STAT606 Computing for Data Science and Statistics

Lecture 11: `numpy` and `scipy`

Some examples adapted from A. Tewari @ UMichigan

Numerical computing in Python: `numpy`

One of a few increasingly-popular, free competitors to MATLAB

Numpy quickstart guide: https://numpy.org/doc/stable/user/quickstart.html

For MATLAB fans:

https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

Closely related package `scipy` is for optimization

See https://docs.scipy.org/doc/

Installing packages

So far, we have only used built-in modules

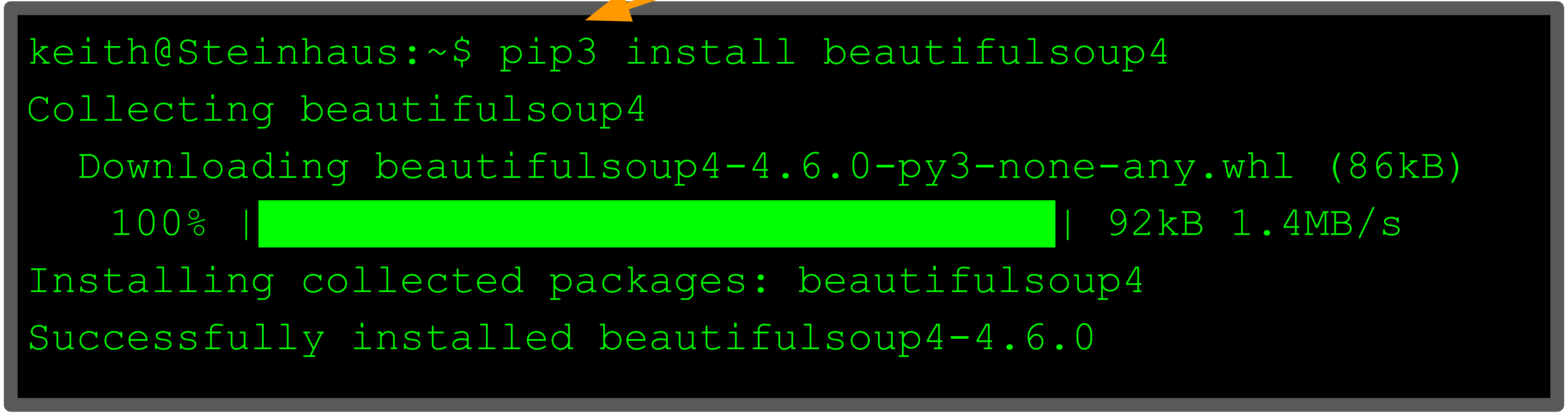
- But there are many modules/packages that do not come preinstalled

Ways to install packages:

- At the `conda` prompt or in terminal: `conda install numpy`
  - https://conda.io/docs/user-guide/tasks/manage-pkgs.html
- Using `pip` (recommended): `pip install numpy`
  - https://pip.pypa.io/en/stable/
- Using UNIX/Linux package manager (not recommended)
- From source (not recommended)

Installing packages with `pip`

If you have both Python 2 and Python 3 installed, make sure you specify which one you want to install in!



```
keith@Steinhaus:~$ pip3 install beautifulsoup4
Collecting beautifulsoup4
  Downloading beautifulsoup4-4.6.0-py3-none-any.whl (86kB)
    100% |████████████████████████████████| 92kB 1.4MB/s
Installing collected packages: beautifulsoup4
Successfully installed beautifulsoup4-4.6.0
```

The above command installs the package `beautifulsoup4`. We will use that later in the semester. To install `numpy`, type the same command, but use `numpy` in place of `beautifulsoup4`.

`numpy` data types

`import … as …` lets us import a package and give it a shorter name.

