NUMPY

Numpy – fast array interface

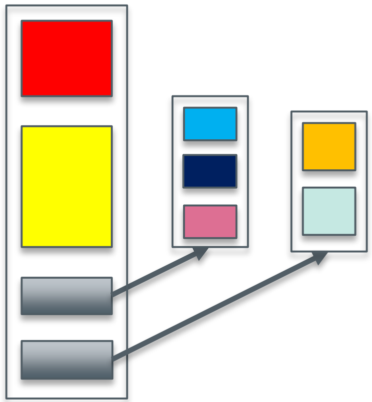
- Standard Python is not well suitable for numerical computations
  - lists are very flexible but also slow to process in numerical computations
- Numpy adds a new **array** data type
  - static, multidimensional
  - fast processing of arrays
  - some linear algebra, random numbers

Numpy arrays

- All elements of an array have the same type
- Array can have multiple dimensions
- The number of elements in the array is fixed, shape can be changed

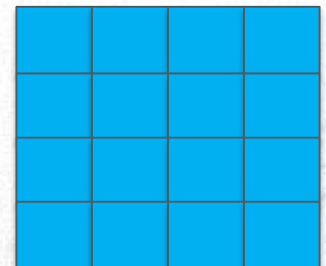
Python list vs. NumPy array

Python list



Memory layout

NumPy array



Memory layout

Creating numpy arrays

- From a list:

```
>>> import numpy as np
>>> a = np.array((1, 2, 3, 4), float)
>>> a
array([ 1.,  2.,  3.,  4.])
>>>
>>> list1 = [[1, 2, 3], [4,5,6]]
>>> mat = np.array(list1, complex)
>>> mat
array([[ 1.+0.j,  2.+0.j,  3.+0.j],
       [ 4.+0.j,  5.+0.j,  6.+0.j]])
>>> mat.shape
(2, 3)
>>> mat.size
6
```

Creating numpy arrays

- More ways for creating arrays:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>
>>> b = np.linspace(-4.5, 4.5, 5)
>>> b
array([-4.5 , -2.25,  0.  ,  2.25,  4.5 ])
>>>
>>> c = np.zeros((4, 6), float)
>>> c.shape
(4, 6)
>>>
>>> d = np.ones((2, 4))
>>> d
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

Indexing and slicing arrays

- Simple indexing:

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat[0,2]
3
>>> mat[1,-2]
>>> 5
```

- Slicing:

```
>>> a = np.arange(5)
>>> a[2:]
array([2, 3, 4])
>>> a[:-1]
array([0, 1, 2, 3])
>>> a[1:3] = -1
>>> a
array([0, -1, -1, 3, 4])
```

Indexing and slicing arrays

- Slicing is possible over all dimensions:

```
>>> a = np.arange(10)
>>> a[1:7:2]
array([1, 3, 5])
>>>
>>> a = np.zeros((4, 4))
>>> a[1:3, 1:3] = 2.0
>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  2.,  2.,  0.],
       [ 0.,  2.,  2.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Views and copies of arrays

- Simple assignment creates references to arrays
- Slicing creates “views” to the arrays
- Use copy() for real copying of arrays

example.py

```
a = np.arange(10)
b = a            # reference, changing values in b changes a
b = a.copy()     # true copy

c = a[1:4]       # view, changing c changes elements [1:4] of a
c = a[1:4].copy() # true copy of subarray
```

Array manipulation

- reshape : change the shape of array

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

- ravel : flatten array to 1-d

```
>>> mat.ravel()
array([1, 2, 3, 4, 5, 6])
```

Array manipulation

- concatenate : join arrays together

```
>>> mat1 = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat2 = np.array([[7, 8, 9], [10, 11, 12]])
>>> np.concatenate((mat1, mat2))
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> np.concatenate((mat1, mat2), axis=1)
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

- split : split array to N pieces

```
>>> np.split(mat1, 3, axis=1)
[array([[1],
       [4]]), array([[2],
       [5]]), array([[3],
       [6]])]
```

Array operations

- Most operations for numpy arrays are done element-wise
  - +, -, *, /, **

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
>>> a + b
array([ 3.,  4.,  5.])
>>> a * a
array([ 1.,  4.,  9.])
```

