Numpy

SciCoder 2017 • Vanderbilt

Demitri Muna & Adrian Price-Whelan

Numpy

- Python supports lists of objects (as we've seen), but this is extremely inefficient for numerical operations.
- Mathematical operations are fastest when numbers are contiguously allocated in memory and homogenous.
- Numpy is designed to make numeric arrays a basic data type and optimize their performance (e.g. vectorization).

To use Numpy objects, you need to import the module:

```
import numpy
```

Though typically people shorten the package name (namespace) like this:

```
import numpy as np
```

Data Types

A numpy array can only contain one type of data; below is a list of some of them. Typically you should use the smallest data type as appropriate to your data.

Type	Range
np.int_	same as np.int32 or np.int64
np.int8	-128-127
np.int16	-32768 to 32767
np.int32	-2147483648 to 2147483647
np.float_	same as np.float64
np.float32	single precision float
np.float64	double precision float
np.str_	string type
np.object_	any Python object

See this for the full list and sizes:
http://docs.scipy.org/doc/numpy/user/basics.types.html
http://docs.scipy.org/doc/numpy/reference/arrays.scalars.html#arrays-scalars-built-in

Creating Numpy Arrays

New 10 element array filled with zeros or ones (the default type is “float”):

```
n1 = np.zeros(10, dtype=np.int)
n2 = np.zeros(10, dtype=np.float)
```

The default data type is “float”. A new empty array:

```
n3 = np.empty() # typically you don't use this
```

Create an array from a Python list of numbers:

```
a_list = [1,2,3,4,5]
n4 = np.array(a_list)
```

The *shape* is the dimensionality of the array (1D, 2D, etc.) – row x columns.
Create a 4x5 2D array:

shape is a tuple

```
n5 = np.array((4,5), dtype=np.float)
```

Can use generators:

```
n6 = np.array(range(4)) # array([0, 1, 2, 3])
n7 = np.array([x for x in range(10) if x % 2 == 0]) # array([0, 2, 4, 6, 8])
```

Creating Numpy Arrays

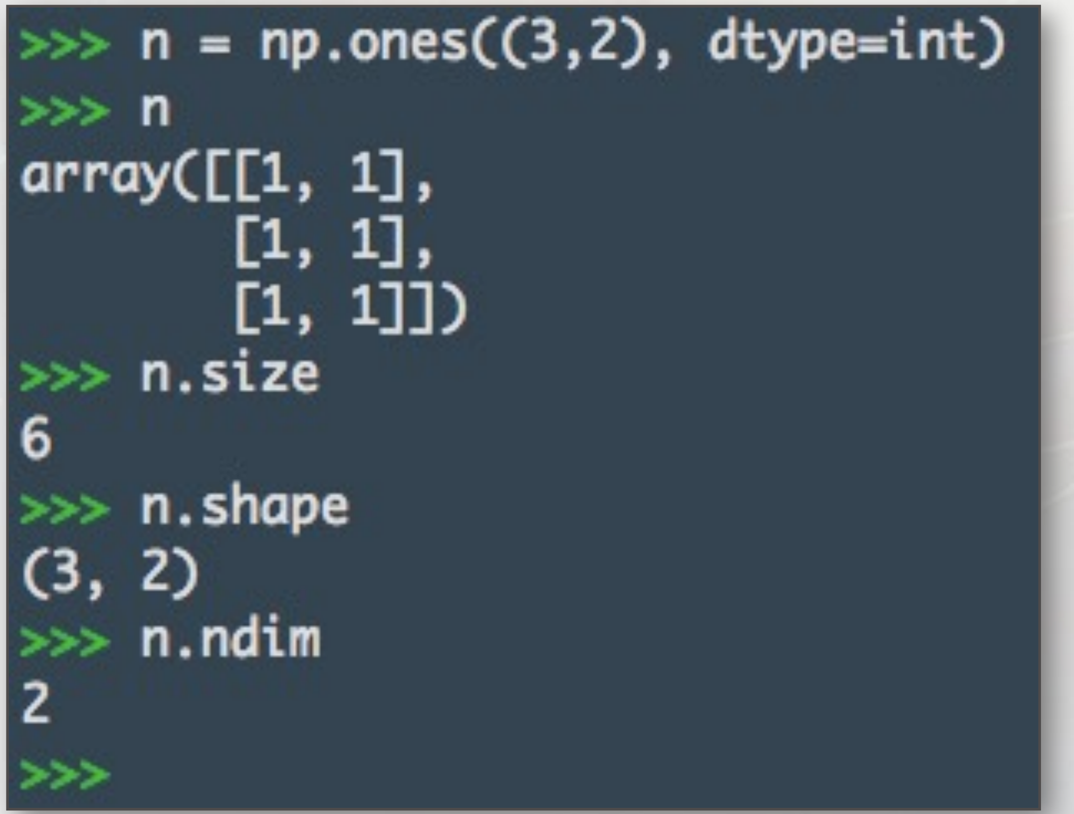
More ways to create an array:

```
>>> np.arange(start=10, stop=20, step=2)
array([10, 12, 14, 16, 18])

>>> np.linspace(start=0, stop=1, num=5) # num = number of elements
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])

>>> np.eye(3, dtype=np.int)
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

Array Properties

Numpy arrays are proper objects– they know how to do things and are introspective.

(but not emo)