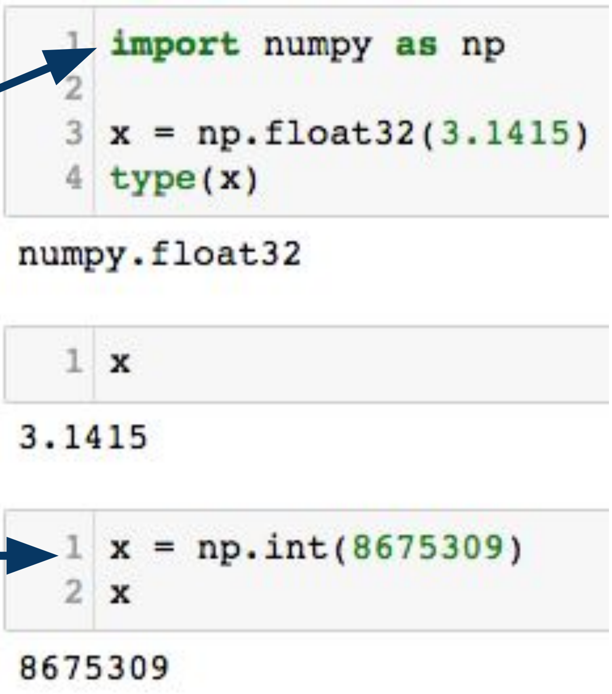
Five basic numerical data types:

- boolean (`bool`)
- integer (`int`)
- unsigned integer (`uint`)
- floating point (`float`)
- complex (`complex`)

Note that this is not the same as a Python int.

Many more complicated data types are available

- e.g., each of the numerical types can vary in how many bits it uses
- https://docs.scipy.org/doc/numpy/user/basics.types.html



```
import numpy as np

x = np.float32(3.1415)
type(x)
```

```
numpy.float32
```

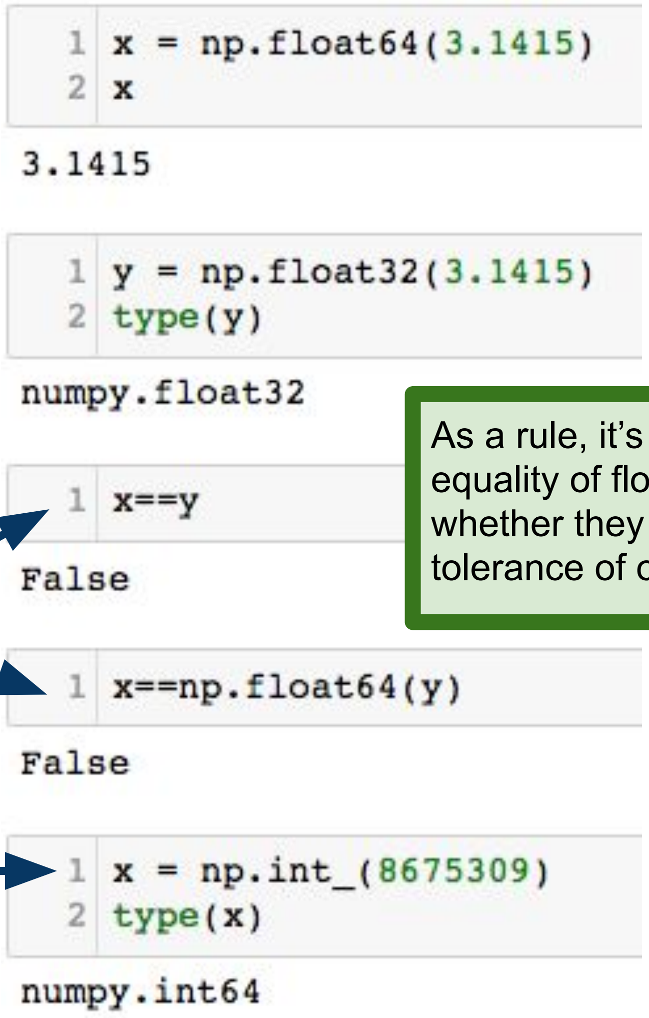
```
x
```

```
3.1415
```

```
x = np.int(8675309)
x
```

```
8675309
```

`numpy` data types



```
x = np.float64(3.1415)
x
```

```
3.1415
```

```
y = np.float32(3.1415)
type(y)
```

```
numpy.float32
```

```
x==y
```

```
False
```

```
x==np.float64(y)
```

```
False
```

```
x = np.int_(8675309)
type(x)
```

```
numpy.int64
```

