

CS 357: Numerical Methods

Lecture 2: Basis and Numpy

Eric Shaffer

Adapted from the slides of Phillip Klein

Unresolved stuff

- Shape of $(X,)$ versus $(X,1)$
 - Can think of it as *single list* versus *list of lists*
 - Or maybe a row vector versus a column vector

From Stack Overflow:

Here the *shape* `(12,)` means the array is indexed by a single index which runs from 0 to 11. Conceptually, if we label this single index `i`, the array `a` looks like this:

```
i= 0  1  2  3  4  5  6  7  8  9 10 11
```

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

```
>>> d = a.reshape((12, 1))
```

the array `d` is indexed by two indices, the first of which runs from 0 to 11, and the second index is always 0:

```
i= 0  1  2  3  4  5  6  7  8  9 10 11
```

```
j= 0  0  0  0  0  0  0  0  0  0  0  0
```

0	1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	---	----	----

Basis

Definition: Let \mathcal{V} be a vector space. A *basis* for \mathcal{V} is a linearly independent set of generators for \mathcal{V} .

Thus a set S of vectors of \mathcal{V} is a *basis* for \mathcal{V} if S satisfies two properties:

Property B1 (*Spanning*) $\text{Span } S = \mathcal{V}$, and

Property B2 (*Independent*) S is linearly independent.

Most important definition in linear algebra.

Basis: Examples

A set S of vectors of \mathcal{V} is a *basis* for \mathcal{V} if S satisfies two properties:

Property B1 (*Spanning*) $\text{Span } S = \mathcal{V}$, and

Property B2 (*Independent*) S is linearly independent.

Example: Let $\mathcal{V} = \text{Span} \{[1, 0, 2, 0], [0, -1, 0, -2], [2, 2, 4, 4]\}$.

Is $\{[1, 0, 2, 0], [0, -1, 0, -2], [2, 2, 4, 4]\}$ a basis for \mathcal{V} ?

The set *is* spanning but is *not* independent

$$1 [1, 0, 2, 0] - 1 [0, -1, 0, -2] - \frac{1}{2} [2, 2, 4, 4] = \mathbf{0}$$

so not a basis

However, $\{[1, 0, 2, 0], [0, -1, 0, -2]\}$ *is* a basis:

- ▶ Obvious that these vectors are independent because each has a nonzero entry where the other has a zero.
- ▶ To show

$\text{Span} \{[1, 0, 2, 0], [0, -1, 0, -2]\} = \text{Span} \{[1, 0, 2, 0], [0, -1, 0, -2], [2, 2, 4, 4]\}$,
can use Superfluous-Vector Lemma:

$$[2, 2, 4, 4] = 2 [1, 0, 2, 0] - 2 [0, -1, 0, -2]$$

Basis: Examples

Example: A simple basis for \mathbb{R}^3 : the standard generators $\mathbf{e}_1 = [1, 0, 0]$, $\mathbf{e}_2 = [0, 1, 0]$, $\mathbf{e}_3 = [0, 0, 1]$.

- ▶ *Spanning:* For any vector $[x, y, z] \in \mathbb{R}^3$,

$$[x, y, z] = x [1, 0, 0] + y [0, 1, 0] + z [0, 0, 1]$$

- ▶ *Independent:* Suppose

$$\mathbf{0} = \alpha_1 [1, 0, 0] + \alpha_2 [0, 1, 0] + \alpha_3 [0, 0, 1] = [\alpha_1, \alpha_2, \alpha_3]$$

Then $\alpha_1 = \alpha_2 = \alpha_3 = 0$.

Instead of “standard generators”, we call them *standard basis vectors*.

We refer to $\{[1, 0, 0], [0, 1, 0], [0, 0, 1]\}$ as *standard basis* for \mathbb{R}^3 .

In general the standard generators are usually called *standard basis vectors*.

René Descartes



Born 1596.

After studying law in college,....

I entirely abandoned the study of letters. Resolving to seek no knowledge other than that of which could be found in myself or else in the great book of the world, I spent the rest of my youth traveling, visiting courts and armies, mixing with people of diverse temperaments and ranks, gathering various experiences, testing myself in the situations which fortune offered me, and at all times reflecting upon whatever came my way so as to derive some profit from it.

He had a practice of lying in bed in the morning, thinking about mathematics....

Coordinate systems

In 1618, he had an idea...

while lying in bed and watching a fly on the ceiling.

He could describe the location of the fly in terms of two numbers: its distance from the two walls.

He realized that this works even if the two walls were not perpendicular.

He realized that you could express geometry in algebra.

