

Rules  
ooooo

Scalar case  
o

2D cases  
ooooooo

Extended example  
oooo

# Broadcasting

Jean Mark Gawron

Linguistics 572  
San Diego State University

September 29, 2020

## Dimension compatibility

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions and works its way forward. Two dimensions are compatible when

- ① they are equal; or
- ② one of them is 1.

## Arrays of different dimensionality

All dimensions of lower dimensionality array match **trailing** dimensions of the other. Scale the 3 color layers in an RGB image by different amounts:

Image (3d array): 256 x 256 x 3

Scales (1d array): 3 [1. .9 1.4]

Result (3d array): 256 x 256 x 3

## Trailing dimensions don't match

```
>>> import numpy as np
>>> A, B = np.arange(20).reshape((5,4)),np.arange(4)
>>> Result1 = A * B
>>> C = np.arange(5)
>>> Result2 = A * C
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shape
```

## Two mismatched 1D arrays

ValueError: operands could not be broadcast together

```
>>> A1d, B1d = np.arange(4),np.arange(5)  
>>> A1d + B1d
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: operands could not be broadcast together with shape

Rules  
oooo●

Scalar case  
○

2D cases  
oooooooo

Extended example  
oooo

## Examples

A      5    x    4  
B                4  
Result    5    x    4

A      5    x    4  
B                1    Scalar!  
Result    5    x    4

A      15    x    3    x    5  
B      15    x    1    x    5  
Result  15    x    3    x    5

A      15    x    3    x    5  
B                3    x    5  
Result  15    x    3    x    5

# Multiply by 5

$$5 + \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$



$$\begin{bmatrix} 5 & 5 & 5 & 5 \\ 5 & 5 & 5 & 5 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

## Two Matched 2D arrays

```
>>> a = np.arange(4)[:,np.newaxis]
>>> b = np.arange(5)[np.newaxis,:]
>>> print(a,a.shape)
[[0]
 [1]
 [2]
 [3]] (4, 1)
>>> print(b,b.shape)
[[0 1 2 3 4]] (1, 5)
>>> print((a+b).shape)
(4, 5)
```

## “Outer” addition

a is 4x1, b is 1x5.

```
>>> print(a + b)
[[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]]
```

```
r,c = a.shape[0],b.shape[1]
M = np.zeros((r,c),dtype=int)
for i in range(r):
    for j in range(c):
        M[i,j] = a[i,0] + b[0,j]
```

## Broadcasting: size-one dimensions copied

$4 \times 1$

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix} \implies$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

$1 \times 5$

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \end{bmatrix} \downarrow$$

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

Rules  
ooooo

Scalar case  
o

2D cases  
oooo●oooo

Extended example  
oooo

## Shapes?

```
>>> import numpy as np
>>> a1d, b1d = np.arange(4),np.arange(5)
>>> a, b = a1d.reshape((4,1)), b1d.reshape((5,1))

>>> a1d_p_b = a1d + b
>>> a_p_b1d = a + b1d
```

## Values

```
>>> print(a.shape, b.shape)
(4, 1) (5, 1)
```

```
>>> a1d_p_b
array([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5],
       [3, 4, 5, 6],
       [4, 5, 6, 7]])
```

```
>>> a_p_b1d
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

# K Nearest Neighbors

- ➊ To illustrate the power of broadcasting with a somewhat more practical example, we'll compute the **K Nearest Neighbors** (KNN) for a set of points, where  $k$  is an integer.
- ➋ Typically the KNN problem is solved for points in a high dimensional space; each point represents a sample; each coordinate represents a value for some numerical feature of the sample. For presentational purposes we'll solve KNN for a small set of 2D points.
- ➌ The KNN problem is important in a number of applications. For example, in a machine learning context, one strategy for classifying point  $X$  is to find the KNNs of  $X$  from among a set of points whose classes are known, and let the KNNs "vote" on the class of  $X$ .

## VanderPlas's K-Nearest Neighbor calculation

```
>>> Y = (np.array([np.arange(5),np.arange(5,0,-1)])).T
>>> Y
array([[0, 5],
       [1, 4],
       [2, 3],
       [3, 2],
       [4, 1]])
>>> Y[:, np.newaxis, :].shape, Y[np.newaxis, :, :].shape
((5, 1, 2), (1, 5, 2))
>>> deltas = Y[:,np.newaxis,:]-Y[np.newaxis,:,:] # (5, 5, 2)
>>> deltas[0,:,:1] # Y-coord diffs for Pt 0
array([0, 1, 2, 3, 4])
```

Rules  
ooooo

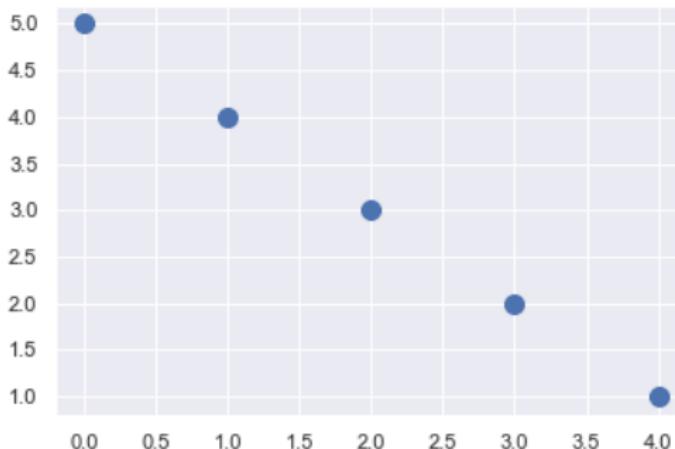
Scalar case  
○

2D cases  
oooooooo

Extended example  
●ooo

## The points

$$\text{dist}(\mathbf{Y}[i], \mathbf{Y}[j])^2 = (\text{deltas}[i][j][0])^2 + (\text{deltas}[i][j][1])^2$$



## Distance calc completed (looplessly)

```
>>> dists = np.sum(deltas**2, axis=2) #  $x^{**2} \text{ lyr} + y^{**2} \text{ lyr}$ 
>>> print(dists) #  $dists[i, j] = \text{sqd dist from } i \text{ to } j$ 
[[ 0  2  8 18 32]
 [ 2  0  2  8 18]
 [ 8  2  0  2  8]
 [18  8  2  0  2]
 [32 18  8  2  0]]

>>> nearest = np.argsort(dists, axis=1)
>>> print(nearest) #  $\text{nearest}[i, :] = \text{sorted nbrs of point } i$ 
[[0 1 2 3 4]
 [1 0 2 3 4]
 [2 1 3 0 4]
 [3 2 4 1 0]
 [4 3 2 1 0]]
```

## Finding the K nearest ( $k = 2$ )

Less work: Partition the set of points into the top  $K + 1$  ( $k = 2$ ) and all the rest:

```
>>> nearest_k = np.argpartition(dists, kth = 3, axis=1)
>>> print(nearest_k) # nearest[i,:k] = unsorted K nearest nbrs
[[1 0 2 3 4]
 [1 2 0 3 4]
 [3 2 1 0 4]
 [3 2 4 1 0]
 [3 4 2 1 0]]
```

# Takeaways

- ➊ Basic ‘numpy’ op: elementwise arithmetic between ‘ndarray’s of the same shape.
- ➋ Broadcasting licenses stretching operations on dimensions of size 1, or lower dimensionality arrays when “trailing” dimensions match.
- ➌ The broadcasting operation can be used to achieve the effect of a loop, performing a single operation on all the elements. To achieve this we sometimes increase the dimensionality of the data (‘:newaxis’).
- ➍ Complementing broadcasting (in the KNN example):
  - U-functions (universal functions) [‘deltas\*\*2’]
  - Aggregation operations [‘np.sum(deltas\*\*2, axis=2)’]