ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Introduction to Numerical Computing with Numpy

Enthought, Inc.
www.enthought.com

Enthought, Inc.
515 Congress Avenue
Suite 2100
Austin, TX 78701

www.enthought.com

Q2-2017
letter
1.0.0

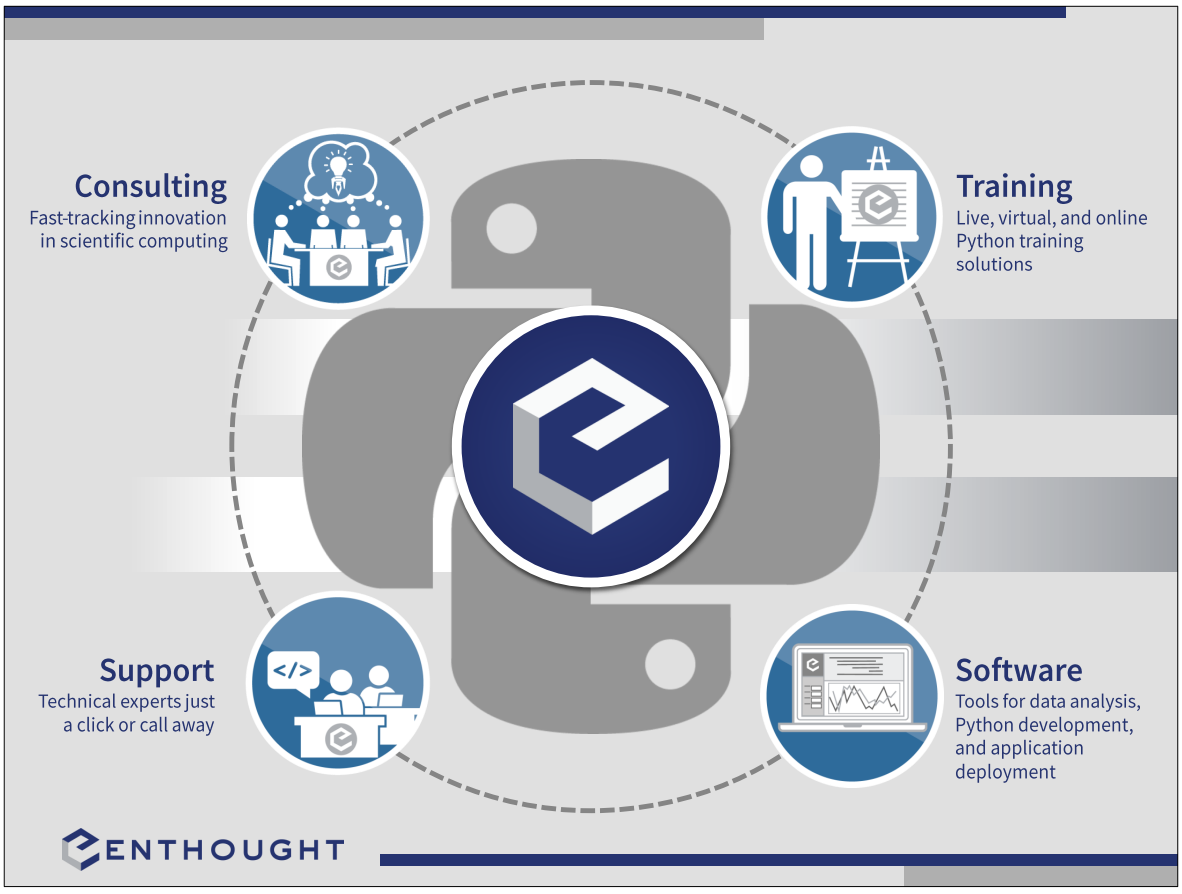
Introduction to Numerical Computing with Numpy

Enthought, Inc.
www.enthought.com

ENTHOUGHT

What We Do

Consulting | Training | Software & Support



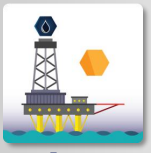
Consulting

For more than fifteen years, Enthought has solved technical computing challenges in industries such as energy, industrial control, consumer products, technology, finance, aerospace, and more. We understand the unique challenges faced by scientists, engineers, and analysts – because we've faced them too.

Our domain aptitude and development expertise allows us to fill the gaps between data and insight, ideas and results – translating needs into solutions.


Aerospace & Defense


Finance


Energy


Manufacturing


Biotechnology

Consumer Products

Semiconductors

…and more

ENTHOUGHT

Training

Enthought's intensive Python training courses are tailored for scientists, engineers, financial analysts, and data scientists who would like to learn how to use Python in their specific work environments. Our targeted courses address both the scientific and programming best practices when working in Python.

Virtual Training

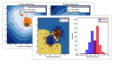

Pandas Crash Course

On-Demand Training


ENTHOUGHT TRAINING ON DEMAND

Live Training

Python for Data Science
Pandas Master Class



Live Training

Python for Scientists & Engineers
Python Foundations

ENTHOUGHT

Software and Support

Enthought offers a variety of commercial software tools and applications for development, data analysis, and visualization, both general and for specific industries. We have over a million users worldwide.

Our support plans ensure users always have someone to contact with their software or Python-related questions.

Integrated Analysis Environment and Python Package Distribution — ENTHOUGHT CANOPY

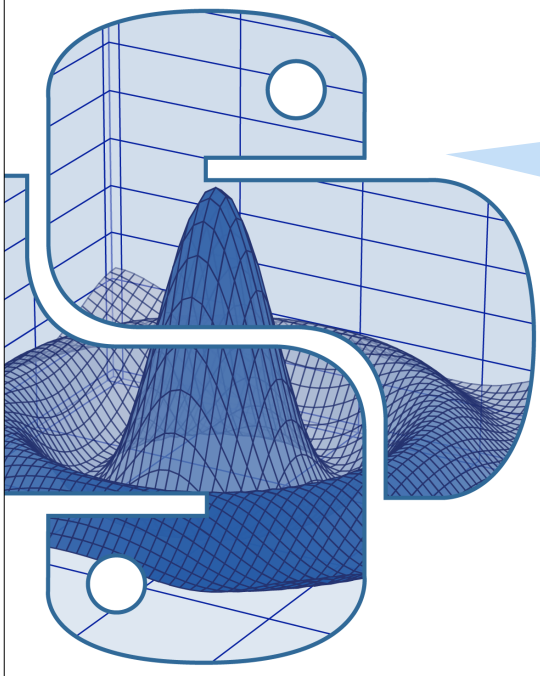
Deployment Solutions — Enthought Deployment Server

Python for Excel — pyxll

Industry-Specific Software — VIRTUAL CORE, GEOSCIENCE CANOPY

ENTHOUGHT

ENTHOUGHT



Hi there!