- ▶ The walls play role of what we now call *axes*.
- ▶ The two numbers are what we now call *coordinates*

Coordinate systems

In terms of vectors (and generalized beyond two dimensions),

- ▶ *coordinate system* for a vector space \mathcal{V} is specified by generators $\mathbf{a}_1, \dots, \mathbf{a}_n$ of \mathcal{V}
- ▶ Every vector \mathbf{v} in \mathcal{V} can be written as a linear combination

$$\mathbf{v} = \alpha_1 \mathbf{a}_1 + \dots + \alpha_n \mathbf{a}_n$$

- ▶ We represent vector \mathbf{v} by the vector $[\alpha_1, \dots, \alpha_n]$ of coefficients. called the *coordinate representation* of \mathbf{v} in terms of $\mathbf{a}_1, \dots, \mathbf{a}_n$.

But assigning coordinates to points is not enough. In order to avoid confusion, we must ensure that each point is assigned coordinates in exactly one way. How?

We will discuss unique representation later.

Coordinate representation

Definition: The *coordinate representation* of \mathbf{v} in terms of $\mathbf{a}_1, \dots, \mathbf{a}_n$ is the vector $[\alpha_1, \dots, \alpha_n]$ such that

$$\mathbf{v} = \alpha_1 \mathbf{a}_1 + \dots + \alpha_n \mathbf{a}_n$$

In this context, the coefficients are called the *coordinates*.

Example: The vector $\mathbf{v} = [1, 3, 5, 3]$ is equal to

$$1 [1, 1, 0, 0] + 2 [0, 1, 1, 0] + 3 [0, 0, 1, 1]$$

so the coordinate representation of \mathbf{v} in terms of the vectors $[1, 1, 0, 0], [0, 1, 1, 0], [0, 0, 1, 1]$ is $[1, 2, 3]$.

Example: What is the coordinate representation of the vector $[6, 3, 2, 5]$ in terms of the vectors $[2, 2, 2, 3], [1, 0, -1, 0], [0, 1, 0, 1]$?

Since

$$[6, 3, 2, 5] = 2 [2, 2, 2, 3] + 2 [1, 0, -1, 0] - 1 [0, 1, 0, 1],$$

the coordinate representation is $[2, 2, -1]$.

Coordinate representation

Definition: The *coordinate representation* of \mathbf{v} in terms of $\mathbf{a}_1, \dots, \mathbf{a}_n$ is the vector $[\alpha_1, \dots, \alpha_n]$ such that

$$\mathbf{v} = \alpha_1 \mathbf{a}_1 + \dots + \alpha_n \mathbf{a}_n$$

In this context, the coefficients are called the *coordinates*.

Why put the coordinates in a vector?

Makes sense in view of linear-combinations definitions of matrix-vector multiplication.

$$\text{Let } A = \left[\begin{array}{c|c|c} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{array} \right].$$

- ▶ “ \mathbf{u} is the coordinate representation of \mathbf{v} in terms of $\mathbf{a}_1, \dots, \mathbf{a}_n$ ” can be written as matrix-vector equation $A\mathbf{u} = \mathbf{v}$
- ▶ To go from a coordinate representation \mathbf{u} to the vector being represented, we multiply A times \mathbf{u} .
- ▶ To go from a vector \mathbf{v} to its coordinate representation, we can solve the matrix-vector equation $A\mathbf{x} = \mathbf{v}$.
(Because the columns of A are generators for \mathcal{V} and \mathbf{v} belongs to \mathcal{V} , the equation must have at least one solution.)

