



Introduction to Numerical Computing with Numpy

© 2001-2019, Enthought, Inc.

All Rights Reserved. Use only permitted under license. Copying, sharing, redistributing or other unauthorized use strictly prohibited.

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

Q2-2019
letter



Introduction to Numerical Computing with Numpy

Enthought, Inc.
www.enthought.com

Introduction 1

NumPy 2

An interlude: Matplotlib basics 4

Introducing NumPy Arrays 17

Multi-Dimensional Arrays 21

Slicing/Indexing Arrays 22

Fancy Indexing 28

Creating arrays 32

Array Creation Functions 34

Array Calculation Methods 37

Array Broadcasting 45

Universal Function Methods 53

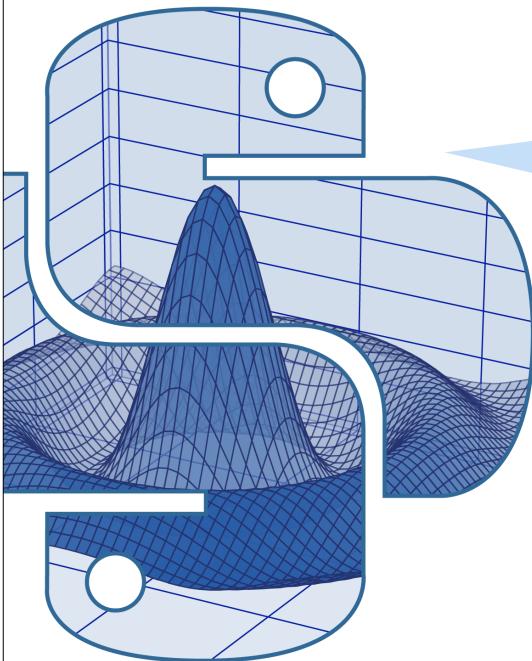
The array data structure 59



Introduction to Numerical Computing with Numpy



1

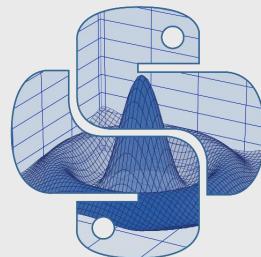


Hi there!

NumPy
The Standard Numerical Library
for Python

NumPy Arrays

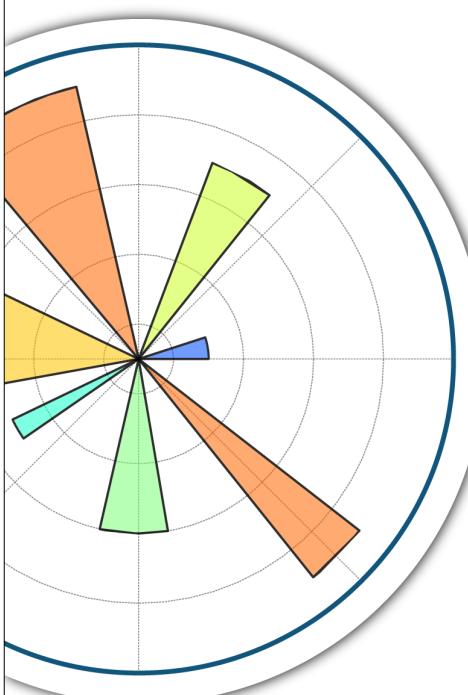
- Defining Arrays
- Indexing and Slicing
- Creating Arrays
- Array Calculations
- The Array Data Structure
- Advanced NumPy, Overview



© 2008-2018 Enthought, Inc.

 ENTHOUGHT 3

 ENTHOUGHT



matplotlib

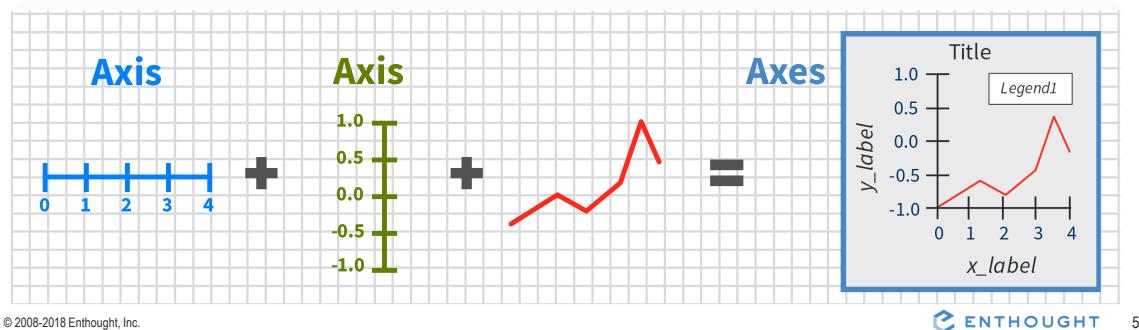
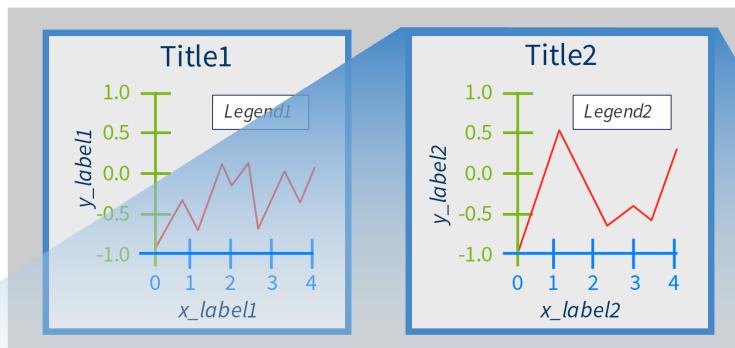
Matplotlib
An Interlude

© 2008-2018 Enthought, Inc.

4

Matplotlib Object Model

Figure



© 2008-2018 Enthought, Inc.

ENTHOUGHT

5

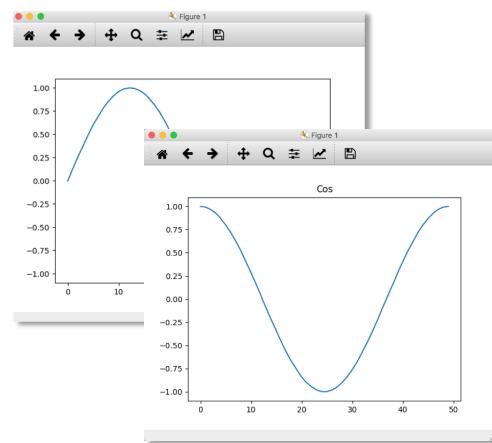
Matplotlib's "State Machine"

Matplotlib behaves like a state machine.
Any command is applied to the current plotting area:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> t = np.linspace(0,2*np.pi,
...                  50)
>>> x = np.sin(t)
>>> y = np.cos(t)

# Now create a figure
>>> plt.figure()
# and plot x inside it
>>> plt.plot(x)

# Now create a new figure
>>> plt.figure()
# and plot y inside it...
>>> plt.plot(y)
# ...and add a title
>>> plt.title("Cos")
```



© 2008-2018 Enthought, Inc.

ENTHOUGHT

6

Line Plots

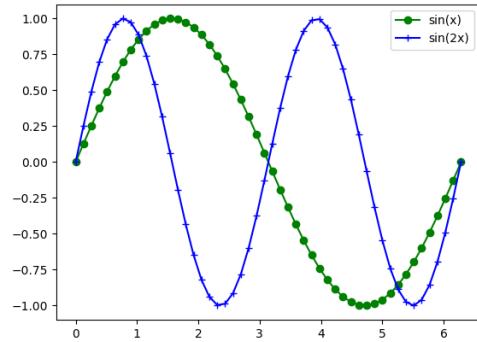
```
>>> x = np.linspace(0, 2*np.pi, 50)
>>> y1 = np.sin(x)
>>> y2 = np.sin(2*x)