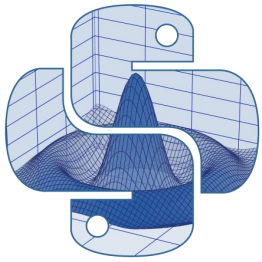
NumPy

The Standard Numerical Library for Python

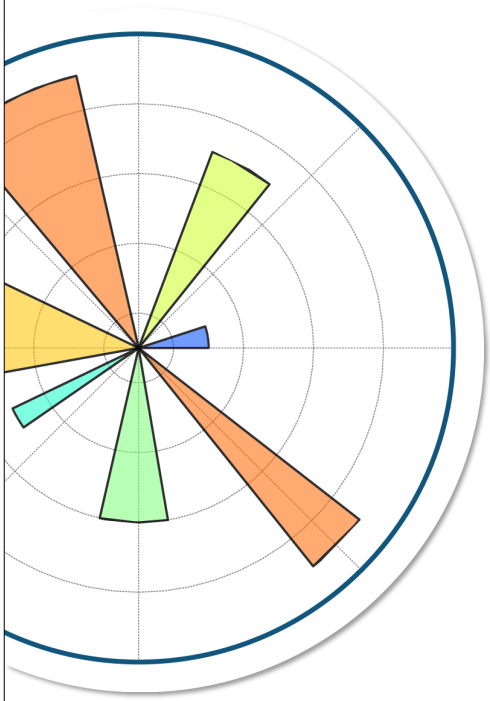
NumPy Arrays

ENTHOUGHT

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



ENTHOUGHT



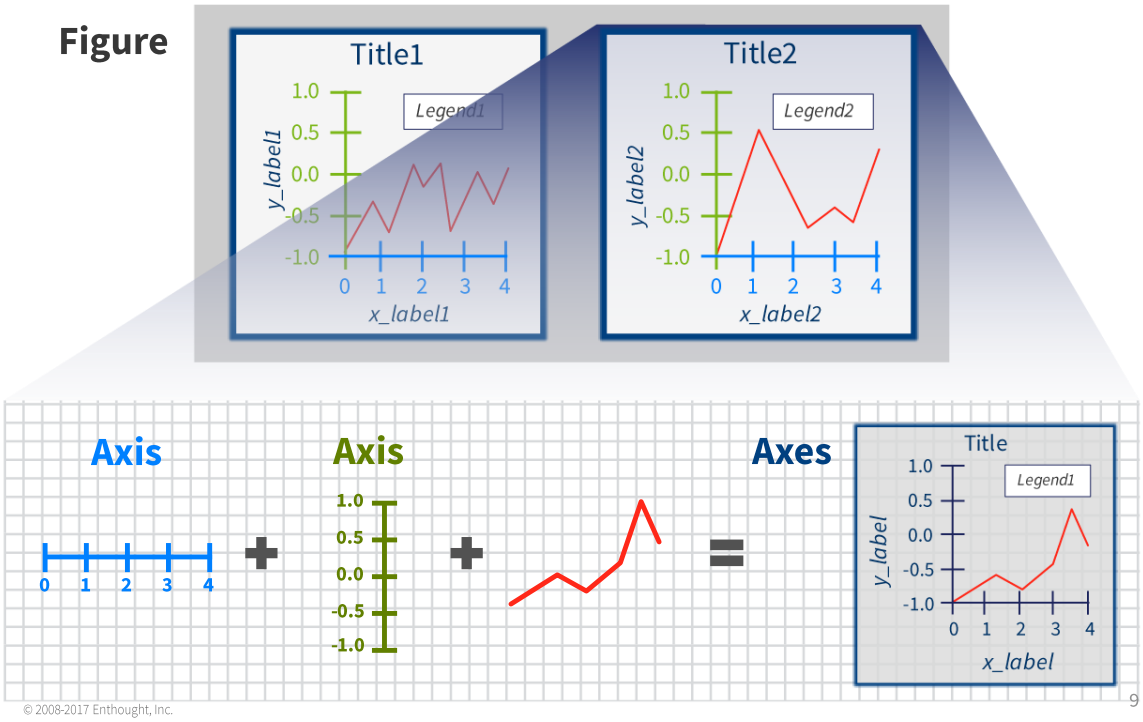
matplotlib

Matplotlib

An Interlude

Matplotlib Object Model

ENTHOUGHT

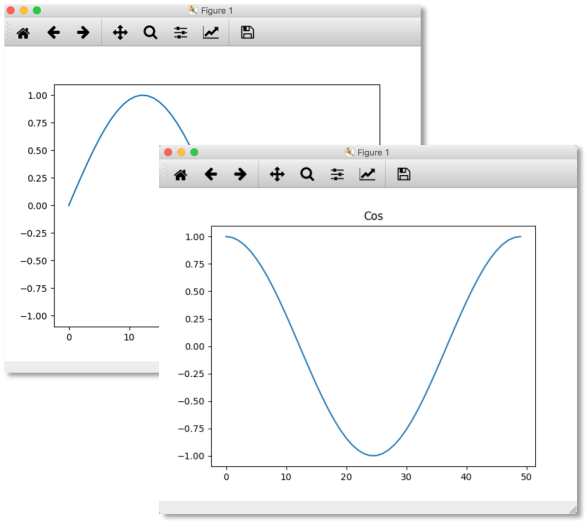


Matplotlib's "State Machine"

ENTHOUGHT

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

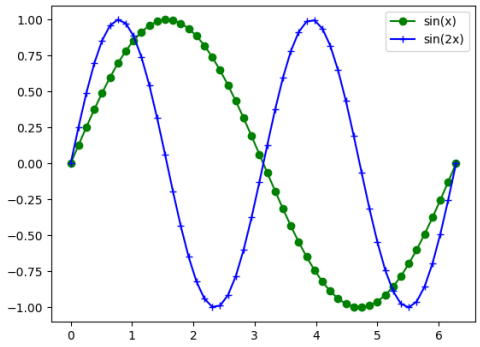
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> t = np.linspace(0,2*np.pi,
...                 50)
>>> x = np.sin(t)
>>> y = np.cos(t)

# 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")
```



Line Plots

ENTHOUGHT

```
>>> 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')
>>> plt.plot(x, y1, 'g-o',
...          x, y2, 'b-+')
>>> plt.legend(['sin(x)',
...              'sin(2x)'])
```



Symbol	Color
b	Blue
g	Green
r	Red
c	Cyan
m	Magenta
y	Yellow
k	Black
w	White

Symbol	Marker
.	Point
o	Circle
< > ^ v	Triangles
8	Octagon
s	Square
*	Star
+	Plus
	...and more

Scatter Plots

ENTHOUGHT

```
>>> 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")
```

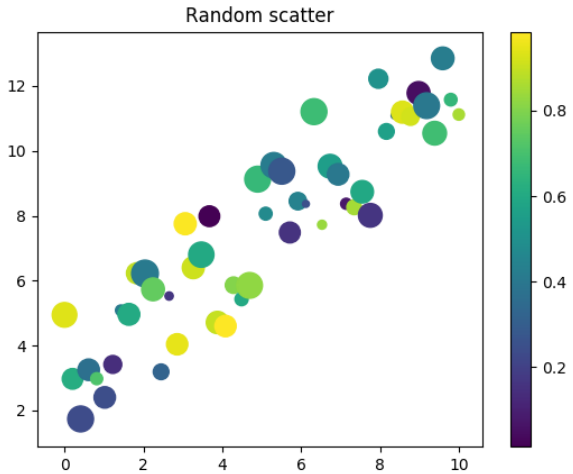
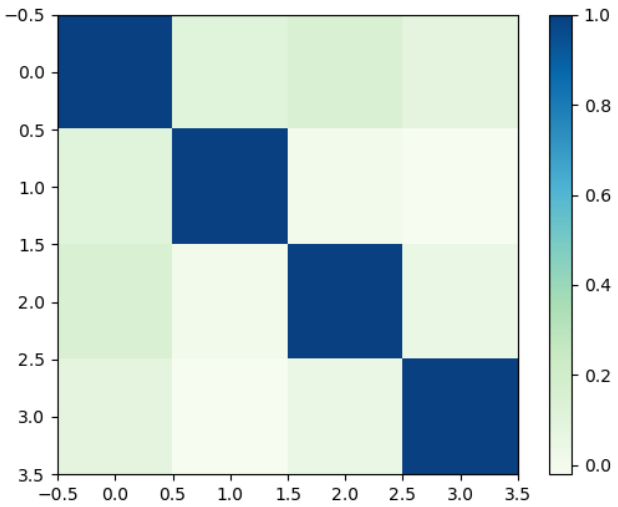


Image "Plots"

ENTHOUGHT

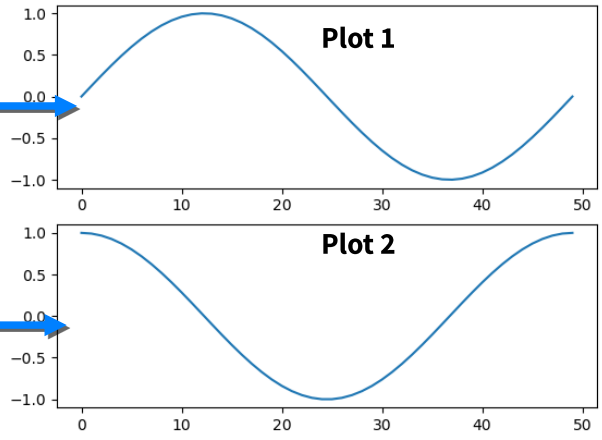
```
>>> # Create some data
>>> e1 = rand(100)
>>> e2 = rand(100)*2
>>> e3 = rand(100)*10
>>> e4 = rand(100)*100
>>> corrmatrix = \
...   np.corrcoef([e1, e2,
...                e3, e4])

