Numpy, Scipy and Matplotlib
Numpy

- Basis for numerical computation
- List can be used for numerical data
  - Inefficient for that purpose
  - No native mathematical operators
- Numpy defines an array type
  - Called \texttt{ndarray}
Numpy

- **ndarray** Type
  - Implements n-dimensional array
  - Elements of fixed size
  - Implemented as compiled code
  - Efficient and fast operations
  - Supports the same indexing operations as lists
Creating **ndarray**

- Numpy arrays have a type determined at creation time
- Stored in attribute `dtype`
- Types are themselves objects of type `dtype`
- `dtype` represents the underlying low level type of the array
Numpy

- Creating `ndarray`
  - `array()` function creates a numpy array from a sequence type or array-like type
  - Example from a list of numeric types
Example

```python
>>> import numpy as np
>>> a = np.array([1, 2, 3])

>>> print type(a)
<type 'numpy.ndarray'>

>>> print a
[1 2 3]

>>> print a.dtype
int64
```
Numpy

- Nested sequence → multi-dimensional array

```python
>>> mlist = [[1, 2, 3],
...           [4, 5, 6],
...           [7, 8, 9]]
>>> a = np.array(mlist)

>>> print a
[[1 2 3]
 [4 5 6]
 [7 8 9]]

>>> print a.ndim
2

>>> print a.shape
(3, 3)
```
Numpy

- Numpy determines the dtype from the elements of the array
- A numpy array is of uniform dtype
- Casting to make all elements of same type
- dtype can be specified explicitly during array creation
Numpy

```python
>>> a = np.array([1, 2, 3, 0.1])

>>> print a.dtype
float64

>>> b = np.array([1, 2, 3], dtype=np.float)

>>> print b
[ 1.  2.  3.]

>>> print b.dtype
float64
```
Numpy

- Other array creation functions
  - `arange()`

```python
>>> a = np.arange(10)

>>> type(a)
<type 'numpy.ndarray'>

>>> print a
[0 1 2 3 4 5 6 7 8 9]

>>> a = np.arange(0, 1, 0.1)

>>> print a
[ 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```
Numpy

- Other array creation functions
  - `linspace()`

```python
>>> a = np.linspace(1, 2.5, 5)

>>> print a
[ 1.     1.375  1.75   2.125  2.5  ]
```
Other array creation functions

- `empty()`, `zeros()`, `ones()`

```python
>>> earr = np.empty((2,2), dtype=np.float)
>>> print earr
[[ 6.90637353e-310  1.93980291e-316]
 [ 6.90637277e-310  6.90637277e-310]]

>>> oarr = np.ones((2,2,3), dtype=np.int)
>>> print oarr
[[[1 1 1]
  [1 1 1]]
[[1 1 1]
 [1 1 1]]]
```
## Numpy

- **Other array creation functions**
  - `empty()`, `zeros()`, `ones()`

```python
>>> darr = np.eye(4)
>>> print(darr)
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```
Numpy

- Other array creation functions
  - loadtxt()

```shell
$ cat npinput.txt
1.2 3.3 4.4 0.0
0.11 4.5 6.6 33.222
0.01 45.3 3.4 2.3
```
Other array creation functions

- `loadtxt()`

```python
>>> import numpy as np
>>> a = np.loadtxt('npinput.txt')
>>> print a
[[  1.20000000e+00   3.30000000e+00   4.40000000e+00   0.00000000e+00]
[  1.10000000e-01   4.50000000e+00   6.60000000e+00   3.32220000e+01]
[  1.00000000e-02   4.53000000e+01   3.40000000e+00   2.30000000e+00]]
```
Array types

