math 480 - 20080519 - numpy

```
%hide
%html
<font size=+1>
<h1>Math 480 -- 20080519 -- numpy</h1>
<h1>Numpy: Dense Numerical Matrices</h1>

<h2>Documentation</h2>
The <a target="_new" href="numpybook.pdf">Guide to NumPy</a> is
Travis Oliphant's book about numpy.  The description of
functions in this worksheet draws heavily on that.

<h2>Introduction</h2>
Numpy arrays are <i>vastly different</i> than Sage matrices, to put
it mildly.  Essentially
every rule, constraint, convention, etc., is different.  They
server a very different purpose.
They are meant to be the ultimate Python general $n$-dimensional
dense array object.  In contrast, Sage matrices are meant to be
very mathematical objects with clear mathematically meaningful
semantics and very fast algorithms mainly for algebraic
computations.

<br><br>
Numpy array have many similarities with matlab arrays, given that
they have very similar computational domains.
See <a href="http://www.scipy.org/NumPy_for_Matlab_Users">Numpy for
Matlab Users</a> for a table that compares
and contrasts their functions.  That page also has an excellent
table showing equivalent commands in Matlab and Numpy.
```

Math 480 -- 20080519 -- numpy

Numpy: Dense Numerical Matrices

Documentation

The Guide to NumPy is Travis Oliphant's book about numpy. The description of functions in this worksheet draws heavily on that.

Introduction

Numpy arrays are *vastly different* than Sage matrices, to put it mildly. Essentially every rule, constraint, convention, etc., is different. They server a very different purpose. They are meant to be the ultimate Python general n-dimensional dense array object. In contrast, Sage matrices are meant to be very mathematical objects with clear mathematically meaningful semantics and very fast algorithms mainly for algebraic computations.

Numpy array have many similarities with matlab arrays, given that they have very similar computational domains. See Numpy for Matlab Users for a table that compares and contrasts their functions. That page also has an excellent table showing equivalent commands in Matlab and Numpy.

```
import numpy
```

```
%hide
%html
<font size=+1>
Create a numpy array as follows.  For numeric types, one can and
```

```
should explicitly
give the datatype (dtype), and it can be any numerical type or
generic
Python objects.  This is already much different than Sage's very
mathematical
matrices.
```

Create a numpy array as follows. For numeric types, one can and should explicitly give the datatype (dtype), and it can be any numerical type or generic Python objects. This is already much different than Sage's very mathematical matrices.

```
Z = numpy.array([[sqrt(2) + 3, random_matrix(QQ, 2)], [1.5, 2/3]])
Z
```

```
array([[
                                  sqrt(2) + 3, [0 0]
[0 1]],
       [1.50000000000000, 2/3]], dtype=object)
```

```
Z + Z
```

```
array([[
                                 2 sqrt(2) + 6, [0 0]
[0 2]],
       [3.00000000000000, 4/3]], dtype=object)
```

```
Z.
```

```
%hide
%html
<font size=+1>
Symbolics look funny above because numpy matrices's use the "wrong"
formatting
option when converting them to strings. We thus write a good
pretty printer for numpy arrays for the Sage notebook...
```

Symbolics look funny above because numpy matrices's use the "wrong" formatting option when converting them to strings. It would be nice to write a good pretty printer for numpy arrays for the Sage notebook...

```
def shownp(A):
   if len(A.shape) > 2:
      for i in range(A.shape[0]):
          shownp(A[i])
      return
   if len(A.shape) == 1:
      A = A.reshape(A.shape[0], 1)
   nrows, ncols = A.shape
   s = r'\left(\begin{array}{'
   s += 'r'*ncols + '}\n'
   for i in range(nrows):
       s += ' & '.join([latex(A[i,j]) for j in range(ncols)]) +
'\\\\\n'
   s += r'\end{array}\right)'
   html('$%s$'%s)
```

```
shownp(Z)
```

$$\left(\begin{array}{rr} \sqrt{2} + 3 & \left(\begin{array}{rr} 0 & 0 \\ 0 & 1 \end{array}\right) \\ 1.50000000000000 & \frac{2}{3} \end{array}\right)$$

```
shownp(numpy.dot(Z,Z))
```

$$\left(\begin{array}{rr} \left(\begin{array}{rr} \left(\sqrt{2} + 3\right)^{2} & 0.000000000000000 \\ 0.000000000000000 & \left(\sqrt{2} + 3\right)^{2} + 1.50000000000000 \end{array}\right) & \\ 1.50000000000000 \left(\sqrt{2} + 3\right) + 1.00000000000000 & \left(\begin{array}{rr} 0.444444444444444 & 0.0 \\ 0.000000000000000 & 1. \end{array}\right. \end{array}\right.$$

```
# An 1-d array
A = numpy.array([1..10])
shownp(A)
```

$$\left(\begin{array}{r} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}\right)$$

```
%hide
%html
<font size=+1>Numpy arrays can have any number of dimensions!
 Think of these higher dimensional arrays as arrays of arrays (of
arrays...).  They are useful, e.g., in image processing, where
there is one array for each color channel.
```