>>> # Plot corr matrix as image
>>> plt.imshow(corrmatrix,
...     interpolation='none',
...     cmap='GnBu')
>>> plt.colorbar()
```



Multiple Plots Using `subplot`

ENTHOUGHT

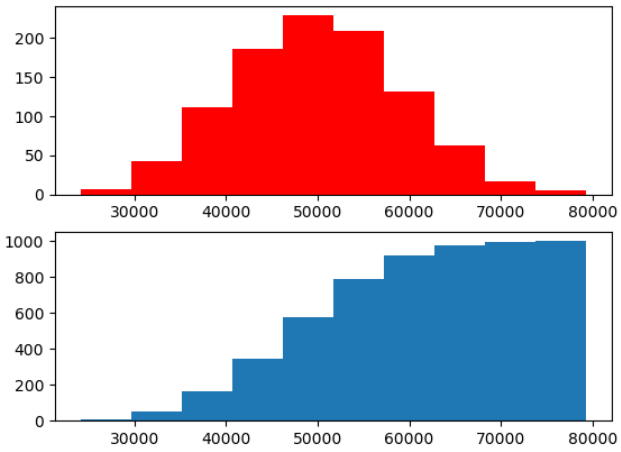
```
>>> t = np.linspace(0, 2*np.pi)
>>> x = np.sin(t)
>>> y = np.cos(t)

# To divide the plotting area
#                 columns
#                    |
>>> plt.subplot(2, 1, 1)
>>> plt.plot(x)
#              |      |
#             rows  active plot


# Now activate a new plot
# area.
>>> plt.subplot(2, 1, 2)
>>> plt.plot(y)
```



Histogram Plots

ENTHOUGHT

```
>>> # 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, …])
```

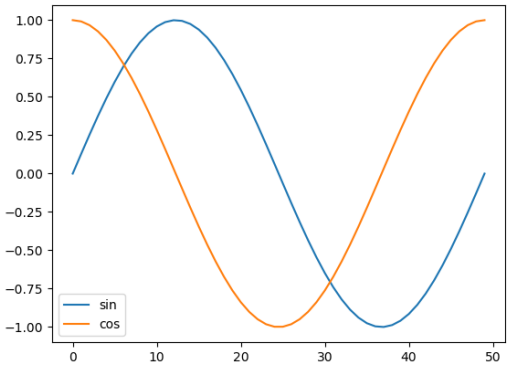


Legends, Titles, and Axis Labels

ENTHOUGHT

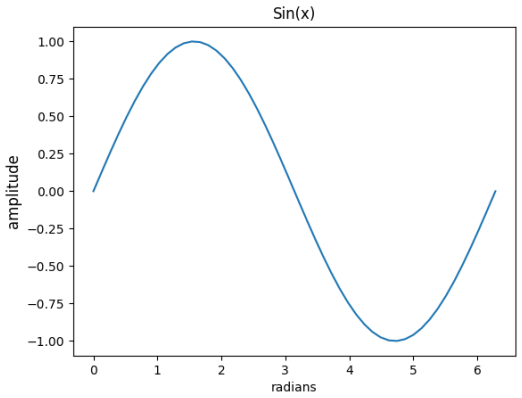
LEGEND LABELS WITH PLOT

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



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)')
```



Plotting from Scripts

ENTHOUGHT

INTERACTIVE MODE

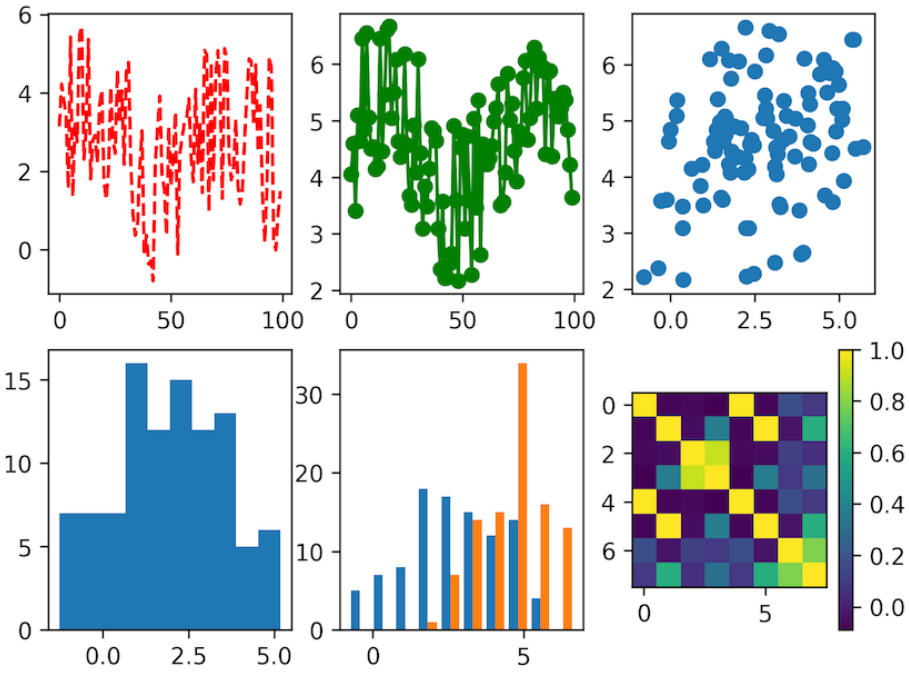
```
# In IPython, plots show up
# as soon as a plot command
# is issued:
>>> plt.figure()
>>> plt.plot(np.sin(t))
>>> plt.figure()
>>> plt.plot(np.cos(t))
```

NON-INTERACTIVE MODE

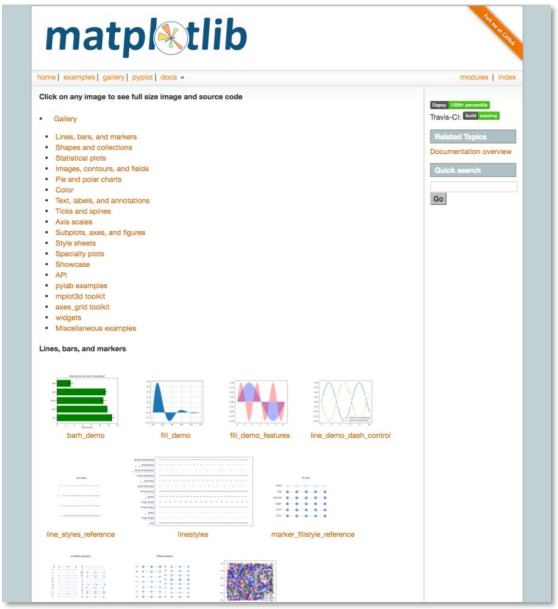
```
# In a script, you must call
# the show() command to display
# plots. Call it at the end of
# all your plot commands for
# best performance.
import matplotlib.pyplot as plt
import numpy as np
t = np.linspace(0,2*np.pi,
...                   50)
plt.figure()
plt.plot(np.sin(t))
plt.figure()
plt.plot(np.cos(t))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

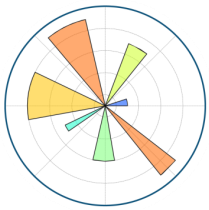

MPL Exercise: Desired Output

ENTHOUGHT



Additional Matplotlib Resources





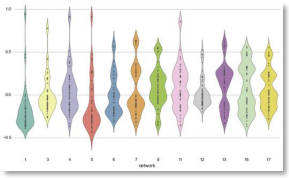
- Simple examples with increasing difficulty
  http://matplotlib.org/examples/index.html
- Gallery (huge)
  http://matplotlib.org/gallery.html
- See appendix for reference materials
- Usage FAQ
  http://matplotlib.org/faq/usage_faq.html

Further Learning Online

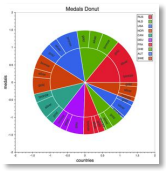
Additional material and exercises available online:

Other Visualization Libraries

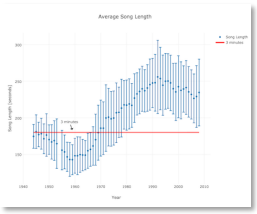
ENTHOUGHT



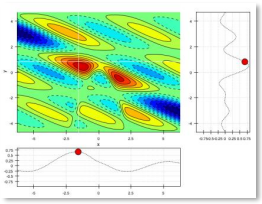
Seaborn: Better looking, high-level plots based on matplotlib



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

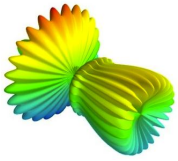


plot.ly: D3-based visualization with support for multiple languages (R, Python, Matlab, Julia and more.)

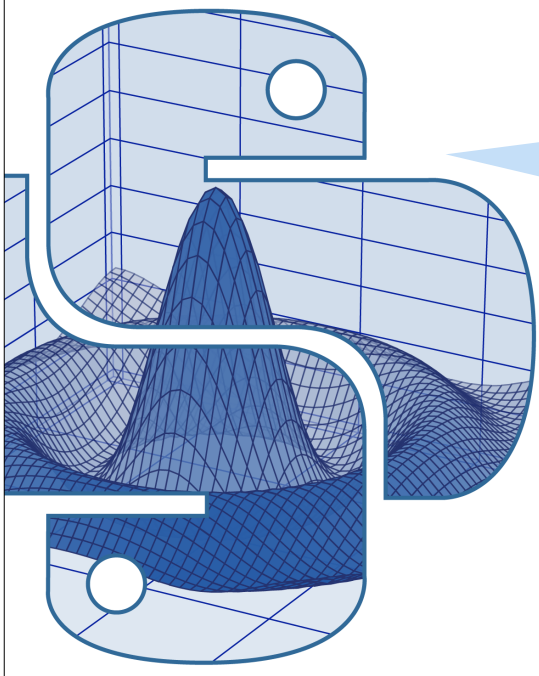


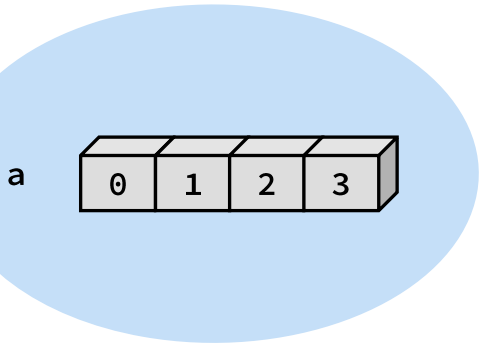
chaco: For interactive, custom plots embedded in user applications

mayavi: 3D visualization, based on VTK



ENTHOUGHT





NumPy

Introducing Arrays

Introducing NumPy Arrays

ENTHOUGHT

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
```