```
>>> n = np.ones((3,2), dtype=int)
>>> n
array([[1, 1],
       [1, 1],
       [1, 1]])
>>> n.size
6
>>> n.shape
(3, 2)
>>> n.ndim
2
>>>
```

Other useful methods:

```
n.max()
n.min()
n.mean()
n.sort() # sort array in place (returns None)
```

Functions

```
np.sum(n)
np.mean(n)
np.median(n)
```

Operations can be chained (aka *generative*)

```
s = np.array([1,2,3,4]).min()
```

Make an array read-only (protection against accidentally changing fixed values):

```
n.flags.writeable = False
```

Appending Items to a Numpy Array

In short, don't.

Creating an array is "expensive". Arrays must be contiguous – appending a value to an array actually creates a new one. If you can, initialize your array, then fill in the values.

```
from __future__ import print_function

import numpy as np
import time

n = 10000
start = time.time()

a = np.empty(0)
for i in range(10000):
    a = np.append(a, i)

print("Time 1: {0:.4f} " +
     "seconds".format(time.time() - start))

start = time.time()
a = np.zeros(10000)
for i in range(10000):
    a[i] = i

print("Time 2: {0:.4f} " +
    "seconds".format(time.time() - start))

# Time 1: 0.0687 seconds
# Time 2: 0.0015 seconds
```

10,000 objects created

1 object created

Operating on Arrays

Array operations work on a per-element basis. This is different than Python lists.

```
>>> [1,2,3,4] * 5
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> np.array([1,2,3,4]) * 10
array([10, 20, 30, 40])

>>> [1,2,3,4] + [5,6,7,8]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> np.array([1,2,3,4]) + np.array([5,6,7,8])
array([ 6,  8, 10, 12])
```

Working with arrays is much faster than Python lists.

```
>>> a = [x**2 for x in range(10000)]
>>> a = np.arange(10000)**2
```

Numpy is 70x faster here!

Reshaping An Array

An array’s dimensions can be changed (reshaped). For example, let’s say I want to change a 1x9 array to a 3x3 matrix:

```
>>> a = np.array([0,1,2,3,4,5,6,7,8])
>>> b = a.reshape(3,3)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> b
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> print(a.shape, b.shape)
((9,), (3, 3))
```

Array Indexing

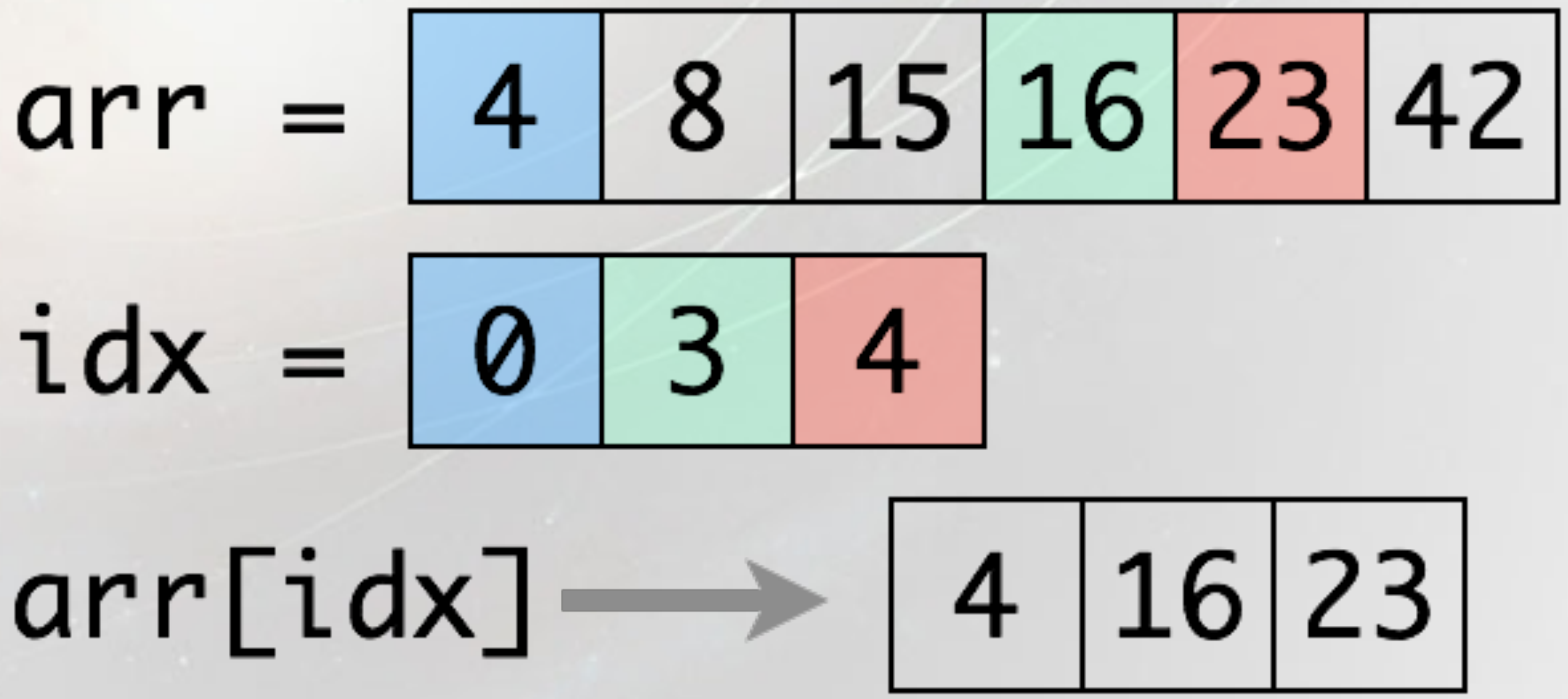
Array indexing works in the same way that Python lists do. Indexing can be performed on multiple dimensions:

