NUMERIC

PYTHON

(NUMPY)

Overview

- add-on module for for scientific computing
- built around ndarray()
- (multi) dimensional arrays of objects
- vectorized operations

Arrays

- similar to lists
- but: all elements of the same type
- support indexing/slicing
- multi-dimensional (through reshaping)
- efficient, vectorized operations

Lists vs. Arrays

```
> from random import random
> import numpy as np
> n = 1000
> x_list = [ random() for i in range(n)]
> y_list = [ random() for i in range(n)]
> timeit z_list = [ x_list[i] + y_list[i]
                           for i in range(n)]
91.8 ms  2.46 ms per loop (mean  std. dev.
of 7 runs, 10000 loops each)
> x_vector = np.array(x)
> y_vector = np.array(y)
> timeit z_vector = x_vector + y_vector
922 ns  12.7 ns per loop (mean  std. dev.
of 7 runs, 1000000 loops each)
```

- vectorized operations are faster
- avoid iterations

Lists vs. Arrays (cont'd)

```
> from random import random
> import numpy as np
> n = 1000
> x_list = [ random() for i in range(n)]
> x_list.__sizeof__()
9000
> x_vector = np.array(x)
> x_vector.__sizeof__()
8096
```

- numpy arrays are more memory efficient

Basic Constructors

```
> import numpy as np
> np.arange(0, 5, 1)
array([ 0,  1,  2,  3,  4])
> np.zeros(5, dtype=float)
array([ 0.,  0.,  0.,  0.,  0.])
> np.linspace(0, 2, 6)
array([ 0. ,  0.4,  0.8,  1.2,  1.6,  2. ])
> np.eye(2)
array([[ 1.,  0.],
       [ 0.,  1.]])
> np.diag([5, 10])
array([[ 5,  0],
       [ 0, 10]])
```

Universal Functions (ufunc)

- ufunc perform element-wise operations
- allow to apply scalar arithmetic to vectors

```
> import numpy as np
> x = np.linspace(0, 1, 5)
array([ 0.  , 0.25, 0.5, 0.75, 1. ])
> y = 2*x
array([ 0.  , 0.5, 1., 1.5, 2.])
> z = x/7
array([ 0.  , 0.0625, 0.125 , 0.1875, 0.25 ])
> w = np.sin(x)
array([ 0., 0.24740396, 0.47942554,
                    0.68163876, 0.84147098])
```

Multidimensional Arrays

```
> import numpy as np
> v = np.array([[1,2,3], [4,5,6], [7,8,9]])
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
> v.shape
(3, 3)
> v.ndim
2
> v.dtype
dtype('int32')
```

Broadcasting

- can apply functions to arrays of different sizes

```
> x = np.array([1,2,3])
> y = np.array([4,5,6])
> z = x + y
> z
array([5, 7, 9])
> w = 2 + y
> w
array([6, 7, 8])
```

Numeric Python

- focus on n-dimensional arrays (vectors and matrices)
- all objects of the same type
- vectorized (more efficient than Python objects)
- many math and statistical functions

Creating a Numpy Vector

```
> import numpy as np
> x = np.array([1,2,3,4,5])
> type(x)
numpy.ndarray
> x.ndim
1
> x.shape
(5,)
> x.size
5
```

Creating a Numpy Vector (cont'd)

- implicit typing

```
> import numpy as np
> x = np.array([1,2,3,4,5])
> x.dtype
int32
```

- explicit typing

```
> x = np.array([1,2,3,4,5], dtype=float)
> x.dtype
float64
```

Creating Sequences

- evenly spaced with step 1

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

- evenly spaced with step 2

```
> x = np.arange(0, 10, step=2)
array([0, 2, 4, 6, 8])
```

- evenly spaced, specified number of elements

```
> x = np.linspace(start=0,stop=10,
                   num=5)
array([0. , 2.5, 5. , 7.5, 10.])
```

Creating Sequences (cont'd)

- repeat value 1 six times

```
> x = np.ones(shape=10)
> array([ 1.,1.,1.,1.,1.,1.])
```