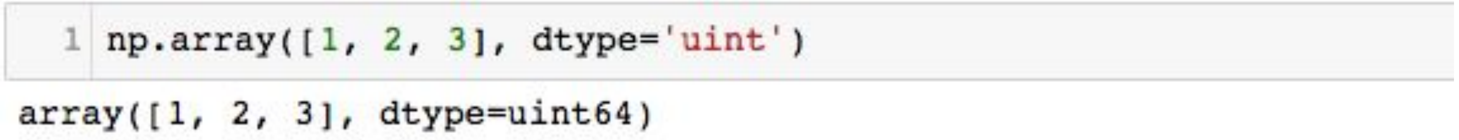
As a rule, it's best never to check for equality of floats. Instead, check whether they are within some error tolerance of one another.

32-bit and 64-bit representations are distinct!

Data type followed by underscore uses the default number of bits. This default varies by system.

`numpy.array`: `numpy`’s version of Python array (i.e., list)

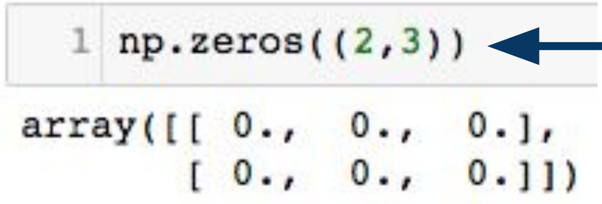
Can be created from a Python list…



```
np.array([1, 2, 3], dtype='uint')
```

```
array([1, 2, 3], dtype=uint64)
```

...by “shaping” an array…

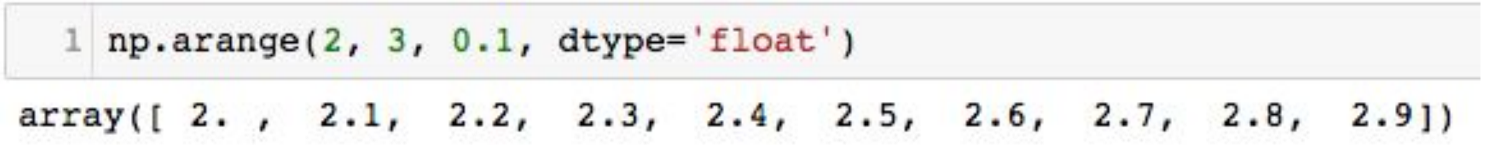


```
np.zeros((2,3))
```

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

`np.zeros` and `np.ones` generate arrays of 0s or 1s, respectively. Shape parameter (2,3) means to create a 2-D array with two rows and three columns.

...by “ranges”...



```
np.arange(2, 3, 0.1, dtype='float')
```

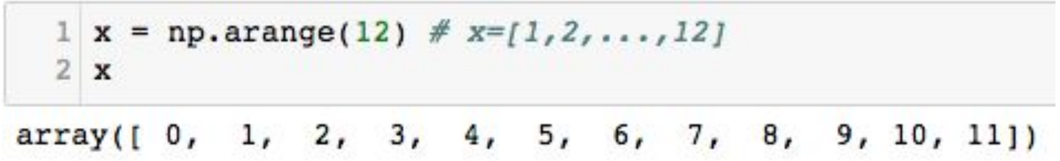
```
array([ 2. ,  2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9])
```

...or reading directly from a file

see https://docs.scipy.org/doc/numpy/user/basics.creation.html

`numpy` allows arrays of arbitrary dimension (tensors)