Introduction to NumPy

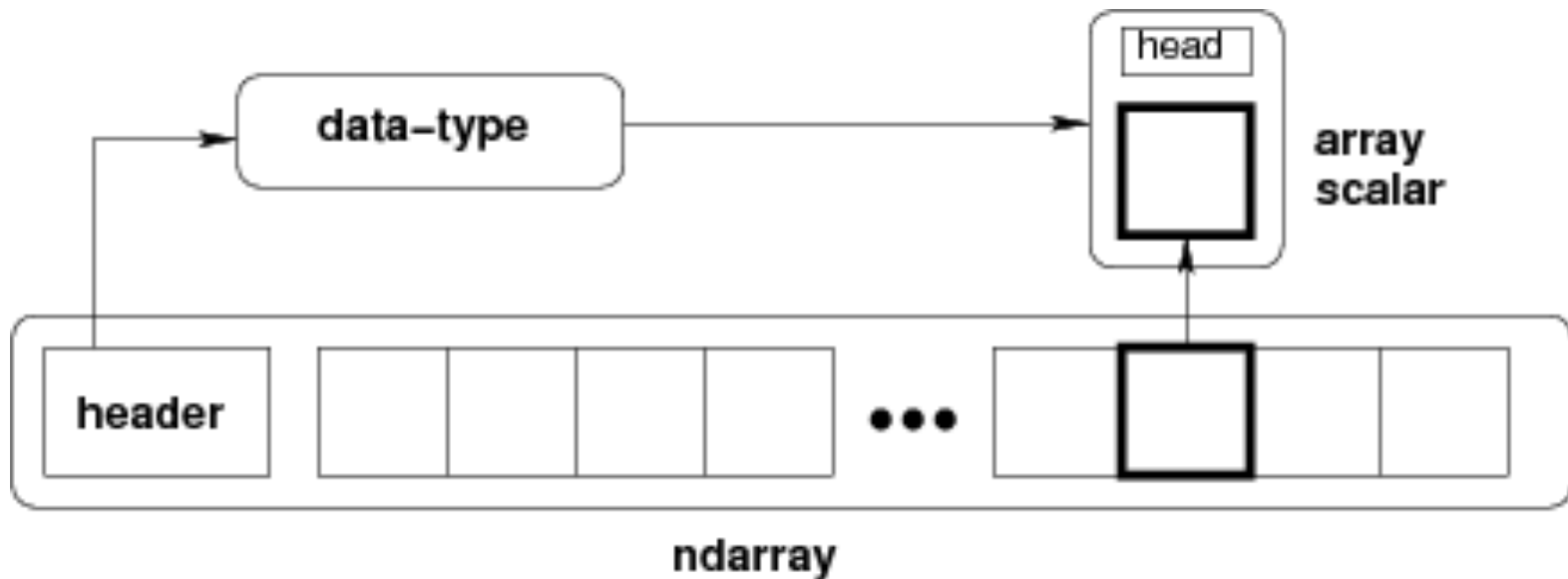
Slides adapted from Travis E. Oliphant
Enthought, Inc.

▣ www.enthought.com

- ▣ Python has no built-in multi-dimensional array
- ▣ NumPy provides a **fast** built-in object (ndarray) which is a multi-dimensional array of a homogeneous data-type.

NumPy Array

A NumPy array is an N-dimensional homogeneous collection of “items” of the same “kind”. The kind can be any arbitrary structure and is specified using the data-type.



Introducing NumPy Arrays

SIMPLE ARRAY CREATION

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

CHECKING THE TYPE

```
>>> type(a)
<type 'array'>
```

NUMERIC 'TYPE' OF ELEMENTS

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

BYTES PER ELEMENT

```
>>> a.itemsize # per element
4
```

ARRAY SHAPE

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

ARRAY SIZE

```
# size reports the entire
# number of elements in an
# array.
>>> a.size
4
>>> size(a)
4
```

Introducing NumPy Arrays

BYTES OF MEMORY USED

```
# returns the number of bytes
# used by the data portion of
# the array.
>>> a.nbytes
12
```

NUMBER OF DIMENSIONS

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

ARRAY COPY

```
# create a copy of the array
>>> b = a.copy()
>>> b
array([0, 1, 2, 3])
```

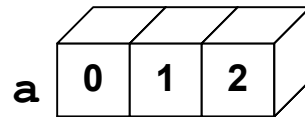
CONVERSION TO LIST

```
# convert a numpy array to a
# python list.
>>> a.tolist()
[0, 1, 2, 3]

# For 1D arrays, list also
# works equivalently, but
# is slower.
>>> list(a)
[0, 1, 2, 3]
```

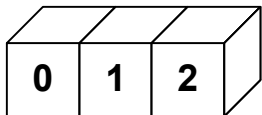
Indexing with None

`None` is a special index that inserts a new axis in the array at the specified location. Each `None` increases the array's dimensionality by 1.



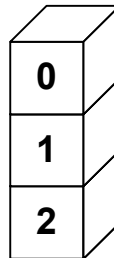
1 X 3

```
>>> y = a[None, :]  
>>> shape(y)  
(1, 3)
```



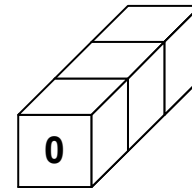
3 X 1

```
>>> y = a[:, None]  
>>> shape(y)  
(3, 1)
```



3 X 1 X 1

```
>>> y = a[:, None, None]  
>>> shape(y)  
(3, 1, 1)
```



Setting Array Elements

ARRAY INDEXING

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

FILL

set all values in an array.

```
>>> a.fill(0)
>>> a
[0, 0, 0, 0]
```

This also works, but may
be slower.

```
>>> a[:] = 1
>>> a
[1, 1, 1, 1]
```



BEWARE OF TYPE COERSION

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

assigning a float to into # an
int32 array will
truncate decimal part.