- repeat value 3 ten times

```
> x = np.full(shape=10, fill_value=3)
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3])
```

Creating from a List

```
> x = [1,2,3,4,5]
> type(x)
<class ’list’>
> y = np.asarray(x, dtype=float)
> type(y)
<class ’numpy.ndarray’>
```

- can append (like in a list)

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

Arrays Manipulations

- can delete at some position like in a list

```
> x = np.array([1,2,3,4, 5])
> y = np.delete(x,2)
> y
array([1,2,4,5])
```

- add extra zeros

```
> x = np.array([1,2,3,4,5])
> x.resize(new-shape=7)
> x
array[1,2,3,4,5,6,7]
```

Arrays Manipulations (cont'd)

- can concatenate (like lists)

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

> y = np.array([6,7,8,9,10])

> z = np.append(x,y)

> z

array([1,2,3,4,5,6,7,8,9,10])
```

Sorting and Searching

- get maximum value

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

- index of the maximum value

```
> np.argmax(x)
2
```

- sort

```
> np.sort(x)
> array([1, 2, 3, 4, 5])
```

Sorting and Searching (cont'd)

- indices for sorting

```
> np.argsort(x)
array([4, 3, 1, 0, 2], dtype=int64)
```

- get unique elements

```
> y = np.array([1,2,2,3,4,4])
> z = np.unique(y)
> z
array([1, 2, 3, 4])
```

Data Processing With Arrays

- evaluate $f(x) = \sqrt{x^2 + y^2}$
- interested in some range of x and y
- can create a mesh

```
> x_points = np.arange(-2, 3, 1)
array([-2, -1,  0,  1,  2])
> y_points = np.arange(-1, 2, 1)
array([-1,  0,  1])
> xs, ys = np.meshgrid(x_points, y_points)
```

Data Processing Wit Arrays (cont'd)

```
> xs
array([[-2, -1,  0,  1,  2],
       [-2, -1,  0,  1,  2],
       [-2, -1,  0,  1,  2]])
> ys
array([[-1, -1, -1, -1, -1],
       [ 0,  0,  0,  0,  0],
       [ 1,  1,  1,  1,  1]])
> z = np.round(np.sqrt(xs**2 + ys**2), 2)
array([[ 2.24,  1.41,  1.  ,  1.41,  2.24],
       [ 2.  ,  1.  ,  0.  ,  1.  ,  2.  ],
       [ 2.24,  1.41,  1.  ,  1.41,  2.24]])
```

Multi-Dimensional Arrays

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> import numpy as np
> X = np.arange(1,37).reshape(6,6)
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]])
```

Transposing Arrays

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> import numpy as np
> X = np.arange(1,37).reshape(6,6)
> Y = X.transpose()
array([[ 1,  7, 13, 19, 25, 31],
       [ 2,  8, 14, 20, 26, 32],
       [ 3,  9, 15, 21, 27, 33],
       [ 4, 10, 16, 22, 28, 34],
       [ 5, 11, 17, 23, 29, 35],
       [ 6, 12, 18, 24, 30, 36]])
```

Reshaping Arrays

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> import numpy as np
> X = np.arange(1,37).reshape(6,6)
> Y = X.reshape(4, 9)
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34, 35, 36]])
```

Dot (Inner) Product

$$\begin{aligned}(x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n) \\ = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n\end{aligned}$$

```
> x = np.array([1,2,3])
array([1, 2, 3])
> y = np.array([3,9, 2])
array([3, 9, 2])
> z = np.dot(x, y)
27
```

- geometric meaning

Linear Algebra

```
> x = np.array([[1,2,3], 1,4,2],[1,10, -5]])
array([[ 1,  2,  3],
       [ 1,  4,  2],
       [ 1, 10, -5]])
> x_inv = np.linalg.inv(x)
array([[ 5.   , -5.   ,  1.   ],
       [-0.875,  1.   , -0.125],
       [-0.75 ,  1.   , -0.25 ]])
> y = x.dot(x_inv)  # or  y = np.dot(x, x_inv)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Applying Numpy Functions to Rows and Columns