Array Operations

ENTHOUGHT

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

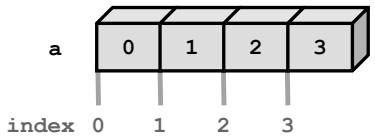
```
# create array from 0. to 10.
>>> x = np.arange(11.)

# 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

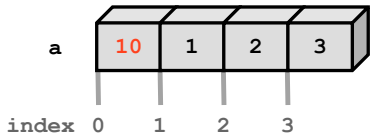
ENTHOUGHT

ARRAY INDEXING

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




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



BEWARE OF TYPE COERCION

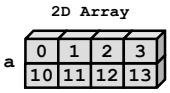
```
>>> a.dtype
dtype('int32')

# 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

ENTHOUGHT

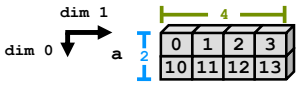
MULTI-DIMENSIONAL ARRAYS

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



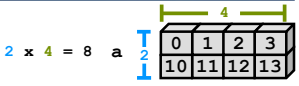
SHAPE = (ROWS, COLUMNS)

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



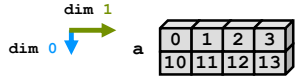
ELEMENT COUNT

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



NUMBER OF DIMENSIONS

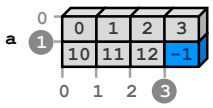
```
>>> a.ndim
2
```



GET / SET ELEMENTS

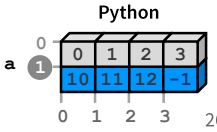
```
>>> a[1, 3]
13
```


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

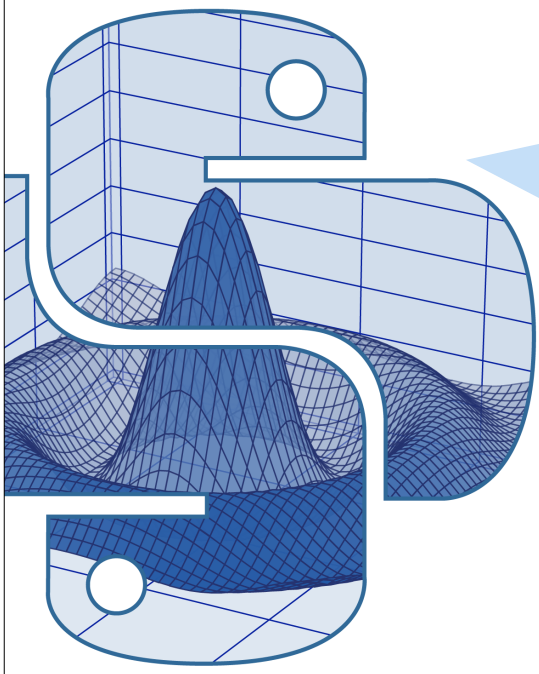


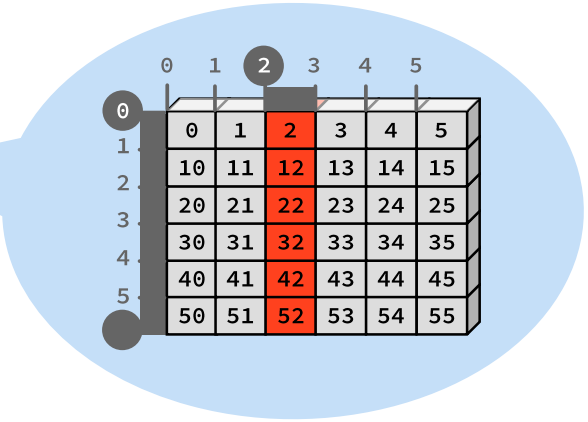
ADDRESS SECOND (ONETH) ROW USING SINGLE INDEX

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



ENTHOUGHT





NumPy

Indexing and Slicing

ENTHOUGHT

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])
```

OMITTING INDICIES

```
# 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

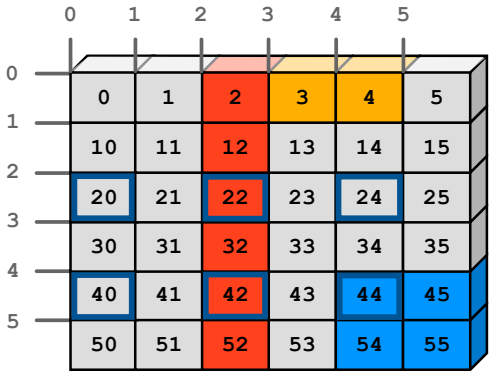
ENTHOUGHT

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]])
```



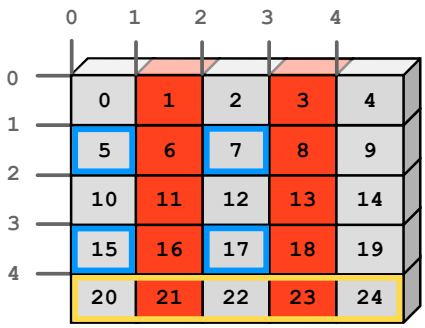
Give it a try!

ENTHOUGHT

Create the array below with the command

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

and extract the slices as indicated.



Slices are References

ENTHOUGHT

Slices are references to memory in the original array.
Changing values in a slice also changes the original array.

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

# create a slice containing only the last
# element of a
>>> b = a[2:4]
>>> b
array([2, 3])
>>> b[0] = 10

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


Fancy Indexing

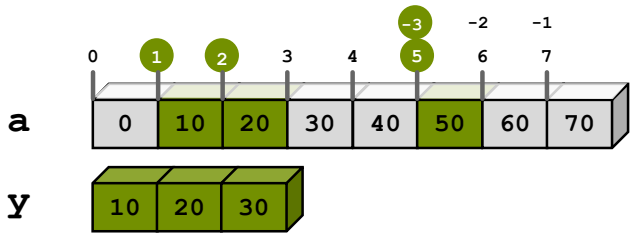
ENTHOUGHT

INDEXING BY POSITION

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


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

INDEXING WITH BOOLEANS

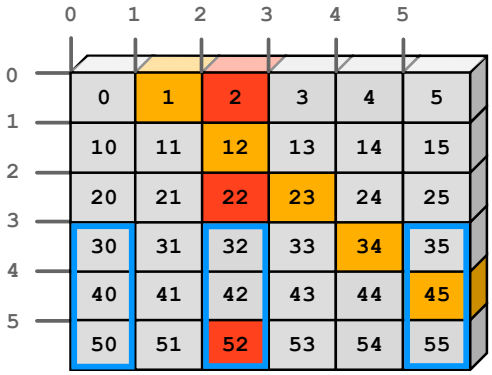
```
# manual creation of masks
>>> mask = np.array(
...      [0, 1, 1, 0, 0, 1, 0, 0],
...      dtype=bool)

