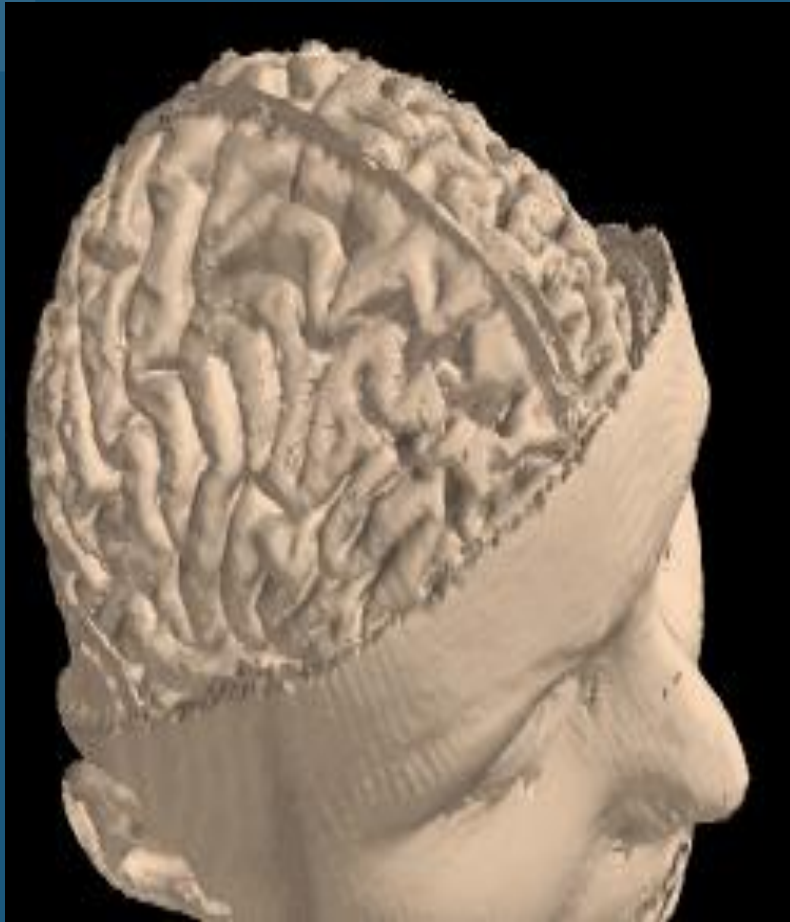


Programming with Python

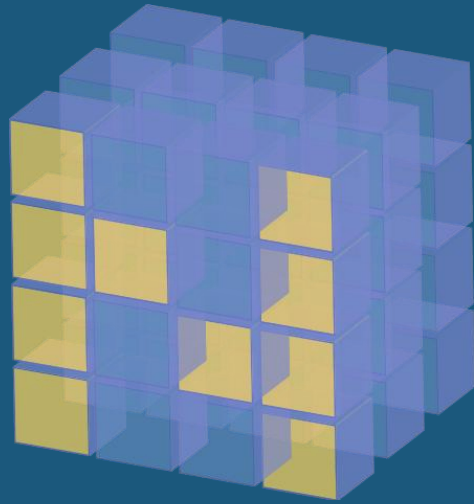


View MRI of brain with cut plane



Steps

- ✓ Get an MRI scan of brain
- ✓ Convert it into a **3D Array**
- ✓ Extract some internal structures of the brain
- ✓ Display **two cut planes**
- ✓ Display the outer surface



NumPy

1. Introduction
2. NumPy Constants
3. NDArray (Array Objects)
4. Universal Functions
5. Broadcasting
6. Structured Data Types
7. Examples

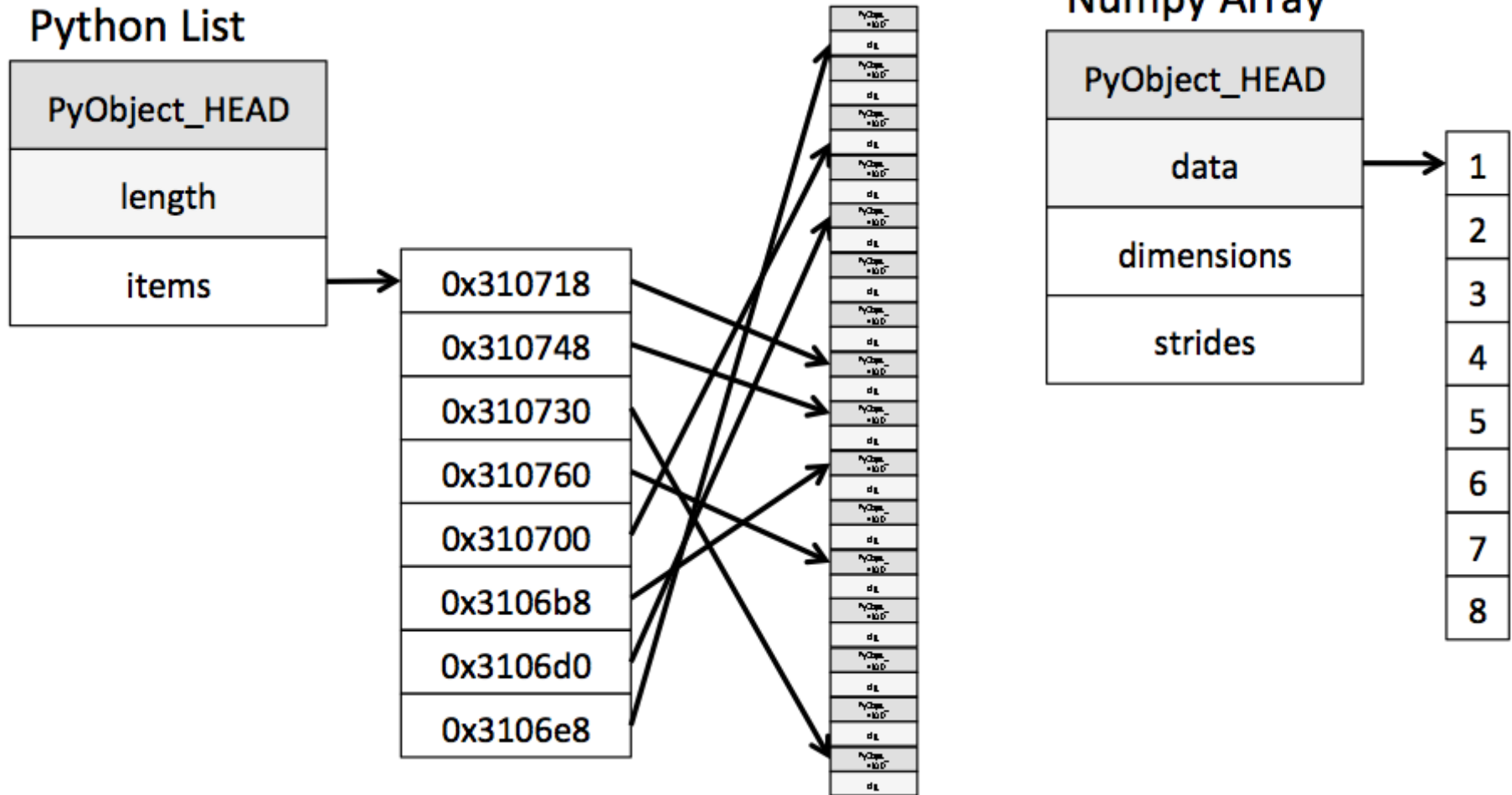
What is Numpy?

