Introduction to Numpy and Scipy

Yang Liu
Associate Research Scientist
High Performance Research Computing
Texas A&M University

June 6, 2017

Numpy and Scipy

Numpy: fundamental package for scientific computing in Python. It provides

- Multidimensional array object and various derived objects (matrices, etc)
- Routines for fast operations on array: mathematical, shape manipulation, sorting, basic linear algebra, etc.

Scipy: a collection of mathematical algorithms and functions built on Numpy. It is organized into subpackages covering various computing domains

- cluster: clustering algorithm
- fftpack: fast Fourier Transform routines
- linalg: linear algebra
- optimize: optimzation and root-finding routines
- sparse: sparse matrices and associated routines
- ...

Numpy Array

A Numpy array (ndarray class) is a table of elements of the same type

- Axis: dimensions
- Rank: number of axis (dimensions)

An ndarray object has important attributes

- ndarray.ndim: rank
- ndarray.size: total number elements
- ndarray.shape: a tuple of integers indicating the size of the array in each dimension
- ndarray.dtype: the type of the elements

```
[ [ 1, 2, 3],
  [ 4, 5, 6] ]

Axis 1: [1, 2, 3]
Axis 2: [4, 5, 6]
Rank: 2
```

```
[ [ 1, 2, 3],
  [ 4, 5, 6] ]

dim: 2
size: 6
shape: (3,3)
dtype: integer
```

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a.ndim
2
>>> a.size
6
>>> a.shape
(2, 3)
>>> a.dtype
dtype('int64')
```

Array of Zeros/Ones

- Function ‘zeros’ creates an array full of zeros
- Function ‘ones’ creates an array full of ones
- Data type by default is float64, and can be set when creating arrays from funciton ‘zeros’ and ‘ones’.

```
>>> a = np.zeros((3,4))
>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> b = np.ones((2, 3, 4))
>>> b
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.]]])
>>> a = np.zeros((3,4), dtype=np.int16)
>>> a
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int16)
```

Array of Numbers in a Sequence: arange and linspace

- arange(start, end, step)
- linespace(start, end, num_elements)
- Due to finite floating point precision, it is difficult to predict the number of elements for arange

```
>>> np.arange(1, 10, 0.3)
array([ 1. ,  1.3,  1.6,  1.9,  2.2,  2.5,  2.8,  3.1,  3.4,  3.7,  4. ,
        4.3,  4.6,  4.9,  5.2,  5.5,  5.8,  6.1,  6.4,  6.7,  7. ,  7.3,
        7.6,  7.9,  8.2,  8.5,  8.8,  9.1,  9.4,  9.7])
>>> np.linspace(1, 10, 30)
array([  1.        ,   1.31034483,   1.62068966,   1.93103448,
         2.24137931,   2.55172414,   2.86206897,   3.17241379,
         3.48275862,   3.79310345,   4.10344828,   4.4137931 ,
         4.72413793,   5.03448276,   5.34482759,   5.65517241,
         5.96551724,   6.27586207,   6.5862069 ,   6.89655172,
         7.20689655,   7.51724138,   7.82758621,   8.13793103,
         8.44827586,   8.75862069,   9.06896552,   9.37931034,
         9.68965517,  10.        ])
```

Random Array

Numpy random.random generates an array of random floating numbers between 0 and 1

```
>>> a = np.random.random((2, 3))
>>> print(a)
[[ 0.02336401  0.19639539  0.338913  ]
 [ 0.79757397  0.16154741  0.06970195]]
```

Operations on Array

Arithmetic operations on array are element-wise

```
>>> a = np.array([10, 20, 30])
>>> b = np.random.random((3))
>>> print(a)
[10 20 30]
>>> print(b)
[ 0.77840463  0.89514907  0.41390085]
>>> a + b
array([ 10.77840463,  20.89514907,  30.41390085])
>>> a - b
array([  9.22159537,  19.10485093,  29.58609915])
>>> a < b
array([False, False, False], dtype=bool)
>>> np.sin(a)
array([-0.54402111,  0.91294525, -0.98803162])
>>> np.sqrt(a)
array([ 3.16227766,  4.47213595,  5.47722558])
```

Matrix Product: dot

Numpy ‘dot’ function creates a matrix product.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.ones((2,2))
>>> print(a)
[[1 2]
 [3 4]]
>>> print(b)
[[ 1.  1.]
 [ 1.  1.]]
>>> a.dot(b)
array([[ 3.,  3.],
       [ 7.,  7.]])
>>> np.dot(a, b)
array([[ 3.,  3.],
       [ 7.,  7.]])
```

