13.3 Datatypes

- Unlike lists, arrays have a specific element type.
- Be careful with integers:

```
In [114]: arr = np.arange(10)  # either 32 or 64 bit!
          arr = arr.astype(np.int64)
          arr += 0.5  # Will be an error in the future!
          arr
```

```
Out[114]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [121]: arr + np.int64(2**63-1)
```

```
Out[121]: array([9223372036854775807, -9223372036854775808, -9223372036854775807,
                 -9223372036854775806, -9223372036854775805, -9223372036854775804,
                 -9223372036854775803, -9223372036854775802, -9223372036854775801,
                 -9223372036854775800])
```

Floats have finite precision. Especial float32 (single precision):

```
In [123]: arr = np.ones((10**8, 2)).astype(np.float32)
          print(arr.mean(0))
```

```
[ 0.16777216  0.16777216]
```

# 14 SciPy

Scipy is a collection of many packages such as:

- integrate: Integration and ordinary differential equation solvers
- interpolate: Interpolation and smoothing splines
- linalg: Linear algebra
- ndimage: N-dimensional image processing
- optimize: Optimization and root-finding routines
- signal: Signal processing
- spatial: Spatial data structures and algorithms
- stats: Statistical distributions and functions
- ...

## 14.1 Usage

- Import a subpackage directly for example:
- `from scipy import integrate`
- `from scipy.spatial import KDTree`
- Documentation available at (also numpy): http://scipy.org
- Main namespace is (basically) just numpy → do not use it.

## 14.2 Other packages

1. SciPy is BSD license, but some packages (sometimes "scikits") are either to large or copyleft.
2. The ecosystem is constantly growing.
3. Some packages to keep in mind are for example:

- scikits-learn (`sklearn` as a package)
- scikits-image (`skimage`)
- networkx
- astropy
- statsmodels
- ...

# 15 Plotting with matplotlib

```
In [89]: # In the ipython notebook, make it aware of matplotlib:
         %matplotlib inline
         import matplotlib
         # And fix the default savefig dpi that ipython sets very low
         matplotlib.rcParams['savefig.dpi'] = 120
```

### 15.0.1 Pyplot interface

```
In [90]: # Import interactive interface to matplotlib:
         from matplotlib import pyplot as plt
```

- Pyplot interface will "remember" the last figure you worked with
- It will redraw automatically
- convenient for most cases

**But:** * Complex things or GUI programming → use the figure/axis objects directly. * Please do not use `pylab`

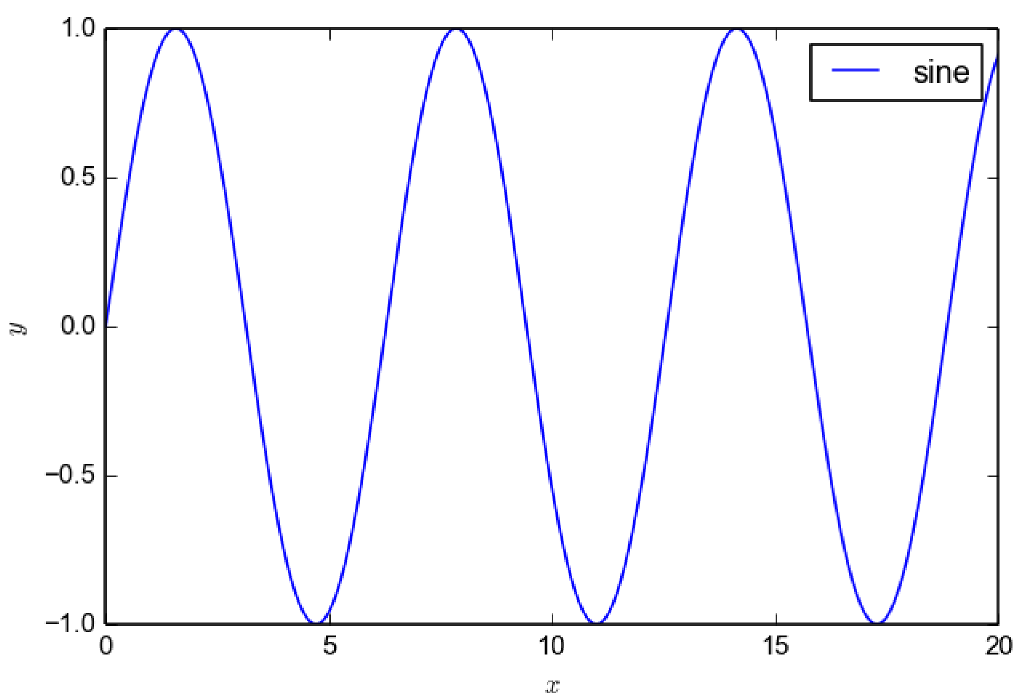
## 15.1 Plotting from a script, etc.

