Numpy

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

NumPy

- Stands for Numerical Python
- Is the fundamental package required for high performance computing and data analysis
- NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data.

NumPy

- Numpy provides
  - ndarray for creating multiple dimensional arrays
  - Internally stores data in a contiguous block of memory, independent of other built-in Python objects, use much less memory than built-in Python sequences.
  - Standard math functions for fast operations on entire arrays of data without having to write loops
  - NumPy Arrays are important because they enable you to express batch operations on data without writing any *for* loops. We call this *vectorization*.

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

NumPy Overview

- Arrays
- Shaping
- Mathematical Operations
- Indexing and slicing

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

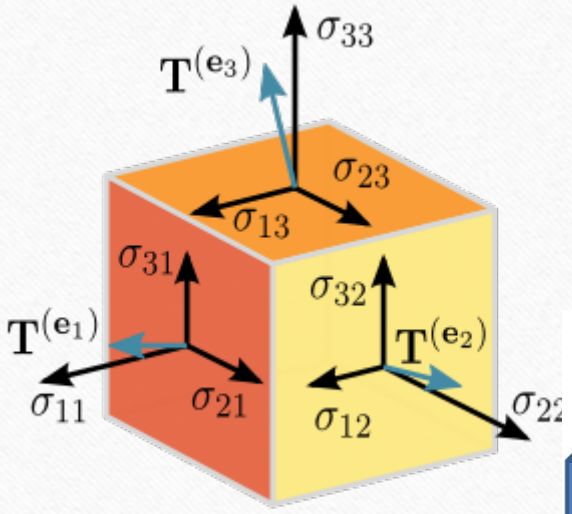
Structured lists of numbers.

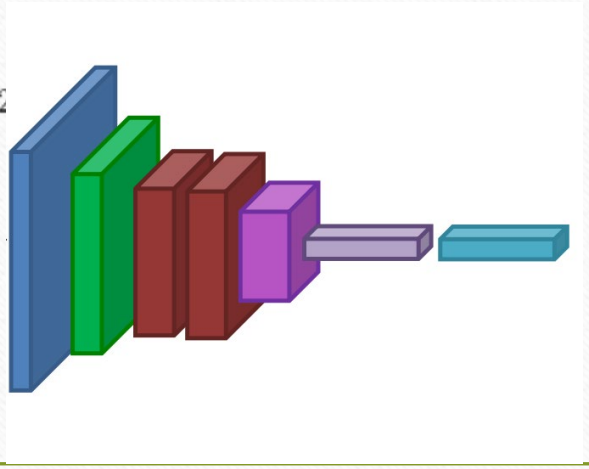
- Vectors
- Matrices
- **Images**
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

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





Arrays

- Structured lists of numbers.
  - Vectors
  - Matrices
  - Images
  - Tensors
  - ConvNets

```
import numpy as np

a = np.array([[1,2,3],[4,5,6]],dtype=np.float32)

print(a.ndim)
print(a.shape)
print(a.dtype)
```

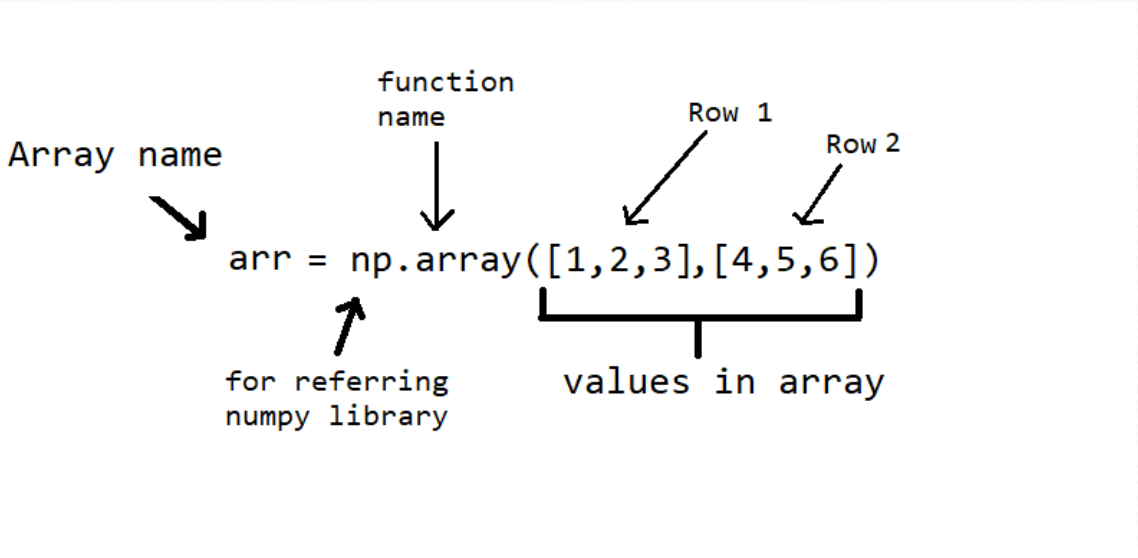
```
2
(2, 3)
float32
```

What are Arrays?

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.

Create Arrays

- 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)
```

Create Arrays

- 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])
```

Create Arrays

- 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.]])
>>>
```

Create Arrays

- 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_li
- 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]]
```

Create Arrays

- 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)
```

Create Arrays

- 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]])
```

Create Arrays

- 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

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

```
[2]: import numpy as np

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

[5]: array([[1, 2],
            [3, 4],
            [5, 6]])

[6]: a = a.reshape(2,-1)
     a

[6]: array([[1, 2, 3],
            [4, 5, 6]])

[7]: a = a.ravel()
     a

[7]: array([1, 2, 3, 4, 5, 6])
```

Return values

- Numpy functions return either **views** or **copies**.
- Views share data with the original array.
- The numpy documentation says which functions return views or copies
- Np.copy, np.view make explicit copies and views.

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, 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)

Slicing

```
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
```

- Slices are **views**. Writing to a slice overwrites the original array.
- 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)
```