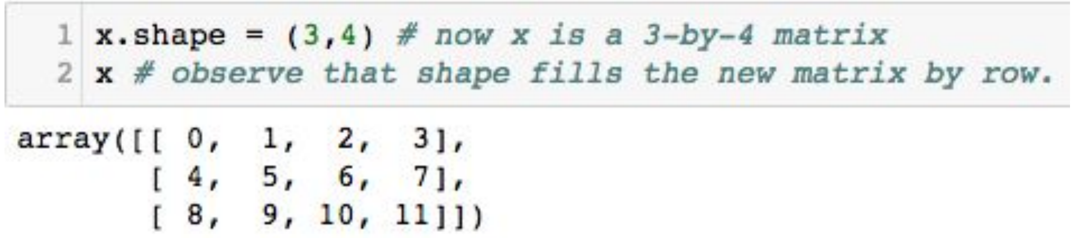
1-dimensional arrays:



```
x = np.arange(12) # x=[1,2,...,12]
x
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

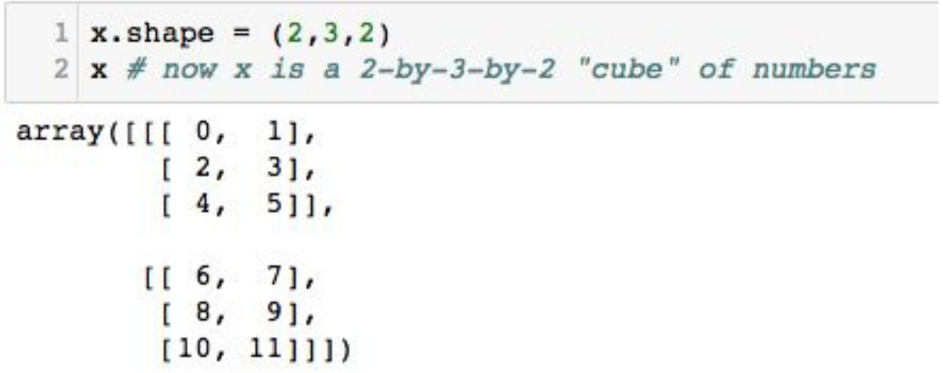
2-dimensional arrays (matrices):



```
x.shape = (3,4) # now x is a 3-by-4 matrix
x # observe that shape fills the new matrix by row.
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

3-dimensional arrays (“3-tensor”):



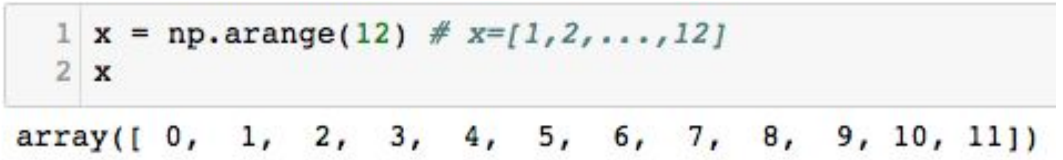
```
x.shape = (2,3,2)
x # now x is a 2-by-3-by-2 "cube" of numbers
```

```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
```

`numpy` allows arrays of arbitrary dimension (tensors)

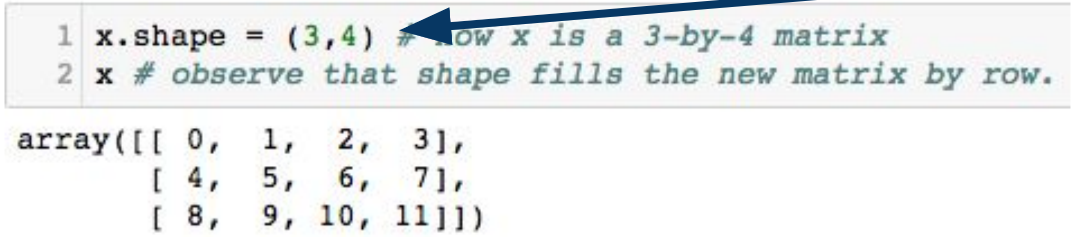
1-dimensional arrays:



```
x = np.arange(12) # x=[1,2,...,12]
x
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

2-dimensional arrays (matrices):

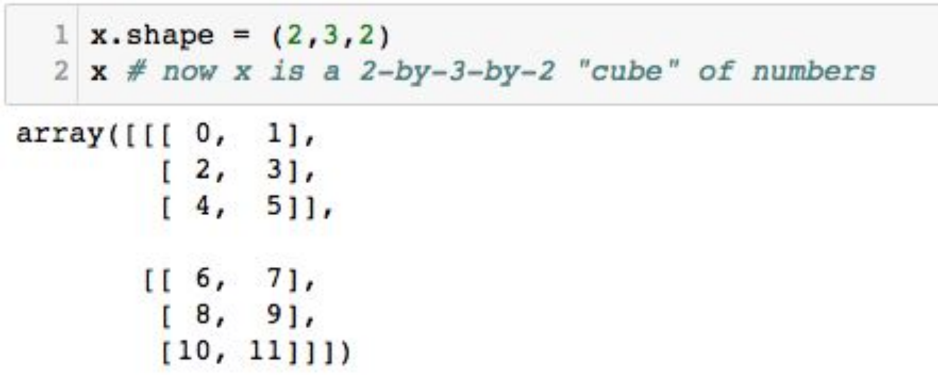