>>> plt.figure() # Create figure
>>> plt.plot(y1)
>>> plt.plot(x, y1)

# red dot-dash circle
>>> plt.plot(x, y1, 'r-o')

# red marker only circle
>>> plt.plot(x, y1, 'ro')

# clear figure then plot 2 curves
>>> plt.clf()
>>> plt.plot(x, y1, 'g-o',
...             x, y2, 'b-+')
>>> plt.legend(['sin(x)',
...             'sin(2x)'])
```



Symbol	Color	Symbol	Marker
b	Blue	.	Point
g	Green	o	Circle
r	Red	< > ^ v	Triangles
c	Cyan	8	Octagon
m	Magenta	s	Square
y	Yellow	*	Star
k	Black	+	Plus
w	White		...and more

© 2008-2018 Enthought, Inc.

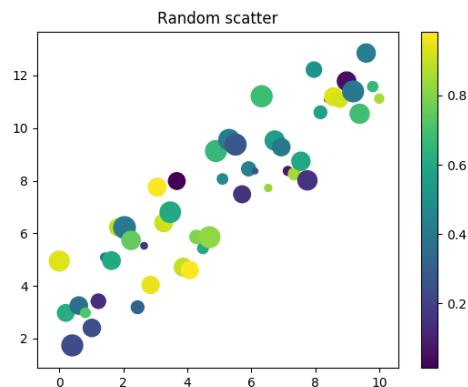
ENTHOUGHT

7

Scatter Plots

```
>>> N = 50 # no. of points
>>> x = np.linspace(0, 10, N)
>>> from numpy.random \
...     import rand
>>> e = rand(N)*5.0 # noise
>>> y1 = x + e

>>> areas = rand(N)*300
>>> plt.scatter(x, y1, s=areas)
>>> colors = rand(N)
>>> plt.scatter(x, y1, s=areas,
...               c=colors)
>>> plt.colorbar()
>>> plt.title("Random scatter")
```



© 2008-2018 Enthought, Inc.

ENTHOUGHT

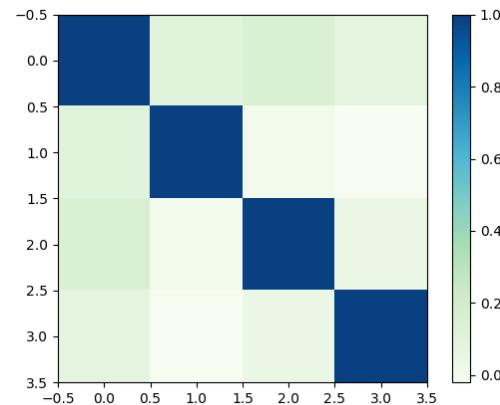
8

Image "Plots"

```
>>> # Create some data
>>> e1 = rand(100)
>>> e2 = rand(100)*2
>>> e3 = rand(100)*10
>>> e4 = rand(100)*100
>>> corrmatrix = \
...     np.corrcoef([e1, e2,
...                  e3, e4])
```



```
>>> # Plot corr matrix as image
>>> plt.imshow(corrmatrix,
...             cmap='GnBu')
>>> plt.colorbar()
```



© 2008-2018 Enthought, Inc.

 ENTHOUGHT 9

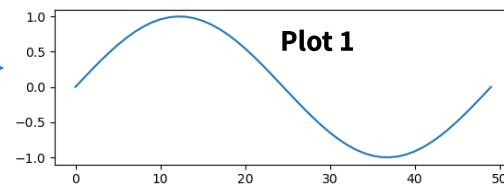
Multiple Plots Using subplot

```
>>> t = np.linspace(0, 2*np.pi)
>>> x = np.sin(t)
>>> y = np.cos(t)
```

```
# To divide the plotting area
```

columns
|
|
rows active plot

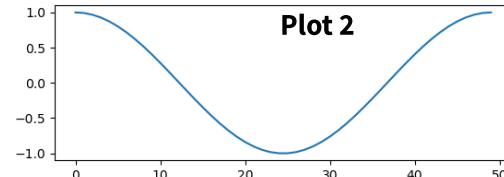
```
>>> plt.subplot(2, 1, 1) →
>>> plt.plot(x)
```



```
# Now activate a new plot
```

```
# area.
```

```
>>> plt.subplot(2, 1, 2) →
>>> plt.plot(y)
```



© 2008-2018 Enthought, Inc.

 ENTHOUGHT 10

Histogram Plots

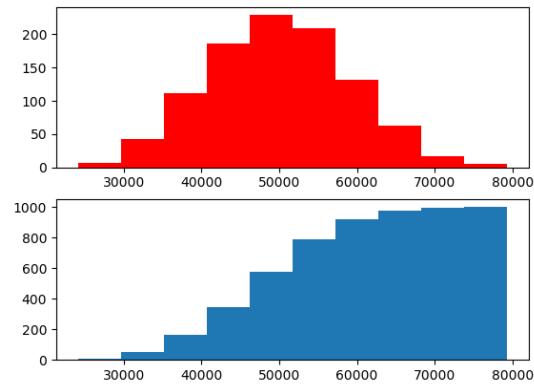
```
>>> # Create array of data
>>> from numpy.random import \
...     randint
>>> data = randint(10000,
...     size=(10,1000))

>>> # Approx norm distribution
>>> x = np.sum(data, axis=0)

>>> # Set up for stacked plots
>>> plt.subplot(2,1,1)
>>> plt.hist(x, color='r')

>>> # Plot cumulative dist
>>> plt.subplot(2,1,2)
>>> plt.hist(x, cumulative=True)

>>> # For multiple histograms:
>>> plt.hist([d1, d2, ...])
```



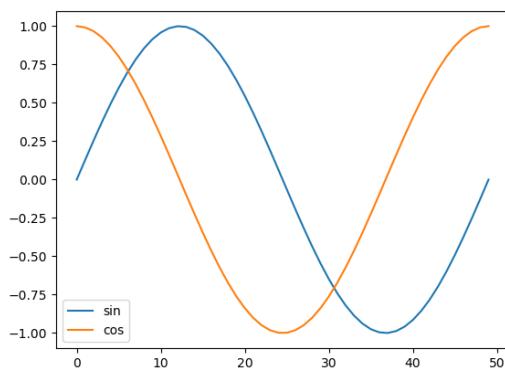
© 2008-2018 Enthought, Inc.

 ENTHOUGHT 11

Legends, Titles, and Axis Labels

LEGEND LABELS WITH PLOT

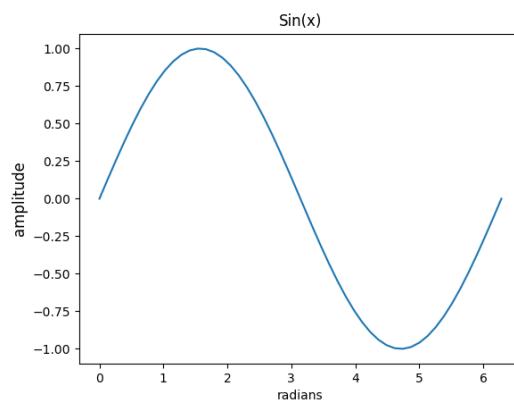
```
# Add labels in plot command.
>>> plt.plot(np.sin(t),
...           label='sin')
>>> plt.plot(np.cos(t),
...           label='cos')
>>> plt.legend()
```



© 2008-2018 Enthought, Inc.

TITLES AND AXIS LABELS