```
>>> a = np.array(range(9)).reshape((3,3))
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> a[1:2]
array([[3, 4, 5]])
>>> a[2,1:2]
array([7])
```

You can also use an array as an index list for another array. For example, let's say we have the following array, and we want to get the 0th, 3rd, and 4th items.

Array Indexing

You can also use an array as an index list for another array. For example, let's say we have the following array, and we want to get the 0th, 3rd, and 4th items.



can be a Python list or a Numpy array

```
>>> a = np.array([4,8,15,16,23,42])
>>> a[[0,3,4]]
array([ 4, 16, 23])
>>> a[np.array([0,3,4])]
array([ 4, 16, 23])
>>> a[np.array([True, False, False, True, True, False])]
array([ 4, 16, 23])
```

Boolean values work too, but must be a Numpy Boolean array

Selecting Elements with Some Criteria

You can select elements that match some criteria (similar to "where" in IDL):

```
>>> a = np.array([1,2,3,5,8,13])
>>> a
array([ 1,  2,  3,  5,  8, 13])
>>> a > 3
array([False, False, False,  True,  True,  True], dtype=bool)
```

result is a Boolean array

Expressions can be combined:

must use "|" for OR and "&" for —
"or", "and" don't work

```
>>> (a > 3) | (a == 1)
array([ True, False, False,  True,  True,  True], dtype=bool)
>>> (a > 2) & (a < 10)
array([False, False,  True,  True,  True, False], dtype=bool)
```

The inverse operator:

```
>>> ~np.array([True, True, False])
array([False, False,  True], dtype=bool)
```

Selecting Elements with Some Criteria

Example: selecting elements from a catalog:

```
idx = (ra > 11.1324) & (ra < 31.5134)
selected = ra[idx]
not_selected = ra[~idx]
```

Structured Arrays

Standard arrays have homogeneous data types (e.g. all integers, all floats). A structured array can contain different types of data – think of a table with each column having a different type.

Name	ID	Height	Active
"Mulder"	11605	6.0	False
"Scully"	42115	5.5	True

```
data = [("mulder", 11605, 6.0, False), ("scully", 42115, 5.5, True)]
arr = np.array(data, dtype=[("Name", str), ("ID", int),
                            ("Height", float), ("Active", bool)])
arr["ID"] -> [11605, 42115]
arr[0] -> ("mulder", 11605, 6.0, True)
```

access a column by name

access a row by the row number

Dimensionality Reduction

Aggregate functions "flatten" an array:

```
>>> a = np.array(range(9)).reshape((3,3))
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.sum(a)
36
```

These operations can act on a specified axis:

```
>>> np.sum(a, axis=0)
array([ 9, 12, 15])
>>> np.sum(a, axis=1)
array([ 3, 12, 21])
```