```
x.shape = (3,4) # now x is a 3-by-4 matrix
x # observe that shape fills the new matrix by row.
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Every numpy array has a `shape` attribute specifying its dimensions. For example, an array with shape (3,4) has three rows and four columns. An array with shape (2,3,2) is a 2-by-3-by-2 “box” of numbers.

3-dimensional arrays (“3-tensor”):



```
x.shape = (2,3,2)
x # now x is a 2-by-3-by-2 "cube" of numbers
```

```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
```

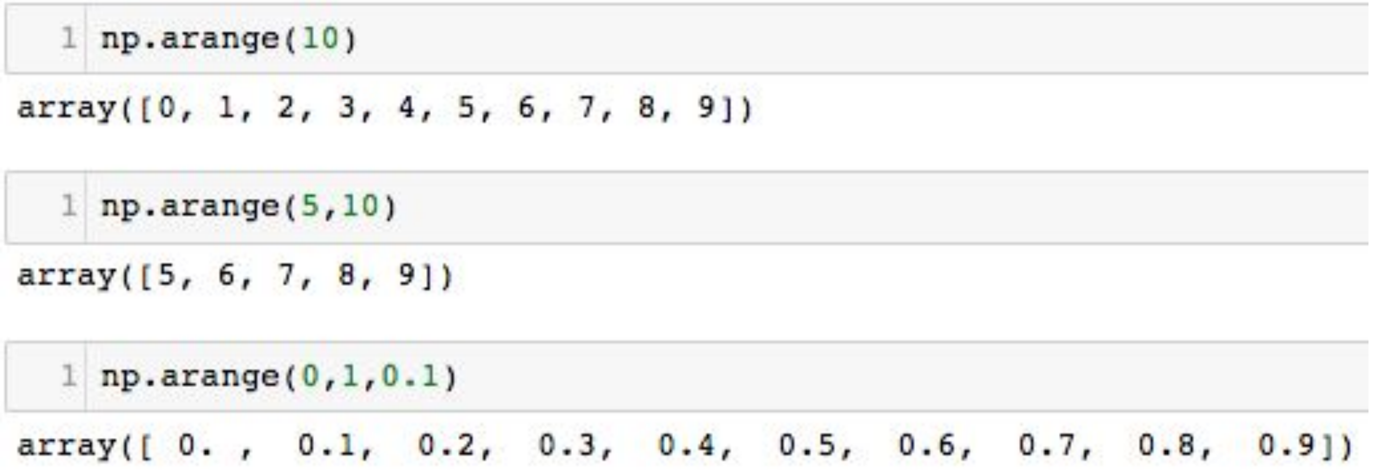
Think of the shape of an array as specifying how many indices we need to pick out an entry of the array. For example, to pick out a number from a 3-by-4 matrix, we must specify a row and a column.