- *IPython notebook* like before or `ipython notebook --matplotlib=inline`
- Run *IPython* with: `ipython --matplotlib`
- In a script calling `plt.show()` will show all figures and pause the program
- `plt.savefig()` or 'figure.savefig()" to save to almost arbitrary file types.
- Animations are easy → check the tutorials

## 15.2 Simple plot examples:

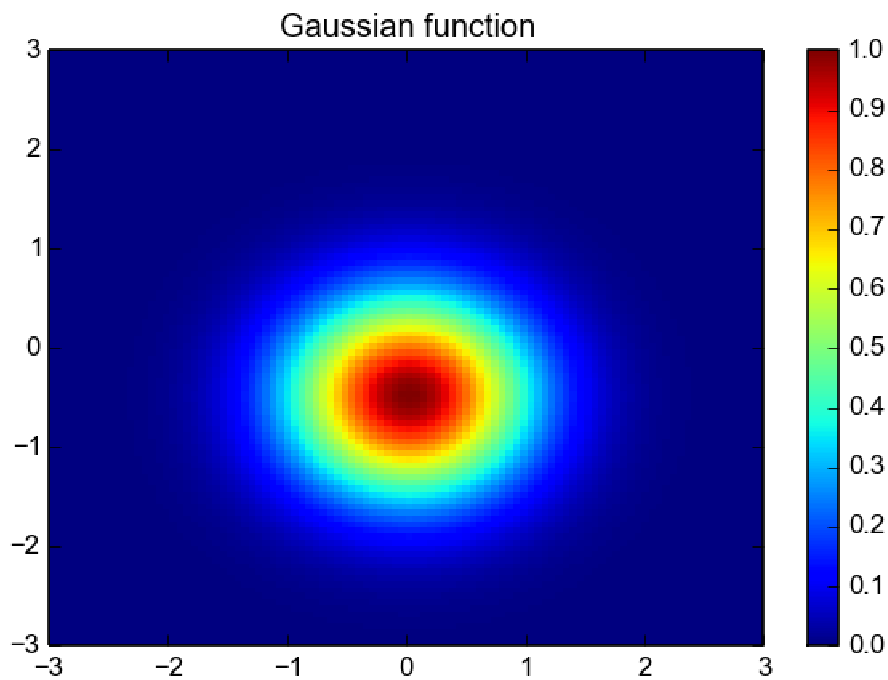
```
In [128]: x = np.linspace(0, 20, 301)
          plt.plot(x, np.sin(x), label='sine')
          plt.legend()
          plt.xlabel('$x$')
          plt.ylabel('$y$')

Out[128]: <matplotlib.text.Text at 0x7f2e1420d950>
```



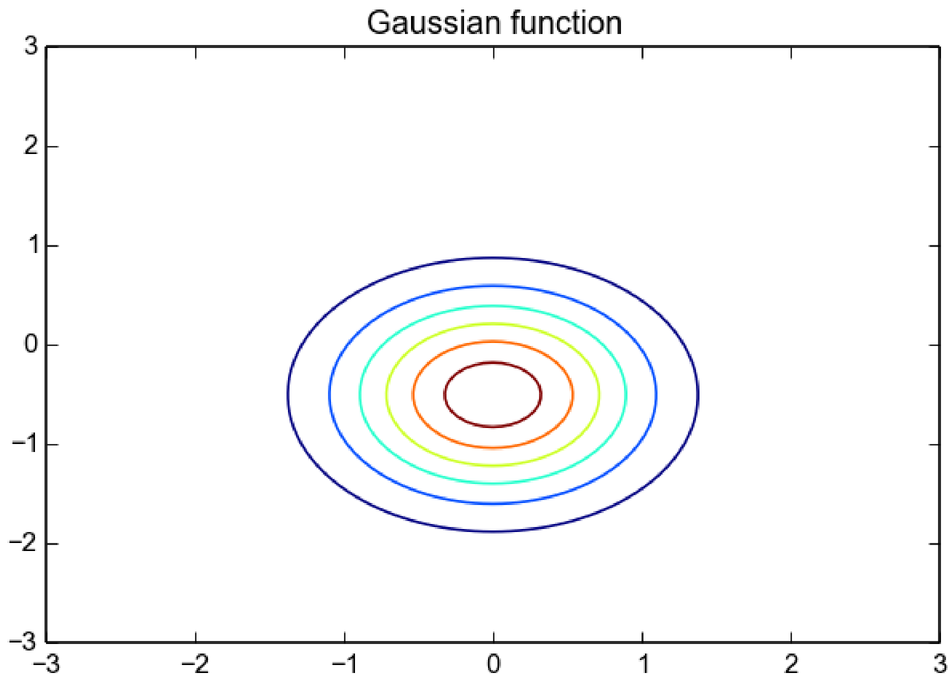
```
In [127]: x = y = np.linspace(-3, 3, 101)
          plt.title('Gaussian function')
          values = np.exp(-(x**2 + (y[:, None] + 0.5)**2))
          plt.pcolormesh(x, y, values)
          plt.colorbar()
```

```
Out[127]: <matplotlib.colorbar.Colorbar instance at 0x7f2e141a0e60>
```