```
>>> plt.plot(t, np.sin(t))
>>> plt.xlabel('radians')
# Keywords set text properties.
>>> plt.ylabel('amplitude',
...             fontsize='large')
>>> plt.title('Sin(x)')
```



 ENTHOUGHT 12

Plotting from Scripts

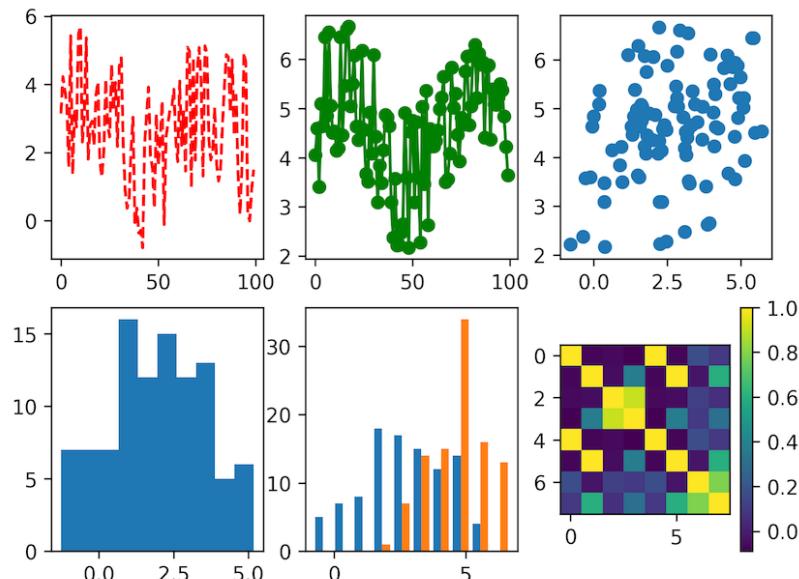
INTERACTIVE MODE

```
# In IPython, plots show up
# as soon as a plot command
# is issued:
>>> plt.figure()
>>> plt.plot(np.sin(t))
>>> plt.figure()
>>> plt.plot(np.cos(t))
```

NON-INTERACTIVE MODE

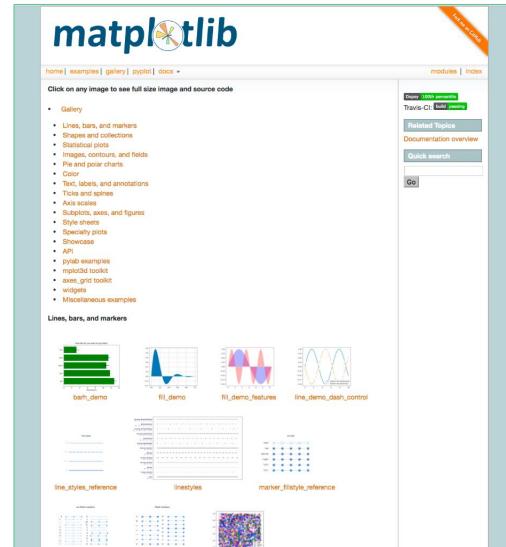
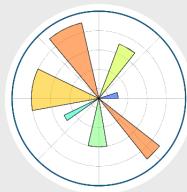
```
# In a script, you must call
# the show() command to display
# plots. Call it at the end of
# all your plot commands for
# best performance.
import matplotlib.pyplot as plt
import numpy as np
t = np.linspace(0,2*np.pi,
                50)
plt.figure()
plt.plot(np.sin(t))
plt.figure()
plt.plot(np.cos(t))
# Plots will not appear until
# this command is run:
plt.show()
```

MPL Exercise: Desired Output



Additional Matplotlib Resources

- Simple examples with increasing difficulty
<http://matplotlib.org/examples/index.html>
- Gallery (huge)
<http://matplotlib.org/gallery.html>
- See appendix for reference materials
- Usage FAQ
http://matplotlib.org/faq/usage_faq.html

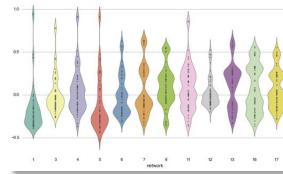
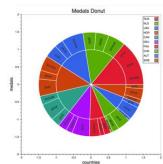


© 2008-2018 Enthought, Inc.

 ENTHOUGHT 15

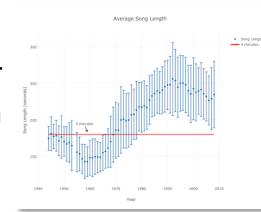
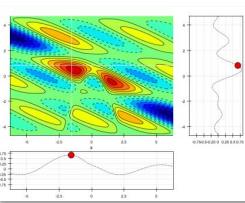
Other Visualization Libraries

Seaborn: Better looking, high-level plots based on matplotlib



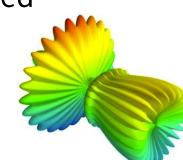
bokeh: D3-like visualization in web browsers (HTML file, or inline in IPython notebooks)

plot.ly: D3-based visualization with support for multiple languages (R, Python, Matlab, Julia and more.)



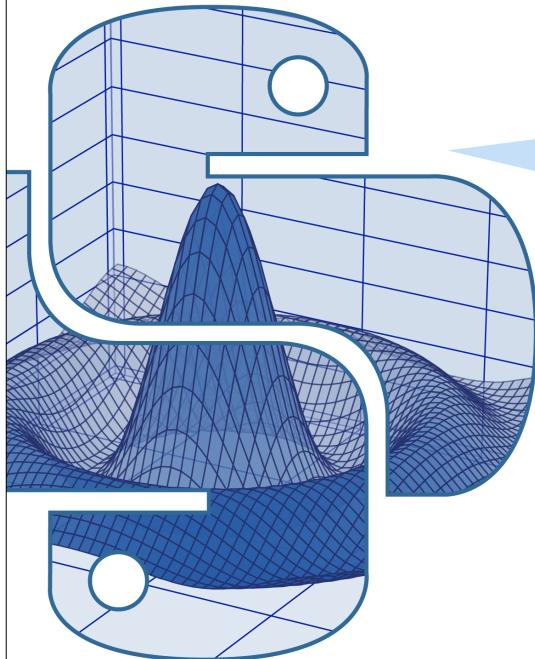
chaco: For interactive, custom plots embedded in user applications

mayavi: 3D visualization, based on VTK



© 2008-2018 Enthought, Inc.

 ENTHOUGHT 16



© 2008-2018 Enthought, Inc.



NumPy

Introducing Arrays

17

Introducing NumPy Arrays

SIMPLE ARRAY CREATION

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

CHECKING THE TYPE

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

NUMERIC "TYPE" OF ELEMENTS

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

NUMBER OF DIMENSIONS

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

ARRAY SHAPE

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

BYTES PER ELEMENT

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

BYTES OF MEMORY USED

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

© 2008-2018 Enthought, Inc.

Array Operations

SIMPLE ARRAY MATH

```
>>> 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:
 $\pi = 3.14159265359$
 $e = 2.71828182846$