Array Indexing and Slicing

- An element of an array a can be referred as a[i1, i2, ..] (index starts from 0—c style)
- A slicing start:end:step refers to one or more elements

```
>>> a = np.arange(10)
>>> a[0]
0
>>> a[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 10 is out of bounds for axis 0 with size 10
>>> b = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
>>> b[1, 3]
9
>>> b[1, ::2]
array([ 6,  8, 10])
>>> b[1, :4:2]
array([6, 8])
>>> b[1, 1:4:2]
array([7, 9])
```

Reshape Array

The shape of an array a can be changed by

- a.shape=(…): a is changed
- a.resize(…): a is changed
- a.reshape(…): a does not change

```
>>> a = np.arange(24)
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> b = a.reshape(3,8)
>>> print(b)
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]]
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> a.shape = (2, 12)
>>> print(a)
[[ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]]
>>> a.resize(4, 6)
>>> print(a)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

Stacking Arrays

Two arrays a and b can be stacked together either horizontally or vertically

- np.vstack(a,b): a and b are stacked vertically (first axes)
- np.hstack(a,b): a and b are stacked horizontally (second axes)

```
>>> a = np.arange(6).reshape(2,3)
>>> print(a)
[[0 1 2]
 [3 4 5]]
>>> b = np.ones((2,3))
>>> print(b)
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
>>> np.hstack((a, b))
array([[ 0.,  1.,  2.,  1.,  1.,  1.],
       [ 3.,  4.,  5.,  1.,  1.,  1.]])
>>> np.vstack((a, b))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Splitting an Array

An array a can be splited into several smaller arrays

- np.hsplit(a, 3): split a into 3 smaller arrays horizontally (second axis)
- np.vsplit(a, 3): split a into 3 smaller arrays vertically (first axis)

```
>>> a = np.arange(24).reshape(2,12)
>>> print(a)
[[ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]]
>>> x, y, z = np.hsplit(a, 3)
>>> print(x)
[[ 0  1  2  3]
 [12 13 14 15]]
>>> print(y)
[[ 4  5  6  7]
 [16 17 18 19]]
>>> print(z)
[[ 8  9 10 11]
 [20 21 22 23]]
>>> x, y = np.vsplit(a, 2)
>>> print(x)
[[ 0  1  2  3  4  5  6  7  8  9 10 11]]
>>> print(y)
[[12 13 14 15 16 17 18 19 20 21 22 23]]
```

No Copy of an Array for Assignment

Assignment “a = b” does not copy array b. Changing a will actually change b.

```
>>> a = np.arange(12)
>>> b = a
>>> b is a
True
>>> id(a)
139978214113328
>>> id(b)
139978214113328
>>> b.shape = (3,4)
>>> print(a)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

View (Shallow Copy) of an Array

For an array a, ‘a.view()’ creates a new array object which shares the same data of array a. Note that slicing of an array returns a view of that array.

```
>>> a = np.arange(12)
>>> c = a.view()
>>> print(c)
[ 0  1  2  3  4  5  6  7  8  9 10 11]
>>> c.resize((3,4))
>>> print(c)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11]
>>> c[1, 2] = -3
>>> print(a)
[ 0  1  2  3  4  5 -3  7  8  9 10 11]
>>> b = c[:, 1:2]
>>> print(b)
[[1]
 [5]
 [9]]
>>> b[:] = -10
>>> print(c)
[[  0 -10   2   3]
 [  4 -10  -3   7]
 [  8 -10  10  11]]
```

Deep Copy of an Array

For an array a, ‘a.copy()’ creates a new object which has its own data (not sharing with array a).

```
>>> a = np.arange(24)
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
>>> d = a.copy()
>>> d is a
False
>>> d[5] = -10
>>> print(d)
[  0   1   2   3   4 -10   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23]
>>> print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

Overview of Numpy Functions and Methods

- Array creation: arange, array, copy, ...
- Manipulcations: vstack, vsplit, reshape, transpose, ...
- Operations: sum, fill, ...
- Basic statistics: mean, std, var, ...
- Basic linear algebra: cross, dot, outer, vdot, ...
- ...