- **NumPy** is the fundamental package for scientific computing with Python. It contains among other things:
 - A powerful **N-dimensional array object**
 - Sophisticated **functions (ufuncs)**
 - Tools for **integrating C/C++** and FORTRAN code
 - Useful **linear algebra, Fourier transform, and random number** capabilities

NumPy Array .vs. List

- **NumPy Array:** A NumPy array in its simplest form is a Python object build around a C array.
- That is, it has **a pointer to a *contiguous* data buffer of values.**
- **Python List:** A Python list, on the other hand, has **a pointer to a contiguous buffer of pointers, each of which points to a Python object which in turn has references to its data** (in this case, integers).
- If you're doing some operation which steps through data in sequence, the **numpy layout will be much more efficient than the Python layout**, both in the cost of **storage** and the **cost of access**.

NumPy Array .vs. List



So, NumPy Array performs better when compared to List

NumPy Array .vs. List

■ Similarities

- Both are used for storing data
- Both are mutable
- Both can be indexed and iterated through
- Both can be sliced

NumPy Array .vs. List

- The important advantages of **NumPy Array** are:
 - **Memory/Size** – Consumes less space
 - **Performance** – Faster than Lists
 - **Functionality** – Provides very rich set of functions
 - Linear Algebra
 - Fourier Transformation
 - Random Numbers

NumPy Array .vs. List

- The important advantages of **List** are:
 - Flexible
 - Heterogeneous elements can be stored
 - Easy to use

How to install NumPy?

https://www.scipy.org/scipylib/download.html



SciPy.org



Sponsored By
ENTHOUGHT

SciPy.org

SciPy library

Obtaining NumPy & SciPy libraries

See also:

[Installing packages](#)

Official Source and Binary Releases

For each official release of NumPy and SciPy, we provide source code (tarball) as well as binary wheels for several major platforms (Windows, OSX, Linux).

Project	Available packages	Download location
NumPy	Official <i>source code</i> (all platforms) and <i>binaries</i> for Windows, Linux and Mac OS X	PyPI page for NumPy
SciPy	Official <i>source code</i> (all platforms) and <i>binaries</i> for Windows, Linux and Mac OS X	SciPy release page (sources) PyPI page for SciPy (all)

Source Code Repository Access

SCIPY LIBRARY:

[SciPy \(library\)](#)

[License](#)

[Code of
Conduct](#)

[Download](#)

[Documentation](#)



[Build
instructions](#)



[Mailing Lists](#)

[Report Bugs](#)

[Developer
Zone](#)

[Donations](#)

[FAQ](#)

NumPy Constants (numpy_math.h)

```
#define NPY_INFINITYF __numpy_inff()
#define NPY_NANF __numpy_nanf()
#define NPY_PZEROF __numpy_pzerof()
#define NPY_NZEROF __numpy_nzerof()

#define NPY_INFINITY ((numpy_double)NPY_INFINITYF)
#define NPY_NAN ((numpy_double)NPY_NANF)
#define NPY_PZERO ((numpy_double)NPY_PZEROF)
#define NPY_NZERO ((numpy_double)NPY_NZEROF)

#define NPY_INFINITYL ((numpy_longdouble)NPY_INFINITYF)
#define NPY_NANL ((numpy_longdouble)NPY_NANF)
#define NPY_PZEROL ((numpy_longdouble)NPY_PZEROF)
#define NPY_NZEROL ((numpy_longdouble)NPY_NZEROF)

/*
 * Useful constants
 */
#define NPY_E 2.718281828459045235360287471352662498 /* e */
#define NPY_LOG2E 1.442695040888963407359924681001892137 /* log_2 e */
#define NPY_LOG10E 0.434294481903251827651128918916605082 /* log_10 e */
#define NPY_LOGE2 0.693147180559945309417232121458176568 /* log_e 2 */
#define NPY_LOGE10 2.302585092994045684017991454684364208 /* log_e 10 */
#define NPY_PI 3.141592653589793238462643383279502884 /* pi */
#define NPY_PI_2 1.570796326794896619231321691639751442 /* pi/2 */
#define NPY_PI_4 0.785398163397448309615660845819875721 /* pi/4 */
```



NumPy Data Types

Data Type Objects (dtype)

- **A data type object** - an instance of [numpy.dtype](#) class
- It describes the following aspects of the data:
 - **Type of the data** (integer, float, Python object, etc.)
 - **Size of the data** (how many bytes is in *e.g.* the integer)
 - **Byte order of the data** (little-endian or big-endian)
 - If the data type is **structured**, **an aggregate of other data types**, (*e.g.*, describing an array item consisting of an integer and a float),
 - what are the names of the “[fields](#)” of the structure, by which they can be [accessed](#),
 - what is the data-type of each [field](#), and
 - which part of the memory block each field takes.
 - If the data type is a sub-array, what is its shape and data type.

Constructing dtype objects

Examples:

```
>>> dt = np.dtype(np.int32) # 32-bit integer
```

```
>>> dt = np.dtype(np.complex128) # 128-bit complex floating-  
point number
```

Generic Types

Generic Types

number , inexact , floating	float
complexfloating	cfloat
integer , signedinteger	int_
unsignedinteger	uint
character	string
generic, flexible	void

Built-In Python Types

int	int_	-
bool	bool_	-
float	float_	-
complex	cfloat	-
bytes	bytes_	-
str	bytes_ (Python2) or unicode_ (Python3)	-

Array Protocol Type Strings

- The first character specifies the **kind of data** and the remaining characters specify the **number of bytes per item**.
- For Unicode, it is interpreted as the number of characters.
- The item size must correspond to an existing type, or an error will be raised.

'?'	boolean
'b'	(signed) byte
'B'	unsigned byte
'i'	(signed) integer
'u'	unsigned integer
'f'	floating-point
'c'	complex-floating po
'm'	timedelta
'M'	datetime
'O'	(Python) objects
'S' , 'a'	zero-terminated bytes (not recommended)
'U'	Unicode string
'V'	raw data (void)

```
dt = np.dtype('i4') # 32-bit signed integer
```

```
dt = np.dtype('f8') # 64-bit floating-point number
```

```
dt = np.dtype('c16') # 128-bit complex floating-point number
```

```
dt = np.dtype('a25') # 25-length zero-terminated bytes
```

```
dt = np.dtype('U25') # 25-character string
```


Structured Data Types

- A short-hand notation for specifying the format of a **structured data type** is a comma-separated string of basic formats.
- The generated data-type fields are named 'f0', 'f1', ..., 'f<N-1>' where N (>1) is the number of comma-separated basic formats in the string.

Example:

- field named f0 containing a 32-bit integer
- field named f1 containing a 2 x 3 sub-array of 64-bit floating-point numbers
- field named f2 containing a 32-bit floating-point number

```
>>> dt = np.dtype("i4, (2,3)f8, f4")
```

- field named f0 containing a 3-character string
- field named f1 containing a sub-array of shape (3,) containing 64-bit unsigned integers
- field named f2 containing a 3 x 4 sub-array containing 10-character strings

```
>>> dt = np.dtype("a3, 3u8, (3,4)a10")
```

dtype Attributes

- dtype.char** A unique character code for each of the 21 different built-in types.
- dtype.num** A unique number for each of the 21 different built-in types.
- dtype.str** The array-protocol typestring of this data-type object.

Size of the data is in turn described by:

- dtype.name** A bit-width name for this data-type.
- dtype.itemsize** The element size of this data-type object.

Endianness of this data:

- dtype.byteorder** A character indicating the byte-order of this data-type object.

Information about sub-data-types in a structured data type:

- dtype.fields** Dictionary of named fields defined for this data type, or `None`.
- dtype.names** Ordered list of field names, or `None` if there are no fields.

For data types that describe sub-arrays:

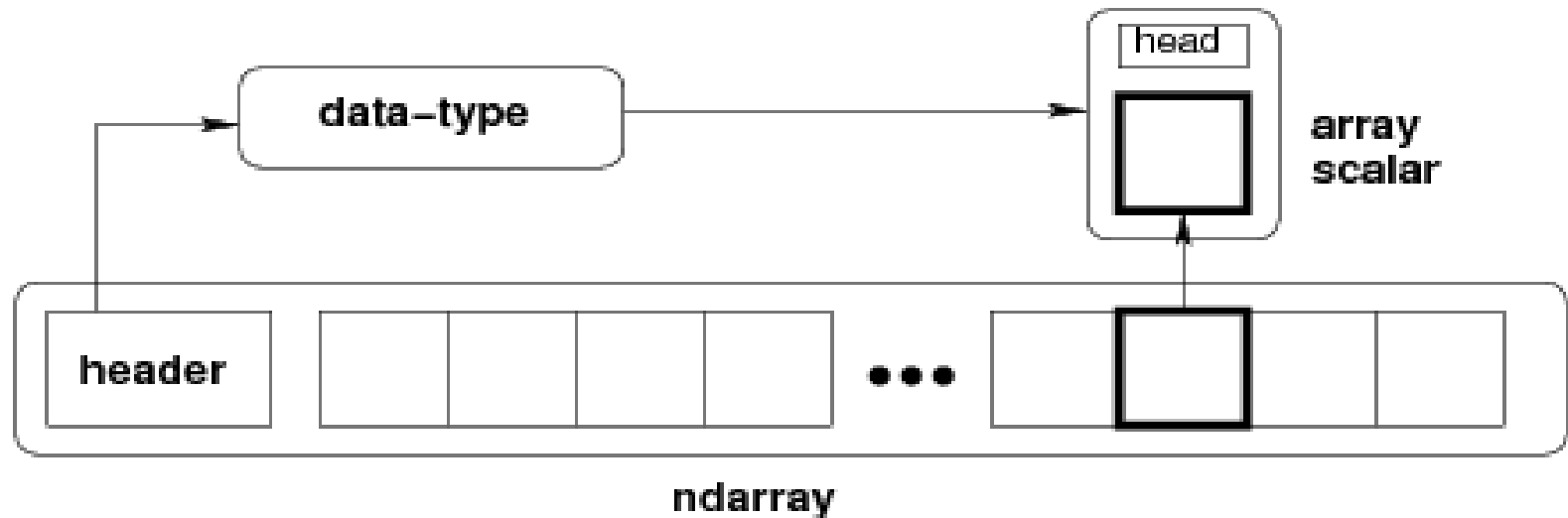
- dtype.subdtype** Tuple `(item_dtype, shape)` if this **dtype** describes a sub-array, and `None` otherwise.
- dtype.shape** Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise.



NumPy Arrays

NumPy Array (ndarray)

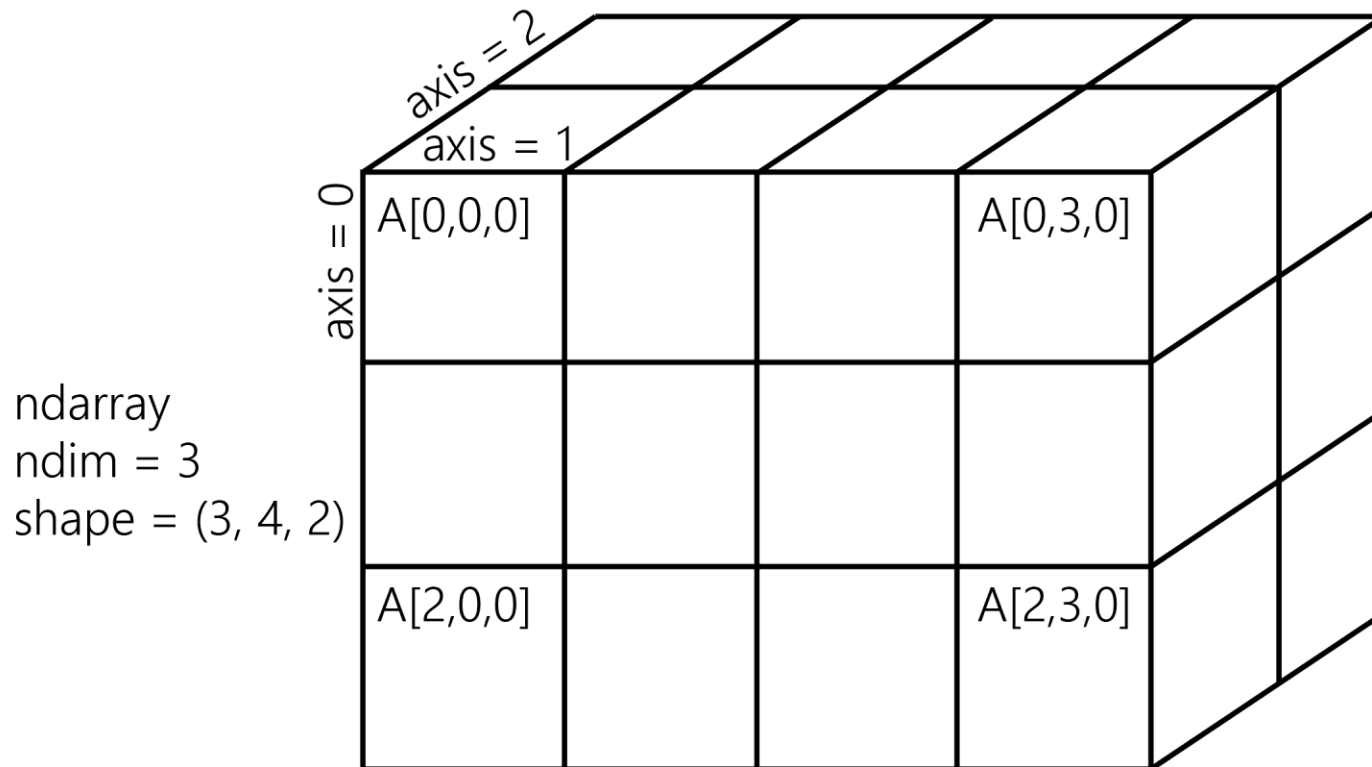
A NumPy array is an **N-dimensional homogeneous** collection of “items” of same kind. The **kind can be any arbitrary structure** and is specified using the data-type.



- 1) the **ndarray** itself,
- 2) the **data-type object** that describes the layout of a single fixed-size element of the array,
- 3) the **array-scalar Python object or Array Element** that is returned when a single element of the array is accessed.

NumPy Array (ndarray)...

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.



Array Creation

There are 5 general mechanisms:

1. Intrinsic numpy array creation (e.g., arange, ones, zeros, etc.)
2. Conversion from other Python structures (e.g., lists, tuples)
3. Reading arrays from disk (either from standard or custom formats)
4. Creating arrays from raw bytes (through the use of strings or buffers)
5. Use of special library functions (e.g., random)

Other Ways: Replicating, Joining, Expanding, Structured Arrays etc..

Array Creation – Intrinsic Methods

Ones and zeros

empty(shape[, dtype, order])

Return a new array of given shape and type, without initializing entries.

empty_like(prototype[, dtype, order, subok])

Return a new array with the same shape and type as a given array.

eye(N[, M, k, dtype, order])

Return a 2-D array with ones on the diagonal and zeros elsewhere.

identity(n[, dtype])

Return the identity array.

ones(shape[, dtype, order])

Return a new array of given shape and type, filled with ones.

ones_like(a[, dtype, order, subok])

Return an array of ones with the same shape and type as a given array.

zeros(shape[, dtype, order])

Return a new array of given shape and type, filled with zeros.

zeros_like(a[, dtype, order, subok])

Return an array of zeros with the same shape and type as a given array.

full(shape, fill_value[, dtype, order])

Return a new array of given shape and type, filled with *fill_value*.

full_like(a, fill_value[, dtype, order, subok])

Return a full array with the same shape and type as a given array.

Array Creation - Example

Array Creation Example

Refer Jupyter Notebooks

Array Creation – From existing data

- [array](#)(object[, dtype, copy, order, subok, ndmin])
Create an array.
- [asarray](#)(a[, dtype, order])
Convert the input to an array.
- [asanyarray](#)(a[, dtype, order])
Convert the input to an ndarray, but pass ndarray subclasses through.
- [ascontiguousarray](#)(a[, dtype])
Return a contiguous array in memory (C order).
- [asmatrix](#)(data[, dtype])
Interpret the input as a matrix.
- [copy](#)(a[, order])
Return an array copy of the given object.

Array Attributes

Array attributes reflect information that is intrinsic to the array itself.

Memory layout

The following attributes contain information about the memory layout of the array:

<code>ndarray.flags</code>	Information about the memory layout of the array.
<code>ndarray.shape</code>	Tuple of array dimensions.
<code>ndarray.strides</code>	Tuple of bytes to step in each dimension when traversing an array.
<code>ndarray.ndim</code>	Number of array dimensions.
<code>ndarray.data</code>	Python buffer object pointing to the start of the array's data.
<code>ndarray.size</code>	Number of elements in the array.
<code>ndarray.itemsize</code>	Length of one array element in bytes.
<code>ndarray.nbytes</code>	Total bytes consumed by the elements of the array.
<code>ndarray.base</code>	Base object if memory is from some other object.

Array Attributes...

Array attributes reflect information that is intrinsic to the array itself.

The data type object associated with the array can be found in the `dtype` attribute:

`ndarray.dtype` Data-type of the array's elements.

Other attributes

<code>ndarray.T</code>	Same as <code>self.transpose()</code> , except that <code>self</code> is returned if <code>self.ndim < 2</code> .
<code>ndarray.real</code>	The real part of the array.
<code>ndarray.imag</code>	The imaginary part of the array.
<code>ndarray.flat</code>	A 1-D iterator over the array.
<code>ndarray.ctypes</code>	An object to simplify the interaction of the array with the <code>ctypes</code> module.

Array Indexing/Slicing

[ndarrays](#) can be indexed using the standard Python `x[obj]` syntax, where *x* is the array and *obj* the selection.

There are three kinds of indexing available:

- field access,
- basic slicing,
- advanced indexing

Syntax:

`A[start : stop: step]`

Array Slicing - Example

Refer EXAMPLES

Iterating over arrays

The iterator object [nditer](#), introduced in NumPy 1.6, provides many **flexible ways to visit all the elements of one or more arrays** in a systematic fashion.

- An important thing to be aware of for this iteration is that the **order is chosen to match the memory layout of the array** instead of using a standard C or Fortran ordering.

Example:

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a):
...     print(x, end=' ')
...
0 1 2 3 4 5
```

Iterator - Example

Refer EXAMPLES

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]
```

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
```



```
>>> 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

Problem: Get the below elements

```
array([[20, 22, 24],  
       [40, 42, 44]])
```

Solution:

```
>>> a[2::2,::2]
```

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

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)
```

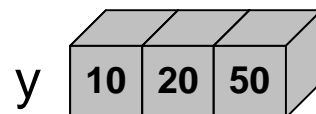
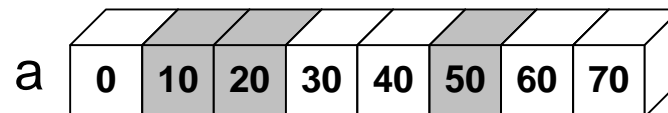
```
[10,20,50]
```

```
# 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.

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) #column-wise  
array([5., 7., 9.]
```

```
# supply the keyword axis to  
# sum along the last axis.
```

```
>>> sum(a, axis=-1) # Here it is row  
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])
```

Summary of (most) array

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

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 <-> 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

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

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)



Universal Functions

Universal Functions

- A universal function (or [ufunc](#)) is a function that operates on [ndarrays](#) in an element-by-element fashion,
- Supports [array broadcasting](#), [type casting](#), and several other standard features.
- **ufunc** is a “**vectorized**” **wrapper** for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.
- Ufuncs are instances of the **numpy.ufunc class**.
- Many of the built-in functions are implemented in compiled C code.
- The basic ufuncs operate on scalars, but there is also a generalized kind for which the basic elements are sub-arrays (vectors, matrices, etc.), and broadcasting is done over other dimensions.

Universal Functions - Methods

- All ufuncs have **five methods**.
- However, these methods only make sense on scalar ufuncs that take two input arguments and return one output argument.
- Attempting to call these methods on other ufuncs will cause a [ValueError](#).

<code>ufunc.reduce(a[, axis, dtype, out, keepdims])</code>	Reduces <i>a</i> 's dimension by one, by applying ufunc along one axis.
<code>ufunc.accumulate(array[, axis, dtype, out])</code>	Accumulate the result of applying the operator to all elements.
<code>ufunc.reduceat(a, indices[, axis, dtype, out])</code>	Performs a (local) reduce with specified slices over a single axis.
<code>ufunc.outer(A, B, **kwargs)</code>	Apply the ufunc <i>op</i> to all pairs (a, b) with a in <i>A</i> and b in <i>B</i> .
<code>ufunc.at(a, indices[, b])</code>	Performs unbuffered in place operation on operand 'a' for elements specified by 'indices'.

Universal Functions – > 60 ufuncs

Math operations

add (x1, x2, /[, out, where, casting, order, ...])	Add arguments element-wise.
subtract (x1, x2, /[, out, where, casting, ...])	Subtract arguments, element-wise.
multiply (x1, x2, /[, out, where, casting, ...])	Multiply arguments element-wise.
divide (x1, x2, /[, out, where, casting, ...])	Returns a true division of the inputs, element-wise.
logaddexp (x1, x2, /[, out, where, casting, ...])	Logarithm of the sum of exponentiations of the inputs.
logaddexp2 (x1, x2, /[, out, where, casting, ...])	Logarithm of the sum of exponentiations of the inputs in base-2.
true_divide (x1, x2, /[, out, where, ...])	Returns a true division of the inputs, element-wise.
floor_divide (x1, x2, /[, out, where, ...])	Return the largest integer smaller or equal to the division of the inputs.
negative (x, /[, out, where, casting, order, ...])	Numerical negative, element-wise.
positive (x, /[, out, where, casting, order, ...])	Numerical positive, element-wise.
power (x1, x2, /[, out, where, casting, ...])	First array elements raised to

Universal Functions – > 60 ufuncs

<code>remainder(x1, x2, /[, out, where, casting, ...])</code>	Return element-wise remainder of division.
<code>mod(x1, x2, /[, out, where, casting, order, ...])</code>	Return element-wise remainder of division.
<code>fmod(x1, x2, /[, out, where, casting, ...])</code>	Return the element-wise remainder of division.
<code>divmod(x1, x2[, out1, out2], / [[, out, ...])</code>	Return element-wise quotient and remainder simultaneously.
<code>absolute(x, /[, out, where, casting, order, ...])</code>	Calculate the absolute value element-wise.
<code>fabs(x, /[, out, where, casting, order, ...])</code>	Compute the absolute values element-wise.
<code>rint(x, /[, out, where, casting, order, ...])</code>	Round elements of the array to the nearest integer.
<code>sign(x, /[, out, where, casting, order, ...])</code>	Returns an element-wise indication of the sign of a number.
<code>heaviside(x1, x2, /[, out, where, casting, ...])</code>	Compute the Heaviside step function.
<code>conj(x, /[, out, where, casting, order, ...])</code>	Return the complex conjugate, element-wise.
<code>exp(x, /[, out, where, casting, order, ...])</code>	Calculate the exponential of all elements in the input array

Universal Functions – > 60 ufuncs

`log(x, /[, out, where, casting, order, ...])`

Natural logarithm, element-wise.

`log2(x, /[, out, where, casting, order, ...])`

Base-2 logarithm of x .

`log10(x, /[, out, where, casting, order, ...])`

Return the base 10 logarithm of the input array, element-wise.

`expm1(x, /[, out, where, casting, order, ...])`

Calculate $\exp(x) - 1$ for all elements in the array.

`log1p(x, /[, out, where, casting, order, ...])`

Return the natural logarithm of one plus the input array, element-wise.

`sqrt(x, /[, out, where, casting, order, ...])`

Return the non-negative square-root of an array, element-wise.

`square(x, /[, out, where, casting, order, ...])`

Return the element-wise square of the input.

`cbrt(x, /[, out, where, casting, order, ...])`

Return the cube-root of an array, element-wise.

`reciprocal(x, /[, out, where, casting, ...])`

Return the reciprocal of the argument, element-wise.

`gcd(x1, x2, /[, out, where, casting, order, ...])`

Returns the greatest common divisor of $|x1|$ and $|x2|$

`lcm(x1, x2, /[, out, where, casting, order, ...])`

Returns the lowest common multiple of $|x1|$ and $|x2|$

Universal Functions – > 60 ufuncs

Trigonometric functions

All trigonometric functions use radians when an angle is called for. The ratio of degrees to radians is $180^\circ/\pi$.

sin(x, /[, out, where, casting, order, ...])

Trigonometric sine, element-wise.

cos(x, /[, out, where, casting, order, ...])

Cosine element-wise.

tan(x, /[, out, where, casting, order, ...])

Compute tangent element-wise.

arcsin(x, /[, out, where, casting, order, ...])

Inverse sine, element-wise.

arccos(x, /[, out, where, casting, order, ...])

Trigonometric inverse cosine, element-wise.

arctan(x, /[, out, where, casting, order, ...])

Trigonometric inverse tangent, element-wise.

arctan2(x1, x2, /[, out, where, casting, ...])

Element-wise arc tangent of $x1/x2$ choosing the quadrant correctly.

hypot(x1, x2, /[, out, where, casting, ...])

Given the “legs” of a right triangle, return its hypotenuse.

sinh(x, /[, out, where, casting, order, ...])

Hyperbolic sine, element-wise.

cosh(x, /[, out, where, casting, order, ...])

Hyperbolic cosine, element-wise.

tanh(x, /[, out, where, casting, order, ...])

Compute hyperbolic tangent

Universal Functions – > 60 ufuncs

Bit-twiddling functions

These functions all require integer arguments and they manipulate the bit-pattern of those arguments.

<code>bitwise_and(x1, x2, /[, out, where, ...])</code>	Compute the bit-wise AND of two arrays element-wise.
<code>bitwise_or(x1, x2, /[, out, where, casting, ...])</code>	Compute the bit-wise OR of two arrays element-wise.
<code>bitwise_xor(x1, x2, /[, out, where, ...])</code>	Compute the bit-wise XOR of two arrays element-wise.
<code>invert(x, /[, out, where, casting, order, ...])</code>	Compute bit-wise inversion, or bit-wise NOT, element-wise.
<code>left_shift(x1, x2, /[, out, where, casting, ...])</code>	Shift the bits of an integer to the left.
<code>right_shift(x1, x2, /[, out, where, ...])</code>	Shift the bits of an integer to the right.

Universal Functions – > 60 ufuncs

Comparison functions

<code>greater(x1, x2, /[, out, where, casting, ...])</code>	Return the truth value of $(x1 > x2)$ element-wise.
<code>greater_equal(x1, x2, /[, out, where, ...])</code>	Return the truth value of $(x1 \geq x2)$ element-wise.
<code>less(x1, x2, /[, out, where, casting, ...])</code>	Return the truth value of $(x1 < x2)$ element-wise.
<code>less_equal(x1, x2, /[, out, where, casting, ...])</code>	Return the truth value of $(x1 \leq x2)$ element-wise.
<code>not_equal(x1, x2, /[, out, where, casting, ...])</code>	Return $(x1 \neq x2)$ element-wise.
<code>equal(x1, x2, /[, out, where, casting, ...])</code>	Return $(x1 == x2)$ element-wise.

Mathematic Binary Operators

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

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])
```

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

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

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

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]])
```


Bitwise Operators

bitwise_and	(&)	invert	(~)	right_shift(a,shifts)
bitwise_or	()	bitwise_xor		left_shift (a,shifts)

BITWISE EXAMPLES

```
>>> a = array((1,2,4,8))
>>> b = array((16,32,64,128))
>>> bitwise_or(a,b)
array([ 17, 34, 68, 136])
```

bit inversion

```
>>> a = array((1,2,3,4), uint8)
>>> invert(a)
array([254, 253, 252, 251], dtype=uint8)
```

left shift operation

```
>>> left_shift(a,3)
array([ 8, 16, 24, 32], dtype=uint8)
```

Trig and Other Functions

TRIGONOMETRIC

<code>sin(x)</code>	<code>sinh(x)</code>
<code>cos(x)</code>	<code>cosh(x)</code>
<code>arccos(x)</code>	<code>arccosh(x)</code>
<code>arctan(x)</code>	<code>arctanh(x)</code>
<code>arcsin(x)</code>	<code>arcsinh(x)</code>
<code>arctan2(x,y)</code>	

OTHERS

<code>exp(x)</code>	<code>log(x)</code>
<code>log10(x)</code>	<code>sqrt(x)</code>
<code>absolute(x)</code>	<code>conjugate(x)</code>
<code>negative(x)</code>	<code>ceil(x)</code>
<code>floor(x)</code>	<code>fabs(x)</code>
<code>hypot(x,y)</code>	<code>fmod(x,y)</code>
<code>maximum(x,y)</code>	<code>minimum(x,y)</code>

`hypot(x,y)`

Element by element distance
calculation using $\sqrt{x^2 + y^2}$

Broadcasting using ufuncs

- **broadcasting** refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations.
- Arithmetic operations on arrays are usually done on corresponding elements.
- If two arrays are of exactly the same shape, then these operations are smoothly performed.
- Broadcasting is used throughout NumPy to decide how to handle disparately shaped arrays

Broadcasting using ufuncs...

A set of arrays is called “broadcastable” to the same shape if they follow any of the following rules:

- The arrays all have exactly the same shape.
- The arrays all have the same number of dimensions and the length of each dimensions is either a common length or 1.
- The arrays that have too few dimensions can have their shapes prepended with a dimension of length 1 to satisfy property 2.

Example:

If `a.shape` is (5,1), `b.shape` is (1,6), `c.shape` is (6,) and `d.shape` is () so that *d* is a scalar, then *a*, *b*, *c*, and *d* are all broadcastable to dimension (5,6); and

- *a* acts like a (5,6) array where `a[:,0]` is broadcast to the other columns,
- *b* acts like a (5,6) array where `b[0,:]` is broadcast to the other rows,
- *c* acts like a (1,6) array and therefore like a (5,6) array where `c[:]` is broadcast to every row, and finally,
- *d* acts like a (5,6) array where the single value is repeated.

Broadcasting - Examples

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.

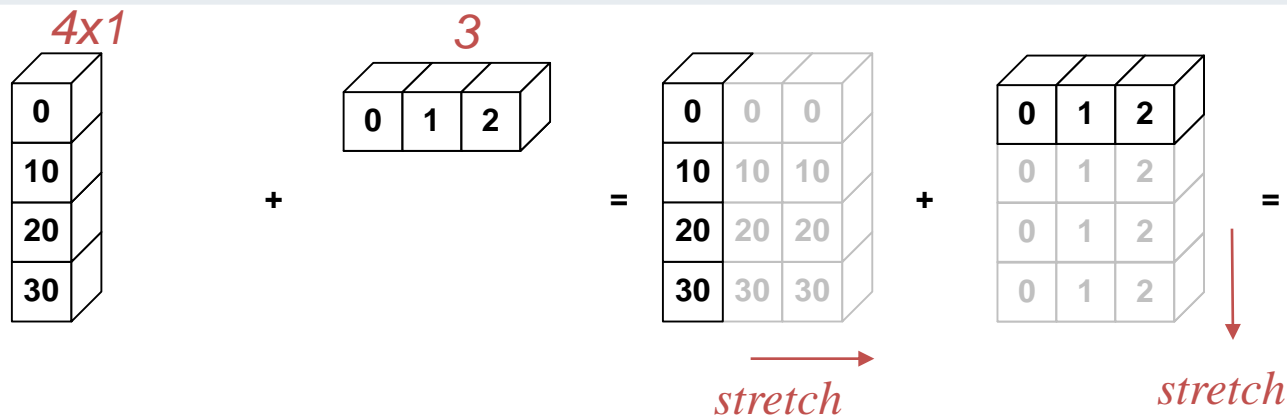
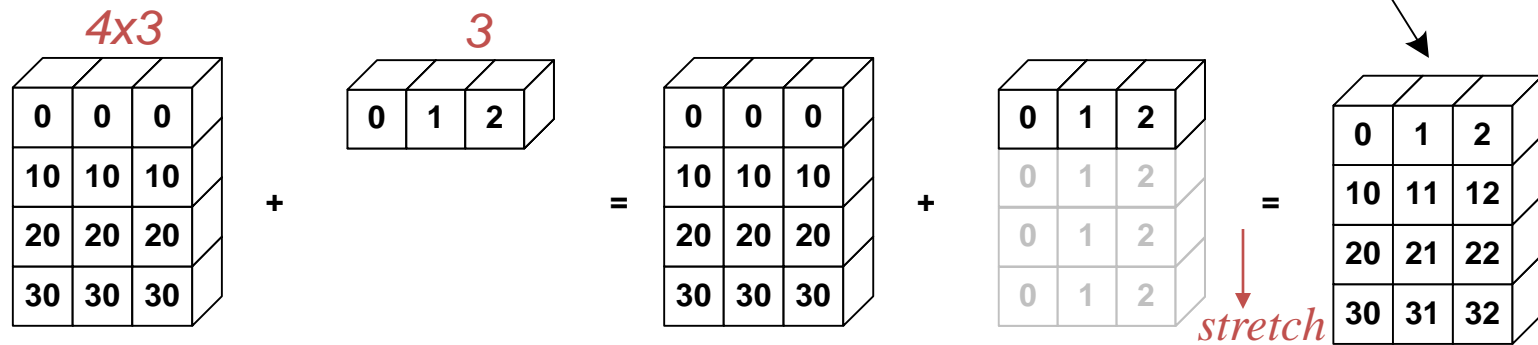
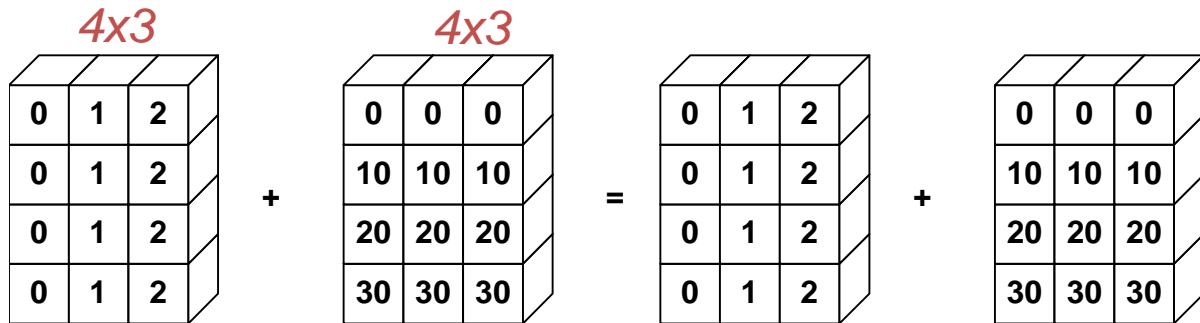
```
>>> x = [1,2,3,4];
>>> y = [[10],[20],[30]]
>>> print N.add(x,y)
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
>>> x = array(x)
>>> y = array(y)
>>> print x+y
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