- `dtypes` represent the low level type used to represent elements of an array
- `dtype` objects passed as arguments to set `dtype` of array
- `dtypes` with specific bit widths are useful when interfacing with compiled libraries (e.g., C, Fortran)
- Some types
# Array types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Underlying Python int (int64 or int32)</td>
</tr>
<tr>
<td>int8</td>
<td>Signed 8 bit integer</td>
</tr>
<tr>
<td>int16</td>
<td>Signed 16-bit integer</td>
</tr>
<tr>
<td>int32</td>
<td>Signed 32-bit integer</td>
</tr>
<tr>
<td>int64</td>
<td>Signed 64-bit integer</td>
</tr>
<tr>
<td>uint8/16/32/64</td>
<td>Corresponding unsigned integers</td>
</tr>
<tr>
<td>float</td>
<td>Shortcut to float64 type</td>
</tr>
<tr>
<td>float16/32/64</td>
<td>Floating point values of various bit sizes</td>
</tr>
<tr>
<td>complex</td>
<td>Shorthand for complex128</td>
</tr>
<tr>
<td>complex64/128</td>
<td>Complex numbers of 64 and 128-bits</td>
</tr>
</tbody>
</table>
Structured Arrays

- Elements of array composed of a collection of dtypes
- Similar to C struct
- Used to represent a record
- User creates a new dtype that is composed of the basic dtypes
- Specification passed as argument to dtype(). Returns new dtype
Structured Arrays

- Argument to `dype()`
  - List of tuples
  - Each tuple represents one field within the record
  - Tuple composed of 2-3 elements
    - Name of field as string
    - `dtype` of field (e.g. `np.int16`)
    - Optional: shape of the field
      - Integer (1D size)
      - Tuple (multi-dimensional)
      - Creates a field that is a subarray
Structured Arrays

```python
>>> d = np.dtype([('x', np.int), ('y', np.float)])
>>> a = np.zeros(3, dtype=d)

>>> print a
[(0, 0.0) (0, 0.0) (0, 0.0)]

>>> print repr(a)
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('x', '<i8'), ('y', '<f8')])
```
Structured Arrays

```python
>>> d = np.dtype([('x', np.int, (2,3)), ('y', np.float)])
>>> a = np.zeros(3, dtype=d)
>>> print repr(a)
array([[[[0, 0, 0], [0, 0, 0]], 0.0],
       [[[0, 0, 0], [0, 0, 0]], 0.0],
       [[[0, 0, 0], [0, 0, 0]], 0.0]],
       dtype=[('x', '<i8', (2, 3)), ('y', '<f8')])
```
Array Methods

- `ndarray` has several methods
  - Methods usually return an array
  - Returned array could be a
    - `copy`: does not share any memory space with input arrays
    - `view`: shares memory space with input
    - Usually specified in the documentation for the method
Array Methods

- View of an array shares underlying data with the array
- `transpose()` method returns a view

```python
>>> a = np.random.rand(3,2)
>>> print a
[[ 0.35845424  0.87955596]
 [ 0.83735742  0.18780326]
 [ 0.66418267  0.46915679]]

>>> b = a.transpose()
>>> print b
[[ 0.35845424  0.83735742  0.66418267]
 [ 0.87955596  0.18780326  0.46915679]]
```
Array Methods

- View of an array shares underlying data with the array
- `transpose()` method returns a view

```python
>>> b[0][0] = 0.0
>>> print a
[[ 0.          0.87955596]
 [ 0.83735742  0.18780326]
 [ 0.66418267  0.46915679]]

>>> np.may_share_memory(a,b)
True
```
Array Methods

- `copy()`

```python
>>> b = a.transpose()
>>> np.may_share_memory(a,b)
True
>>> b = a.transpose().copy()
>>> np.may_share_memory(a,b)
False
```
Array Methods

- `fill()`

```python
>>> a = np.zeros((3,3), dtype=int)
>>> a.fill(3.21)
>>> print a
[[3 3 3]
 [3 3 3]
 [3 3 3]]
```
Array Methods

- `reshape()`
  - Returns array view or copy with new shape
  - Argument is integer or tuple
  - Same size as input array
  - One dimension can be -1
Array Methods

- **reshape()**

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

>>> print a.reshape((2,6))
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
```
Array Methods

- reshape()

```python
>>> print a.reshape((12,-1))
[[ 1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12]]