More on `numpy.arange` creation

`np.arange(x)`: array version of Python's `range(x)`, like `[0,1,2,...,x-1]`

`np.arange(x,y)`: array version of `range(x,y)`, like `[x,x+1,...,y-1]`

`np.arange(x,y,z)`: array of elements `[x,y)` in `z`-size increments.



```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange(5,10)
```

```
array([5, 6, 7, 8, 9])
```

```
np.arange(0,1,0.1)
```

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

More on `numpy.arange` creation

`np.arange(x)`: array version of Python's `range(x)`, like `[0,1,2,...,x-1]`

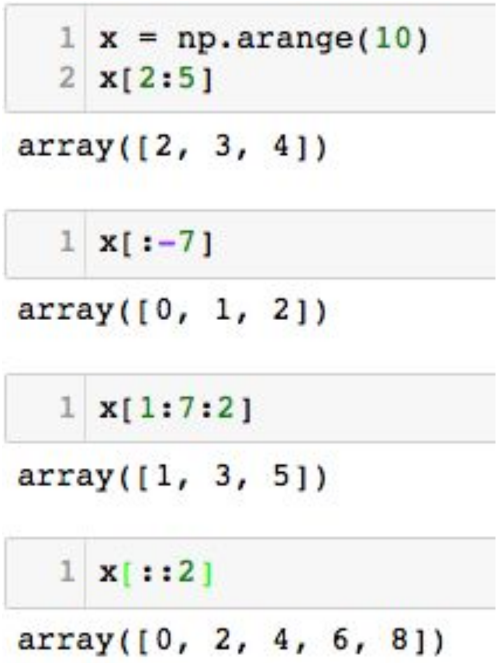
`np.arange(x,y)`: array version of `range(x,y)`, like `[x,x+1,...,y-1]`

`np.arange(x,y,z)`: array of elements `[x,y)` in `z`-size increments.

Related useful functions, that give better/clearer control of start/endpoints and allow for multidimensional arrays:

- https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.ogrid.html
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.mgrid.html

`numpy` array indexing is highly expressive



```
x = np.arange(10)
x[2:5]
```

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

```
x[:-7]
```

```
array([0, 1, 2])
```

```
x[1:7:2]
```

```
array([1, 3, 5])
```

```
x[::2]
```

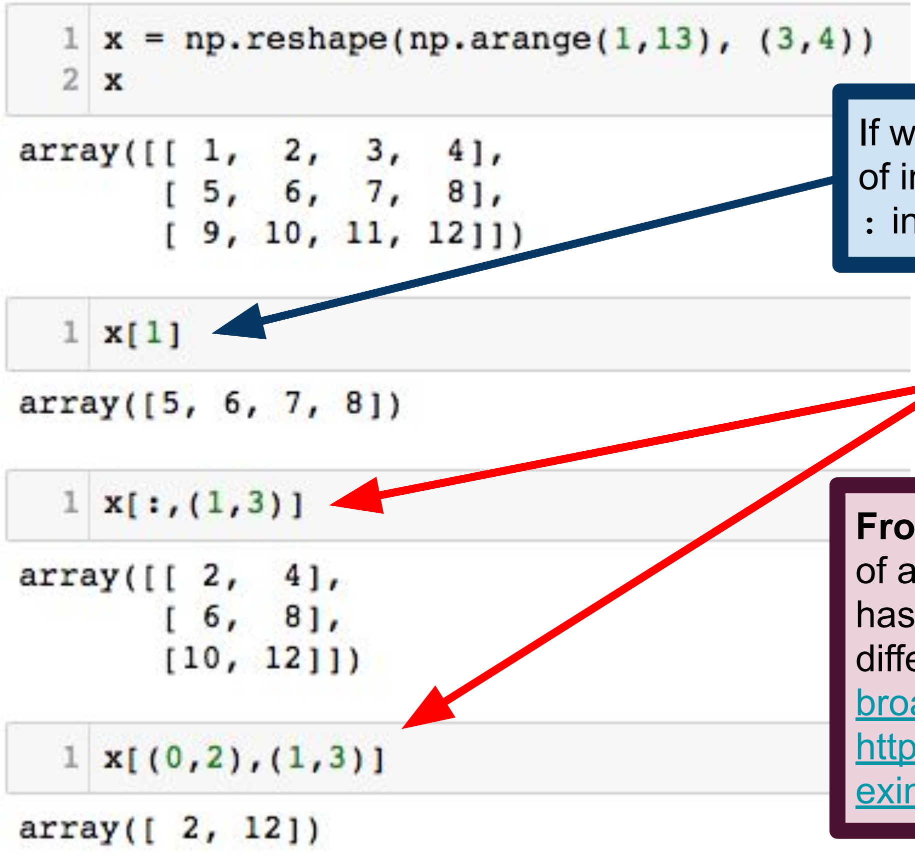
```
array([0, 2, 4, 6, 8])
```

Slices, strides, indexing from the end, etc.
Just like with Python lists.