Numpy arrays can have any number of dimensions! Think of these higher dimensional arrays as arrays of arrays (of arrays...). They are useful, e.g., in image processing, where there is one array for each color channel.

```
M = numpy.array([[[1,2], [3,4]], [[1/3,5], [10,2]], [[8,1],
[-5,2]]], dtype=int)
M
```

```
array([[[ 1,  2],
        [ 3,  4]],

       [[ 0,  5],
        [10,  2]],

       [[ 8,  1],
        [-5,  2]]])
```

```
# The actual array entries are *NOT* Python int's!
type(M[0,0,0])
```

```
<type 'numpy.int32'>
```

```
M[0,0,0].item()
```

```
1
```

```
shownp(M)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$
$$\begin{pmatrix} 0 & 5 \\ 10 & 2 \end{pmatrix}$$
$$\begin{pmatrix} 8 & 1 \\ -5 & 2 \end{pmatrix}$$

```
# dtype: A data-type object that fully describes each
# fixed-length item in the array.
# The dtype attribute can be set to anything that can be
# interpreted as a data type (numpy includes 21 of these).
# Setting this attribute allows you to change the interpretation of
# the data in the array. The new data type must be compatible with
the array%u2019s
# current data type.
```

```
M.dtype
```

```
dtype('int32')
```

```
M.dtype = float
```

```
shownp(M)
```

$$\begin{pmatrix} 4.24399158242e-314 \\ 8.48798316534e-314 \end{pmatrix}$$
$$\begin{pmatrix} 1.06099789548e-313 \\ 4.24399158687e-314 \end{pmatrix}$$

$$\begin{pmatrix} 2.12199579492e - 314 \\ 6.36598737043e - 314 \end{pmatrix}$$

```
M.dtype=numpy.int32
```

```
shownp(M)
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$
$$\begin{pmatrix} 0 & 5 \\ 10 & 2 \end{pmatrix}$$
$$\begin{pmatrix} 8 & 1 \\ -5 & 2 \end{pmatrix}$$

```
shownp(M.astype(float))
```

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}$$
$$\begin{pmatrix} 0.0 & 5.0 \\ 10.0 & 2.0 \end{pmatrix}$$
$$\begin{pmatrix} 8.0 & 1.0 \\ -5.0 & 2.0 \end{pmatrix}$$

```
# flags: (not settable) special array-connected dictionary-like
object with
# attributes showing the state of %uFB02ags in this array;
# only the %uFB02ags WRITEABLE, ALIGNED, and
# UPDATEIFCOPY can be modi%uFB01ed by setting
# attributes of this object
M.flags
```

```
C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

```
# shape:  (settable) tuple showing the array shape; setting this
# attribute re-shapes the array
M.shape
```

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

```
M.shape = (3,4)
shownp(M)
```

$$\begin{pmatrix} 20090283 & 2 & 3 & 4 \\ 0 & 5 & 10 & 2 \\ 8 & 1 & -5 & 2 \end{pmatrix}$$

```
M.shape = (3,2,2)
```

```
# Reshape makes a reshaped copy:
shownp(M.reshape((4,3)))
```

$$\begin{pmatrix} 20090283 & 2 & 3 \\ 4 & 0 & 5 \\ 10 & 2 & 8 \\ 1 & -5 & 2 \end{pmatrix}$$

```
M.shape
```

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

```
# strides: (settable) tuple showing how many bytes must be jumped
in
# the data segment to get from one entry to the next
M.strides
```

```
(16, 8, 4)
```

```
# ndim: (not settable) number of dimensions in array
M.ndim
```

```
3
```

```
# data: (settable) bu%uFB00er object loosely wrapping the array
data
# (only  works for single-segment arrays)
d = M.data; d
```

```
<read-write buffer for 0x7cc3470, size 48, offset 0 at 0x7fda800>
```

```
d[0:10]
```

```
'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00'
```

```
type(d)
```

```
<type 'buffer'>
```

```
# size: (not settable) total number of elements
M.size
```

```
12
```

```
# itemsize: (settable) one-dimensional, indexable iterator
# object that acts somewhat like a 1-d array
M.flat
```

```
<numpy.flatiter object at 0xe83e00>
```

```
len(M.flat)
```

```
12
```

```
list(M.flat)
```

```
[1, 2, 3, 4, 0, 5, 10, 2, 8, 1, -5, 2]
```

```
M.flat[0] = 20090283
```

```
shownp(M)
```

$$\begin{pmatrix} 20090283 & 2 \\ 3 & 4 \end{pmatrix}$$
$$\begin{pmatrix} 0 & 5 \\ 10 & 2 \end{pmatrix}$$
$$\begin{pmatrix} 8 & 1 \\ -5 & 2 \end{pmatrix}$$