>>> print a.reshape((12,-1)).shape
(12, 1)
```
Array Methods

- `reshape()`

```python
>>> print a.reshape(12)
[ 1  2  3  4  5  6  7  8  9 10 11 12]

>>> print a.reshape(12).shape
(12,)
```
Array Methods

- `reshape()` can sometimes return a copy
- `ravel()`
  - Returns 1D array of elements
  - `order` argument. Default order is 'C'
  - Other orders are 'F', 'A', and 'K'.

Array Methods

- `ravel()`

```python
>>> print a
[[ 1  2  3]
 [ 5  6  7]
 [ 9 10 11]]

>>> print a.ravel(order='C')
[ 1  2  3  5  6  7  9 10 11]

>>> print a.ravel(order='F')
[ 1  5  9  2  6 10  3  7 11]
```
Array Methods

- `sort()`
  - Sorts an array in place
  - Sorts along the last axis by default
    - Changed with `axis` argument
  - `np.sort()` returns a sorted copy
Array Methods

- `sort()`

```python
>>> print a
[[0 5 8 7]
 [1 5 6 0]
 [6 9 5 0]
 [4 5 8 1]]

>>> a.sort(axis=1)

>>> print a
[[0 5 7 8]
 [0 1 5 6]
 [0 5 6 9]
 [1 4 5 8]]
```
Array Indexing

- Numpy arrays support all the indexing operations of Python sequence types
- Additionally supports some extended indexing syntax
Array Indexing

- Multidimensional indexing
- Python sequence

>>> a[3][0]

- Numpy array
  - Accepts tuple indexes

>>> a[3,0]
Array Indexing

- Colon operator still available

```python
>>> print a
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

>>> print a[:,::2]
[[ 1  3]
 [ 5  7]
 [ 9 11]]
```
Array Indexing

- Unlike sequences, Numpy slicing returns a view

```python
>>> print a
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
>>> b = a[:,::2]
>>> print b
[[ 1  3]
 [ 5  7]
 [ 9 11]]
>>> b[0,0] = 44
>>> print a
[[44  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```
Array Indexing

- `newaxis` keyword in index, add a dimension to a view
Array Indexing

```python
>>> vsarr = np.array([1, 2, 3])
>>> print vsarr.shape
(3,)

>>> vsarr2d = vsarr[:,np.newaxis]
>>> print vsarr2d.shape
(3, 1)

>>> vsarr3d = vsarr2d[np.newaxis,:,::]
>>> print vsarr3d.shape
(1, 3, 1)
```
Fancy Indexing

- Numpy allows other arrays as indexes

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

>>> print a[i]
[1 3 3 1 1 5 4]
```
Fancy Indexing

- Multi-dimensional arrays indexed with tuples of arrays

```python
>>> print b
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

>>> irow = np.array([0, 1, 2])
>>> icol = np.array([3, 3, 2])

>>> print b[irow, icol]
[ 4  8 11]

>>> print b[0,3], b[1,3], b[2,2]
4 8 11
```
Fancy Indexing

- Fancy indexing returns a copy

```python
>>> a = np.array([1,2,3,4,5,6])
>>> i = np.array([[1,1,0],[2,3,0]])

>>> b = a[i]
>>> print b
[[2 2 1]
 [3 4 1]]

>>> np.may_share_memory(a,b)
False
```
Basic Operations

- Basic operators like +  -  *  /  //  **
  - Apply element wise
- Remember: if a and b are 2D square matrices
  - a*b is not matrix multiplication
Basic Operations

```python
>>> a = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float)
>>> b = np.array([[7, 8, 9], [10, 11, 12]], dtype=np.float)

>>> print a
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

>>> print b
[[  7.   8.   9.]
 [ 10.  11.  12.]]
```
Basic Operations

- Elementwise operators

```python
>>> a = np.array([[1,2,3],[4,5,6]], dtype=np.float)
>>> b = np.array([[7,8,9],[10,11,12]], dtype=np.float)

>>> print a+b
[[  8.  10.  12.]
 [ 14.  16.  18.]]

>>> print a*b
[[  7.  16.  27.]
 [ 40.  55.  72.]]
```
Basic Operations

- np.dot() performs generalized matrix multiplication

```python
>>> print a
[[1 2]
 [3 4]]

>>> print b
[[5 6]
 [7 8]]

>>> print np.dot(a, b)
[[19 22]
 [43 50]]
```
Universal Functions (ufuncs)

- Mathematical and other functions that operate on `ndarray` types
- Applies function element-wise to the array
- Implemented as precompiled code (usually C)
  - Vectorized versions faster and more efficient than equivalent Python looping construct
- Equivalent functions in `math` module cannot operate on arrays
Universal Functions (ufuncs)

- ufuncs

```python
>>> import numpy as np
>>> print a
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
>>> print np.sqrt(a)
[[ 1.          1.41421356  1.73205081]
 [ 2.          2.23606798  2.44948974]]
```
Universal Functions (ufuncs)

- ufuncs versions of the standard operators

```python
>>> a = np.array([[1,2,3],[4,5,6]], dtype=np.float)
>>> b = np.array([[7,8,9],[10,11,12]], dtype=np.float)

>>> print np.add(a,b)
[[ 8. 10. 12.]
 [14. 16. 18.]]

>>> print np.multiply(a,b)
[[  7.  16.  27.]
 [ 40.  55.  72.]]
```
Array Broadcasting

- ufuncs operate on two arrays of the same shape
  - Example `add()`
- If they are not the same shape
- Numpy tries to broadcast them to the same shape
- Then the ufunc can operate on them
- Broadcasting performed according to certain rules
- Can be understood by the following 3 cases
  - Case 1: What we have seen until now
    - Both arrays of the same shape
Array Broadcasting

- Case 2
  - Arrays have same number of dimensions or $\text{ndim}$ is equal
  - Shape is compared. i.e size in each dimension compared.
  - If corresponding sizes in each dimension are equal or one of the array has size 1. Then arrays are broadcastable
Array Broadcasting

- Case 2

```python
>>> print a
[[1 2]
 [3 4]]

>>> print b
[[1]
 [3]]

>>> print a.shape
(2, 2)

>>> print b.shape
(2, 1)
```
Array Broadcasting

```python
>>> print a
[[1 2]
 [3 4]]

>>> print b
[[[1]
  [3]]

>>> print c
[[[1 1]
  [3 3]]

>>> print a*b
[[ 1  2]
 [ 9 12]]

>>> print a*c
[[ 1  2]
 [ 9 12]]
```
Array Broadcasting

- Case 2: Broadcasting can happen in more than one dimension

```python
>>> print a
[[1 2]
 [3 4]]

>>> print b
[[12]]

>>> print c
[[12 12]
 [12 12]]
```
Array Broadcasting

- Case 2: Broadcasting can happen in more than one dimension

```python
>>> print a+b
[[13 14]
 [15 16]]

>>> print a+c
[[13 14]
 [15 16]]
```
Array Broadcasting

- Case 2: Broadcasting can happen in both operands

```python
>>> a = np.array([[[1, 2, 3], [4, 5, 6]]])
>>> b = np.array([[[1, 2, 3]], [[4, 5, 6]]])
>>> print a.shape
(1, 2, 3)
>>> print b.shape
(2, 1, 3)
>>> print a + b
[[[ 2  4  6]
  [ 5  7  9]]
[[ 5  7  9]
  [ 8 10 12]]]
>>> print (a+b).shape
(2, 2, 3)
```
Case 3
- Number of dimensions are not equal
  - Or ndim not equal
- Shape of array with small ndim is prepended with ones
- Until ndim are equal
- Then rules for case 2 are applied
  - And broadcasted if satisfied
Array Broadcasting

```python
>>> a = np.array([[1,2],[3,4]])
>>> s = np.array(4)

>>> print a.shape
(2, 2)

>>> print s.shape
()

>>> print s.ndim
0

>>> print a+s
[[5 6]
 [7 8]]
```
Array Derived Types

- Types that are derived or built-on the \texttt{ndarray} class

- \texttt{matrix} type

- Identical to \texttt{ndarray} except
Array Derived Types

- Allows MATLAB-style creation syntax

```python
>>> mmat = np.matrix('1 2 3; 4 5 6; 7 8 9')
>>> print mmat
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> print type(mmat)
<class 'numpy.matrixlib.defmatrix.matrix'>
```
Array Derived Types

- `matrix` is *always* 2-dimensional

```python
>>> mmat = np.matrix([[1, 2, 3, 4, 5]])
>>> print mmat.shape
(1, 5)
>>> print mmat.ravel().shape
(1, 5)
```
Array Derived Types

- * is overloaded to do matrix multiplication

```python
>>> a = np.matrix([[1,2],[3,4]])
>>> b = np.matrix([[5,6],[7,8]])

>>> print a*b
[[19 22]
 [43 50]]

>>> print np.multiply(a,b)
[[ 5 12]
 [21 32]]
```
Array Derived Types

- ** is overloaded to do matrix raised to power

```python
>>> print a
[[1 2]
 [3 4]]

>>> print a**3
[[ 37  54]
 [81 118]]

>>> print np.power(a,3)
[[ 1  8]
 [27 64]]
```
Array Derived Types

- `matrix` instance has the following attributes
  - `matrix.T`
    - Transpose of matrix
  - `matrix.H`
    - Hermitian transpose
  - `matrix.I`
    - Inverse of the matrix
  - `matrix.A`
    - Base ndarray type
Scipy

- Library of mathematical and scientific modules
- Comparable to MATLAB toolboxes
- Built on numpy arrays
- `import`ing scipy also imports most functions in numpy
  - Need not import both
- Consists of several subpackages
  - May need to be imported separately
- See

```python
>>> import scipy as sp
>>> help(sp)
```
Scipy

odr
--- Orthogonal Distance Regression [*]

misc
--- Various utilities that don't have another home.

cluster
--- Vector Quantization / Kmeans [*]

fftpack
--- Discrete Fourier Transform algorithms [*]

io
--- Data input and output [*]

sparse.linalg.eigen.lobpcg
--- Locally Optimal Block Preconditioned Conjugate Gradient Method (LOBPCG) [*]

special
--- Airy Functions [*]

lib.blas
--- Wrappers to BLAS library [*]

sparse.linalg.eigen
--- Sparse Eigenvalue Solvers [*]

stats
--- Statistical Functions [*]

lib
--- Python wrappers to external libraries [*]

lib.lapack
--- Wrappers to LAPACK library [*]

maxentropy
--- Routines for fitting maximum entropy models [*]
Scipy

- Try

```python
>>> import scipy.linalg as la
>>> help(la)
```
Scipy

inv - Find the inverse of a square matrix
solve - Solve a linear system of equations
solve_banded - Solve a banded linear system
solveh_banded - Solve a Hermitian or symmetric banded system
det - Find the determinant of a square matrix

Eigenvalue Problems
---------------

.. autosummary::
   :toctree: generated/

eig - Find the eigenvalues and eigenvectors of a square matrix
eigvals - Find just the eigenvalues of a square matrix
```python
>>> amat = sp.random.random_integers(1,100, size=(4,4))
>>> bvec = sp.random.random_integers(1,100, size=4)

>>> print amat
[[39 72 80 12]
 [71 9 74 4]
 [23 58 75 39]
 [41 63 12 21]]

>>> print bvec
[23 51 86 86]

>>> la.det(amat)
-10039045.0

>>> la.solve(amat,bvec)
array([[ 1.10460716, -0.19100821, -0.49857053,  2.79654619]])
```
Matplotlib

- Python 2D plotting library
- Line plots, histograms, scatterplots, pie charts, image plots...
  - And a lot more...
- Backend can produce output for various media
  - Display
  - Raster/Vector image formats
  - PDF, PS, EPS
  - Etc..
Matplotlib

- Two ways to work with matplotlib
  - Pylab interface
  - Object oriented approach
- Plotting can be done from scripts or the interactive interpreter
- iPython is an alternate python interpreter that integrates matplotlib well and provides a MATLAB-like environment
  - ipython.org
Pylab interface

- Need to import `matplotlib.pyplot`
- This mode of usage is stateful
- Each plotting command updates the current `figure`
- Once all plotting commands are applied, figure is displayed with `show()` function
Pylab interface

