Scientific Computing in Python – NumPy, SciPy, Matplotlib

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

Based on Lecture Material by
Shawn Brown, PSC
David Grellscheid, Durham

Python for Scientific Computing?

- Pro:
  - Programming in Python is convenient
  - Development is fast (no compilation, no linking)
- Con:
  - Interpreted language is slower than compiled code
  - Lists are wasteful and inefficient for large data sets

=> NumPy to the rescue

- NumPy is also a great example for using OO-programming to hide implementation details

NumPy and SciPy

- NumPy provides functionality to create, delete, manage and operate on large arrays of typed “raw” data (like Fortran and C/C++ arrays)
- SciPy extends NumPy with a collection of useful algorithms like minimization, Fourier transform, regression and many other applied mathematical techniques
- Both packages are add-on packages (not part of the Python standard library) containing Python code and compiled code (fftpack, BLAS)

Installation of NumPy / SciPy

- Package manager of your Linux distribution
- Listed on PyPi
  → Installation via “pip install numpy scipy”
- See http://www.scipy.org/install.html for other alternatives, suitable for your platform
- After successful installation, “numpy” and “scipy” can be imported like other packages:

```
import numpy as np
import scipy as sp
```

The Basic Data Structure in NumPy

- The essential component of NumPy is the “**array**”, which is a container similar to the C++ std::array, but more powerful and flexible
- Data is stored “raw” and all elements of one array have to have the same type (efficient!)
- Data access similar to Python list:

```
>>> a = np.array([1, 4, 9, 16], np.float32)
>>> print(a[0],a[-1])
(1.0, 16.0)
>>> a
array([ 1.,  4.,  9., 16.], dtype=float32)
```

NumPy Data Types

- Numpy supports a larger number of data types, and similar to compiled languages, you can specify how many bits are used, e.g.: bool, int, int8, int16, uint32, uint64, float32, float64, complex64, complex128

```
>>> a = np.array([0,2,3,4],np.complex128)
>>> a
array([ 0.+0.j,  2.+0.j,  3.+0.j,  4.+0.j])
```

```
>>> a = np.array([0,2,3,4],dtype=np.int8)
>>> a[1] += 128
>>> print (a[1])
-126
```

Multi-dimensional Arrays

- Multi-dimensional arrays are like lists of lists:

```
>>> b = np.array([[0,1,-1],[2,3,4]],np.int8)
>>> b
array([[ 0,  1, -1],
       [ 2,  3,  4]], dtype=int8)
>>> b.shape
(2, 3)
>>> b[1][0]
2
>>> b[0,1]
1
```

Reshaping Arrays

- Same as in Fortran, arrays can be recast into different shapes, while data remains in place:

```
>>> a = np.array(range(10),np.float64)
>>> a
Array([0.,1.,2.,3.,4.,5.,6.,7.,8.,9.])
>>> b = a.reshape(2,5)
>>> a
array([0.,1.,2.,3.,4.,5.,6.,7.,8.,9.])
>>> b
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
```

Array Assignments are Shallow

- Plain assignments creates a new "view" of the same data. Array copies must be explicit:

```
>>> a = np.array(range(10),np.float64)
>>> b = a.reshape(2,5)
>>> c = a.copy()
>>> a[0] = 1
>>> a
Array([1.,1.,2.,3.,4.,5.,6.,7.,8.,9.])
>>> b
array([[ 1.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> c
Array([0.,1.,2.,3.,4.,5.,6.,7.,8.,9.])
```

More Array operations

- Arrays can be filled with a single value
- Arrays can be resized (if only one reference)

```
>>> a = np.array(range(6),float)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.])
>>> a.fill(1)
>>> a
Array([ 1.,  1.,  1.,  1.,  1.,  1.])

>>> a = np.array(range(6),float)
>>> a.resize(9)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  0.,  0.,  0.])
```

More Array operations

- Multi-dimensional arrays can be transposed

```
>>> a = np.array(range(6),float).reshape(2,3)
>>> b = a.transpose()
>>> a
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
>>> b
array([[ 0.,  3.],
       [ 1.,  4.],
       [ 2.,  5.]])
```

More Array operations

- Combine multiple arrays through concatenate

```
>>> a = np.array([1,2], np.float)
>>> b = np.array([3,4,5,6], np.float)
>>> c = np.array([7,8,9], np.float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

More Array operations

- Some more ways to create arrays