# conditional creation of masks
>>> mask2 = a < 30

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



Fancy Indexing in 2-D

ENTHOUGHT

```
>>> 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])
```

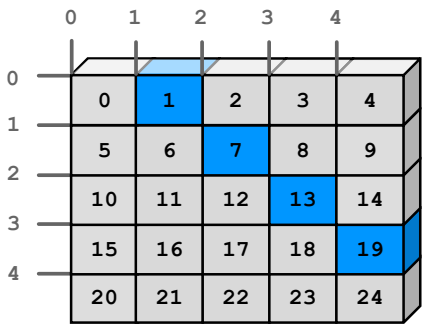


Unlike slicing, fancy indexing creates copies instead of a view into original array.

Give it a try!

ENTHOUGHT

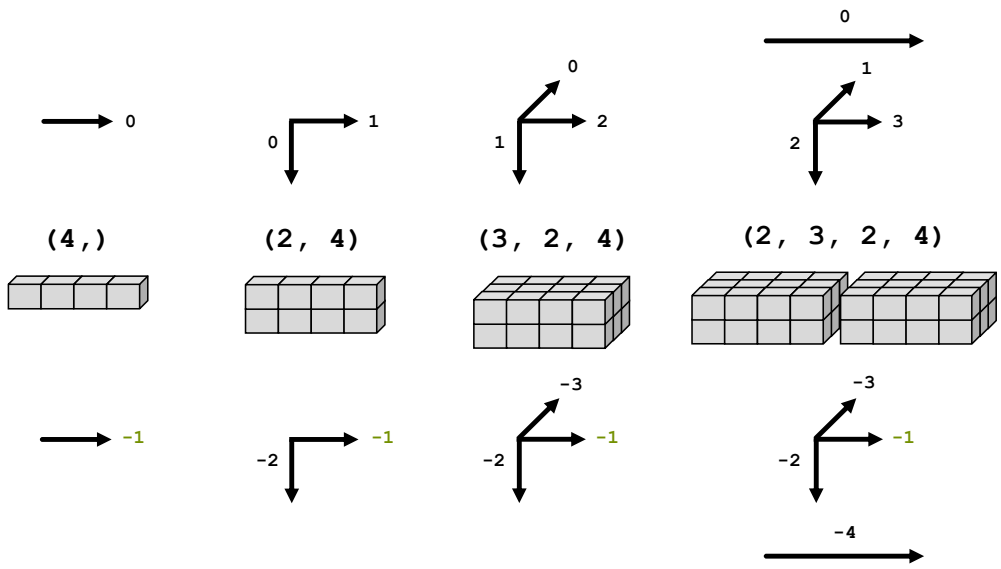
1. Create the array below with
   `a = np.arange(25).reshape(5, 5)`
   and extract the elements indicated in blue.
2. Extract all the numbers divisible by 3 using a boolean mask.



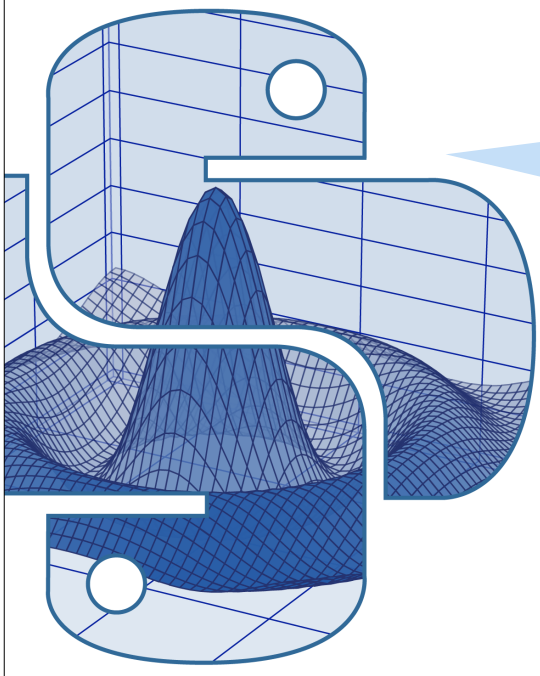
Multi-Dimensional Arrays

ENTHOUGHT

VISUALIZING MULTI-DIMENSIONAL ARRAYS



ENTHOUGHT



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

NumPy

Creating Arrays

Array Constructor Examples

ENTHOUGHT

FLOATING POINT ARRAYS

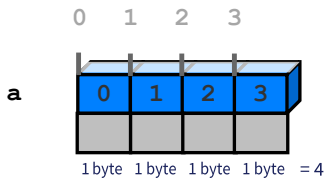
```
# Default to double precision
>>> a = np.array([0,1.0,2,3])
>>> a.dtype
dtype('float64')
>>> a.nbytes
32
```

REDUCING PRECISION

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

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
```


Array Creation Functions

ENTHOUGHT

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.

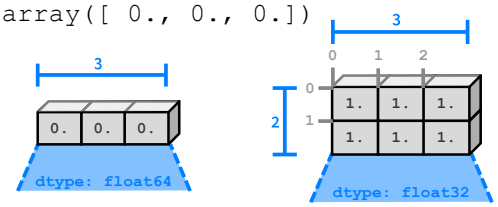
```
>>> np.arange(4)
array([0, 1, 2, 3])
>>> np.arange(0, 2*pi, pi/4)
array([ 0.000, 0.785, 1.571,
2.356, 3.142, 3.927, 4.712,
5.497])

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

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,))`**

Array Creation Functions (cont'd)

ENTHOUGHT

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')
>>> a = np.empty(2)
>>> a
array([1.78021120e-306,
 6.95357225e-308])

# fill array with 5.0
>>> a.fill(5.0)
array([5., 5.])

# alternative approach
# (slightly slower)
>>> a[:] = 4.0
array([4., 4.])
```


Array Creation Functions (cont'd)

ENTHOUGHT

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])
```

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., 1.77, 3.16,  5.62, 10.])
```

ARRAYS FROM/TO TXT FILES

Data.txt

```
-- 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
```

```
# 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)
```

"Structured" Arrays

```
>>> import numpy as np
# "Data structure" (dtype) that describes the fields and
# type of the items in each array element.
>>> particle_dtype = np.dtype([('mass','float32'),
...                            ('velocity', 'float32')])
# This must be a list of tuples.
>>> particles = np.array([(1,1), (1,2), (2,1), (1,3)],
                         dtype=particle_dtype)
>>> print(particles)
[(1.0, 1.0) (1.0, 2.0) (2.0, 1.0) (1.0, 3.0)]
# Retrieve the mass for all particles through indexing.
>>> print(particles['mass'])
[ 1.  1.  2.  1.]
# Retrieve particle 0 through indexing.
>>> particles[0]
(1.0, 1.0)
# Sort particles in place, with velocity as the primary field and
# mass as the secondary field.
>>> particles.sort(order=('velocity','mass'))
>>> print(particles)
[(1.0, 1.0) (2.0, 1.0) (1.0, 2.0) (1.0, 3.0)]

# See demo/multitype_array/particle.py.
```


"Structured" Arrays

Elements of an array can be any fixed-size data structure!

