Introduction to NumPy

Benjamin Roth

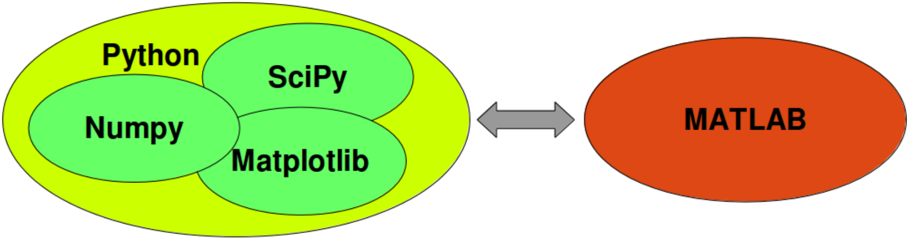
CIS LMU München

What is NumPy?

- Acronym for "Numeric Python"
- Open source extension module for Python.
- Powerful data structures for efficient computation of multi-dimensional arrays and matrices.
- Fast precompiled functions for mathematical and numerical routines.
- Used by many scientific computing and machine learning packages. For example
  - *Scipy* (Scientific Python): Useful functions for minimization, regression, Fourier-transformation and many others.
  - *Theano*: Deep learning, mimimization of custom objective functions, auto-gradients.
- Downloading and installing numpy: `www.numpy.org`

The Python Alternative to Matlab

- Python in combination with Numpy, Scipy and Matplotlib can be used as a replacement for MATLAB.
- Matplotlib provides MATLAB-like plotting functionality.



Comparison between Core Python and Numpy

- *"Core Python"*: Python without any special modules, i.e. especially without NumPy.
- 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

- NumPy needs to be imported. Convention: use short name `np`

```
import numpy as np
```

- Turn a list of temperatures in Celsius into a one-dimensional numpy array:

```
>>> cvalues = [25.3, 24.8, 26.9, 23.9]
>>> np.array(cvalues)
[ 25.3  24.8  26.9  23.9]
```

- Turn temperature values into degrees Fahrenheit:

```
>>> C * 9 / 5 + 32
[ 77.54  76.64  80.42  75.02]
```

- Compare to using core python only:

```
>>> [ x*9/5 + 32 for x in cvalues]
[77.54, 76.64, 80.42, 75.02]
```

Creation of evenly spaced values (given stepsize)

- Useful for plotting: Generate values for x and compute $y = f(x)$
- Syntax:

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

- Similar to core python `range`, but returns `ndarray` rather than a `list iterator`.
- Defaults for `start` and `step`: 0 and 1
- `dtype`: If it is not given, the type will be automatically inferred from the other input arguments.
- Don't use non-integer step sizes (use `linspace` instead).
- Examples:

```
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(1,5,2)
array([1, 3])
```

Creation of evenly spaced values (given number of values)

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

- Creates `ndarray` with `num` values equally distributed between `start` (included) and `stop` (excluded).
- If `endpoint=True`, the end point is included additionally.

```
>>> np.linspace(1, 3, 5)
array([ 1. ,  1.5,  2. ,  2.5,  3. ])
>>> np.linspace(1, 3, 4, endpoint=False)
array([ 1. ,  1.5,  2. ,  2.5])
```

- If `retstep=True`, the stepsize is returned additionally:

```
>>> np.linspace(1, 3, 4, endpoint=False, \
      retstep=True)
(array([ 1. ,  1.5,  2. ,  2.5]), 0.5)
```

Exercise

- Compare the speed of vector addition in core Python and Numpy

Multidimensional Arrays

- NumPy arrays can be of arbitrary dimension.
- 0 dimensions (scalar):
  `np.array(42)`
- 1 dimension (vector):
  `np.array([3.4, 6.9, 99.8, 12.8])`
- 2 dimensions (matrix):

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

- 3 or more dimensions (tensor):

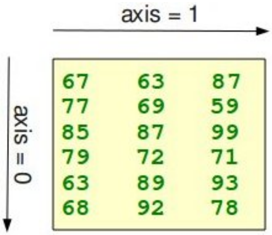
```
np.array([ [[111, 112], [121, 122]],
           [[211, 212], [221, 222]],
           [[311, 312], [321, 322]] ])
```

Question

- When can a 3 dimensional array be an appropriate representation?