Indexing with Arrays of Indices for One Dimensional Array

For a one dimensional array a, an array array_indices can be used to select elements from array a

- Each element is array_indices is an index of array a
- The new array has the same shape as array_indices

```
>>> a = np.arange(12) + 10
>>> print(a)
[10 11 12 13 14 15 16 17 18 19 20 21]
>>> array_indices = np.array([0, 0, 3, 4, 3])
>>> a[array_indices]
array([10, 10, 13, 14, 13])
>>> array_indices = np.array([[0, 0, 2], [1, 3, 4]])
>>> a[array_indices]
array([[10, 10, 12],
       [11, 13, 14]])
```

Indexing with Arrays of Indices for Two Dimensional Array

For a two dimensional array a, an array array_indices can be used to select elements from array a

- Each element is array_indices is the first dimension index of array a
- The new array has the same shape as array_indices

```
>>> a = np.arange(24).reshape(4,6)
>>> print(a)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
>>> array_indices = np.array([[0, 0, 2], [1, 3, 3]])
>>> b = a[array_indices]
>>> print(b)
[[[ 0  1  2  3  4  5]
  [ 0  1  2  3  4  5]
  [12 13 14 15 16 17]]

 [[ 6  7  8  9 10 11]
  [18 19 20 21 22 23]
  [18 19 20 21 22 23]]]
>>> b.shape
(2, 3, 6)
```

Indexing with Arrays of Indices: Multidimensional Indices

More than one dimension can be used to create a new array from array a. For example

- Each element in row_indices is the first dimension index of a
- Each element in column_indices is the second dimension index of a
- row_indices and column_indices can be put together (e.g., as a list)

```
>>> a = np.arange(24).reshape(4,6)
>>> print(a)
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
>>> row_indices = np.array([[0, 0], [1, 3]])
>>> column_indices = np.array([[2, 2], [0, 4]])
>>> a[row_indices, column_indices]
array([[ 2,  2],
       [ 6, 22]])
>>> a[[row_indices, column_indices]]
array([[ 2,  2],
       [ 6, 22]])
```

Indexing with Boolean Arrays of Indices:

Boolean array b can be used to select elements of a

- a[b]: returns the selected element of a
- a[b] = c: set the selected elements in a to be c

```
>>> a = np.random.random(12).reshape(3,4)
>>> print(a)
[[ 0.7574143   0.5185174   0.10785612  0.952208  ]
 [ 0.00364275  0.33689668  0.44678831  0.56410663]
 [ 0.27069099  0.91649263  0.99866993  0.1147783 ]]
>>> b = a > 0.5
>>> b
array([[ True,  True, False,  True],
       [False, False, False,  True],
       [False,  True,  True, False]], dtype=bool)
>>> a[b]
array([ 0.7574143 ,  0.5185174 ,  0.952208  ,  0.56410663,  0.91649263,
        0.99866993])
>>> a[b] = 1
>>> print(a)
[[ 1.          1.          0.10785612  1.        ]
 [ 0.00364275  0.33689668  0.44678831  1.        ]
 [ 0.27069099  1.          1.          0.1147783 ]]
```

Broadcasting

- Broadcasting in Numpy allows operations on arrays of different shapes.
- Numpy compares the shapes of two arrays element-wise by starting with the trailing dimensions and working its way backward. Two dimensions are compatible if (1) they are equal or (2) one of them is 1
- Numpy copies the array along the dimension of size 1

Examples of compatible arrays

(1) A 2 x 3 x 3
B 2 x 3 x 1

(2) A 2 x 3 x 3
B 3

```
>>> a = np.arange(18).reshape(2, 3, 3)
>>> print(a)
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]]

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]]
>>> b = np.arange(6).reshape(2,3,1)
>>> print(b)
[[[0]
  [1]
  [2]]

 [[3]
  [4]
  [5]]]
>>> a + b
array([[[ 0,  1,  2],
        [ 4,  5,  6],
        [ 8,  9, 10]],

       [[12, 13, 14],
        [16, 17, 18],
        [20, 21, 22]]])
```