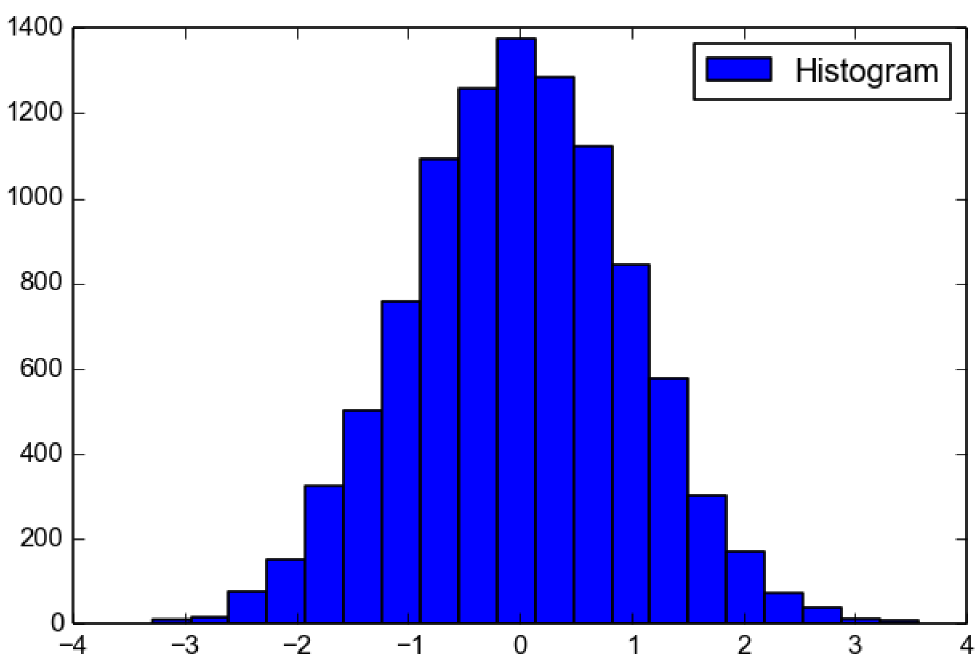

```
In [93]: x = y = np.linspace(-3, 3, 101)
         plt.title('Gaussian function')
         values = np.exp(-(x**2 + (y[:, None] + 0.5)**2))
         plt.contour(x, y, values)
```

```
Out[93]: <matplotlib.contour.QuadContourSet instance at 0x7f2e1982e0e0>
```



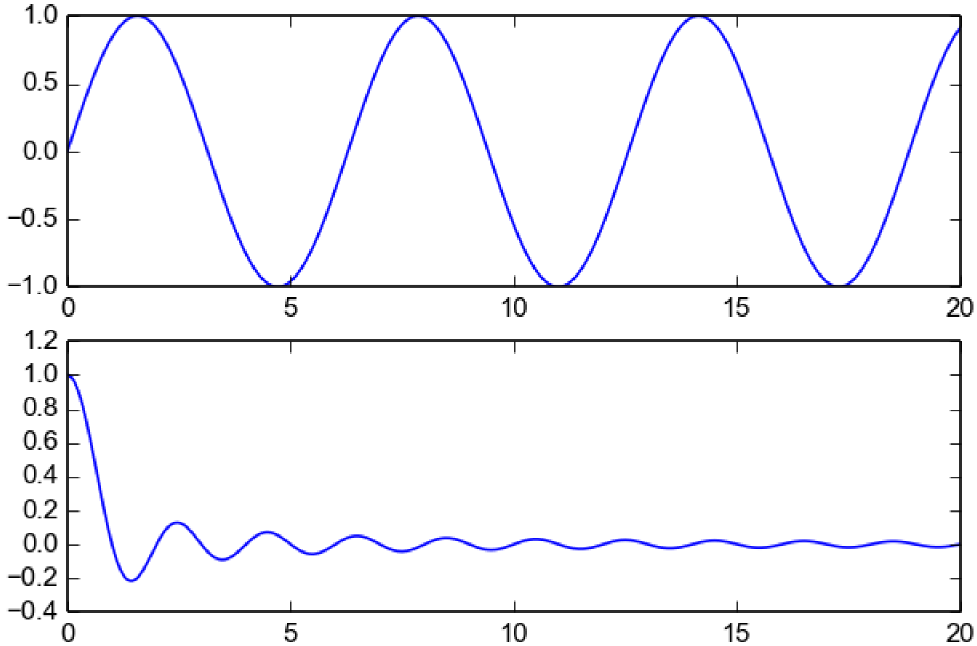

```
In [94]: from scipy.stats import norm
         random = norm.rvs(size=10000)
         plt.hist(random, bins=20, label='Histogram')
         plt.legend()
```

```
Out[94]: <matplotlib.legend.Legend at 0x7f2e19af0cd0>
```