Not very relevant to us right now…

...but this will come up again in a few weeks when we cover TensorFlow

More array indexing



```
x = np.reshape(np.arange(1,13), (3,4))
x
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```
x[1]
```

```
array([5, 6, 7, 8])
```

```
x[:,(1,3)]
```

```
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
```

```
x[(0,2),(1,3)]
```

```
array([ 2, 12])
```

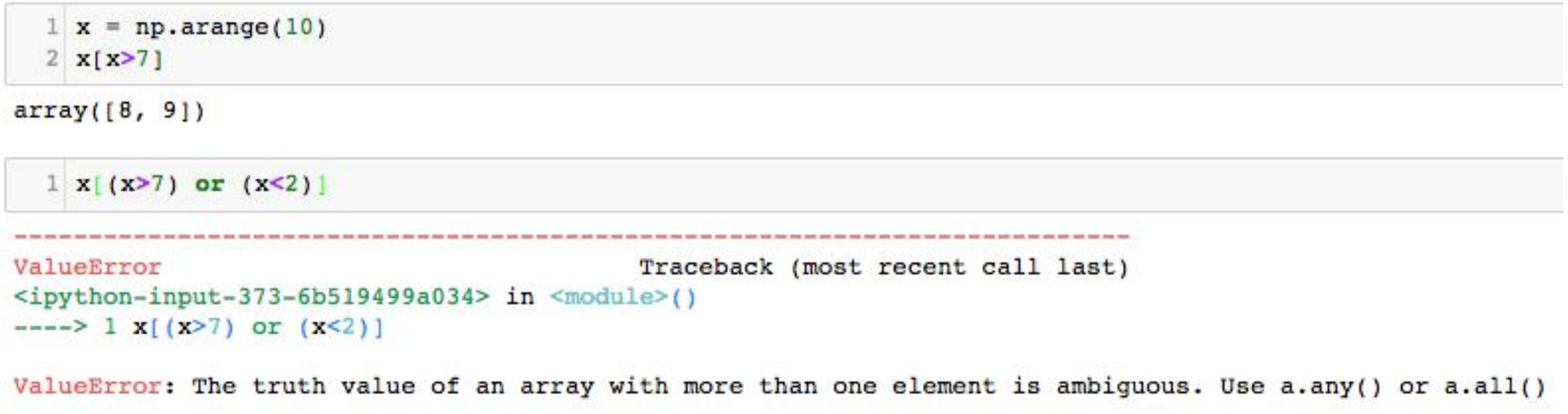
If we specify fewer than the number of indices, `numpy` assumes we mean `:` in the remaining indices.

Warning: if you're used to MATLAB or R, this behavior will seem weird to you.

From the documentation: When the index consists of as many integer arrays as the array being indexed has dimensions, the indexing is straight forward, but different from slicing. Advanced indexes always are broadcast and iterated as *one*.
https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#integer-array-indexing

More array indexing

Numpy allows MATLAB/R-like indexing by Booleans



```
x = np.arange(10)
x[x>7]
```

```
array([8, 9])
```