```
>>> np.linspace(30,40,5)
Array([ 30. , 32.5, 35. , 37.5, 40. ])
>>> np.ones((2,3), dtype=float32)
Array([[ 1., 1., 1.],
       [ 1., 1., 1.]], dtype=float32)
>>> np.zeros(7, dtype=int)
array([0, 0, 0, 0, 0, 0, 0])
>>> a = np.array([[1, 2, 3],[4, 5, 6]])
>>> np.zeros_like(a)
array([[ 0, 0, 0],
       [ 0, 0, 0]])
```

Element-by-Element Operations

```
>>> a = np.array([1,2,3],float)
>>> b = np.array([5,2,6],float)
>>> a + b
array([ 6.,  4.,  9.])
>>> a - b
array([-4.,  0., -3.])
>>> a * b
array([  5.,   4.,  18.])
>>> b / a
array([ 5.,  1.,  2.])
>>> a % b
array([ 1.,  0.,  3.])
>>> b ** a
array([   5.,    4.,  216.])
```

Mathematical Operations

- NumPy has a large set of mathematical functions that can be applied to arrays, e.g.:

  abs, sign, sqrt, log, log10, exp, sin, cos, tan, ...

```
>>> a = np.linspace(0.3,0.6,4)
>>> print(a)
[ 0.3  0.4  0.5  0.6]
>>> np.sin(a)
array([ 0.29552021,  0.38941834,  0.47942554,
 0.56464247])
>>> np.exp(a)
array([ 1.34985881,  1.4918247 ,  1.64872127,
 1.8221188 ])
```

Reduction Operations

```
>>> a = np.array([2,4,3],dtype=np.float64)
>>> a.sum()
9.0
>>> a.prod()
24.0
>>> np.sum(a)
9.0
>>> a.mean()
3.0
>>> a.var()
0.66666666666666663
>>> a.std()
0.81649658092772603
```

Boolean Operations

- Boolean operators can be used on whole arrays and then produces an array of booleans.
- Comparisons can be used as “filters”.

```
>>> a = np.array([[6,4],[5,9]])
>>> print (a >= 6)
[[ True False]
 [False  True]]
>>> print (a[ a >= 6])
[6 9]
>>> b = a < 6
>>> print (a[b])
[4 5]
```

Linear Algebra Operations

- Operations on matrices and vectors in NumPy are very efficient because they are linked to compiled in BLAS/LAPACK code (can use MKL, OpenBLAS, ACML, ATLAS, etc.)
- => vector-vector, vector-matrix, matrix-matrix multiplication are supported with dot()
- Also available inner(), outer(), cross()

Linear Algebra Operations

```
>>> a = np.array([[0,1],[2,3]],float)
>>> b = np.array([2,3],float)
>>> c = np.array([[1,1],[4,0]],float)
>>> np.dot(b,a)
array([  6.,  11.])
>>> np.dot(a,b)
array([  3.,  13.])
>>> np.dot(a,c)
array([[  4.,   0.],
       [ 14.,   2.]])
>>> np.outer(b,b)
array([[ 4.,  6.],
       [ 6.,  9.]])
```

Linear Algebra Operations

- Several built-in linear algebra operations are located in the `linalg` submodule