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

fill has the same behavior

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


Multi-Dimensional Arrays

MULTI-DIMENSIONAL ARRAYS

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

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

(ROWS,COLUMNS)

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

ELEMENT COUNT


```
>>> a.size  
8  
>>> size(a)  
8
```

NUMBER OF DIMENSIONS

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

GET/SET ELEMENTS

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



The diagram shows a 2D array with two rows and four columns. The first row is [0, 1, 2, 3] and the second row is [10, 11, 12, 13]. The element at row 1, column 3 is 13. The label 'row' points to the first index (1) and 'column' points to the second index (3).

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

ADDRESS FIRST ROW USING SINGLE INDEX

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

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])
```

STRIDES ARE ALSO POSSIBLE

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

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

Slices Are References

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

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

```
# create a slice containing only the  
# last element of a
```

```
>>> b = a[2:4]
```

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

```
# changing b changed a!
```

```
>>> a
```

```
array([ 1,  2, 10,  3,  4])
```

Fancy Indexing

INDEXING BY POSITION

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

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

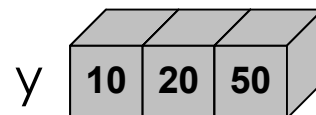
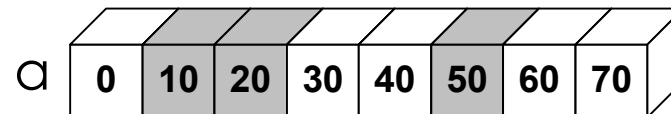
```
# using take
>>> y = take(a,[1,2,-3])
>>> print y
[10 20 50]
```

INDEXING WITH BOOLEANS

```
>>> mask = array([0,1,1,0,0,1,0,0],
...              dtype=bool)
```

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

```
# using compress
>>> y = compress(mask, a)
>>> print y
[10,20,50]
```



Fancy Indexing in 2D

```
>>> 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 = array([1,0,1,0,0,1],  
                 dtype=bool)
```

```
>>> a[mask,2]  
array([2,22,52])
```

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



Unlike slicing, fancy indexing creates copies instead of views into original arrays.

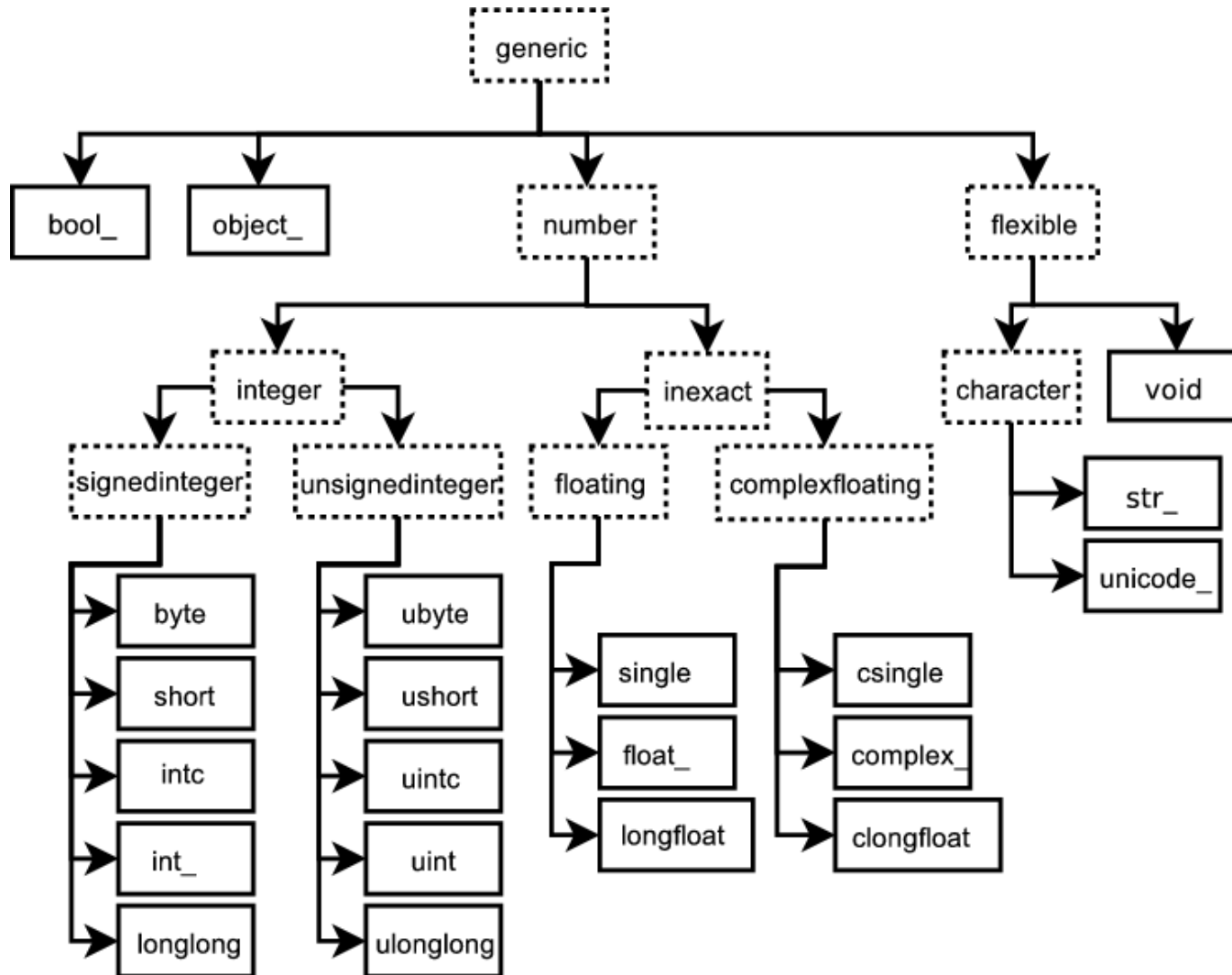
Data-types

- There are two related concepts of “type”
 - The data-type object (dtype)
 - The Python “type” of the object created from a single array item (hierarchy of scalar types)
- The **dtype** object provides the details of how to interpret the memory for an item. It's an instance of a single dtype class.
- The “type” of the extracted elements are true Python classes that exist in a hierarchy of Python classes
- Every dtype object has a type attribute which provides the Python object returned when an element is selected from the array

NumPy dtypes

Basic Type	Available NumPy types	Comments
Boolean	<code>bool</code>	Elements are 1 byte in size
Integer	<code>int8, int16, int32, int64, int128, int</code>	<code>int</code> defaults to the size of <code>int</code> in C for the platform
Unsigned Integer	<code>uint8, uint16, uint32, uint64, uint128, uint</code>	<code>uint</code> defaults to the size of unsigned <code>int</code> in C for the platform
Float	<code>float32, float64, float, longfloat,</code>	Float is always a double precision floating point value (64 bits). <code>longfloat</code> represents large precision floats. Its size is platform dependent.
Complex	<code>complex64, complex128, complex</code>	The real and complex elements of a <code>complex64</code> are each represented by a single precision (32 bit) value for a total size of 64 bits.
Strings	<code>str, unicode</code>	Unicode is always UTF32 (UCS4)
Object	<code>object</code>	Represent items in array as Python objects.
Records	<code>void</code>	Used for arbitrary data structures in record arrays.

Built-in “scalar” types



Array Calculation Methods

SUM FUNCTION

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

# Sum defaults to summing all
# *all* array values.
>>> sum(a)
21.

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

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

SUM ARRAY METHOD