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

y has shape (3,1)

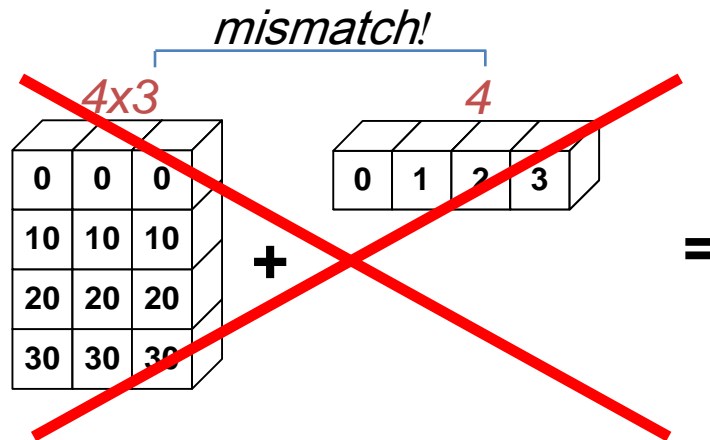
The ufunc result has shape
(3,4)

Array Broadcasting



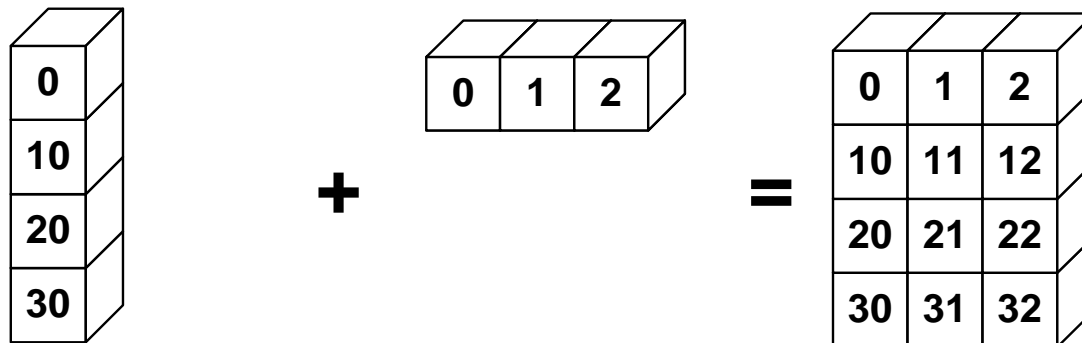
Broadcasting Rules

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
```



Summary

1. Introduction
2. NumPy Constants
3. NDArray (Array Objects)
4. Universal Functions
5. Broadcasting
6. Structured Data Types
7. Examples

Thank You