```
name char[10]
age  int
weight double
```

Brad	Jane	John	Fred
33	25	47	54
135.0	105.0	225.0	140.0
Henry	George	Brian	Amy
29	61	32	27
154.0	202.0	137.0	187.0
Ron	Susan	Jennifer	Jill
19	33	18	54
188.0	135.0	88.0	145.0

EXAMPLE

```
# structured data format
>>> fmt = np.dtype([('name', 'S10'),
                    ('age', int),
                    ('weight', float)
                   ])
>>> a = np.empty((3,4), dtype=fmt)
>>> a.itemsize
22
>>> a['name'] = [['Brad', … ,'Jill']]
>>> a['age'] = [[33, … , 54]]
>>> a['weight'] = [[135, … , 145]]
>>> print(a)
[[('Brad', 33, 135.0)
  …
  ('Jill', 54, 145.0)]]
```

Nested Datatype

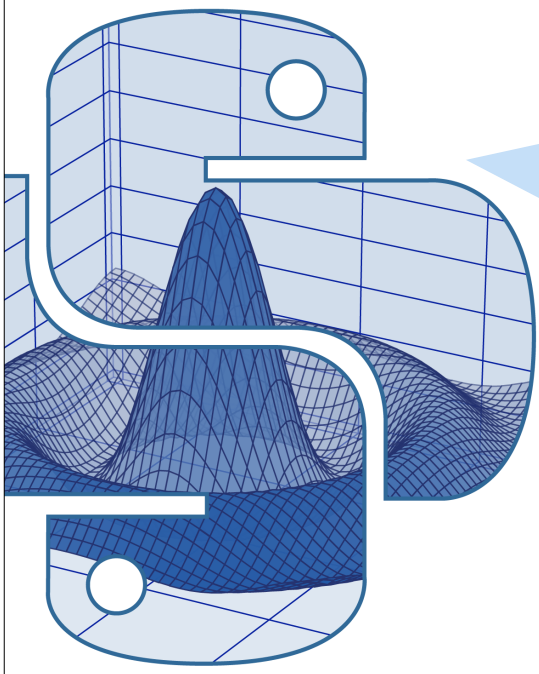
nested.dat

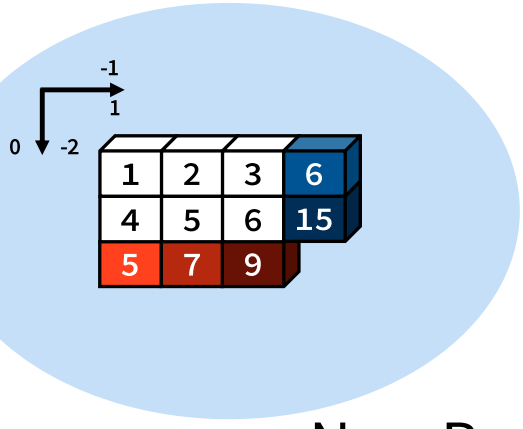
Time	Size	Position				Gain	Samples (2048) ...			
		Az	El	Type	ID					
1172581077060	4108	0.715594	−0.148407	1	4	40	561	1467	997	−3(
1172581077091	4108	0.706876	−0.148407	1	4	40	7	591	423	
1172581077123	4108	0.698157	−0.148407	1	4	40	49	−367	−565	−3[
1172581077153	4108	0.689423	−0.148407	1	4	40	−55	−953	−1151	−3(
1172581077184	4108	0.680683	−0.148407	1	4	40	−719	−1149	−491	3[
1172581077215	4108	0.671956	−0.148407	1	4	40	−1503	−683	661	14[
1172581077245	4108	0.663232	−0.148407	1	4	40	−2731	−281	2327	291
1172581077276	4108	0.654511	−0.148407	1	4	40	−3493	−159	3277	38(
1172581077306	4108	0.645787	−0.148407	1	4	40	−3255	−247	3145	38[
1172581077339	4108	0.637058	−0.148407	1	4	40	−2303	−101	2079	247
1172581077370	4108	0.628321	−0.148407	1	4	40	−1495	−553	571	107
1172581077402	4108	0.619599	−0.148407	1	4	40	−955	−1491	−1207	−2[
1172581077432	4108	0.61087	−0.148407	1	4	40	−875	−3009	−2987	−9[
1172581077463	4108	0.602148	−0.148407	1	4	40	−491	−3681	−4193	−17[
1172581077497	4108	0.593438	−0.148407	1	4	40	167	−3501	−4573	−25(
1172581077547	4108	0.584696	−0.148407	1	4	40	1007	−2613	−4463	−30[
1172581077599	4108	0.575972	−0.148407	1	4	40	1261	−2155	−4299	−33[
1172581077650	4108	0.567244	−0.148407	1	4	40	1537	−2633	−4945	−367
1172581077702	4108	0.558511	−0.148407	1	4	40	1105	3701	6129	42(

Nested Datatype (cont'd)

The data file can be extracted with the following code:

```
>>> dt = np.dtype([('time', 'uint64'),
...                ('size', 'uint32'),
...                ('position', [('az', 'float32'),
...                              ('el', 'float32'),
...                              ('region_type', 'uint8'),
...                              ('region_ID', 'uint16')]),
...                ('gain', 'uint8'),
...                ('samples', 'int16', 2048)])

>>> data = np.loadtxt('nested.dat', dtype=dt, skiprows=2)
>>> data['position']['az']
array([ 0.71559399,  0.70687598,  0.69815701,  0.68942302,
        0.68068302, ...], dtype=float32)
```





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.

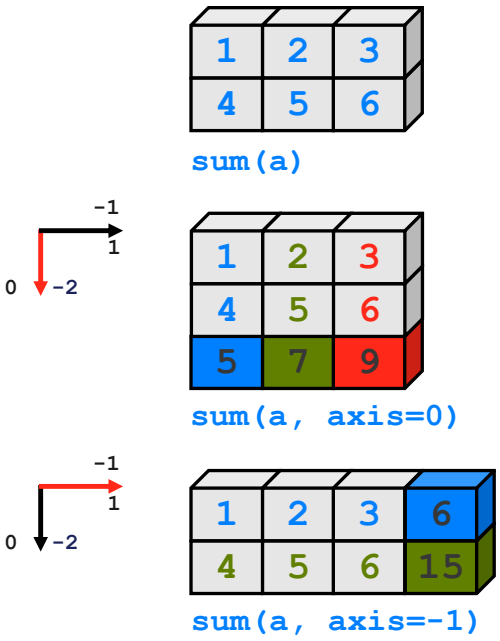
Rule 4: Missing values propagate unless explicitly ignored (nanmean, nansum, ...).

Array Calculation Methods

ENTHOUGHT

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])
```



Other Operations on Arrays

ENTHOUGHT

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`
- `ptp (max - min)`

Statistics

- `mean, std, var`

Truth value testing

- `any, all`

See the Numpy appendix for more.

Min/Max

ENTHOUGHT

MIN

```
>>> a = np.array([2.,3.,0.,1.])
>>> a.min(axis=0)
0.0
# Use NumPy's min() instead
# of Python's built-in min()
# for speedy operations on
# multi-dimensional arrays.
>>> np.min(a, axis=0)
0.0
```

MAX

```
>>> a = np.array([2.,3.,0.,1.])
>>> a.max(axis=0)
3.0


# as a function
>>> np.max(a, axis=0)
3.0
```

ARGMIN

```
# Find index of minimum value.
>>> a.argmin(axis=0)
2
# as a function
>>> np.argmin(a, axis=0)
2
```

ARGMAX

```
# Find index of maximum value.
>>> a.argmax(axis=0)
1
# as a function
>>> np.argmax(a, axis=0)
1
```


Statistics Array Methods

ENTHOUGHT

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([ 1.22,  1.22,  1.22])


# 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!

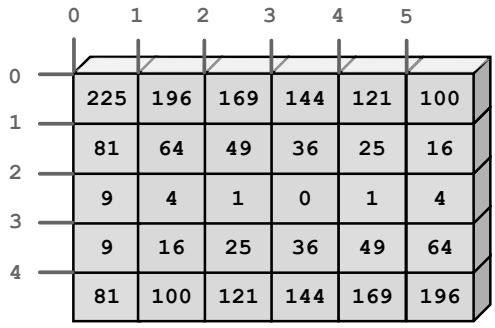
ENTHOUGHT

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

Array Broadcasting

ENTHOUGHT

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**.