MATH FUNCTIONS

```
# create array from 0. to 10.
>>> x = np.arange(11.)

# multiply entire array by
# scalar value
>>> c = (2 * np.pi) / 10.
>>> c
0.62831853071795862
>>> c * x
array([ 0., 0.628,...,6.283])

# in-place operations
>>> x *= c
>>> x
array([ 0., 0.628,...,6.283])

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

Setting Array Elements

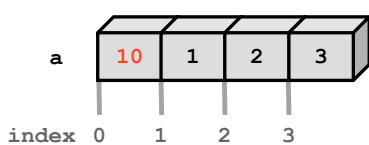
ARRAY INDEXING

```
>>> a[0]
0

a
[[ 0  1  2  3]
 [index 0 1 2 3]]
```

`>>> a[0] = 10`

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



BEWARE OF TYPE COERCION

```
>>> 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])
```

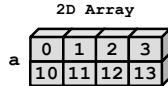
Multi-Dimensional Arrays

MULTI-DIMENSIONAL ARRAYS

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

```
>>> a
```

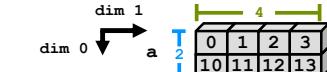
```
array([[ 0,  1,  2,  3],  
       [10,11,12,13]])
```



SHAPE = (ROWS, COLUMNS)

```
>>> a.shape
```

```
(2, 4)
```



ELEMENT COUNT

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

```
8
```



NUMBER OF DIMENSIONS

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

```
2
```



© 2008-2018 Enthought, Inc.

GET / SET ELEMENTS

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

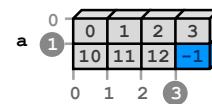
```
13
```

column
row

```
>>> a[1, 3] = -1
```

```
>>> a
```

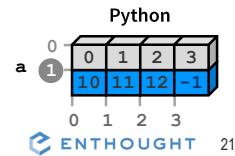
```
array([[ 0,  1,  2,  3],  
       [10,11,12,-1]])
```



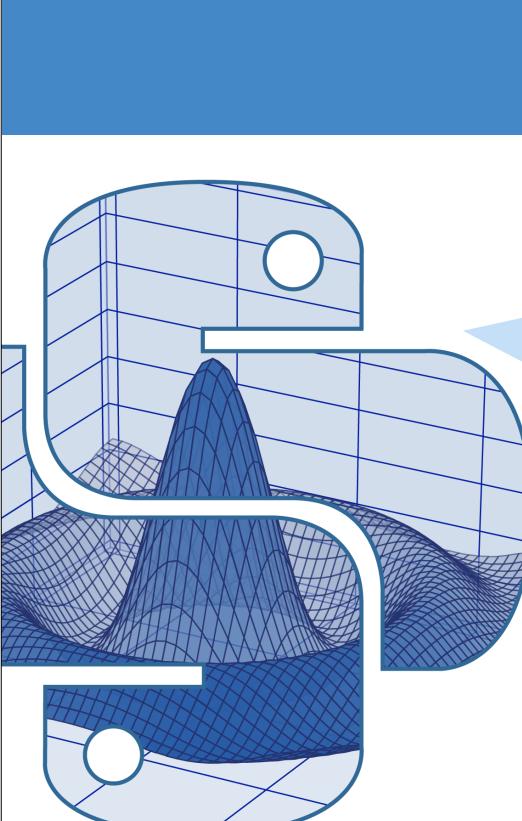
ADDRESS SECOND (ONETH) ROW USING SINGLE INDEX

```
>>> a[1]
```

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



21



The image shows a 3D surface plot of a mathematical function, likely $\sin(x)/x$, rendered in blue. A white rectangular plane cuts through the surface, revealing its underlying grid structure. This visual metaphor represents a multi-dimensional array.

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

NumPy
Indexing and Slicing

© 2008-2018 Enthought, Inc.

22

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.

SLICING ARRAYS

```
#           -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])
```

© 2008-2018 Enthought, Inc.

OMMITTING INDICES

```
# 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])
```

 ENTHOUGHT 23

Array Slicing

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

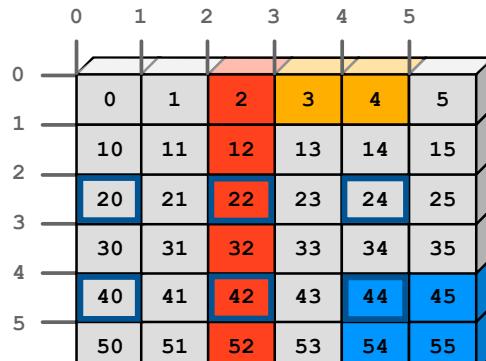
```
>>> a[0, 3:5]
array([3, 4])

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

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

STRIDED ARE ALSO POSSIBLE

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



© 2008-2018 Enthought, Inc.

 ENTHOUGHT 24

Slices are References

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

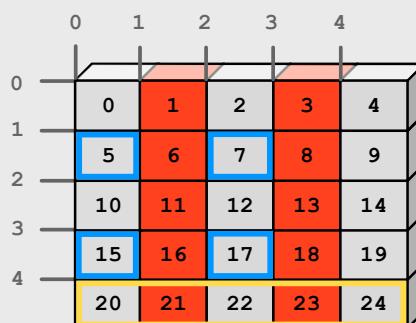
```
>>> a = np.array([0, 1, 2, 3, 4])
# slicing the last two elements returns the data there
>>> a[-2:]
array([3, 4])
# we can insert an iterable of length two
>>> a[-2:] = [-1, -2]
>>> a
array([ 0,  1,  2, -1, -2])
# or a scalar value
>>> a[-2:] = 99
>>> a
array([ 0,  1,  2, 99, 99])
```

Give it a try!

Create the array below with the command

```
a = np.arange(25).reshape(5, 5)
```

and extract the slices as indicated.



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])
```

© 2008-2018 Enthought, Inc.

 ENTHOUGHT 27

Fancy Indexing

INDEXING BY POSITION

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

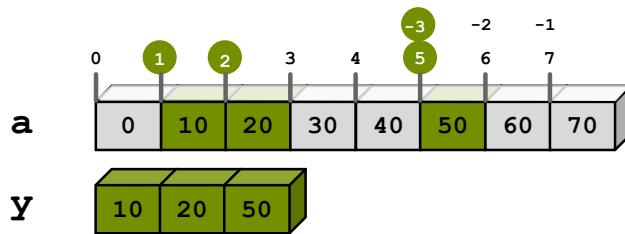
# fancy indexing
>>> indices = [1, 2, -3]
>>> y = a[indices]
>>> print(y)
[10 20 50]

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

INDEXING WITH BOOLEANS

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

# fancy indexing
>>> y = a[mask]
>>> print(y)
array([99, 99, 99])
```



© 2008-2018 Enthought, Inc.

