# Numpy and Scipy

Numerical Computing in Python

# What is Numpy?

- Numpy, Scipy, and Matplotlib provide MATLAB-like functionality in python.
- Numpy Features:
  - Typed multidimentional arrays (matrices)
  - Fast numerical computations (matrix math)
  - High-level math functions

# Why do we need NumPy

Let’s see for ourselves!

# Why do we need NumPy

- Python does numerical computations slowly.
- 1000 x 1000 matrix multiply
  - Python triple loop takes > 10 min.
  - Numpy takes ~0.03 seconds

# Logistics: Versioning

- In this class, your code will be tested with:
  - Python 2.7.6
  - Numpy version: 1.8.2
  - Scipy version: 0.13.3
  - OpenCV version: 2.4.8

- Two easy options:
  - Class virtual machine (always test on the VM)
  - Anaconda 2 (some assembly required)

# NumPy Overview

1. Arrays
2. Shaping and transposition
3. Mathematical Operations
4. Indexing and slicing
5. Broadcasting

# Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- Tensors
- ConvNets

# Arrays

Structured lists of numbers.

- **Vectors**
- **Matrices**
- Images
- Tensors
- ConvNets

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

# Arrays

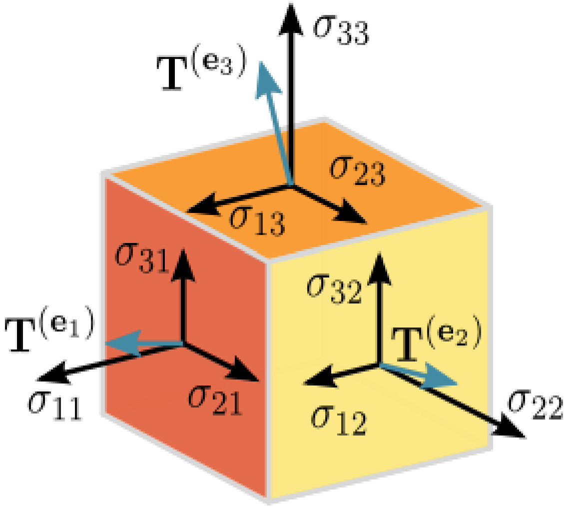
Structured lists of numbers.

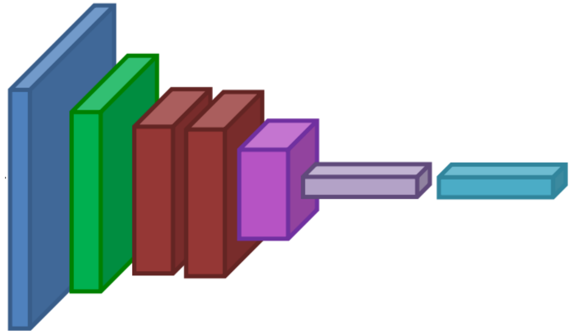
- Vectors
- Matrices
- **Images**
- Tensors
- ConvNets

# Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- **Tensors**
- **ConvNets**



# Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- Tensors
- ConvNets

# Arrays, Basic Properties

```
import numpy as np
a = np.array([[1,2,3],[4,5,6]],dtype=np.float32)
print a.ndim, a.shape, a.dtype
```

1. Arrays can have any number of dimensions, including zero (a scalar).
2. Arrays are typed: np.uint8, np.int64, np.float32, np.float64
3. Arrays are dense. Each element of the array exists and has the same type.

# Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

# Arrays, creation

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

# Arrays, creation

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```

# Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((2,3))
>>> B = np.zeros((4,3))
>>> np.concatenate([A,B])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
```

# Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((4,1))
>>> B = np.zeros((4,2))
>>> np.concatenate([A,B], axis=1)
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

# Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- **np.astype**
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A
array([[ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5]], dtype=float32)
>>> print(A.astype(np.uint16))
[[4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]]
```

# Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- **np.zeros_like, np.ones_like**
- np.random.random

```
>>> a = np.ones((2,2,3))
>>> b = np.zeros_like(a)
>>> print(b.shape)
```

# Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- **np.random.random**

```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

# Arrays, danger zone

- Must be dense, no holes.
- Must be one type
- Cannot combine arrays of different shape

```
>>> np.ones([7,8]) + np.ones([9,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together
 with shapes (7,8) (9,3)
```

# Shaping

```
a = np.array([1,2,3,4,5,6])
a = a.reshape(3,2)
a = a.reshape(2,-1)
a = a.ravel()
```

1. Total number of elements cannot change.
2. Use -1 to infer axis shape
3. Row-major by default (MATLAB is column-major)

# Return values

- Numpy functions return either **views** or **copies**.
- Views share data with the original array, like references in Java/C++. Altering entries of a view, changes the same entries in the original.
- The numpy documentation says which functions return views or copies
- Np.copy, np.view make explicit copies and views.

# Transposition

```
a = np.arange(10).reshape(5,2)

a = a.T

a = a.transpose((1,0))
```

np.transpose permutes axes.

a.T transposes the first two axes.

# Saving and loading arrays