Shape of an array

```
>>> x = np.array([ [67, 63, 87],
...                [77, 69, 59],
...                [85, 87, 99],
...                [79, 72, 71],
...                [63, 89, 93],
...                [68, 92, 78]])
>>> np.shape(x)
(6, 3)
```



Changing the shape

- `reshape` creates new array:

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

- Changing `shape` value (for existing array):

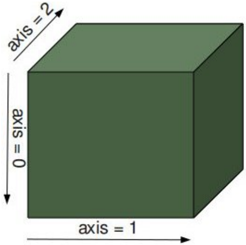
```
>>> a.shape = (2, 6)
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

- Obviously, product of shape sizes must match number of elements!
- If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated.

Shape of 3D Array

```
>>> a = np.arange(24).reshape(2,3, 4)
>>> a
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]]])
```



Transposing an Array

- 2D case:

```
>>> a = np.arange(6).reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.T
array([[0, 3],
       [1, 4],
       [2, 5]])
```

- Multidimensional case:
  - `a.transpose(...)` takes tuple of indices, indicating which axis of the old (input) array is used for each axis of the new (output) array.
  - 3D example:
    `b = a.transpose(1,0,2)`
  - $\Rightarrow$ axis 1 in *a* is used as axis 0 for *b*, axis 0 (*a*) becomes 1 (*b*), and axis 2 (*a*) stays axis 2 (*b*).

Basic Operations

- By default, arithmetic operators on arrays apply *elementwise*:

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.array( [0,1,2,3] )
>>> c = a-b
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> a<35
array([ True, True, False, False], dtype=bool)
```

- In particular, the *elementwise multiplication* ...

```
>>> a * b
array([ 20,  60, 120, 200])
```

- ... is not to be confused with the *dot product*:

```
>>> a.dot(b)
400
```

Unary Operators

- Numpy implements many standard unary (elementwise) operators:

```
>>> np.exp(b)
>>> np.sqrt(b)
>>> np.log(b)
```

- For some operators, an axis can be specified:

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

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

>>> b.min(axis=1)
array([0, 4, 8])
```

Indexing elements

- Indexing single elements:

```
>>> B = np.array([ [[111, 112], [121, 122]],
...                [[211, 212], [221, 222]],
...                [[311, 312], [321, 322]] ])
>>> B[2][1][0]
321
>>> B[2,1,0]
321
```

- Indexing entire sub-array:

```
>>> B[1]
array([[211, 212],
       [221, 222]])
```

- Indexing starting from the end:

```
>>> B[-1,-1]
array([321, 322])
```

Indexing with Arrays/Lists of Indices

```
>>> a = np.arange(12)**2
>>> i = np.array( [ 1,1,3,8,5 ] )
>>> # This also works:
>>> # i = [ 1,1,3,8,5 ]
>>> a[i]
array([ 1,  1,  9, 64, 25])
```

Indexing with Boolean Arrays

Boolean indexing is done with a boolean matrix of the *same shape* (rather than of providing a list of integer indices).

```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)

>>> a[b]
array([ 5,  6,  7,  8,  9, 10, 11])

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

Slicing

- Syntax for slicing lists and tuples can be applied to multiple dimensions in NumPy.
- Syntax:

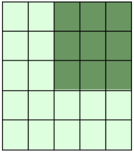
```
A[start0:stop0:step0, start1:stop1:step1, ...]
```

- Example in 1 dimension:

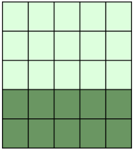
```
>>> S = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> S[3:6:2]
array([3, 5])
>>> S[:4]
array([0, 1, 2, 3])
>>> S[4:]
array([4, 5, 6, 7, 8, 9])
>>> S[:]
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Slicing 2D

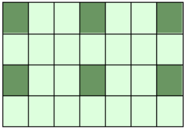
```
A = np.arange(25).reshape(5,5)
B = A[:3,2:]
```



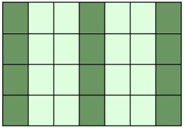
```
B = A[3:,:]
```



```
X = np.arange(28).reshape(4,7)
Y = X[::2, ::3]
```



```
Y = X[:, ::3]
```



Slicing: Caveat

- Slicing only creates a new **view**: the underlying data is shared with the original array.

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

- If you want a deep copy that does not share elements with A, use: `A[2:6].copy()`

Quiz

- What is the value of b?