 ENTHOUGHT 28

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])
```

	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



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

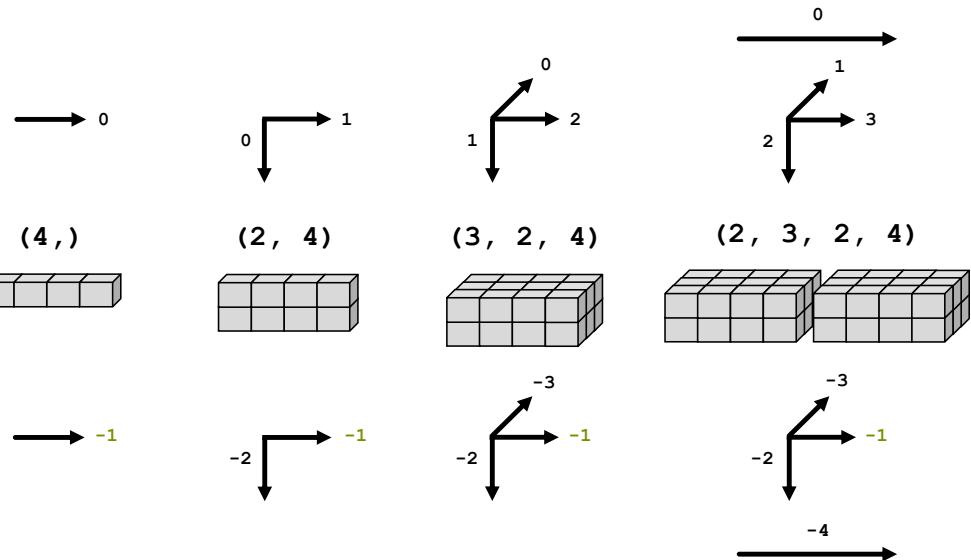
Give it a try!

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	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24

Multi-Dimensional Arrays

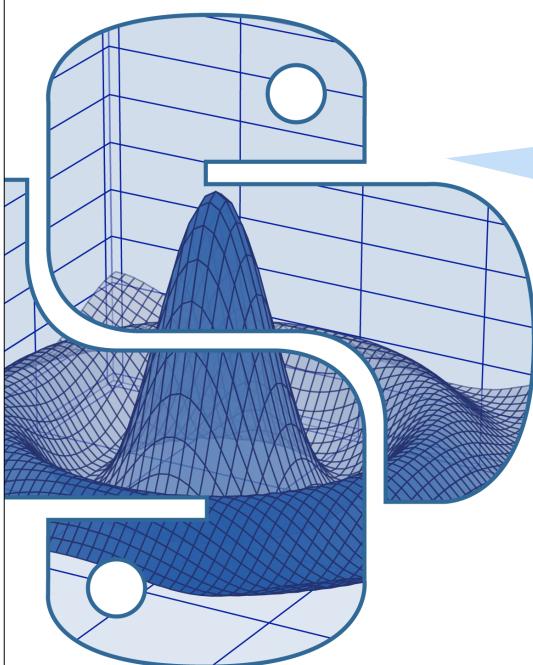
VISUALIZING MULTI-DIMENSIONAL ARRAYS



© 2008-2018 Enthought, Inc.

 ENTHOUGHT 31

**orange()
linspace()
array()
zeros()
ones()**



NumPy
Creating Arrays

© 2008-2018 Enthought, Inc.

32

Array Constructor Examples

FLOATING POINT ARRAYS

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

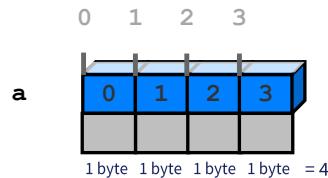
REDUCING PRECISION

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

© 2008-2018 Enthought, Inc.

UNSIGNED INTEGER BYTE

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



Base 2	Base 10
00000000	-> 0 = 0*2**0 + 0*2**1 + ... + 0*2**7
00000001	-> 1 = 1*2**0 + 0*2**1 + ... + 0*2**7
00000010	-> 2 = 0*2**0 + 1*2**1 + ... + 0*2**7
...	
11111111	-> 255 = 1*2**0 + 1*2**1 + ... + 1*2**7

ENTHOUGHT 33

Array Creation Functions

ARANGE

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

Nearly identical to Python's `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])

# Be careful...
>>> np.arange(1.5, 2.1, 0.3)
array([ 1.5,  1.8,  2.1])
```

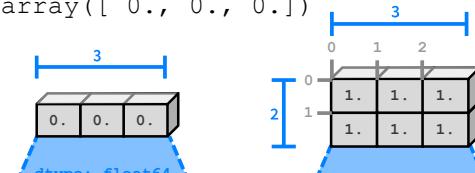
© 2008-2018 Enthought, Inc.

ONES, ZEROS

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



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

ENTHOUGHT 34

Array Creation Functions (cont'd)

IDENTITY

```
# 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 AND FILL

```
# 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.])
```

© 2008-2018 Enthought, Inc.

 ENTHOUGHT 35

Array Creation Functions (cont'd)

LINSPACE

```
# 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])
```

ARRAYS FROM/TO TXT FILES

```
BEGINNING OF THE FILE
% Day, Month, Year, Skip, Avg Power
01, 01, 2000, x876, 13 % crazy day!
% we don't have Jan 03rd
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 = "%")

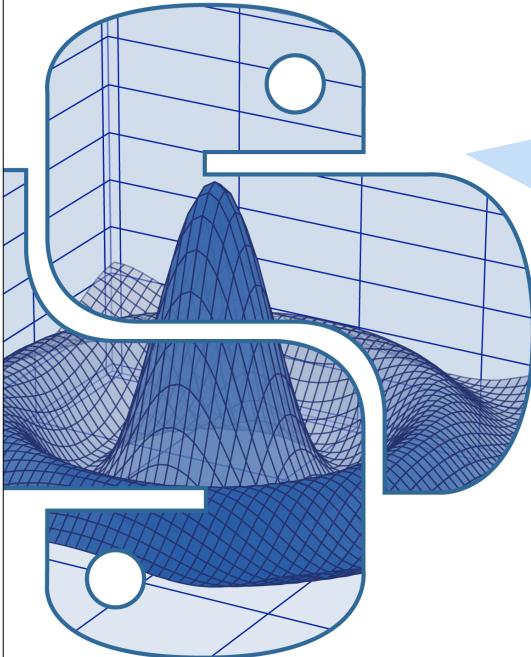
# Save an array into a txt file
np.savetxt('filename', arr)
```

LOGSPACE

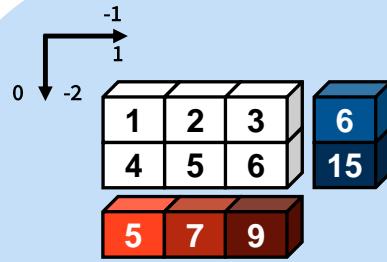
```
# Generate N evenly spaced
# elements on a log scale
# between base**start and
# base**stop (default base=10)
>>> np.logspace(0, 1, 5)
array([1., 1.78, 3.16, 5.62, 10.])
```

© 2008-2018 Enthought, Inc.

 ENTHOUGHT 36



© 2008-2018 Enthought, Inc.



NumPy Array Calculation Methods

37

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, ...).

© 2008-2018 Enthought, Inc.

Array Calculation Methods

SUM METHOD

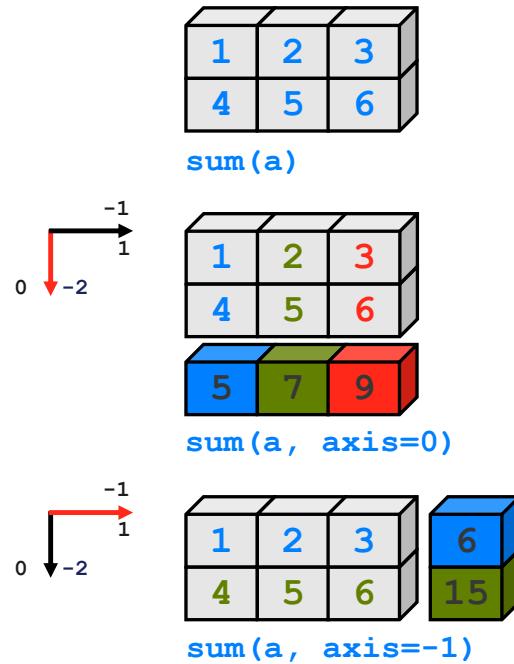
```
# Methods act on data stored
# in the array
>>> a = np.array([[1,2,3],
   [4,5,6]])

# .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.



ENTHOUGHT 39

Other Operations on Arrays

SUM FUNCTION

```
# 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])
```

OTHER METHODS AND FUNCTIONS

Mathematical functions

- sum, prod
- min, max, argmin, argmax
- ptpt (max - min)

Statistics

- mean, std, var

Truth value testing

- any, all

See the Numpy appendix for more.