```
>>> a = np.array([[4,2,0],[9,3,7],[1,2,1]],float)
>>> np.linalg.det(a)
-48.00000000000028
>>> vals,vecs = np.linalg.eig(a)
>>> vals
array([ 8.85591316,  1.9391628 , -2.79507597])
>>> vecs
array([[-0.3663565 , -0.54736745,  0.25928158],
       [-0.88949768,  0.5640176 , -0.88091903],
       [-0.27308752,  0.61828231,  0.39592263]])
```

This is Only the Beginning

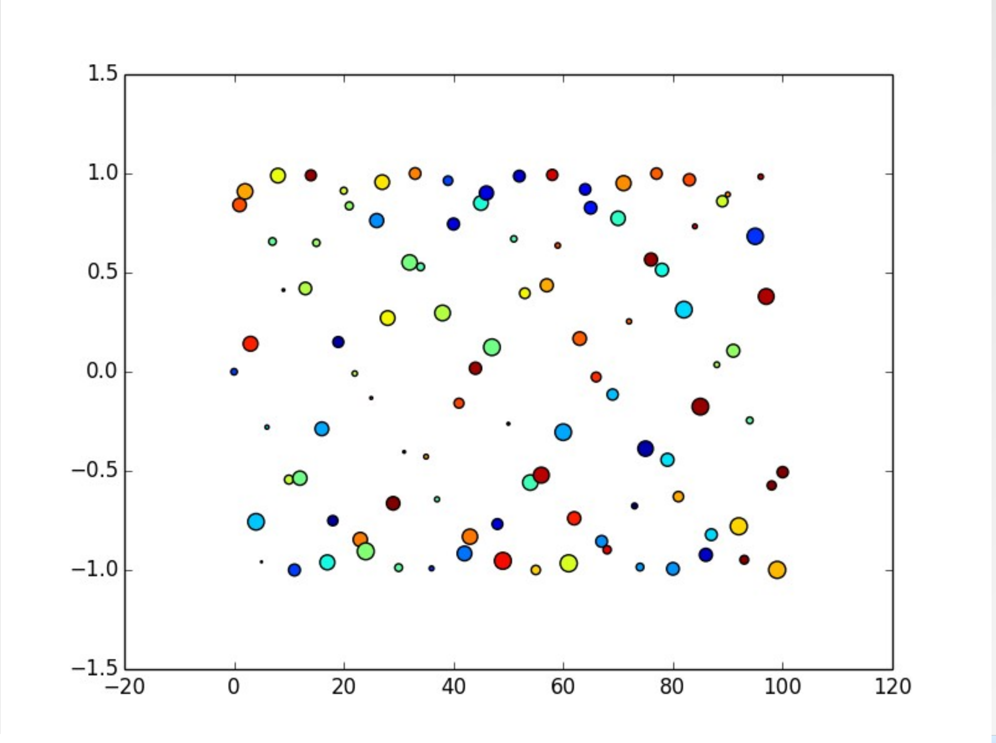
- NumPy has much more functionality:
  - Polynomial mathematics
  - Statistical computations
  - Pseudo random number generators
  - Discrete Fourier transforms
  - Size / shape / type testing of arrays
- To learn more, check out the NumPy docs at: http://docs.scipy.org/doc/

SciPy

- SciPy is built on top of NumPy and implements many specialized scientific computation tools:
  - Clustering, Fourier transforms, numerical integration, interpolations, data I/O, LAPACK, sparse matrices, linear solvers, optimization, signal processing, statistical functions, sparse eigenvalue solvers, ...

Matplotlib

- Powerful library for 2D (and some 3D) plotting
- Well designed, interactive use and scripted, common tasks easy, complex tasks possible



Matplotlib

- Example workflow for plotting with matplotlib.
- Check out: http://matplotlib.org/gallery.html

```
>>> import pylab as pl
>>> xs = pl.linspace(0,100,101)
>>> ys = pl.sin(xs)
>>> cols = pl.random(101)
>>> sizes = 100.0 * pl.random(101)
>>> pl.scatter(xs,ys,c=cols,s=sizes)
<matplotlib.collections.PathCollection object at
0x7fa0b4430ba8>
>>> pl.savefig('scatter-test.png')
```

RSS · Blog · Trac · Report Bugs · Wiki · Ask · Search ...
Online: SMC, sagenb, KAIST · SageCell or Download, Source Code
v6.5 (2015-02-17) · Language

Home Tour Help Library Download Development Links

SageMath is a free open-source mathematics software system licensed under the GPL. It builds on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more. Access their combined power through a common, Python-based language or directly via interfaces or wrappers.
→ Tour, Tutorial, Documentation

Mission: *Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab.*

Join **SageMathCloud**™ online service for free.
Work with SageMath, IPython, Python, R, GAP, M2, ROOT, Julia and more in the Cloud.

Use SageMath Online
other: Sagenb, KAIST, SageMathCell

Download 6.5
Changelog · Source 6.5 · Packages · Git

Help/Documentation
Video · Lists · Tutorial · FAQ · **Ask**

Feature Tour
Quickstart · Research · Graphics

Library
Testimonials · Books · Publications · Press Kit

Search

Scientific Computing in Python – NumPy, SciPy, Matplotlib

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu