



Introduction to Numerical Computing with Numpy

Use Restrictions:

Use only permitted under license or agreement. Copying, sharing, redistributing or other unauthorized use is strictly prohibited. The Enthought Training Materials (ETM) are provided for the individual and sole use of the paid attendee of the class ("Student") for which the Training Services are provided. The virtual training sessions may not be shared, reproduced, transmitted, or retransmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system. Furthermore, neither Customer nor any Student shall:

Copy, disclose, transfer or distribute ETM to any party in any form. Remove, modify or obscure any copyright, trademark, legal notices or other proprietary notations in ETM. Make derivative works of ETM or combine ETM or any part of ETM with any other works. Use ETM in any manner that could be detrimental to Enthought. • 2001-2021, Enthought, Inc. All Rights Reserved. All trademarks and registered trademarks are the property of their respective owners.

Enthought, Inc. 200 W Cesar Chavez Suite 202 Austin, TX 78701
www.enthought.com

Q2-2021
letter
3.3.3



Introduction to Numerical Computing with Numpy

Enthought, Inc.
www.enthought.com

Introduction 1

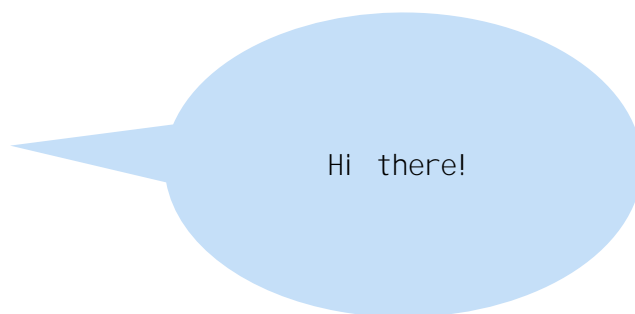
NumPy 2

- Introducing NumPy Arrays 4
- Multi-Dimensional Arrays 8
- Slicing/Indexing Arrays 9
- Fancy Indexing 15
- Creating arrays 18
- Array Creation Functions 20
- Array Calculation Methods 23
- Array Broadcasting 32
- Universal Function Methods 40
- The array data structure 46



Introduction to Numerical Computing with Numpy

1

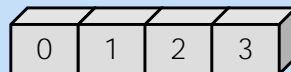


NumPy
The Standard Numerical Library
for Python

NumPy Arrays

- " Introducing Arrays
- " Indexing and Slicing
- " Creating Arrays
- " Array Calculations
- " The Array Data Structure
- " Structure Operations

a



NumPy
Introducing Arrays

Introducing NumPy Arrays

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

```
>>> type(a)
numpy.ndarray
```

```
>>> a.dtype
dtype('int32')
```

```
>>> a.ndim
1
```

```
# Shape returns a tuple
# listing the length of the
# array along each dimension.
>>> a.shape
(4,)
```

```
>>> a.itemsize
4
```

```
# Return the number of bytes
# used by the data portion of
# the array.
>>> a.nbytes
16
```

Array Operations

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([2, 3, 4, 5])
>>> a + b
array([3, 5, 7, 9])
```

```
>>> a * b
array([ 2,  6, 12, 20])
```

```
>>> a ** b
array([ 1,  8, 81, 1024])
```



NumPy defines these constants:
pi = 3.14159265359
e = 2.71828182846

```
# create array from 0.0 to 10.0
>>> x = np.arange(11.0)
```

```
# multiply entire array by
# scalar value
>>> c = (2.0 * np.pi) / 10.0
>>> c
0.62831853071795862
```

```
>>> c * x
áãää]ÇYcÈÁÁÁÊÁcÈWNGÎÊÁÔÊÁWÈGÎÇŸD
```

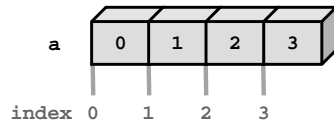
```
# in-place operations
```

```
>>> x *= c
>>> x
áãää]ÇYcÈÁÁÁÊÁcÈWNGÎÊÁÔÊÁWÈGÎÇŸD
```

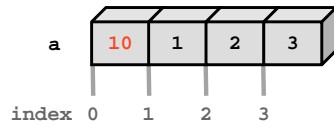
```
# apply functions to array
>>> y = np.sin(x)
```

Setting Array Elements

```
>>> a[0]
0
```



```
>>> a[0] = 10
>>> a
array([10, 1, 2, 3])
```



```
>>> a.dtype
dtype('int32')
```

assigning a float into
an int32 array truncates
the decimal part

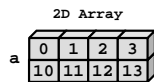
```
>>> a[0] = 10.6
>>> a
array([10, 1, 2, 3])
```

fill has the same behavior

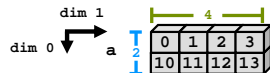
```
>>> a.fill(-4.8)
>>> a
array([-4, -4, -4, -4])
```

Multi-Dimensional Arrays

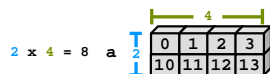
```
>>> a = np.array([[ 0, 1, 2, 3],
...               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```



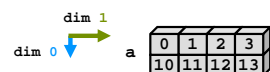
```
>>> a.shape
(2, 4)
```



```
>>> a.size
8
```



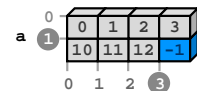
```
>>> a.ndim
2
```



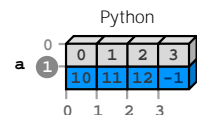
```
>>> a[1, 3]
13
```



```
>>> a[1, 3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```



```
>>> a[1]
array([10, 11, 12, -1])
```



	0	1	2	3	4	5
0	0	1	2	3	4	5
1	10	11	12	13	14	15
2	20	21	22	23	24	25
3	30	31	32	33	34	35
4	40	41	42	43	44	45
5	50	51	52	53	54	55

NumPy

Indexing and Slicing

Slicing

var[lower:upper:step]

Extracts a portion of a sequence by specifying a lower and upper bound.

The lower-bound element is included, but the upper-bound element is **not** included.

Mathematically: [lower, upper). The step value specifies the stride between elements.

```
#           -5 -4 -3 -2 -1
# indices:   0  1  2  3  4
>>> a = np.array([10,11,12,13,14])
```

```
# [10, 11, 12, 13, 14]
>>> a[1:3]
array([11, 12])
```

```
# negative indices work also
>>> a[1:-2]
array([11, 12])
>>> a[-4:3]
array([11, 12])
```

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list
```

```
# grab first three elements
>>> a[:3]
array([10, 11, 12])
```

```
# grab last two elements
>>> a[-2:]
array([13, 14])
```

```
# every other element
>>> a[::2]
array([10, 12, 14])
```


Array Slicing

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 45],
       [54, 55]])

>>> a[:, 2]
array([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	10	11	12	13	14	15
2	20	21	22	23	24	25
3	30	31	32	33	34	35
4	40	41	42	43	44	45
5	50	51	52	53	54	55

Assigning to a Slice

Slices are references to locations in memory.
These memory locations can be used in assignment operations.

```
>>> a = np.array([0, 1, 2, 3, 4])
# slicing the last two elements returns the data there
>>> a[-2:]
array([2, 3])
# we can insert an iterable of length two
>>> a[-2:] = [-1, -2]
>>> a
array([0, 1, -1, -2, 4])
# or a scalar value
>>> a[-2:] = 99
>>> a
array([0, 1, 99, 99, 4])
```

Give it a try!

Create the array below with the command
`a = np.arange(25).reshape(5, 5)`
and extract the slices as indicated.

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24

© 2008-2018 Enthought, Inc.

13

Sliced Arrays Share Data

Arrays created by slicing share data with the originating array.
Changing values in a slice also changes the original array.

```
>>> a = np.array([0, 1, 2, 3, 4])
# create a slice containing two elements of a
>>> b = a[2:4]
>>> b
array([2, 3])
>>> b[0] = 10

# changing b changed a!
>>> a
array([ 0,  1, 10,  3,  4])
>>> np.shares_memory(a, b)
True
```

© 2008-2018 Enthought, Inc.

14

Fancy Indexing

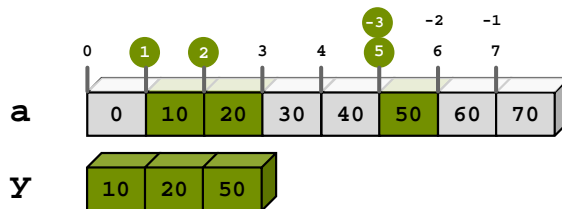
```
>>> a = np.arange(0, 80, 10)

# fancy indexing
>>> indices = [1, 2, -3]
>>> y = a[indices]
>>> y
array([10, 20, 50])

# this also works with setting
>>> a[indices] = 99
>>> a
array([ 0, 99, 99, 30, 40, 99, 60, 70])
```

```
# manual creation of masks
>>> mask = np.array(
...     [0, 1, 1, 0, 0, 1, 0, 0],
...     dtype=bool)

# fancy indexing
>>> y = a[mask]
>>> y
array([10, 20, 50])
```



© 2008-2018 Enthought, Inc.

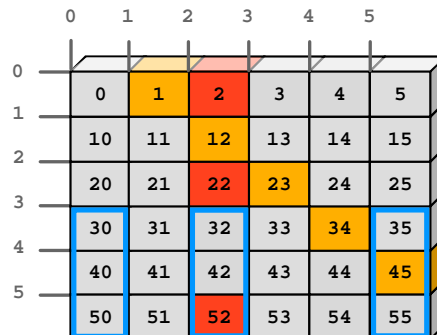
15

Fancy Indexing in 2-D

```
>>> a[[0, 1, 2, 3, 4],
...   [1, 2, 3, 4, 5]]
array([ 1, 12, 23, 34, 45])

>>> a[3:, [0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = np.array(
...     [1, 0, 1, 0, 0, 1],
...     dtype=bool)
>>> a[mask, 2]
array([2, 22, 52])
```



i Unlike slicing, fancy indexing creates copies instead of a view into original array.

© 2008-2018 Enthought, Inc.

16

Give it a try!

1. Create the array below with
`a = np.arange(25).reshape(5, 5)`
and extract the elements indicated in blue.
2. Extract all the numbers divisible by 3 using a boolean mask.

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2					
3	10	11	12	13	14
4	15	16	17	18	19
	20	21	22	23	24

© 2008-2018 Enthought, Inc.

17

arange()
linspace()
array()
zeros()
ones()

NumPy
Creating Arrays

© 2008-2018 Enthought, Inc.

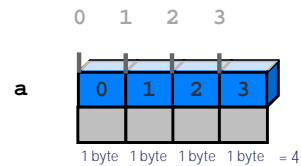
18

Array Constructor Examples

```
# Default to double precision
>>> a = np.array([0,1.0,2,3])
>>> a.dtype
dtype('float64')
>>> a.nbytes
32
```

```
>>> a = np.array([0,1.,2,3],
...               dtype='float32')
>>> a.dtype
dtype('float32')
>>> a.nbytes
16
```

```
>>> a = np.array([0,1,2,3],
...               dtype='uint8')
>>> a.dtype
dtype('uint8')
>>> a.nbytes
4
```



Base 2	Base 10
00000000	-> 0 = $0 \cdot 2^0 + 0 \cdot 2^1 + \dots + 0 \cdot 2^7$
00000001	-> 1 = $1 \cdot 2^0 + 0 \cdot 2^1 + \dots + 0 \cdot 2^7$
00000010	-> 2 = $0 \cdot 2^0 + 1 \cdot 2^1 + \dots + 0 \cdot 2^7$
...	
11111111	-> 255 = $1 \cdot 2^0 + 1 \cdot 2^1 + \dots + 1 \cdot 2^7$

Array Creation Functions

```
arange([start,] stop[, step],
       dtype=None)
```

range()
Creates an array of values in the range [start,stop) with the specified step value. Allows non-integer values for start, stop, and step. Default **dtype** is derived from the start, stop, and step values.

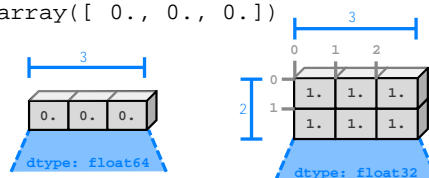
```
>>> np.arange(4)
array([0, 1, 2, 3])
>>> np.arange(0, 2*pi, pi/4)
array([ 0.000, 0.785, 1.571,
        2.356, 3.142, 3.927, 4.712,
        5.497])
```

```
%"Dg"ectghwní
>>> np.arange(1.5, 2.1, 0.3)
array([ 1.5, 1.8, 2.1])
```

```
ones(shape, dtype='float64')
zeros(shape, dtype='float64')
```

shape is a number or sequence specifying the dimensions of the array. If **dtype** is not specified, it defaults to **float64**.

```
>>> np.ones((2, 3),
...         dtype='float32')
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.zeros(3)
array([ 0.,  0.,  0.])
```



zeros(3) is equivalent to **zeros((3,))**

Array Creation Functions (cont'd)

```
# Generate an n by n identity
# array. The default dtype is
# float64.
>>> a = np.identity(4)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
>>> a.dtype
dtype('float64')
>>> np.identity(4, dtype=int)
array([[ 1,  0,  0,  0],
       [ 0,  1,  0,  0],
       [ 0,  0,  1,  0],
       [ 0,  0,  0,  1]])
```

```
# empty(shape, dtype=float64,
#        order='C')
>>> np.empty(2)
array([1.78021120e-306,
       6.95357225e-308])
```

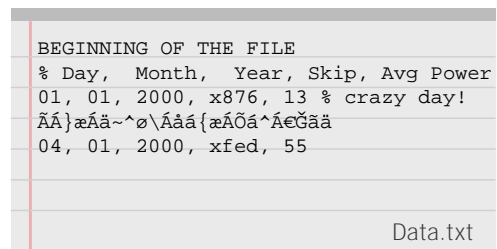
```
# array filled with 5.0
>>> a = np.full(2, 5.0)
array([5.,  5.] )
```

```
# alternative approaches
# (slower)
>>> a = np.empty(2)
>>> a.fill(4.0)
>>> a
array([4.,  4.] )
>>> a[:] = 3.0
>>> a
array([3.,  3.] )
```

Array Creation Functions (cont'd)

```
# Generate N evenly spaced
# elements between (and including)
# start and stop values.
>>> np.linspace(0, 1, 5)
array([0., 0.25, 0.5, 0.75, 1.0])
```

```
# Generate N evenly spaced
# elements on a log scale
# between base**start and
# base**stop (default base=10)
>>> np.logspace(0, 1, 5)
array([1., 1.78, 3.16, 5.62, 10.] )
```



BEGINNING OF THE FILE

% Day, Month, Year, Skip, Avg Power

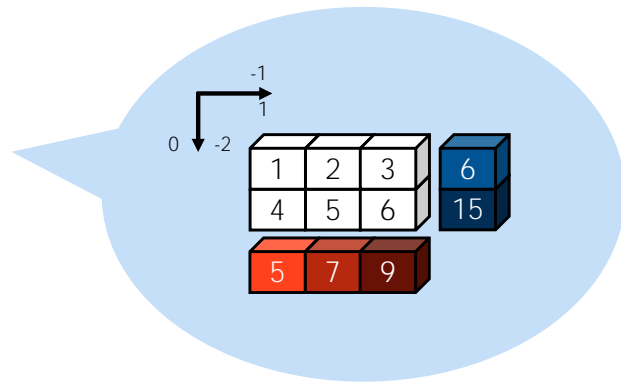
01, 01, 2000, x876, 13 % crazy day!

04, 01, 2000, xfed, 55

Data.txt

```
# loadtxt() automatically
# generates an array from the
# txt file
arr = np.loadtxt('Data.txt',
...             skiprows=1,
...             dtype=int, delimiter=",",
...             usecols = (0,1,2,4),
...             comments = "%")
```

```
# Save an array into a txt file
np.savetxt('filename.txt', arr)
```



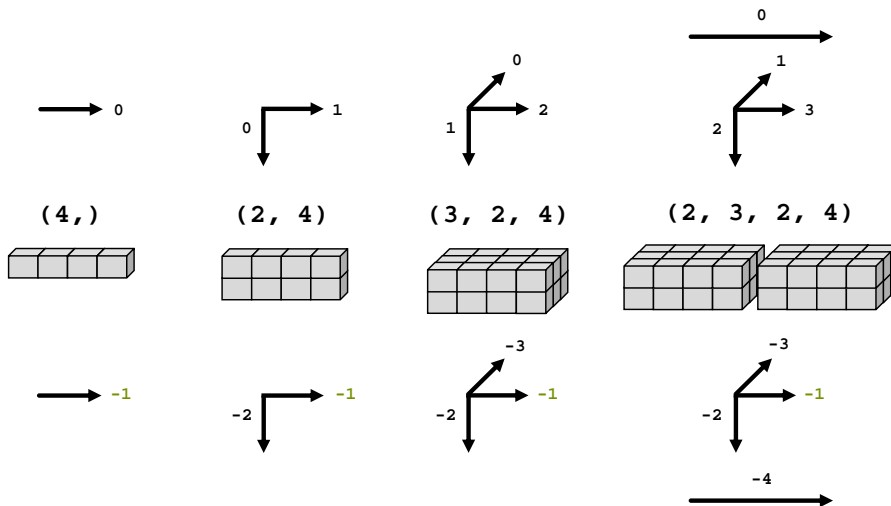
NumPy

Array Calculation Methods

Computations with Arrays

- Rule 1:** Operations between multiple array objects are first checked for proper shape match.
- Rule 2:** Mathematical operators (+ - * / exp, log, ...) apply element by element, on the values.
- Rule 3:** Reduction operations (mean, std, skew, kurt, sum, prod, ...) apply to the whole array, unless an axis is specified.
- Rule 4:** Missing values propagate unless explicitly ignored (nanmean, nansum, ...).

Multi-Dimensional Arrays



© 2008-2018 Enthought, Inc.

25

Array Calculation Methods

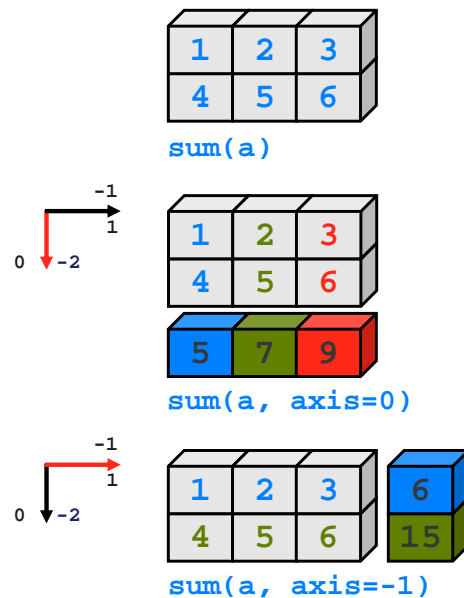
SUM METHOD

```
# Methods act on data stored
# in the array
>>> a = np.array([[1,2,3],
                  [4,5,6]])

# .sum() defaults to adding up
# all the values in an array.
>>> a.sum()
21

# supply the keyword axis to
# sum along the 0th axis
>>> a.sum(axis=0)
array([5, 7, 9])

# supply the keyword axis to
# sum along the last axis
>>> a.sum(axis=-1)
array([ 6, 15])
```



© 2008-2018 Enthought, Inc.

26

Other Operations on Arrays

```
# Functions work on data
# passed to it
>>> a = np.array([[1,2,3],
                  [4,5,6]])

# sum() defaults to adding
# up all values in an array.
>>> np.sum(a)
21

# supply an axis argument to
# sum along a specific axis
>>> np.sum(a, axis=0)
array([5, 7, 9])
```

Mathematical functions

```
" sum, prod
" min, max, argmin, argmax
" ptp (max  $\frac{1}{2}$  min)
```

Statistics

```
" mean, std, var
```

Truth value testing

```
" any, all
```

See the NumPy appendix for more.

Min/Max

```
>>> a = np.array([1,4,5,2,3])
# Prefer NumPy functions to
# builtins when working with
# arrays
>>> np.min(a)
0
# Most NumPy reducers can be used
# as methods as well as functions
>>> a.min()
0

# Use the axis keyword to find
# max values for one dimension
>>> a.max*axis=0+
array([2, 3])
# as a function
>>> np.max(a, axis=1)
array([2, 3])
```

```
# Many tasks (like optimization)
# are interested in the location
# of a min/max, not the value
>>> a.argmax()
1
```

```
# arg methods return the
# location in a 1D, flattened
# version of the original array
>>> np.argmin(a)
2
```

```
# NumPy includes a function
# to un-flatten 1D locations
>>> np.unravel_index(
...     a.argmax(), a.shape)
(0, 1)
```

Where

```
# NumPy's where function has two
# distinct uses. One is to
# provide coordinates from masks.
>>> a = np.arange(-2, 2) ** 2
>>> a
array([4, 1, 0, 1, 4])
>>> mask = a % 2 == 0
>>> mask
array([False,  True,  True,  True, False])
```

```
# Coordinates are returned as
# a tuple of arrays, one for
# each axis.
>>> np.where(mask)
(array([1, 2, 3]),)
```

```
# Where can also be used to
# construct a new array by
# choosing values from other
# arrays of the same shape.
>>> positives = np.arange(1, 5)
>>> negatives = -positives
>>> np.where(mask, positives,
...          negatives)
array([1, 2, 3, 4, 0])
```

```
# Or from scalar values.
# This can be useful for
# recoding arrays.
>>> np.where(mask, 1, 0)
array([1, 0, 1, 0])
```

```
# Or from both.
>>> np.where(mask, positives, 0)
array([1, 2, 3, 4, 0])
```

Statistics Array Methods

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

# mean value of each column
>>> a.mean(axis=0)
array([ 2.5,  3.5,  4.5])
>>> np.mean(a, axis=0)
array([ 2.5,  3.5,  4.5])
```

```
# Standard Deviation
>>> a.std(axis=0)
array([ 1.5,  1.5,  1.5])
# For sample, set ddof=1
>>> a.std(axis=0, ddof=1)
array([ 2.12,  2.12,  2.12])
```

```
# variance
>>> a.var(axis=0)
array([2.25, 2.25, 2.25])
>>> np.var(a, axis=0)
array([2.25, 2.25, 2.25])
```

Give it a try!

Create the array below with

```
a = np.arange(-15, 15).reshape(5, 6) ** 2
```

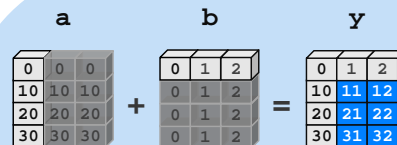
and compute:

1. The maximum of each row
2. The mean of each column
3. The position of the overall minimum

	0	1	2	3	4	5
0	225	196	169	144	121	100
1	81	64	49	36	25	16
2	9	4	1	0	1	4
3	9	16	25	36	49	64
4	81	100	121	144	169	196

© 2008-2018 Enthought, Inc.

31



NumPy
Array Broadcasting

© 2008-2018 Enthought, Inc.

32

Array Broadcasting

NumPy arrays of different dimensionality can be combined in the same expression. Arrays with smaller dimension are **broadcasted** to match the larger arrays, *without copying data*. Broadcasting has **two rules**.

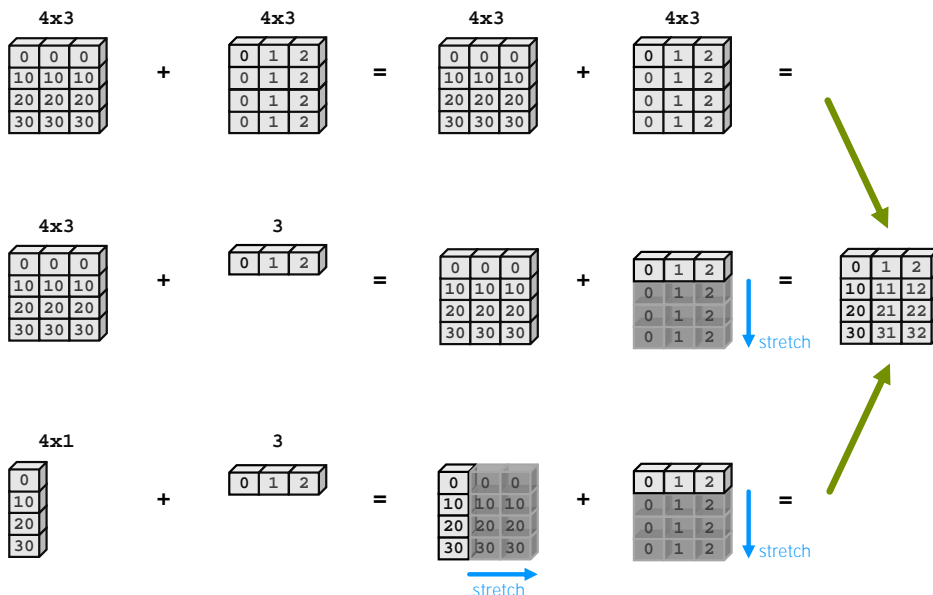
RULE 1: PREPEND ONES TO SMALLER ARRAY'S SHAPE

```
>>> import numpy as np
>>> a = np.ones((3, 5)) # a.shape == (3, 5)
>>> b = np.ones((5,)) # b.shape == (5,)
>>> b.reshape(1, 5) # result is a (1,5)-shaped array.
>>> b[np.newaxis, :] # equivalent, more concise.
```

RULE 2: DIMENSIONS OF SIZE 1 ARE REPEATED WITHOUT COPYING

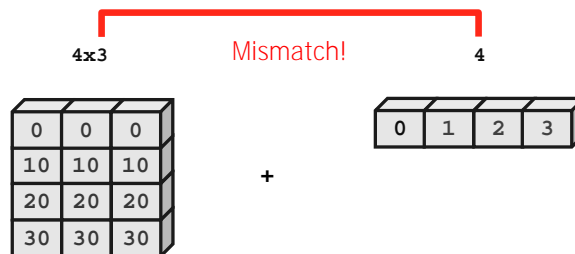
```
>>> c = a + b # c.shape == (3, 5)
# is logically equivalent to...
>>> tmp_b = b.reshape(1, 5)
>>> tmp_b_repeat = tmp_b.repeat(3, axis=0)
>>> c = a + tmp_b_repeat
# But broadcasting makes no copies of "b"s data!
```

Array Broadcasting



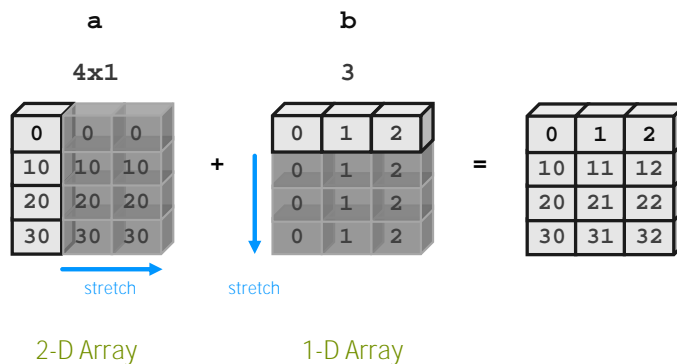
Broadcasting Rules

The trailing axes of either arrays must be 1 or both must have the same size for broadcasting to occur. Otherwise, a `"ValueError: shape mismatch: objects cannot be broadcast to a single shape"` exception is thrown.



Broadcasting in Action

```
>>> a = array([0, 10, 20, 30])
>>> b = array([0, 1, 2])
>>> y = a[:, newaxis] + b
```



Application: Distance from Center

```
>>> import matplotlib.pyplot as plt
>>> a = np.linspace(0, 1, 15) - 0.5
>>> b = a[:, np.newaxis] # b.shape == (15, 1)
>>> dist2 = a**2 + b**2 # broadcasting sum.
>>> dist = np.sqrt(dist2)
>>> plt.imshow(dist)
>>> plt.colorbar()
```

Broadcasting's Usefulness

Broadcasting can often be used to replace needless data replication inside a NumPy array expression.

np.meshgrid() use **newaxis** appropriately in broadcasting expressions.

np.repeat() broadcasting makes repeating an array along a dimension of size 1 unnecessary.

MESHGRID: COPIES DATA

```
>>> x, y = \
...     np.meshgrid([1,2],
...                  [3,4,5])
>>> z = x + y
```

BROADCASTING: NO COPIES

```
>>> x = np.array([1, 2])
>>> y = np.array([3, 4, 5])
>>> z = x[np.newaxis, :] \
...     + y[:, np.newaxis]
```

Broadcasting Indices

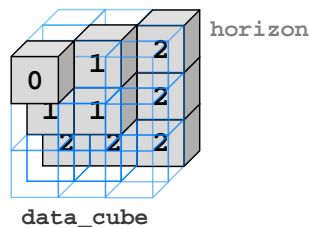
(or any other shape) array. This is a very powerful feature of indexing.

```
data_cube = np.arange(27).reshape(3, 3, 3)
xi, yi, zi = np.meshgrid(np.arange(3), np.arange(3), np.arange(3))
horizon = data_cube[xi, yi, zi]
```

Indices

	yi	0	1	2
xi	0	0	1	2
	1	0	1	2
	2	0	1	2

Selected Data



Universal Function Methods

The mathematical, comparative, logical, and bitwise operators *op* that take two arguments (binary operators) have special methods that operate on arrays:

```
>>> op.reduce(a,axis=0)
>>> op.accumulate(a,axis=0)
>>> op.outer(a,b)
>>> op.reduceat(a,indices)
```

op.reduce()

op.reduce(a) applies **op** to all the elements in a 1-D array **a** reducing it to a single value.

For example:

ADD EXAMPLE

```
>>> a = np.array([1,2,3,4])
>>> np.add.reduce(a)
10
```

STRING LIST EXAMPLE

```
>>> a = np.array(
    ['ab','cd','ef'],
    dtype='object')
>>> np.add.reduce(a)
'abcdef'
```

LOGICAL OP EXAMPLES

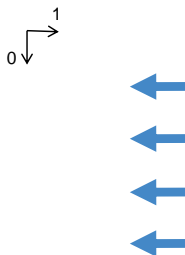
```
>>> a = np.array([1,1,0,1])
>>> np.logical_and.reduce(a)
False
>>> np.logical_or.reduce(a)
True
```

op.reduce()

For multidimensional arrays, **op.reduce(a,axis)** applies **op** to the elements of **a** along the specified **axis**. The resulting array has dimensionality one less than **a**. The default value for **axis** is 0.

SUMMING UP EACH ROW

```
>>> a = np.arange(3) + np.arange(0, 40,
...              10).reshape(-1, 1)
>>> np.add.reduce(a,1)
array([ 3, 33, 63, 93])
```



SUM COLUMNS BY DEFAULT

```
>>> np.add.reduce(a)
array([60, 64, 68])
```



op.accumulate()

op.accumulate(a) creates a new array containing the intermediate results of the **reduce** operation at each element in **a**.

For example:

ADD EXAMPLE

```
>>> a = np.array([1,2,3,4])
>>> np.add.accumulate(a)
array([ 1,  3,  6, 10])
```

STRING LIST EXAMPLE

```
>>> a = np.array(
    ['ab','cd','ef'],
    dtype='object')
>>> np.add.accumulate(a)
array(['ab','abcd','abcdef'],
      dtype=object)
```

LOGICAL OP EXAMPLES

```
>>> a = np.array([1,1,0])
>>> np.logical_and.accumulate(a)
array([True, True, False])
>>> np.logical_or.accumulate(a)
array([True, True, True])
```

op.reduceat()

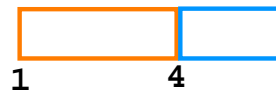
op.reduceat(a,indices) applies **op** to ranges in the 1-D array **a** defined by the values in **indices**. The resulting array has the same length as **indices**.

For example:

y = add.reduceat(a, indices)

EXAMPLE

```
>>> a = np.array(
    [0,10,20,30,40,50])
>>> indices = np.array([1,4])
>>> np.add.reduceat(a,indices)
array([ 60,  90])
```



For multidimensional arrays, **reduceat()** is always applied along the *last* axis (sum of rows for 2-D arrays). This is different from the default for **reduce()** and **accumulate()**.

`np.outer()`

`np.outer(a,b)` forms all possible combinations of elements between `a` and `b` using `np.multiply`. The shape of the resulting array results from concatenating the shapes of `a` and `b`. (Order matters.)

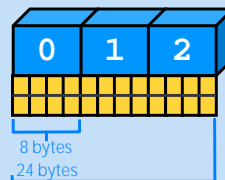
`a`

`b`

```
>>> np.add.outer(a,b)
```

```
>>> np.add.outer(b,a)
```

Memory Block



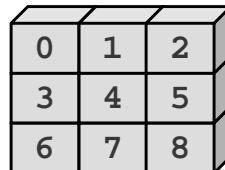
NumPy
The Array Data Structure

Array Data Structure

Memory Block

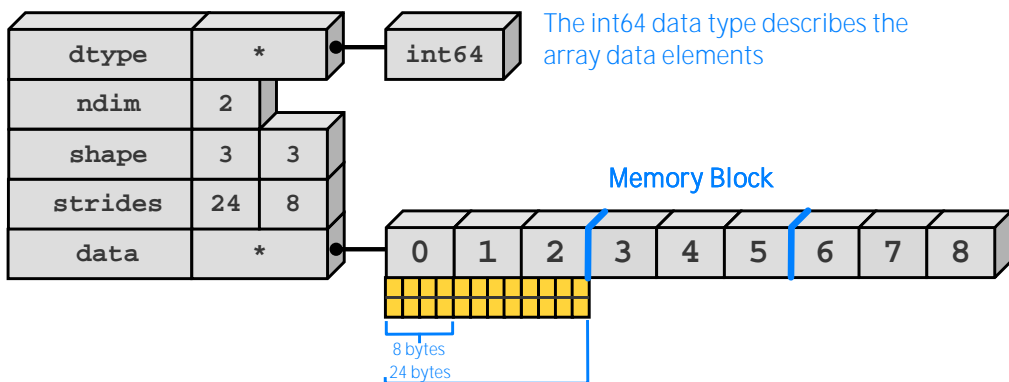


Python View:



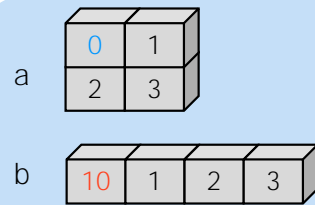
Array Data Structure

NDArray Data Structure



Python View:





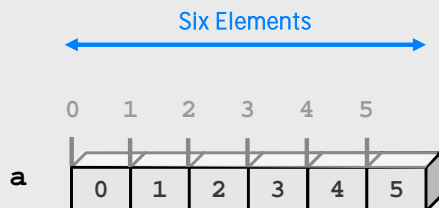
NumPy

Structure Operations

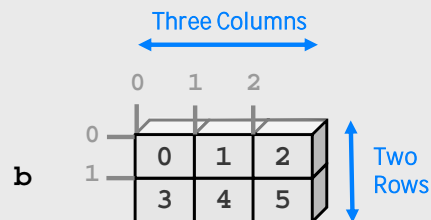
Operations on the Array Structure

Operations that only affect the array structure, not the data, can usually be executed without copying memory.

```
>>> a = np.arange(6)
>>> a
```



```
>>> b = a.reshape(2, 3)
>>> b
```



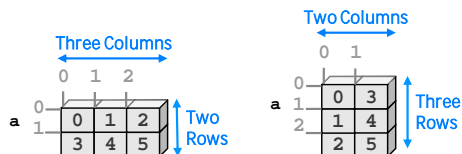
This **is not** a new copy of the data.
The original data **does not** get reordered.

Transpose

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

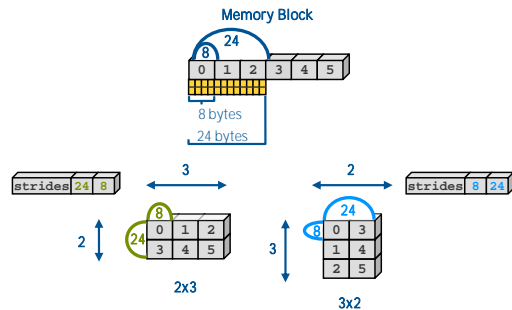
Transpose swaps the order
of axes.

```
>>> a.T
array([[0, 3],
       [1, 4],
       [2, 5]])
>>> a.T.shape
(3,2)
```



Transpose does not move
values around in memory.
It only changes the order
of "strides" in the array

```
>>> a.strides
(24, 8)
>>> a.T.strides
(8, 24)
```



Reshaping Arrays

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

Return a new array with a
different shape (a view
where possible)

```
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Reshape cannot change the
number of elements in an
array

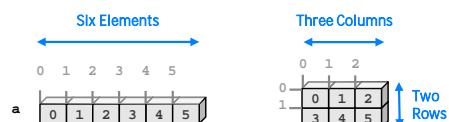
```
>>> a.reshape(4,2)
```

ValueError: total size of new
array must be unchanged

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

Reshape array in-place to
2x3

```
>>> a.shape = (2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
```



Flattening Arrays

a.flatten() converts a multi-dimensional array into a 1-D array. The new array is a *copy* of the original data.

```
# Create a 2D array
```

```
>>> a = np.array([[0,1],  
...               [2,3]])
```

```
# Flatten out elements to 1D
```

```
>>> b = a.flatten()
```

```
>>> b  
array([0,1,2,3])
```

```
# Changing b does not change a
```

```
>>> b[0] = 10
```

```
>>> b
```

```
array([10,1,2,3])
```

```
>>> a  
array([[0, 1],  
       [2, 3]])
```

no change

b 10 1 2 3

a 0 1
2 3

a.ravel() is the same as **a.flatten()**, but returns a *reference (or view)* of the array if possible (i.e., the memory is contiguous). Otherwise the new array copies the data.

```
# Flatten out elements to 1-D
```

```
>>> b = a.ravel()
```

```
>>> b
```

```
array([0,1,2,3])
```

```
# Changing b does change a
```

```
>>> b[0] = 10
```

```
>>> b
```

```
array([10,1,2,3])
```

```
>>> a  
array([[10, 1],  
       [2, 3]])
```

changed!

b 10 1 2 3

a 10 1
2 3