© 2008-2018 Enthought, Inc.

ENTHOUGHT 40

Min/Max

MIN

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

MAX

```
# 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([3, 1])
```

© 2008-2018 Enthought, Inc.

ARGMIN/MAX

```
# 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 1D, on a raveled
# index of the original array
>>> np.argmin(a)
2
```

UNRAVELING

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

 ENTHOUGHT 41

Where

COORDINATE LOCATIONS

```
# 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])
>>> mask = a % 2 == 0
array([ True, False,  True,
False], dtype=bool)
# Coordinates are returned as
# a tuple of arrays, one for
# each axis
>>> np.where(mask)
(array([0, 2]),)
```

CONDITIONAL ARRAY CREATION

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

© 2008-2018 Enthought, Inc.

 ENTHOUGHT 42

Statistics Array Methods

MEAN

```
>>> 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 DEV./VARIANCE

```
# 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!

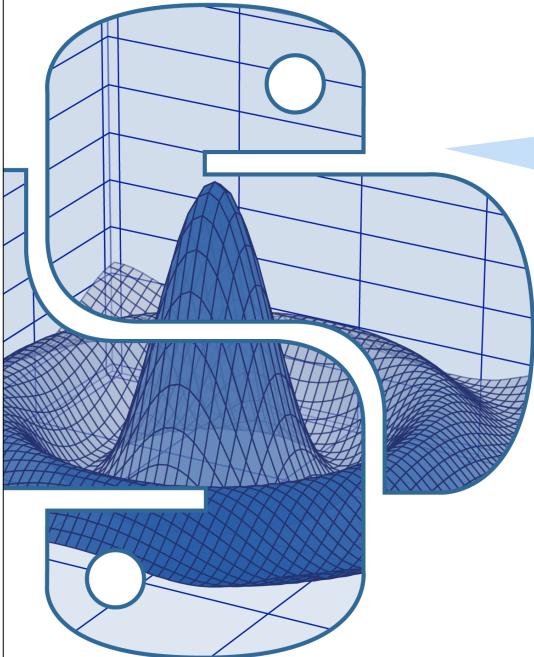
Create the array below with

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

and compute:

1. The maximum of each row (one max per row)
2. The mean of each column (one mean per column)
3. The position of the overall minimum (requires 2-3 steps)

	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



© 2008-2017 Enthought, Inc.

$$\begin{array}{c} \text{a} \\ \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 10 & 10 & 10 \\ \hline 20 & 20 & 20 \\ \hline 30 & 30 & 30 \\ \hline \end{array} \end{array} + \begin{array}{c} \text{b} \\ \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline \end{array} \end{array} = \begin{array}{c} \text{y} \\ \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 10 & 11 & 12 \\ \hline 20 & 21 & 22 \\ \hline 30 & 31 & 32 \\ \hline \end{array} \end{array}$$

NumPy Array Broadcasting

45

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.
```

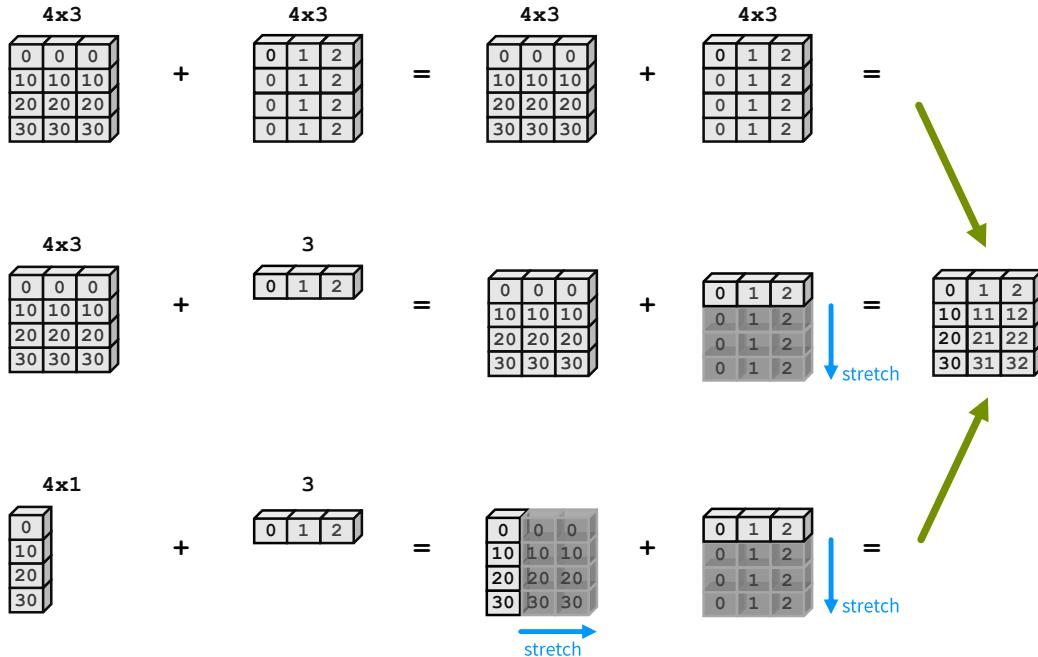
RULE 2: DIMENSIONS OF SIZE 1 ARE REPEATED WITHOUT COPYING

```
>>> 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!
```

© 2008-2017 Enthought, Inc.

46

Array Broadcasting

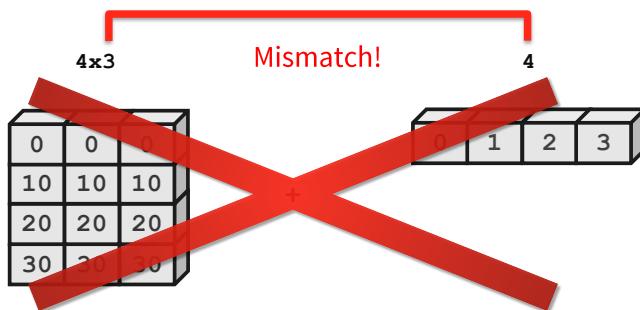


© 2008-2017 Enthought, Inc.

47

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.

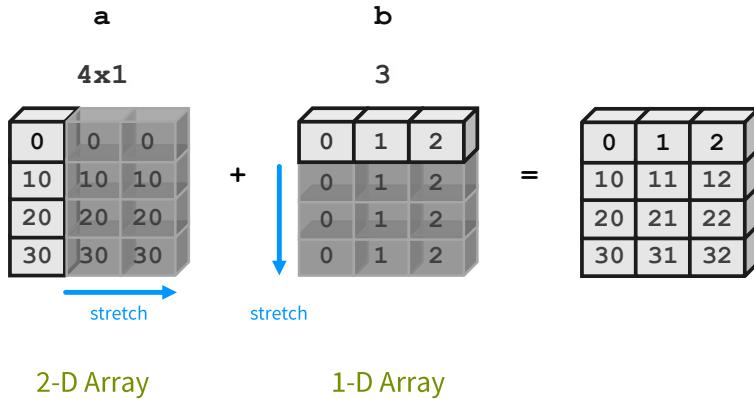


© 2008-2017 Enthought, Inc.

48

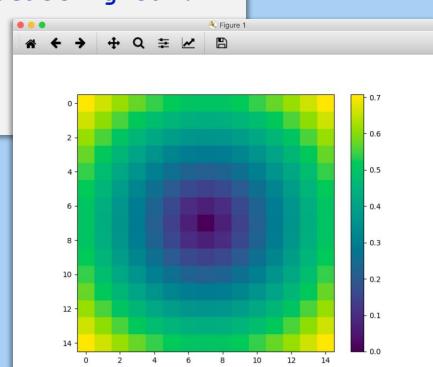
Broadcasting in Action

```
>>> 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

Broadcasting can also be used to slice elements from different “depths” in a 3-D (or any other shape) array. This is a very powerful feature of indexing.

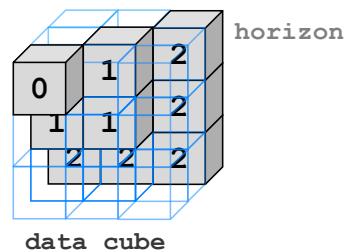
```
>>> data_cube = np.arange(27).reshape(3, 3, 3)
>>> yi, xi = np.meshgrid(np.arange(3), np.arange(3),
...                         sparse=True)
>>> zi = np.array([[0, 1, 2],
...                 [1, 1, 2],
...                 [2, 2, 2]])
>>> horizon = data_cube[yi, xi, zi]
```

Indices

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

zi

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)
```

53

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:

$$\begin{aligned} y &= \text{add.reduce}(a) \\ &= \sum_{n=0}^{N-1} a[n] \\ &= a[0] + a[1] + \dots + a[N-1] \end{aligned}$$

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
```

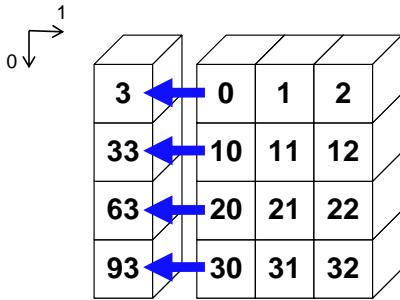
54

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

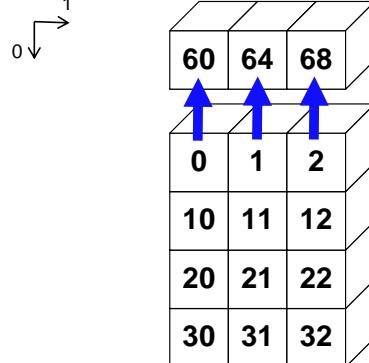
```
>>> a = np.arange(3) + np.arange(0, 40,
...                      10).reshape(-1, 1)
>>> np.add.reduce(a, 1)
array([ 3, 33, 63, 93])
```



SUM COLUMNS BY DEFAULT

```
>>> np.add.reduce(a)
```

```
array([60, 64, 68])
```



55

op.accumulate()

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

For example:

$$y = \text{add.accumulate}(a) \\ = \left[\sum_{n=0}^0 a[n], \sum_{n=0}^1 a[n], \dots, \sum_{n=0}^{N-1} a[n] \right]$$

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

56

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`.

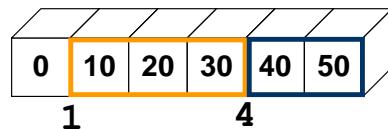
For example:

`y = add.reduceat(a, indices)`

$$y[i] = \sum_{n=indices[i]}^{indices[i+1]} a[n]$$

EXAMPLE

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

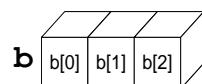
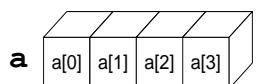


For multidimensional arrays, `reduceat()` is always applied along the *last axis* (sum of rows for 2-D arrays). This is different from the default for `reduce()` and `accumulate()`.

57

op.outer()

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



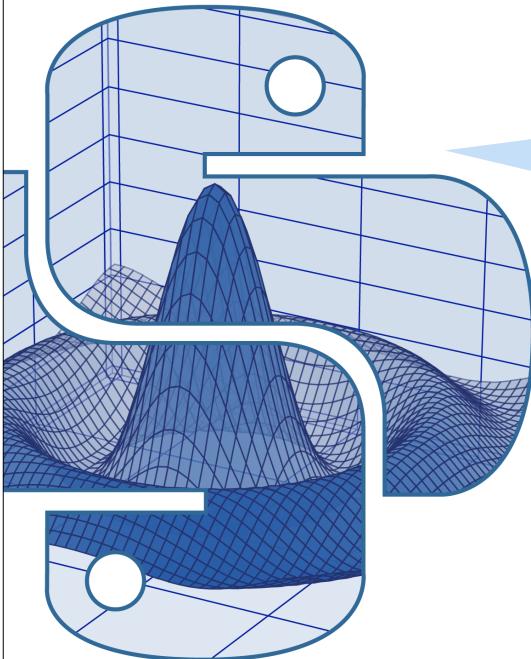
```
>>> np.add.outer(a,b)
```

a[0]+b[0]	a[0]+b[1]	a[0]+b[2]
a[1]+b[0]	a[1]+b[1]	a[1]+b[2]
a[2]+b[0]	a[2]+b[1]	a[2]+b[2]
a[3]+b[0]	a[3]+b[1]	a[3]+b[2]

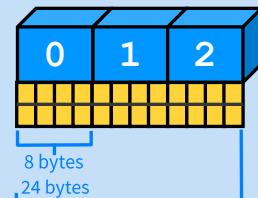
```
>>> np.add.outer(b,a)
```

b[0]+a[0]	b[0]+a[1]	b[0]+a[2]	b[0]+a[3]
b[1]+a[0]	b[1]+a[1]	b[1]+a[2]	b[1]+a[3]
b[2]+a[0]	b[2]+a[1]	b[2]+a[2]	b[2]+a[3]

58



Memory Block



NumPy

The Array Data Structure

© 2008-2018 Enthought, Inc.