```python
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> a = np.linspace(0, 1, 10)
>>> b = np.exp(a)

>>> plt.plot(a)
[<matplotlib.lines.Line2D object at 0x3e3a710>]
>>> plt.plot(b)
[<matplotlib.lines-Line2D object at 0x3e3a910>]

>>> plt.show()
```
>>> plt.subplot(121)
<matplotlib.axes.AxesSubplot object at 0x38e4550>

>>> plt.plot(a)
[<matplotlib.lines.Line2D object at 0x38e7f90>]

>>> plt.subplot(122)
<matplotlib.axes.AxesSubplot object at 0x38be250>

>>> plt.plot(b)
[<matplotlib.lines.Line2D object at 0x38c9a90>]

>>> plt.show()
Pylab interface

```python
>>> plt.plot(a, label='linear')
[<matplotlib.lines.Line2D object at 0x4064bd0>]

>>> plt.plot(b, label='exponential')
[<matplotlib.lines.Line2D object at 0x42a1f50>]

>>> plt.legend()
<matplotlib.legend.Legend object at 0x42a9390>

>>> plt.show()
```
Pylab interface

- Numerous other functions and optional arguments to customize plots
- Familiar ones like `xlim()`, `ylim()`, `xlabel()`, `ylabel()`, `title()`, etc.
Artists Interface

- Object oriented approach to using matplotlib
- Allows more fine grained control
  - Get exactly the type of output you want
- Creating and manipulating instances of various classes that derive from the Artist class
Artists Interface

- General sequence of operations
  - Create an instance of Figure class
    - Represents the window containing various subplots called axes
  - One or more instances of class Axes are created and associated with the figure instance.
    - Axes instances represent each subplot
    - Figure has helper function that can create and associate in one step
  - All graph elements (lines, bars, points, etc..) are derived from Artist class
    - They are created and associated with an Axes instance
Artists Interface

- General sequence of operations
  - Axes instance has numerous helper methods
    - Each creates an artist object (like a histogram or line plot)
    - And automatically associates it with the axes
>>> fig = plt.figure(figsize=(6,5))
>>> ax = fig.add_subplot(1,2,1)
>>> t = np.linspace(0, 2*np.pi, 50)
>>> s = np.sin(t)
>>> print len(ax.lines)
0
Artists interface

```python
>>> ax.plot(t, s, linestyle='--')
[<matplotlib.lines.Line2D object at 0x27e5410>]

>>> print len(ax.lines)
1

>>> ax.plot(t, np.cos(t), linestyle='r')
[<matplotlib.lines.Line2D object at 0x2b73d50>]

>>> print len(ax.lines)
2
```
Artists interface

```python
>>> print len(ax.patches)
0

>>> rarr = np.random.random(50) * 2 * np.pi

>>> ax.hist(rarr)
(array([ 4,  3,  4,  6,  7,  2,  2, 11,  3,  8]),
array([ 0.08967892,
0.66255326,  1.2354276 ,  1.80830194,  2.38117628, 2.95405062,
3.52692496,  4.0997993 ,  4.67267363,  5.24554797, 5.81842231]), <a list of
10 Patch objects>)

>>> print len(ax.patches)
10

>>> fig.show()
```
Fancy Indexing

- Returned array has same shape as indexing array

```python
>>> print a
[1 2 3 4 5]

>>> i = np.array([[0, 2, 2],[0, 0, -1]])
>>> print i.shape
(2, 3)

>>> print a[i]
[[1 3 3]
 [1 1 5]]
>>> print a[i].shape
(2, 3)
```