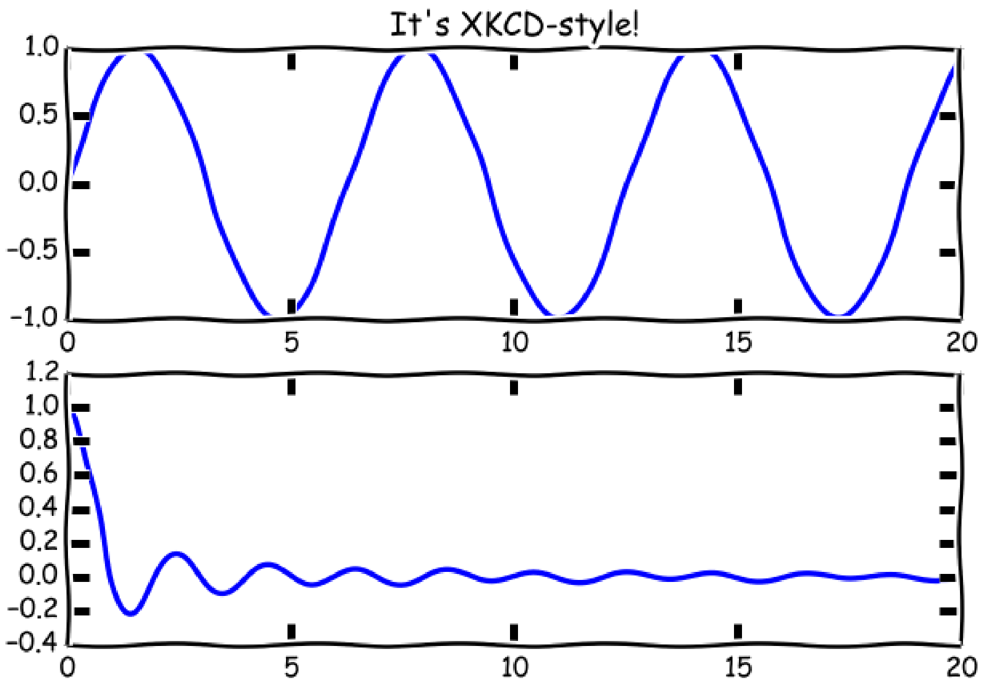


In [95]:
```
x = np.linspace(0, 20, 401)
plt.subplot(2, 1, 1)
plt.plot(x, np.sin(x))
plt.subplot(2, 1, 2)
plt.plot(x, np.sinc(x))
```

Out[95]: `[<matplotlib.lines.Line2D at 0x7f2e142b0ad0>]`




In [129]:
```
with plt.xkcd():
    x = np.linspace(0, 20, 401)
    sub1 = plt.subplot(2, 1, 1)
    plt.title("It's XKCD-style!")
    plt.plot(x, np.sin(x))
    plt.subplot(2, 1, 2)
    plt.plot(x, np.sinc(x))
```




## 15.3 More Examples

- Many, many very good examples at the **Gallery**:
  http://matplotlib.org/gallery
- Examples for animations, Graphical User Interface:
  http://matplotlib.org/examples/index.html

## 15.4 Conclusions

We have seen: 1. How to do convenient, fast math in **NumPy** 2. Where to find many useful tools with **SciPy and friends** 3. How to make beautiful graphics with **matplotlib**

Last point: 1. Do *not* reinvent the wheel 2. Algorithms matter 3. "Premature optimization is the root of all evil" – Donald Knuth