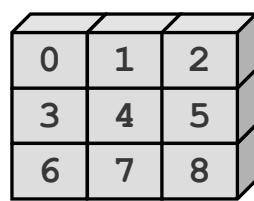
59

Array Data Structure

Memory Block



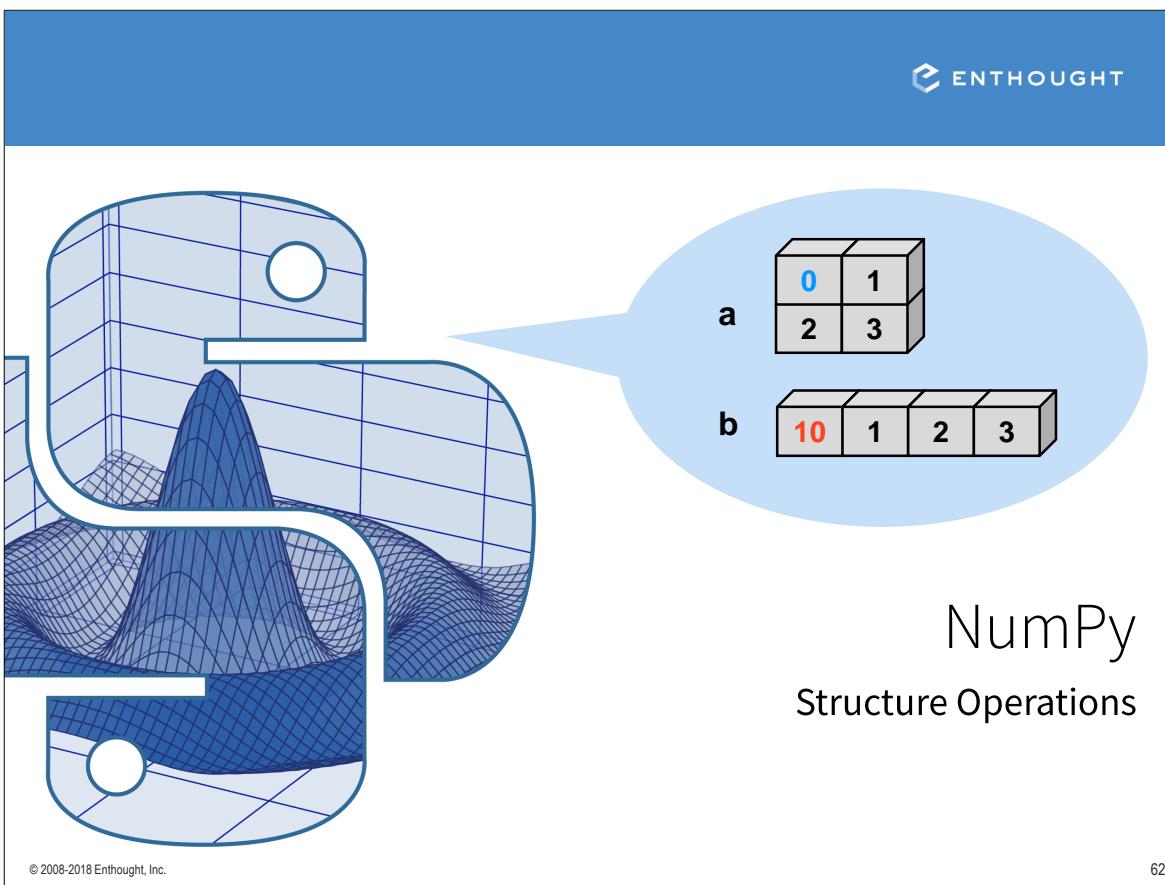
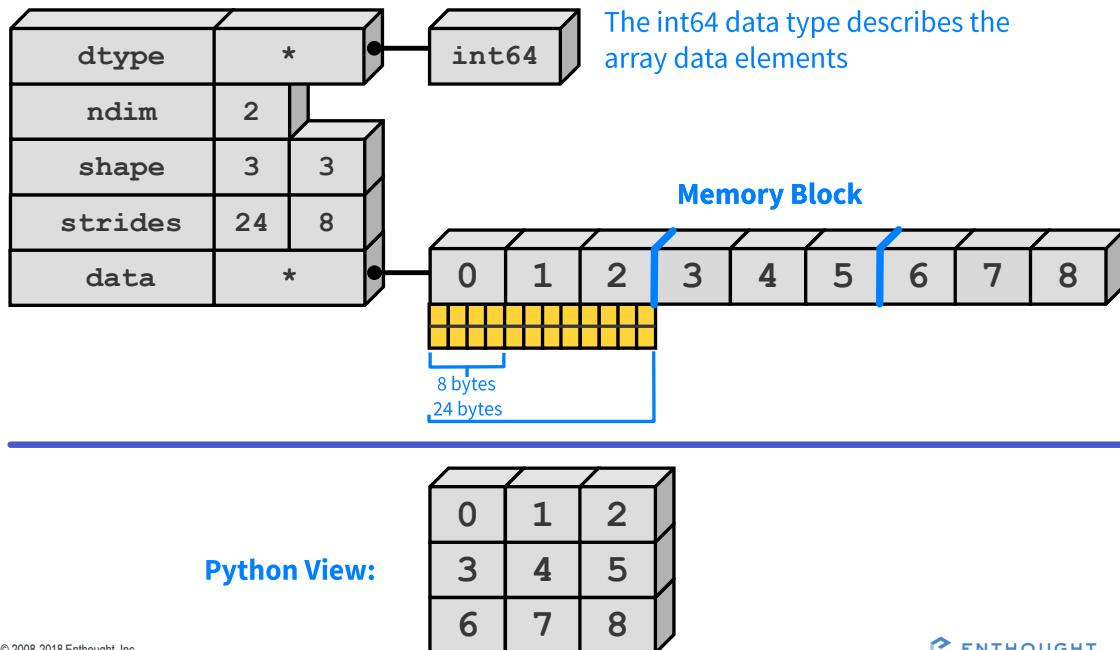
Python View:



© 2008-2018 Enthought, Inc.

Array Data Structure

NDArray Data Structure



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
```

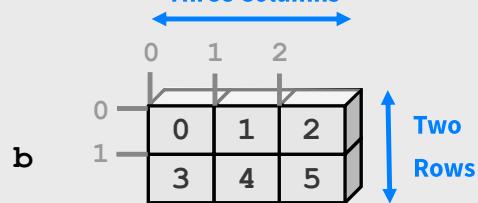
Six Elements



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

```
>>> b
```

Three Columns



This is not a new copy of the data.

The original data does not get reordered.

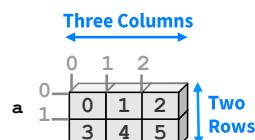
Transpose

TRANSPOSE

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

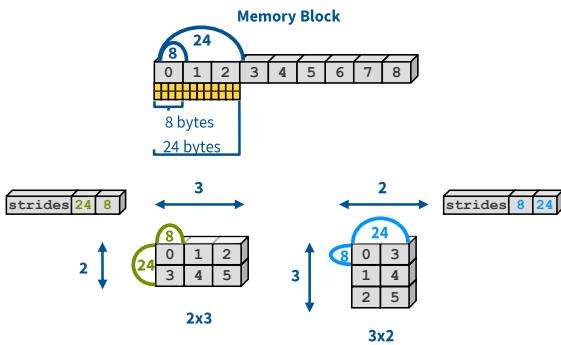
```
# Transpose swaps the order  
# of axes.
```

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



TRANSPOSE RETURNS VIEWS

```
# 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

RESHAPE

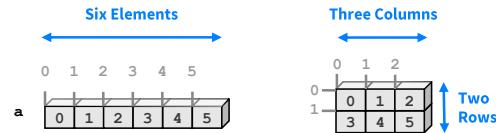
```
>>> a = np.array([[0,1,2],  
...                  [3,4,5]])  
  
# Return a new array with a  
# different shape (a view  
# where possible)  
>>> a.reshape(3,2)  
array([[0, 1],  
      [2, 3],  
      [4, 5]])  
  
# Reshape cannot change the  
# number of elements in an  
# array  
>>> a.reshape(4,2)  
ValueError: total size of new  
array must be unchanged
```

© 2008-2018 Enthought, Inc.

SHAPE

```
>>> 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]])
```



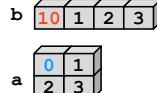
ENTHOUGHT 65

Flattening Arrays

FLATTEN (SAFE)

`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]])
```

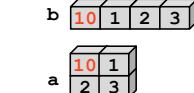


© 2008-2018 Enthought, Inc.

RAVEL (EFFICIENT)

`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]])
```



ENTHOUGHT 66