```
# The a.sum() defaults to
# summing *all* array values
>>> a.sum()
21.
# Supply an axis argument to
# sum along a specific axis.
>>> a.sum(axis=0)
array([5., 7., 9.] )
```

PRODUCT

```
# product along columns.
>>> a.prod(axis=0)
array([ 4., 10., 18.] )

# functional form.
>>> prod(a, axis=0)
array([ 4., 10., 18.] )
```

Min/Max

MIN

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

```
0.
```

```
# use Numpy's amin() instead  
# of Python's builtin min()  
# for speed operations on  
# multi-dimensional arrays.
```

```
>>> amin(a, axis=0)
```

```
0.
```

ARGMIN

```
# Find index of minimum value.
```

```
>>> a.argmin(axis=0)
```

```
2
```

```
# functional form
```

```
>>> argmin(a, axis=0)
```

```
2
```

MAX

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

```
3.
```

```
# functional form
```

```
>>> amax(a, axis=0)
```

```
3.
```

ARGMAX

```
# Find index of maximum value.
```

```
>>> a.argmax(axis=0)
```

```
1
```

```
# functional form
```

```
>>> argmax(a, axis=0)
```

```
1
```

Statistics Array Methods

MEAN

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

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

# average can also calculate
# a weighted average
>>> average(a, weights=[1,2],
...         axis=0)
array([ 3.,  4.,  5.]
```

STANDARD DEV./VARIANCE

```
# Standard Deviation
>>> a.std(axis=0)
array([ 1.5,  1.5,  1.5])

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

Other Array Methods

CLIP

Limit values to a range

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

Set values < 3 equal to 3.

Set values > 5 equal to 5.

```
>>> a.clip(3,5)  
>>> a  
array([[ 3.,  3.,  3.],  
       [ 4.,  5.,  5.]])
```

ROUND

Round values in an array.

Numpy rounds to even, so

1.5 and 2.5 both round to 2.

```
>>> a = array([1.35, 2.5, 1.5])  
>>> a.round()  
array([ 1.,  2.,  2.]
```

Round to first decimal place.

```
>>> a.round(decimals=1)  
array([ 1.4,  2.5,  1.5])
```

POINT TO POINT

Calculate max – min for

array along columns

```
>>> a.ptp(axis=0)  
array([ 3.0,  3.0,  3.0])
```

max – min for entire array.

```
>>> a.ptp(axis=None)  
5.0
```

Summary of (most) array attributes/methods

BASIC ATTRIBUTES

`a.dtype` - Numerical type of array elements. `float32`, `uint8`, etc.
`a.shape` - Shape of the array. `(m,n,o,...)`
`a.size` - Number of elements in entire array.
`a.itemsize` - Number of bytes used by a single element in the array.
`a.nbytes` - Number of bytes used by entire array (data only).
`a.ndim` - Number of dimensions in the array.

SHAPE OPERATIONS

`a.flat` - An iterator to step through array as if it is 1D.
`a.flatten()` - Returns a 1D copy of a multi-dimensional array.
`a.ravel()` - Same as `flatten()`, but returns a 'view' if possible.
`a.resize(new_size)` - Change the size/shape of an array in-place.
`a.swapaxes(axis1, axis2)` - Swap the order of two axes in an array.
`a.transpose(*axes)` - Swap the order of any number of array axes.
`a.T` - Shorthand for `a.transpose()`
`a.squeeze()` - Remove any `length=1` dimensions from an array.

Summary of (most) array attributes/methods

FILL AND COPY

`a.copy()` - Return a copy of the array.
`a.fill(value)` - Fill array with a scalar value.

CONVERSION / COERSION

`a.tolist()` - Convert array into nested lists of values.
`a.tostring()` - raw copy of array memory into a python string.
`a.astype(dtype)` - Return array coerced to given dtype.
`a.byteswap(False)` - Convert byte order (big \leftrightarrow little endian).

COMPLEX NUMBERS

`a.real` - Return the real part of the array.
`a.imag` - Return the imaginary part of the array.
`a.conjugate()` - Return the complex conjugate of the array.
`a.conj()` - Return the complex conjugate of an array. (same as conjugate)

Summary of (most) array attributes/methods

SAVING

`a.dump(file)` - Store a binary array data out to the given file.
`a.dumps()` - returns the binary pickle of the array as a string.
`a.tofile(fid, sep="", format="%s")` Formatted ascii output to file.

SEARCH / SORT

`a.nonzero()` - Return indices for all non-zero elements in `a`.
`a.sort(axis=-1)` - Inplace sort of array elements along axis.
`a.argsort(axis=-1)` - Return indices for element sort order along axis.
`a.searchsorted(b)` - Return index where elements from `b` would go in `a`.

ELEMENT MATH OPERATIONS

`a.clip(low, high)` - Limit values in array to the specified range.
`a.round(decimals=0)` - Round to the specified number of digits.
`a.cumsum(axis=None)` - Cumulative sum of elements along axis.
`a.cumprod(axis=None)` - Cumulative product of elements along axis.

Summary of (most) array attributes/methods

REDUCTION METHODS

All the following methods “reduce” the size of the array by 1 dimension by carrying out an operation along the specified axis. If axis is None, the operation is carried out across the entire array.

`a.sum(axis=None)` - Sum up values along axis.

`a.prod(axis=None)` - Find the product of all values along axis.

`a.min(axis=None)` - Find the minimum value along axis.

`a.max(axis=None)` - Find the maximum value along axis.

`a.argmin(axis=None)` - Find the index of the minimum value along axis.

`a.argmax(axis=None)` - Find the index of the maximum value along axis.

`a.ptp(axis=None)` - Calculate `a.max(axis) - a.min(axis)`

`a.mean(axis=None)` - Find the mean (average) value along axis.

`a.std(axis=None)` - Find the standard deviation along axis.

`a.var(axis=None)` - Find the variance along axis.

`a.any(axis=None)` - True if any value along axis is non-zero. (or)

`a.all(axis=None)` - True if all values along axis are non-zero. (and)

Array Operations

SIMPLE ARRAY MATH

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

Numpy defines the following constants:



```
pi = 3.14159265359
e = 2.71828182846
```

MATH FUNCTIONS

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

```
# multiply entire array by
# scalar value
```

```
>>> a = (2*pi)/10.
```

```
>>> a
```

```
0.62831853071795862
```

```
>>> a*x
```

```
array([ 0.,0.628,...,6.283])
```

```
# inplace operations
```

```
>>> x *= a
```

```
>>> x
```

```
array([ 0.,0.628,...,6.283])
```

```
# apply functions to array.
```

```
>>> y = sin(x)
```

Universal Functions

- ▣ ufuncs are objects that rapidly evaluate a function element-by-element over an array.
- ▣ Core piece is a 1-d loop written in C that performs the operation over the largest dimension of the array
- ▣ For 1-d arrays it is equivalent to but much faster than list comprehension

```
>>> type(N.exp)
<type 'numpy.ufunc'>
>>> x = array([1,2,3,4,5])
>>> print N.exp(x)
[  2.71828183   7.3890561   20.08553692
 54.59815003  148.4131591 ]
>>> print [math.exp(val) for val in x]
[2.7182818284590451,
 7.3890560989306504, 20.085536923187668,
 54.598150033144236, 148.4131591025766]
```

Mathematic Binary Operators

$a + b \rightarrow \text{add}(a,b)$
 $a - b \rightarrow \text{subtract}(a,b)$
 $a \% b \rightarrow \text{remainder}(a,b)$

$a * b \rightarrow \text{multiply}(a,b)$
 $a / b \rightarrow \text{divide}(a,b)$
 $a ** b \rightarrow \text{power}(a,b)$

MULTIPLY BY A SCALAR

```
>>> a = array((1,2))
>>> a*3.
array([3., 6.]
```

ELEMENT BY ELEMENT ADDITION

```
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

ADDITION USING AN OPERATOR FUNCTION

```
>>> add(a,b)
array([4, 6])
```

IN PLACE OPERATION

```
# Overwrite contents of a.
# Saves array creation
# overhead
>>> add(a,b,a) # a += b
array([4, 6])
>>> a
array([4, 6])
```

Comparison and Logical Operators

<code>equal</code>	<code>(==)</code>	<code>not_equal</code>	<code>(!=)</code>	<code>greater</code>	<code>(>)</code>
<code>greater_equal</code>	<code>(>=)</code>	<code>less</code>	<code>(<)</code>	<code>less_equal</code>	<code>(<=)</code>
<code>logical_and</code>		<code>logical_or</code>		<code>logical_xor</code>	
<code>logical_not</code>					

2D EXAMPLE

```
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[True, True, False, True],
       [False, True, True, True]])
# functional equivalent
>>> equal(a,b)
array([[True, True, False, True],
       [False, True, True, True]])
```

Broadcasting

When there are multiple inputs, then they all must be “broadcastable” to the same shape.

- All arrays are promoted to the same number of dimensions (by pre-pending 1's to the shape)
- All dimensions of length 1 are expanded as determined by other inputs with non-unit lengths in that dimension.

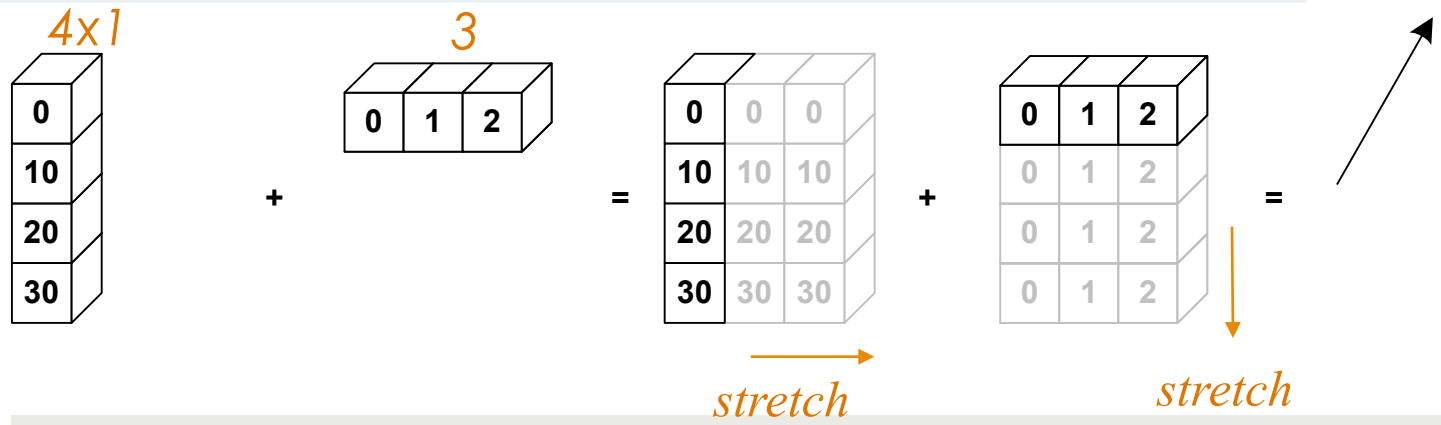
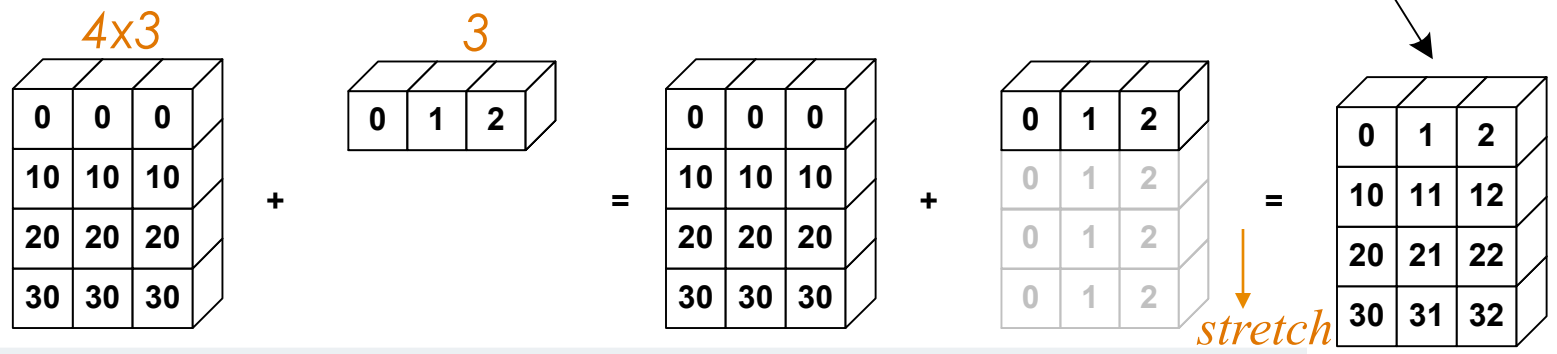
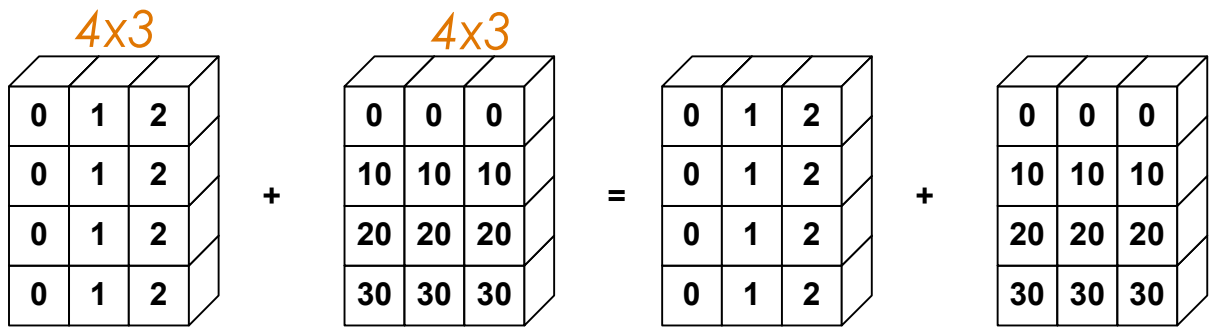
```
>>> x = [1,2,3,4];
>>> y = [[10],[20],[30]]
>>> print N.add(x,y)
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
>>> x = array(x)
>>> y = array(y)
>>> print x+y
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

x has shape (4,) the ufunc sees it as having shape (1,4)

y has shape (3,1)

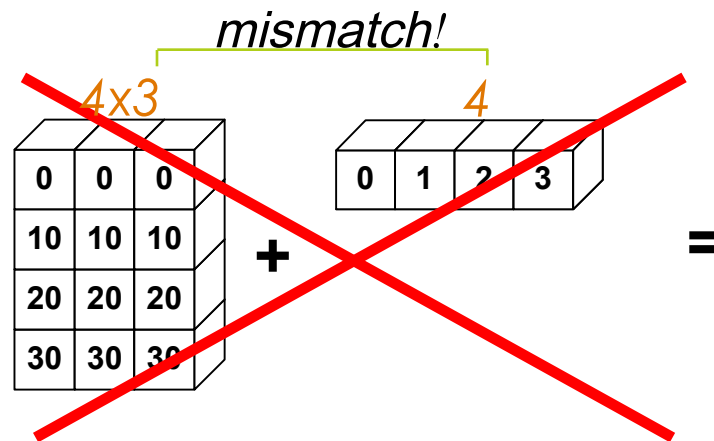
The ufunc result has shape (3,4)

Array Broadcasting



Broadcasting Rules

The *trailing* axes of both arrays must either be 1 or have the same size for broadcasting to occur. Otherwise, a “`ValueError: frames are not aligned`” exception is thrown.



Broadcasting in Action

```
>>> a = array((0,10,20,30))  
>>> b = array((0,1,2))  
>>> y = a[:, None] + b
```