```
>>> a = np.arange(4)
>>> b = a[:]
>>> a *= b
```

Arrays of Ones and of Zeros

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

>>> a = np.ones((3,4),dtype=int)
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])

>>> np.zeros((2,4))
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])

>>> np.zeros_like(a)
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Creating Random Matrices

- Array of floats uniformly drawn from the interval $[0, 1)$:

```
>>> np.random.rand(2,3)
array([[ 0.53604809,  0.54046081,  0.84399025],
       [ 0.59992296,  0.51895053,  0.09988041]])
```

- Generate floats drawn from standard normal distribution $\mathcal{N}(0, 1)$:

```
>>> np.random.randn(2,3)
array([[-1.28520219, -1.02882158, -0.20196267],
       [ 0.48258382, -0.2077209 , -2.03846176]])
```

- For repeatability of your experiment, initialize the seed at the beginning of your script:
  - `>>> np.random.seed = 0`
  - Otherwise, it will be initialized differently at every run (from system clock).
  - If you use core python random numbers, also initialize the seed there:

```
>>> import random
>>> random.seed(9001)
```

Creating Diagonal Matrices

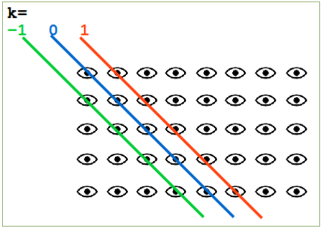
- `eye(N, M=None, k=0, dtype=float)`

  N Number of rows.

  M Number of columns.

  k Diagonal position.
  0: main diagonal, starting at $(0, 0)$
  $+n, -n$: move diagonal n up/down

  dtype Data type (e.g. `int` or `float`)



- $\Rightarrow$ To create an identity matrix (symmetric $N = M$, $k = 1$) the size N is the only argument.

Iterating

- Iterating over rows:

```
>>> for row in b:
...         print(row)
...
[0  1  2  3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

- ⇒ but (!) prefer matrix operations over iterating, if possible.

Stacking of arrays

- Vertical stacking:

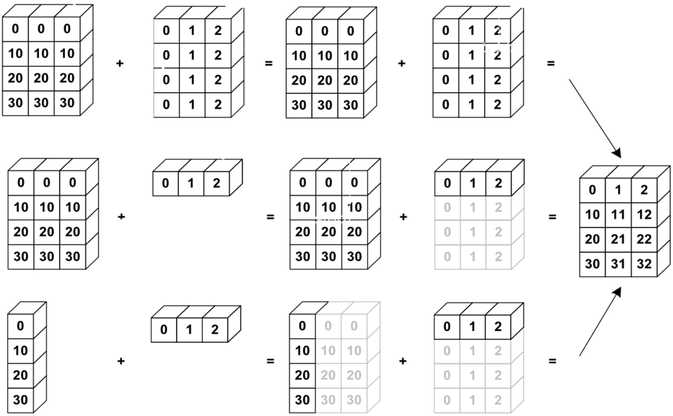
```
>>> a = np.array([[1,2],[3,4]])
>>> b = np.array([[11,22],[33,44]])
>>> np.vstack((a,b))
array([[ 1,  2],
       [ 3,  4],
       [11, 22],
       [33, 44]])
```

- Horizontal stacking:

```
>>> np.hstack((a,b))
array([[ 1,  2, 11, 22],
       [ 3,  4, 33, 44]])
```

Broadcasting

Operations can work on arrays of different sizes if Numpy can **transform** them so that they all have the **same size**!

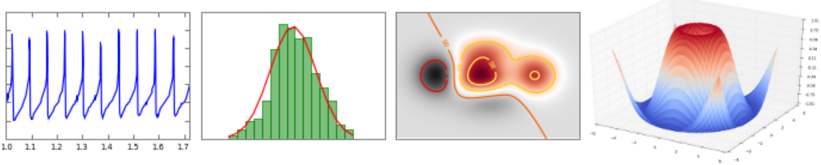


Plotting data

- Often it is a good idea to plot some properties of the data.
  - Verify expectations that you have about the data.
  - Spot trends, maxima/minima, (ir-)regularities and outliers.
  - similiratities / dissimilarities between two data sets.
- Recommended package: Matplotlib/Pyplot

Pyplot

- Plotting data and functions with Python.

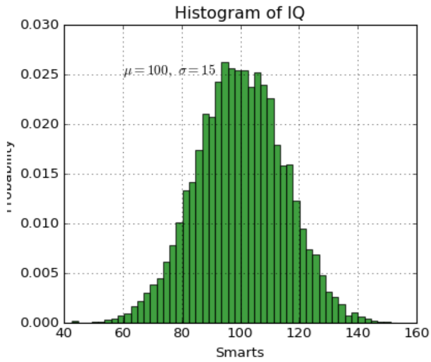


- Package of the `matplotlib` library.
- Uses `numpy` data structures
- Inspired by the `matlab` plotting commands
- Import pyplot as:

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

Example: Histograms

- Show the empirical distribution of one variable.
- Frequency of values with equally-spaced intervals.




```
x = 100 + 15 * np.random.randn(10000)
plt.hist(x, 50)
```

Ressources

- NumPy Quickstart:
  `http://docs.scipy.org/doc/numpy-dev/user/quickstart.html`
- `http://www.python-course.eu/numpy.php`