```
# ctypes: (not settable) object to simplify the interaction of
# this array with the ctypes module
M.ctypes
```

```
    <numpy.core._internal._ctypes object at 0x7fda7d0>
```

```
type(M.ctypes)
```

```
    <class 'numpy.core._internal._ctypes'>
```

```
M.ctypes.data
```

```
    130823360
```

```
# __array_interface__: (not settable) dictionary with keys (data,
# typestr, descr, shape, strides) for compliance with Python side
# of array protocol
M.__array_interface__
```

```
    {'descr': [('', '<i4')], 'strides': None, 'shape': (3, 2, 2),
    'version': 3, 'typestr': '<i4', 'data': (130823360, False)}
```

```
# __array_struct__: (not settable) array interface on C-level
M.__array_struct__
```

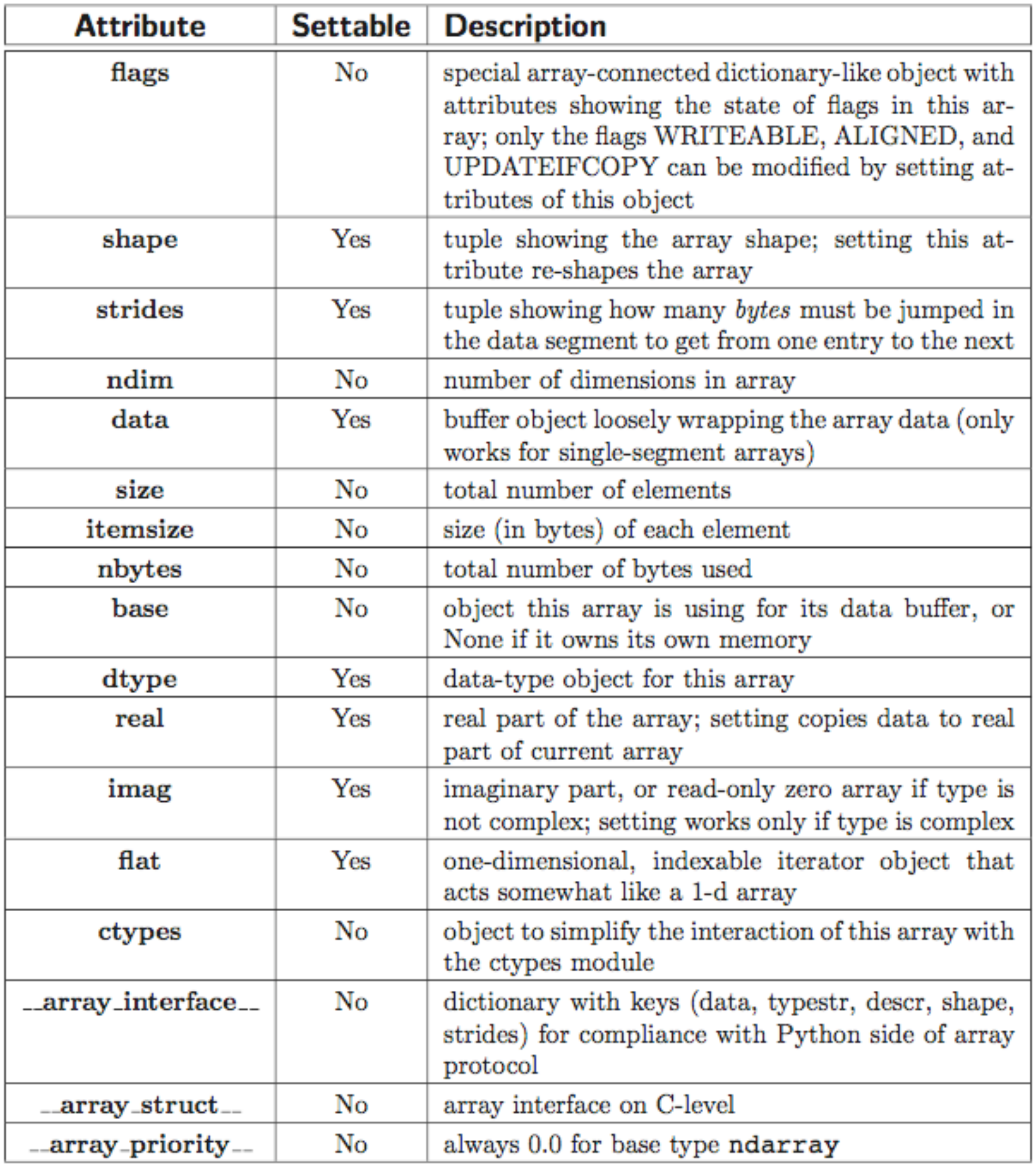
```
    <PyCObject object at 0x7a16740>
```