Array operations

- Numpy has special functions which can work with array arguments
  - sin, cos, exp, sqrt, log, ...

```
>>> import numpy, math
>>> a = numpy.linspace(-math.pi, math.pi, 8)
>>> a
array([-3.14159265, -2.24399475, -1.34639685, -0.44879895,
        0.44879895,  1.34639685,  2.24399475,  3.14159265])
>>> numpy.sin(a)
array([ -1.22464680e-16,  -7.81831482e-01,  -9.74927912e-01,
        -4.33883739e-01,   4.33883739e-01,   9.74927912e-01,
         7.81831482e-01,   1.22464680e-16])
>>>
>>> math.sin(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: only length-1 arrays can be converted to Python scalars
```

Vectorized operations

- for loops in Python are slow
- Use "vectorized" operations when possible
- Example: difference

example.py

```
# brute force using a for loop
arr = np.arange(1000)
dif = np.zeros(999, int)
for i in range(1, len(arr)):
    dif[i-1] = arr[i] - arr[i-1]

# vectorized operation
arr = np.arange(1000)
dif = arr[1:] - arr[:-1]
```

  – for loop is ~80 times slower!

Broadcasting

- If array shapes are different, the smaller array may be broadcasted into a larger shape

```
>>> from numpy import array
>>> a = array([[1,2],[3,4],[5,6]], float)
>>> a
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> b = array([[7,11]], float)
>>> b
array([[  7.,  11.]])
>>>
>>> a * b
array([[  7.,  22.],
       [ 21.,  44.],
       [ 35.,  66.]])
```

Advanced indexing

- Numpy arrays can be indexed also with other arrays (integer or boolean)

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

- Boolean “mask” arrays

```
>>> m = x > 7
>>> m
array([ True,  True,  True, False, False, ...
>>> x[m]
array([10,  9,  8])
```

- Advanced indexing creates copies of arrays

Masked arrays

- Sometimes datasets contain invalid data (faulty measurement, problem in simulation)
- Masked arrays provide a way to perform array operations neglecting invalid data
- Masked array support is provided by numpy.ma module

Masked arrays

- Masked arrays can be created by combining a regular numpy array and a boolean mask

```
>>> import numpy.ma as ma
>>> x = np.array([1, 2, 3, -1, 5])
>>>
>>> m = x < 0
>>> mx = ma.masked_array(x, mask=m)
>>> mx
masked_array(data = [1 2 3 -- 5],
             mask = [False False False True False],
       fill_value = 999999)
>>> x.mean()
2.0
>>> mx.mean()
2.75
```

I/O with Numpy

- Numpy provides functions for reading data from file and for writing data into the files
- Simple text files
  - numpy.loadtxt
  - numpy.savetxt
  - Data in regular column layout
  - Can deal with comments and different column delimiters

Random numbers

- The module numpy.random provides several functions for constructing random arrays
  - random: uniform random numbers
  - normal: normal distribution
  - poisson: Poisson distribution
  - …

```
>>> import numpy.random as rnd
>>> rnd.random((2,2))
array([[ 0.02909142,  0.90848   ],
       [ 0.9471314 ,  0.31424393]])
>>> rnd.poisson(size=(2,2))
```

Polynomials

- Polynomial is defined by array of coefficients p
- $p(x, N) = p[0]\, x^{N-1} + p[1]\, x^{N-2} + \ldots + p[N-1]$
- Least square fitting: numpy.polyfit
- Evaluating polynomials: numpy.polyval
- Roots of polynomial: numpy.roots
- ...

```
>>> x = np.linspace(-4, 4, 7)
>>> y = x**2 + rnd.random(x.shape)
>>>
>>> p = np.polyfit(x, y, 2)
>>> p
array([ 0.96869003, -0.01157275,  0.69352514])
```

Linear algebra

- Numpy can calculate matrix and vector products efficiently: dot, vdot, ...
- Eigenproblems: linalg.eig, linalg.eigvals, ...
- Linear systems and matrix inversion: linalg.solve, linalg.inv

```
>>> A = np.array(((2, 1), (1, 3)))
>>> B = np.array(((-2, 4.2), (4.2, 6)))
>>> C = np.dot(A, B)
>>>
>>> b = np.array((1, 2))
>>> np.linalg.solve(C, b)  # solve C x = b
array([ 0.04453441,  0.06882591])
```

Numpy performance

- Matrix multiplication

  C = A * B

  matrix dimension 200

- pure python: 5.30 s
- naive C: 0.09 s
- numpy.dot: 0.01 s

Summary

- Numpy provides a static array data structure
- Multidimensional arrays
- Fast mathematical operations for arrays
- Arrays can be broadcasted into same shapes
- Tools for linear algebra and random numbers