Linear Algebra for Numpy

Numpy provides basic linear algebra functions and methods.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> print(a)
[[1 2]
 [3 4]]
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> np.trace(a)
5
>>> b = np.array([[10, 20], [30, 40]])
>>> print(b)
[[10 20]
 [30 40]]
>>> np.dot(a, b)
array([[ 70, 100],
       [150, 220]])
```

Linear Algebra for Scipy

Scipy.linalg contains all the function in numpy.linalg, plus some other more advanced ones. So scipy.linalg should be used unless you do not want the dependency on scipy.

- Scipy.linalg is always compiled with BLAS/LAPACK support (faster), while this is optional for numpy
- All of the ral lapack and blas libraries are available for use

Example

$x + 3y = 10$

$2x + 5y = 20$

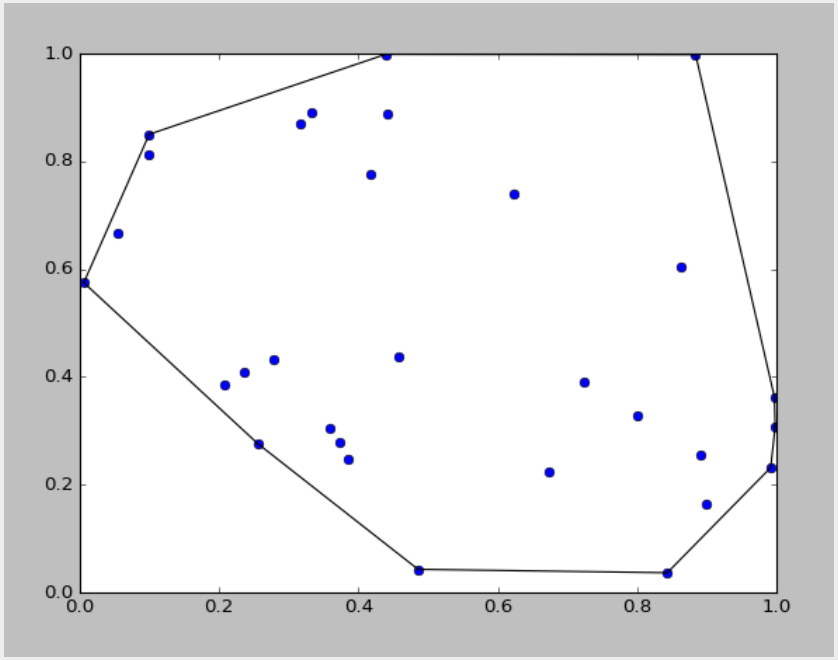
Note: Linalg.solve(A, b) is faster than linalg.inv(A).dot(b)

```
>>> A = np.array([[1, 3], [2, 5]])
>>> print(A)
[[1 3]
 [2 5]]
>>> b = np.array([[10], [20]])
>>> print(b)
[[10]
 [20]]
>>> linalg.solve(A,b)
array([[ 10.],
       [  0.]])
```

Scipy Spactial Data Structures and Algorithms

Scipy.spatial can compute triangulations, Voronoi diagrams, and convex hulls of a set of points. It also contians KDTree and utilities for distnace computations in various metrics.

```
>>> from scipy.spatial import ConvexHull
>>> points = np.random.rand(30, 2)
>>> hull = ConvexHull(points)
>>> import matplotlib.pyplot as plt
>>> plt.plot(points[:, 0], points[:, 1], 'o')
[<matplotlib.lines.Line2D object at 0x7f396f82b080>]
>>> for simplex in hull.simplices:
...     plt.plot(points[simplex, 0], points[simplex, 1], 'k-')
...
[<matplotlib.lines.Line2D object at 0x7f396f82b940>]
[<matplotlib.lines.Line2D object at 0x7f396f82bbe0>]
[<matplotlib.lines.Line2D object at 0x7f396f82f438>]
[<matplotlib.lines.Line2D object at 0x7f396f82fc50>]
[<matplotlib.lines.Line2D object at 0x7f396f8354a8>]
[<matplotlib.lines.Line2D object at 0x7f396f835cc0>]
[<matplotlib.lines.Line2D object at 0x7f396f83a518>]
[<matplotlib.lines.Line2D object at 0x7f396f83ad30>]
[<matplotlib.lines.Line2D object at 0x7f396f840588>]
[<matplotlib.lines.Line2D object at 0x7f396f840da0>]
>>> plt.show()
```



References

- Numpy User Guide: https://docs.scipy.org/doc/numpy/numpy-user-1.12.0.pdf
- Scipy Reference Guide: https://docs.scipy.org/doc/scipy/scipy-ref-0.19.0.pdf