```
# __array_priority__: (not settable) always 0.0 for base type
ndarray
M.__array_priority__
```

```
    0.0
```

```
%hide
%html
<img src="summary-attributes.png">
```

Table 3.1: Attributes of the **ndarray**.



Attribute	Settable	Description
flags	No	special array-connected dictionary-like object with attributes showing the state of flags in this array; only the flags WRITEABLE, ALIGNED, and UPDATEIFCOPY can be modified by setting attributes of this object
shape	Yes	tuple showing the array shape; setting this attribute re-shapes the array
strides	Yes	tuple showing how many *bytes* must be jumped in the data segment to get from one entry to the next
ndim	No	number of dimensions in array
data	Yes	buffer object loosely wrapping the array data (only works for single-segment arrays)
size	No	total number of elements
itemsize	No	size (in bytes) of each element
nbytes	No	total number of bytes used
base	No	object this array is using for its data buffer, or None if it owns its own memory
dtype	Yes	data-type object for this array
real	Yes	real part of the array; setting copies data to real part of current array
imag	Yes	imaginary part, or read-only zero array if type is not complex; setting works only if type is complex
flat	Yes	one-dimensional, indexable iterator object that acts somewhat like a 1-d array
ctypes	No	object to simplify the interaction of this array with the ctypes module
__array_interface__	No	dictionary with keys (data, typestr, descr, shape, strides) for compliance with Python side of array protocol
__array_struct__	No	array interface on C-level
__array_priority__	No	always 0.0 for base type **ndarray**

```
%hide
%html
<font size=+1>A more conventional numerical numpy array:
```

A more conventional numerical numpy array:

```
A = numpy.array([[1/3,2,3], [4,5,6], [7,8,9]], dtype=float)
shownp(A)
```

$$\begin{pmatrix} 0.333333333333 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{pmatrix}$$

```
type(A[0,0])
```

```
<type 'numpy.float64'>
```

```
A[0,0] = 10; shownp(A)
```

$$\begin{pmatrix} 10.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{pmatrix}$$

```
A.flags
```

```
C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

```
A.flags['WRITEABLE'] = False
```

```
A.flags
```

```
C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : True
  UPDATEIFCOPY : False
```

```
A[0,0] = 10
```

```
Traceback (click to the left for traceback)
...
RuntimeError: array is not writeable
```

```
# Mutability (writeability) is not strictly enforced once set,
# since you can trivially change it to true, make a write, then
# change the flag back.
before = A.flags['WRITEABLE']
A.flags['WRITEABLE'] = True
A[0,0] = 20
A.flags['WRITEABLE'] = before
shownp(A)
```

$$\begin{pmatrix} 20.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{pmatrix}$$

```
# This is different than Sage matrix mutability!
B = matrix(RDF, 3, 3, [1..9])
B.set_immutable()
```

```
# B *can't* be changed back to being mutable...
B[0,0] = 20
# Sage is much more about enforcing rigor, etc., since a lot of
subtle
# mathematical errors in complicated algebraic algorithms can arise
# from the lax (but more user-friendly?) numpy approach.
```

```
    Traceback (click to the left for traceback)
    ...
    ValueError: matrix is immutable; please change a copy instead (use
    self.copy()).
```

```
# numpy arrays can be serialized and saved to disk
save(M, DATA+'M.sobj')
print M.dumps()
(M == loads(M.dumps())).all()
```

```
    €cnumpy.core.multiarray
    _reconstruct
    qcnumpy
    ndarray
    qK?…U‡Rq(KKKK‡cnumpy
    dtype
    qUi4K?K‡Rq(KU<NNNJÿÿÿÿJÿÿÿÿK?tb‰U0?y\
    33;????????\
    3;????????&\
    #65533;
    ????????\
    5533;??ûÿÿÿ???tb.
    True
```

```
load(DATA+'M.sobj')
```

```
    array([[[ 1,  2],
            [ 3,  4]],

           [[ 0,  5],
            [10,  2]],

           [[ 8,  1],
            [-5,  2]]])
```

```
%hide
%html
<font size=+1>
<h1>Converting between Numpy and Sage Matrices</h1>
```

Converting between Numpy and

Sage Matrices

```
M = matrix(ZZ, 3, [1..9]); show(M)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
type(M)
```

```
<type
'sage.matrix.matrix_complex_double_dense.Matrix_complex_double_den
'>
```

```
A = M.numpy(); A
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]], dtype=object)
```

```
type(A)
```

```
<type 'numpy.ndarray'>
```

```
A = M.numpy(dtype=int); A
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
A.dtype
```

```
dtype('int32')
```

```
type(A)
```

```
<type 'numpy.ndarray'>
```

```
%hide
%html
<font size=+1>
<b>NOTE:</b> All numpy array have exactly the same Python data type
<tt>numpy.ndarray</tt>, whereas Sage's matrices
all have their own types and their is an object inheritance
scheme.
```