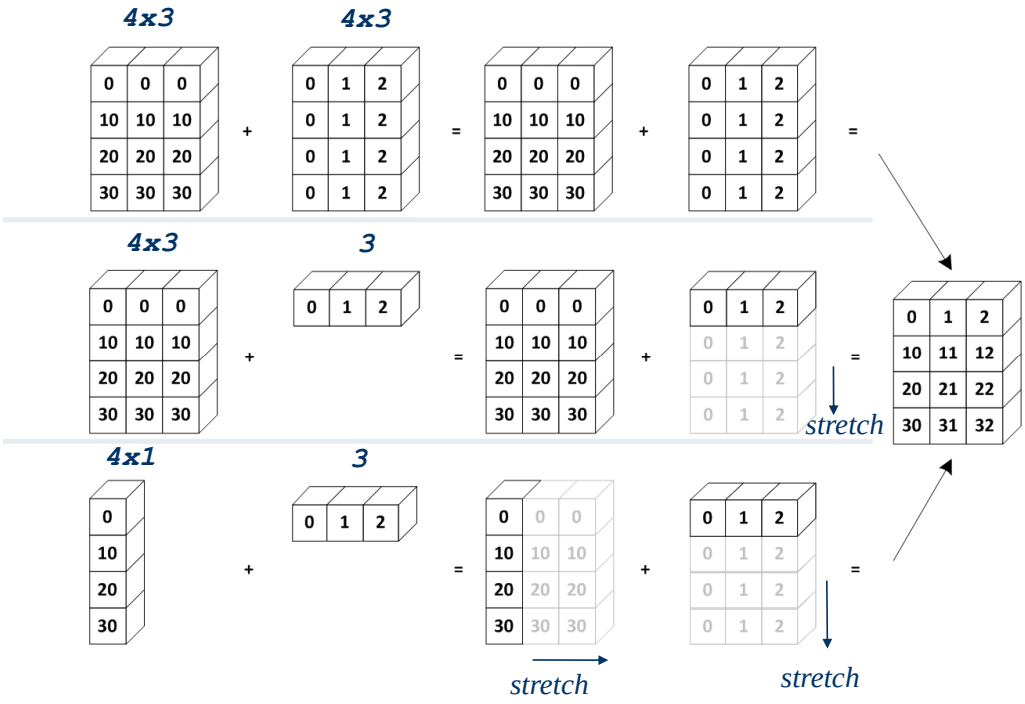
RULE 1: PREPEND ONES TO SMALLER ARRAYS' SHAPE

```
>>> 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

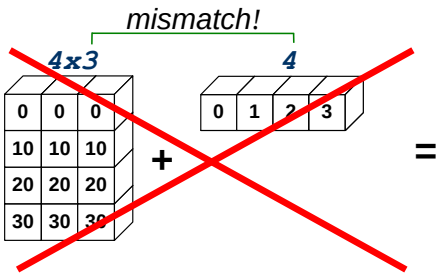
```
>>> c = a + b # c.shape == (3, 5)
# is logically equivalent to...
>>> tmp_b = b.reshape(1, 5)
>>> tmp_b_repeat = tmp_b.repeat(3, axis=0)
>>> c = a + tmp_b_repeat
# But broadcasting makes no copies of "b"s data!
```

Array Broadcasting



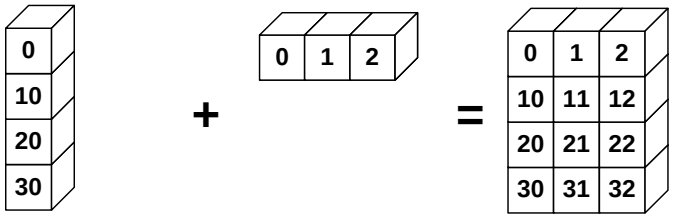
Broadcasting Rules

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.



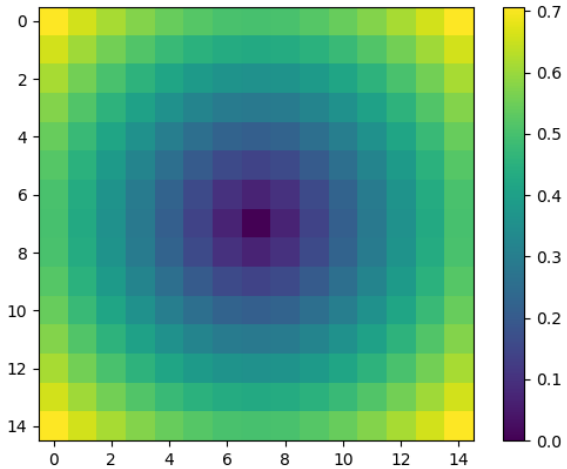
Broadcasting in Action

```
>>> a = np.array((0, 10, 20, 30))
>>> b = np.array((0, 1, 2))
>>> y = a[:, np.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.

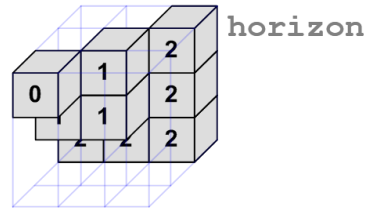
```
>>> yi, xi = np.meshgrid(np.arange(3), np.arange(3),
...                      sparse=True)
>>> zi = np.array([[0, 1, 2],
...                [1, 1, 2],
...                [2, 2, 2]])
>>> horizon = data_cube[xi, yi, zi]
```

Indices

xi \ yi	0	1	2
0	0	1	2
1	1	1	2
2	2	2	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)
```


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:

$$y = \texttt{add.reduce (a)} = \sum_{n=0}^{N-1} a[n] = a[0] + a[1] + \ldots + a[N-1]$$

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

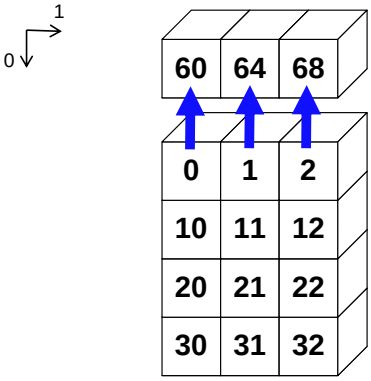
```
>>> a = np.array([1,1,0,1])
>>> np.logical_and.reduce(a)
False
>>> np.logical_or.reduce(a)
True
```

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.

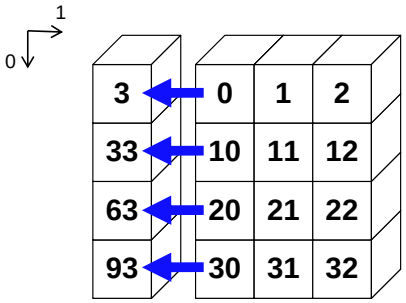
SUM COLUMNS BY DEFAULT

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



SUMMING UP EACH ROW

```
>>> np.add.reduce(a,1)
array([ 3, 33, 63, 93])
```



op.accumulate()

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

For example:

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

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

op.reduceat()

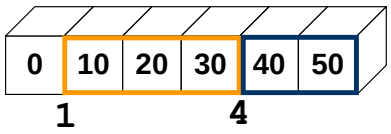
op.reduceat(a,indices)
applies `op` to ranges in the 1-D array `a` defined by the values in `indices`.
The resulting array has the same length as `indices`.

For example:
`y = add.reduceat(a, indices)`

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

EXAMPLE

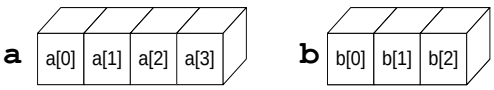
```
>>> a = np.array(
    [0,10,20,30,40,50])
>>> indices = np.array([1,4])
>>> np.add.reduceat(a,indices)
array([60, 90])
```



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()`.

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.)



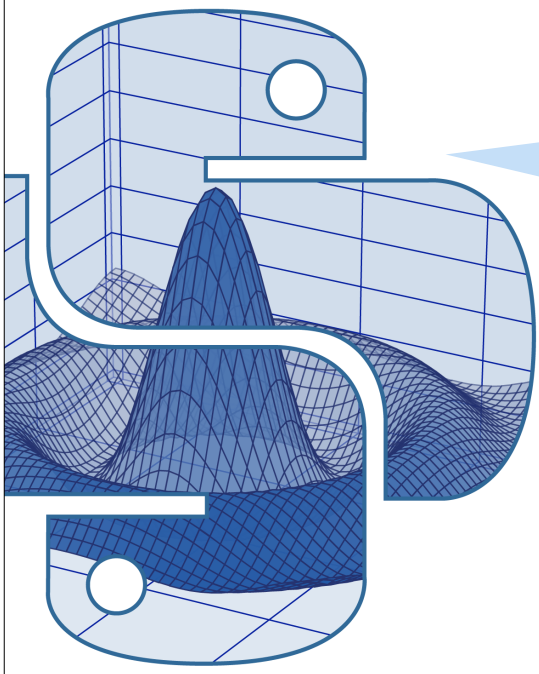