```
np.savez('data.npz', a=a)

data = np.load('data.npz')

a = data['a']
```

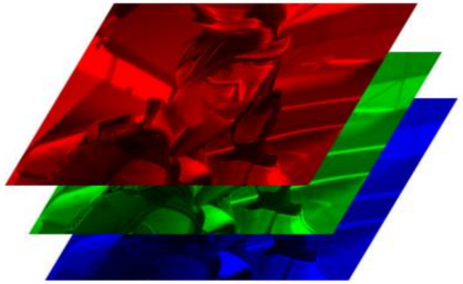
1. NPZ files can hold multiple arrays
2. np.savez_compressed similar.

# Image arrays

Images are 3D arrays: width, height, and channels

Common image formats:

- height x width x RGB (band-interleaved)
- height x width (band-sequential)



Gotchas:

- Channels may also be BGR (OpenCV does this)
- May be [width x height], not [height x width]

# Saving and Loading Images

SciPy: `skimage.io.imread,skimage.io.imsave`

height x width x RGB

PIL / Pillow: `PIL.Image.open, Image.save`

width x height x RGB

OpenCV: `cv2.imread, cv2.imwrite`

height x width x BGR

# Recap

We just saw how to create arrays, reshape them, and permute axes

Questions so far?

# Recap

We just saw how to create arrays, reshape them, and permute axes

Questions so far?

Now: let’s do some math

# Mathematical operators

- Arithmetic operations are element-wise
- Logical operator return a bool array
- In place operations modify the array

# Mathematical operators

- **Arithmetic operations are element-wise**
- Logical operator return a bool array
- In place operations modify the array

```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```

# Mathematical operators

- Arithmetic operations are element-wise
- **Logical operator return a bool array**
- In place operations modify the array

```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

# Mathematical operators

- Arithmetic operations are element-wise
- Logical operator return a bool array
- **In place operations modify the array**

```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```

# Math, upcasting

Just as in Python and Java, the result of a math operator is cast to the more general or precise datatype.

uint64 + uint16 => uint64

float32 / int32 => float32

Warning: upcasting does not prevent overflow/underflow. You must manually cast first.

Use case: images often stored as uint8. You should convert to float32 or float64 before doing math.

# Math, universal functions

Also called ufuncs

Element-wise

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

# Math, universal functions

Also called ufuncs

Element-wise

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

# Math, universal functions

Also called ufuncs

Element-wise

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
>>> a
array([[ 1,  4],
       [ 9, 16],
       [25, 36]])
>>> np.sqrt(a)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

# Indexing

```
x[0,0]    # top-left element
x[0,-1]   # first row, last column
x[0,:]    # first row (many entries)
x[:,0]    # first column (many entries)
```

Notes:

- Zero-indexing
- Multi-dimensional indices are comma-separated (i.e., a tuple)

# Indexing, slices and arrays

```
I[1:-1,1:-1]          # select all but one-pixel border
I = I[:,:,::-1]       # swap channel order
I[I<10] = 0           # set dark pixels to black
I[[1,3], :]           # select 2nd and 4th row
```

1. Slices are **views**. Writing to a slice overwrites the original array.
2. Can also index by a list or boolean array.

# Python Slicing

Syntax: start:stop:step

```
a = list(range(10))

a[:3] # indices 0, 1, 2

a[-3:] # indices 7, 8, 9

a[3:8:2] # indices 3, 5, 7

a[4:1:-1] # indices 4, 3, 2 (this one is tricky)
```

# Axes

```
a.sum() # sum all entries

a.sum(axis=0) # sum over rows

a.sum(axis=1) # sum over columns

a.sum(axis=1, keepdims=True)
```

1. Use the axis parameter to control which axis NumPy operates on
2. Typically, the axis specified will disappear, keepdims keeps all dimensions

# Broadcasting

```
a = a + 1 # add one to every element
```

When operating on multiple arrays, broadcasting rules are used.

Each dimension must match, from right-to-left

1. Dimensions of size 1 will broadcast (as if the value was repeated).
2. Otherwise, the dimension must have the same shape.
3. Extra dimensions of size 1 are added to the left as needed.

# Broadcasting example

Suppose we want to add a color value to an image

a.shape is 100, 200, 3

b.shape is 3

a + b will pad b with two extra dimensions so it has an effective shape of 1 x 1 x 3.

So, the addition will broadcast over the first and second dimensions.

# Broadcasting failures

If a.shape is 100, 200, 3 but b.shape is 4 then a + b will fail. The trailing dimensions must have the same shape (or be 1)

# Tips to avoid bugs

1. Know what your datatypes are.
2. Check whether you have a view or a copy.
3. Use matplotlib for sanity checks.
4. Use pdb to check each step of your computation.
5. Know np.dot vs np.mult.

# Average images

Who is this?

# Practice exercise (not graded)

Compute the average image of faces.

1. Download Labeled Faces in the Wild dataset (google: LFW face dataset). Pick a face with at least 100 images.
2. Call numpy.zeros to create a 250 x 250 x 3 float64 tensor to hold the result
3. Read each image with skimage.io.imread, convert to float and accumulate
4. Write the averaged result with skimage.io.imsave