NOTE: All numpy array have exactly the same Python data type `numpy.ndarray`, whereas Sage's matrices all have their own types and their is an object inheritance scheme.

```
B = matrix(A); B
```

```
[1 2 3]
[4 5 6]
[7 8 9]
```

```
type(B)
```

```
    <type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
```

```
M = matrix(CDF, 3, [1,2,3.4, 1.2, 2.5, -1.3, 2+I,2-I,e]); M
```

```
    [           1.0            2.0            3.4]
    [           1.2            2.5           -1.3]
    [  2.0 + 1.0*I   2.0 - 1.0*I 2.71828182846]
```

```
A = M.numpy(); A
```

```
    array([[ 1.00000000+0.j,  2.00000000+0.j,  3.40000000+0.j],
           [ 1.20000000+0.j,  2.50000000+0.j, -1.30000000+0.j],
           [ 2.00000000+1.j,  2.00000000-1.j,  2.71828183+0.j]])
```

```
A.dtype
```

```
    dtype('complex128')
```

```
%hide
%html
<font size=+1>
<h1>Numpy Arrays Having an Amazingly Rich Range of
Functionality</h1>
```

Numpy Arrays Having an Amazingly Rich Range of Functionality

```
A = random_matrix(RDF,3).numpy()
shownp(A)
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
# max: return the largest value in A.  Do not use max(A)...
A.max()
```

```
    0.69838023969200602
```

```
max(A)
```

```
Traceback (click to the left for traceback)
...
ValueError: The truth value of an array with more than one element
is ambiguous. Use a.any() or a.all()
```

```
# min: Return the smallest value in A.  Much better than min(A)
A.min()
```

```
-0.90064034618193145
```

```
# ptp: Difference of the largest to the smallest value of A
A.ptp()
```

```
1.5990205858739375
```

```
# clip: Return a new array where any element in A less than min is
set to min
# and any element less than max is set to max.
shownp(A.clip(min=-0.5, max=0.5))
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.500000000000000 & -0.152539619277 & -0.500000000000000 \\ -0.500000000000000 & -0.500000000000000 & 0.500000000000000 \end{pmatrix}$$

```
# conj: return a new array with each of the elements of the input
array
# replaced by their complex conjugate
B = random_matrix(CDF, 3).numpy()
print "Matrix B =           ",
shownp(B)
print "\nComplex conjugate = ",
shownp(B.conj())
```

Matrix B =

$$\begin{pmatrix} \text{(0.476948747614-0.694197532337j)} & \text{(0.763736821227+0.7} \\ \text{(-0.106544963575-0.879795738338j)} & \text{(-0.787438060763+0.4} \\ \text{(-0.37540094272+0.835285088525j)} & \text{(-0.67049739463-0.3} \end{pmatrix}$$

Complex conjugate =

$$\begin{pmatrix} \text{(0.476948747614+0.694197532337j)} & \text{(0.763736821227-0.7} \\ \text{(-0.106544963575+0.879795738338j)} & \text{(-0.787438060763-0.4} \\ \text{(-0.37540094272-0.835285088525j)} & \text{(-0.67049739463+0.3} \end{pmatrix}$$

```
shownp(A)
shownp(A.round())
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 \\ -1.0 & -0.0 & -1.0 \\ -1.0 & -1.0 & 1.0 \end{pmatrix}$$

```
shownp(A)
A.trace()
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
0.64038960470808097
```

```
# sum: Sum up all the elements of the array (or along some "axis")
shownp(A)
print "sum = ", A.sum()
print "sum(A.flatten()) = ", sum(A.flatten())
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
sum =  -1.30383464751
sum(A.flatten()) =  -1.30383464751
```

```
# sum: sum along an "axis", i.e., i,axis  -- I do *NOT* understand
this.
shownp(A)
A.sum(0)  # sum of the rows
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
array([-1.42715235, -0.31242654,  0.43574424])
```

```
A[0,0:]
```

```
array([ 0.09454898,  0.39523963,  0.33971528])
```

```
A[1,0:]
```

```
array([-0.62106099, -0.15253962, -0.60235128])
```

```
A[2,0:]
```

```
array([-0.90064035, -0.55512654,  0.69838024])
```

```
sum(A[i,0:] for i in (0..2))
```

```
array([-1.42715235, -0.31242654,  0.43574424])
```

```
# mean -- return the average value
shownp(A)
A.mean()
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
-0.14487051639037921
```

```
# var -- return the variance of the entries of A
# (Like with sum above an axis can be specified.)
# The variance is the mean of the squares of the differences from
the mean.
html(r'$$\frac{1}{N}\sum_{i=0}^{N-1} (a_i - \mu)^2$$')
```

```
shownp(A)
A.var()
```

$$\frac{1}{N}\sum_{i=0}^{N-1}(a_i-\mu)^2$$

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
0.2745043077223801
```

```
# std -- return the standard deviation of the entries in the matrix
# This is the square root of the variance.
shownp(A)
A.std()
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
0.52393158687216035
```