```
>>> np.add.outer(a,b)
```

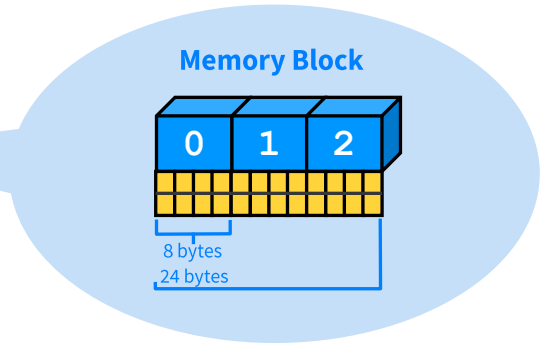
a[0]+b[0]	a[0]+b[1]	a[0]+b[2]
a[1]+b[0]	a[1]+b[1]	a[1]+b[2]
a[2]+b[0]	a[2]+b[1]	a[2]+b[2]
a[3]+b[0]	a[3]+b[1]	a[3]+b[2]

```
>>> 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]

ENTHOUGHT





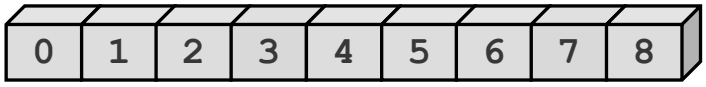
NumPy

The Array Data Structure

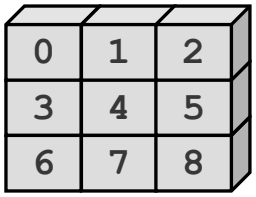
Array Data Structure

ENTHOUGHT

Memory Block



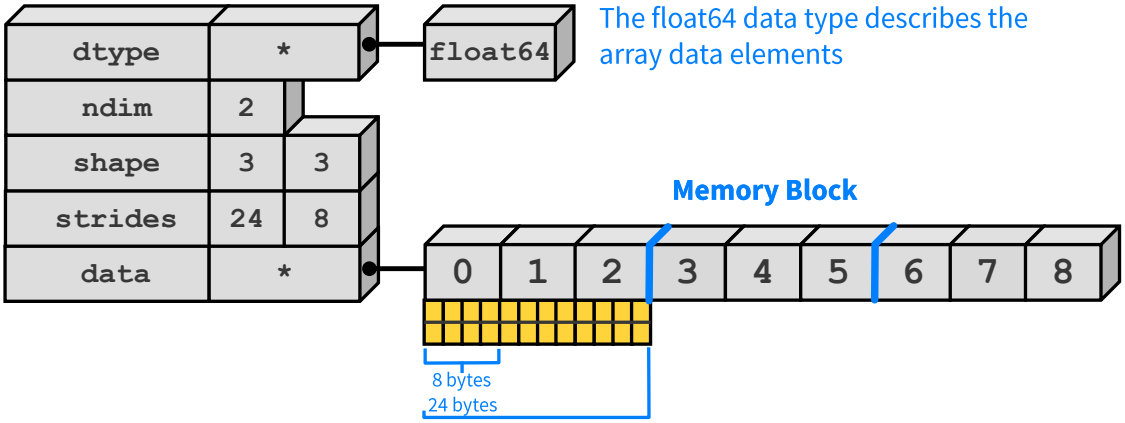
Python View:



Array Data Structure

ENTHOUGHT

NDArray Data Structure

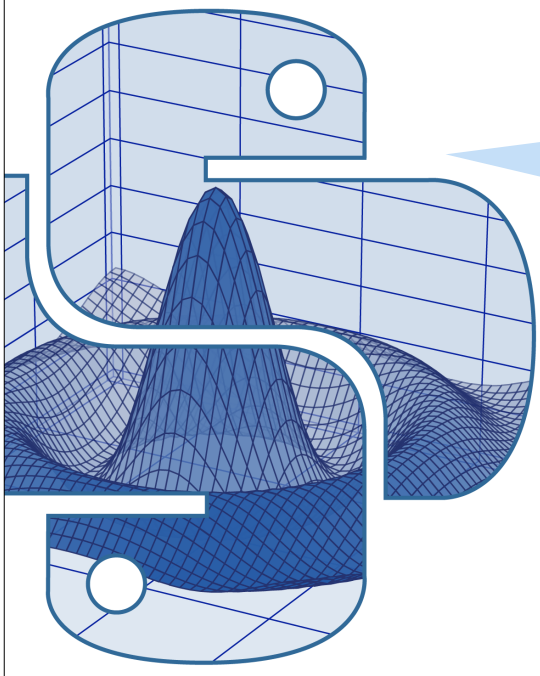


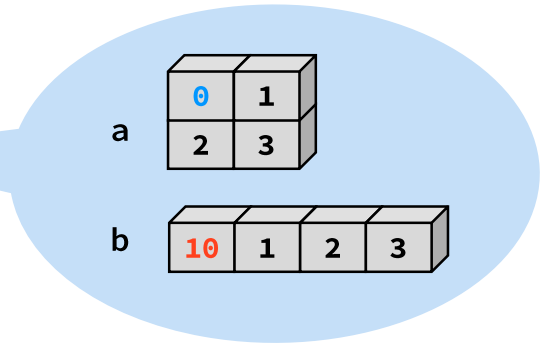
The float64 data type describes the array data elements

Python View:

0	1	2
3	4	5
6	7	8

ENTHOUGHT





NumPy

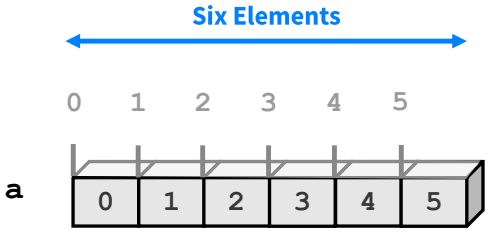
Structure Operations

Operations on the Array Structure

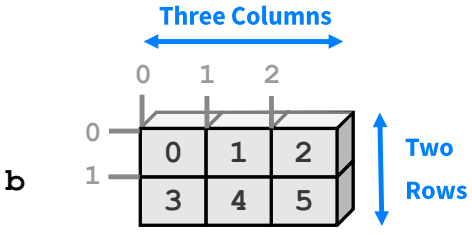
ENTHOUGHT

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

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




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



This is not a new array.

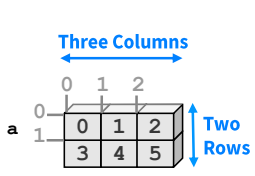
The original data does not get reordered.

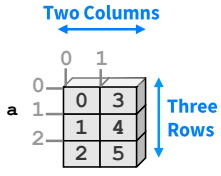
Transpose

ENTHOUGHT

TRANSPOSE

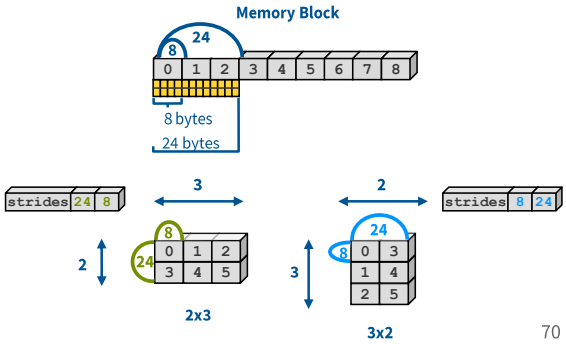
```
>>> a = np.array([[0,1,2],
...               [3,4,5]])
>>> a.shape
(2,3)

# 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

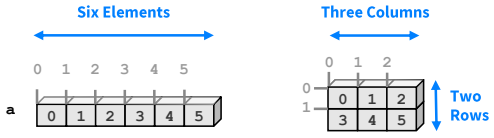
ENTHOUGHT

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
```

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]])
```



Flattening Arrays

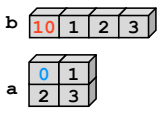
ENTHOUGHT

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]])
```

no change



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]])
```

changed!