```
> x = np.array([[1,2,3], [1,4,2], [1,10, -5]])
array([[ 1,  2,  3],
       [ 1,  4,  2],
       [ 1, 10, -5]])
> sum_columns = np.sum(x, axis = 0)
array([ 3, 16,  0])
> sum_rows = np.sum(x, axis = 1)
array([6, 7, 6])
```

Individual Element

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[1,2]
9
> Y.shape
()
```

- another way: $Y = X[1][2]$

A Single Row

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[2, :]
array([13, 14, 15, 16, 17, 18])
> Y.shape
(6,)
```

- other possibilities:

```
> Y = X[2, ]
> Y = X[2]
```

A List of Rows

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[[1,3], :]
array([[ 7,  8,  9, 10, 11, 12],
       [19, 20, 21, 22, 23, 24]])
> Y.shape
(2, 6)
```

- other possibilities:

```
> Y = X[[1,3], ]
> Y = X[[1,3]]
```

A Range of Rows

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[1: , : ]
array([[ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]])
> Y.shape
(5, 6)
```

- other possibilities:

```
> Y = X[1:, ] ;    Y = X[1:]
```

A (sub)Range of Rows

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[1::2 , : ]
array([[ 7,  8,  9, 10, 11, 12],
       [19, 20, 21, 22, 23, 24],
       [31, 32, 33, 34, 35, 36]])
> Y.shape
(3, 6)
```

- other possibilities:

```
> Y = X[1::2, ]
> Y = X[1::2]
```

A Single Column

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[:, 4]
array([ 5, 11, 17, 23, 29, 35])
> Y.shape
(6,)
```

A Single Column (cont'd)

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[:, [4]]
array([[ 5],
       [11],
       [17],
       [23],
       [29],
       [35]])
> Y.shape
(6, 1)
```

A List of Columns

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[:, [3,4]]
array([[ 4,  5],
       [10, 11],
       [16, 17],
       [22, 23],
       [28, 29],
       [34, 35]])
> Y.shape
(6, 2)
```

A Range of Columns

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[:, 1:6:2]
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18],
       [20, 22, 24],
       [26, 28, 30],
       [32, 34, 36]])
> Y.shape
(6, 3)
```

Row/Column Slicing

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[2: , 1:  ]
array([[14, 15, 16, 17, 18],
       [20, 21, 22, 23, 24],
       [26, 27, 28, 29, 30],
       [32, 33, 34, 35, 36]])
> Y.shape
(4, 5)
```

Row/Column Slicing (cont'd)

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[1:3, 1:3]
array([[ 8,  9],
       [14, 15]])
> Y.shape
(2, 2)
```

Row/Column Slicing (cont'd)

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[1::2, 1:6:2]
array([[ 8, 10, 12],
       [20, 22, 24],
       [32, 34, 36]])
> Y.shape
(3, 3)
```

Reverse Slicing

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[1::2, 5::-2]
array([[12, 10,  8],
       [24, 22, 20],
       [36, 34, 32]])
> Y.shape
(3, 3)
```

Reversing Rows/Cols

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

```
> Y = X[ ::-1, ::-1]
array([[36, 35, 34, 33, 32, 31],
       [30, 29, 28, 27, 26, 25],
       [24, 23, 22, 21, 20, 19],
       [18, 17, 16, 15, 14, 13],
       [12, 11, 10,  9,  8,  7],
       [ 6,  5,  4,  3,  2,  1]])
> Y.shape
(6, 6)
```

Statistical Functions

- many functions available
- much more extensive set in scipy

```
> x = np.array(range(10))
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
> mean = np.mean(x)
4.5
> std = np.std(x)
2.8722813232690143
> var = np.var(x)
8.25
```

Statistical Functions (cont'd)

```
> median = np.percentile(x, 50)
4.5
> cum_sum = np.cumsum(x)
array([ 0,1,3,6,10,15,21,28,36,45], dtype=int32)
```