```
sqrt(A.var())
```

```
0.52393158687216035
```

```
# prod -- product of all entries of self.
A.prod()
```

```
-0.00025294989936084055
```

```
# all -- all entries bool(...) to True
A.all()
```

```
True
```

```
B = A.copy(); B[0,0] = 0
B.all()
```

```
False
```

```
# all -- any entry bool(...) to True
A.any()
```

```
True
```

```
%hide
%html
<h2>Summary of Array Calculation Methods</h2>
<img src="summary-methods.png">
```

Summary of Array Calculation Methods

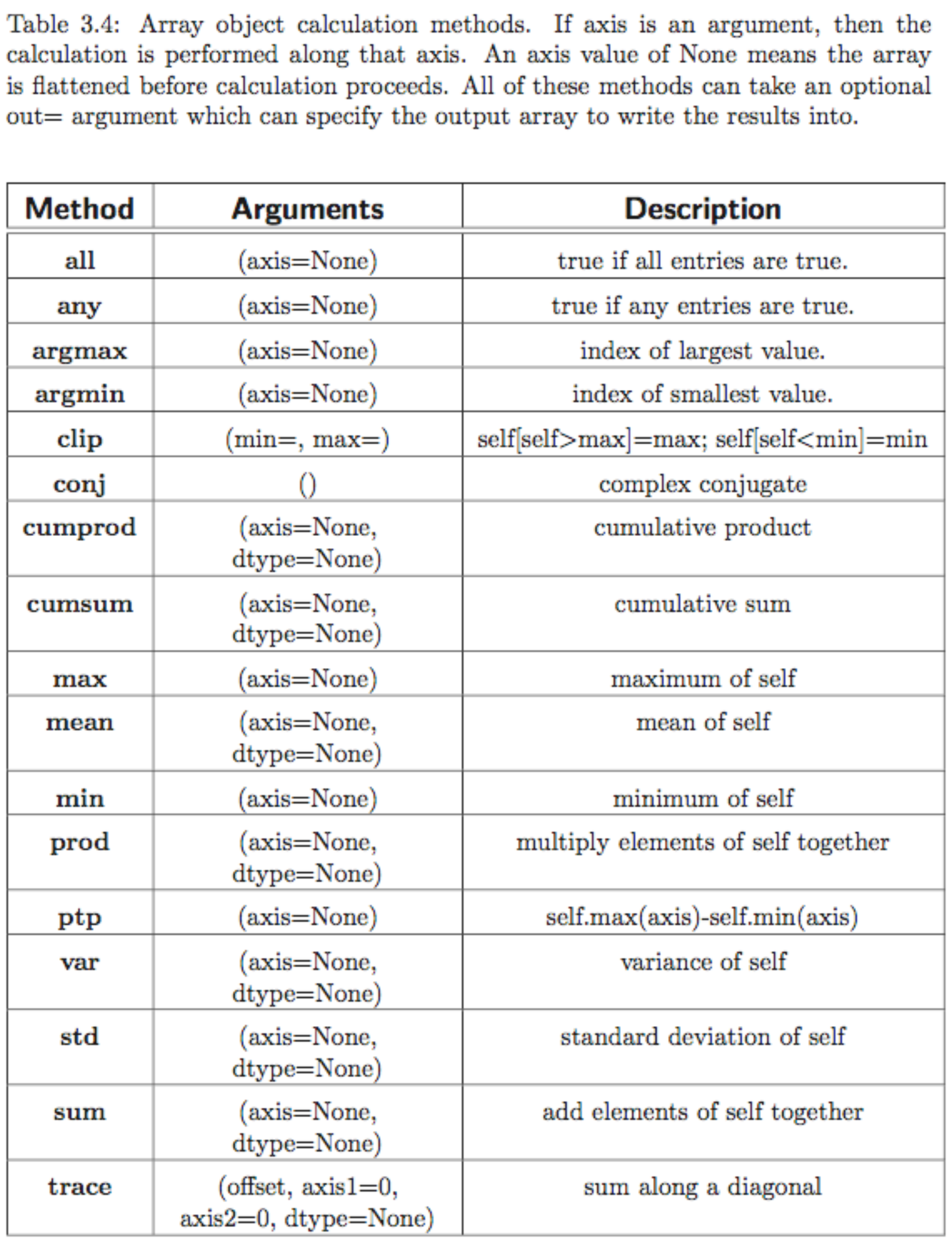


Table 3.4: Array object calculation methods. If axis is an argument, then the calculation is performed along that axis. An axis value of None means the array is flattened before calculation proceeds. All of these methods can take an optional out= argument which can specify the output array to write the results into.

Method	Arguments	Description
all	(axis=None)	true if all entries are true.
any	(axis=None)	true if any entries are true.
argmax	(axis=None)	index of largest value.
argmin	(axis=None)	index of smallest value.
clip	(min=, max=)	self[self>max]=max; self[self<min]=min
conj	()	complex conjugate
cumprod	(axis=None, dtype=None)	cumulative product
cumsum	(axis=None, dtype=None)	cumulative sum
max	(axis=None)	maximum of self
mean	(axis=None, dtype=None)	mean of self
min	(axis=None)	minimum of self
prod	(axis=None, dtype=None)	multiply elements of self together
ptp	(axis=None)	self.max(axis)-self.min(axis)
var	(axis=None, dtype=None)	variance of self
std	(axis=None, dtype=None)	standard deviation of self
sum	(axis=None, dtype=None)	add elements of self together
trace	(offset, axis1=0, axis2=0, dtype=None)	sum along a diagonal

```
# Lots of other functions that aren't methods!
import numpy

# numpy.cov -- compute the covariance matrix of data
shownp(numpy.cov(A))
```

$$\begin{pmatrix} 0.0256007213073 & 0.0284070796819 & 0.0710530963439 \\ 0.0284070796819 & 0.0703654760025 & -0.0620073609308 \\ 0.0710530963439 & -0.0620073609308 & 0.707920979996 \end{pmatrix}$$

```
shownp(numpy.corrcoef(A))
```

$$\left(\begin{array}{rrr} 1.0 & 0.669299766668 & 0.527794081322 \\ 0.669299766668 & 1.0 & -0.277824819792 \\ 0.527794081322 & -0.277824819792 & 1.0 \end{array}\right)$$

```
hist, bins = numpy.histogram(A, bins=4)
shownp(bins)
hist
```

$$\left(\begin{array}{r} -0.900640346182 \\ -0.500885199713 \\ -0.101130053245 \\ 0.298625093224 \end{array}\right)$$

```
array([4, 1, 1, 3])
```

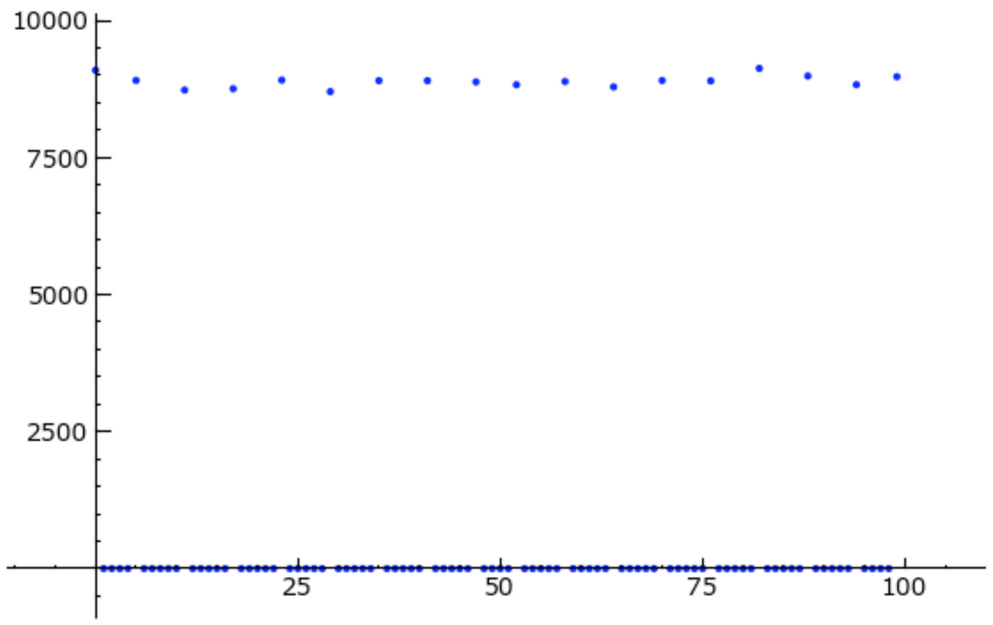
```
B = random_matrix(RDF,100,100).numpy()
hist, bins = numpy.histogram(B, bins=10)
shownp(bins)
hist
```

$$\left(\begin{array}{r} -0.999880634077 \\ -0.799919525802 \\ -0.599958417526 \\ -0.39999730925 \\ -0.200036200974 \\ -7.50926978368e-05 \\ 0.199886015578 \\ 0.399847123854 \\ 0.59980823213 \\ 0.799769340406 \end{array}\right)$$

```
array([1020,  960,  982,  997, 1047,  986,  993, 1044,  977,  994])
```

```
B = random_matrix(ZZ,400,400,x=-9,y=9).numpy(dtype=int)
hist, bins = numpy.histogram(B, bins=100)
print hist
points(list(enumerate(hist)))
```

```
[9092    0    0    0    0 8907    0    0    0    0    0 8730    0
0    0
    0    0 8754    0    0    0    0    0 8912    0    0    0    0
0 8702
    0    0    0    0    0 8902    0    0    0    0    0 8902    0
0    0
    0    0 8879    0    0    0    0 8827    0    0    0    0    0
8886    0
    0    0    0    0 8788    0    0    0    0    0 8905    0    0
0    0
    0 8898    0    0    0    0    0 9126    0    0    0    0    0
8987    0
    0    0    0    0 8828    0    0    0    0 8975]
```



```
%hide
%html
<font size=+1>
<h1>Array Slicing</h1>
Numpy (and Sage) both have sophisticated array slicing somewhat
like Matlab.
```

Array Slicing

Numpy (and Sage) both have sophisticated array slicing somewhat like Matlab.

```
shownp(A)
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
shownp(A[:2,])
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \end{pmatrix}$$

```
shownp(A[:3,0:2])
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 \\ -0.621060987001 & -0.152539619277 \\ -0.900640346182 & -0.555126544917 \end{pmatrix}$$

```
shownp(A[1:3,0:2])
```

$$\begin{pmatrix} -0.621060987001 & -0.152539619277 \\ -0.900640346182 & -0.555126544917 \end{pmatrix}$$

```
shownp(A[1:3,1:3])
```

$$\begin{pmatrix} -0.152539619277 & -0.602351280862 \\ -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
B = matrix(A); show(B)  # sage matrix
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
show(B[1:3, 1:3])
```

$$\begin{pmatrix} -0.152539619277 & -0.602351280862 \\ -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
show(B[:3, 0:2])
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 \\ -0.621060987001 & -0.152539619277 \\ -0.900640346182 & -0.555126544917 \end{pmatrix}$$

```
%hide
%html
<font size=+1>
A <b>key difference</b> between numpy and sage slicing, is that in
Sage the slices are new copies, but in numpy they are references
into the original matrix!
```

A **key difference** between numpy and sage slicing, is that in Sage the slices are new copies, but in numpy they are references into the original matrix!

```
shownp(A)
B = A[0:2,0:2]
```

$$\begin{pmatrix} 0.094548984293 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
B[0,0] = 100
shownp(B)
```

$$\begin{pmatrix} 100.0 & 0.395239626012 \\ -0.621060987001 & -0.152539619277 \end{pmatrix}$$

```
shownp(A)
```

$$\begin{pmatrix} 100.0 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
# But in Sage...
C = matrix(A)  # sage matrix
show(C)
```

$$\begin{pmatrix} 100.0 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
B = C[0:2, 0:2]
B[0,0] = -999
show(B)
```

$$\begin{pmatrix} -999.0 & 0.395239626012 \\ -0.621060987001 & -0.152539619277 \end{pmatrix}$$

```
show(C)
```

$$\begin{pmatrix} 100.0 & 0.395239626012 & 0.339715280728 \\ -0.621060987001 & -0.152539619277 & -0.602351280862 \\ -0.900640346182 & -0.555126544917 & 0.698380239692 \end{pmatrix}$$

```
%hide
%html
<font size=+1>
<h1>There is a LOT more</h1>
Eigenvalues, eigenvectors, decompositions, several different
Fourier transform methods, bit fiddling, random numbers,
distributions, and a C-API, etc. ...

<br>
There is also a lot of work now to make Numpy much more pleasant to
```

```
use from Cython (right now it isn't).
See Dag's GSoC project and Pyx.
```

There is a LOT more

Eigenvalues, eigenvectors, decompositions, several different Fourier transform methods, bit fiddling, random numbers, distributions, and a C-API, etc. ...
There is also a lot of work now to make Numpy much more pleasant to use from Cython (right now it isn't). See Dag's GSoC project and Pyx.

```
numpy.linalg
```

```
<module 'numpy.linalg' from
'/Users/was/build/build/sage-3.0.1/local/lib/python2.5/site-packag
/numpy/linalg/__init__.pyc'>
```

```
numpy.linalg.svd(A)
```

```
(array([[-0.99994001, -0.00480267, -0.0098441 ],
       [ 0.00623649,  0.48919474, -0.87215229],
       [ 0.00900434, -0.87216136, -0.48913544]]), array([
100.00735711,    0.99293485,    0.43942132]), array([[-0.99998627,
-0.00401136, -0.00337138],
       [ 0.00142736,  0.41054067, -0.91184117],
       [-0.00504182,  0.91183346,  0.41052931]]))
```

```
shownp(numpy.fft.fft(A))
```

```
( (100.734954907+0j)    (99.6325225466-0.0480854935439j)   (99.6325225466+0.0480854
  (-1.37595188714+0j)   (-0.243615536931-0.389548325851j)  (-0.243615536931+0.389548
  (-0.757386651407+0j)  (-0.972267193569+1.08556871929j)   (-0.972267193569-1.08556
```