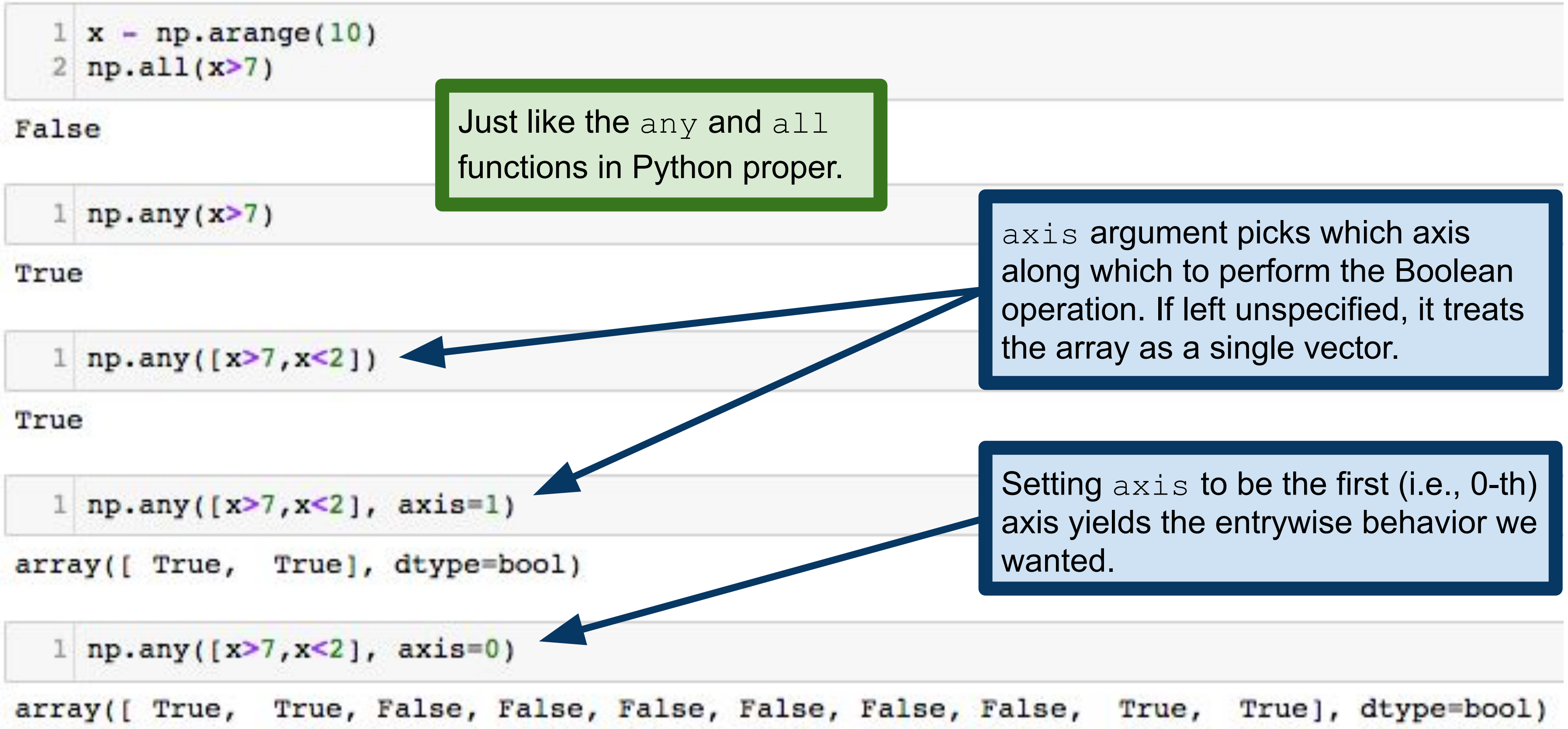
```
x[(x>7) or (x<2)]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-373-6b519499a034> in <module>()
----> 1 x[(x>7) or (x<2)]

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

Believe it or not, this error is by design! The designers of `numpy` were concerned about ambiguities in Boolean vector operations. In essence, should `(x>7) or (x<2)` be a vector of Booleans or a single Boolean?

Boolean operations: `np.any()`, `np.all()`



```
x = np.arange(10)
np.all(x>7)
```

```
False
```

Just like the `any` and `all` functions in Python proper.

```
np.any(x>7)
```

```
True
```

`axis` argument picks which axis along which to perform the Boolean operation. If left unspecified, it treats the array as a single vector.

```
np.any([x>7,x<2])
```

```
True
```

```
np.any([x>7,x<2], axis=1)
```

```
array([ True,  True], dtype=bool)
```

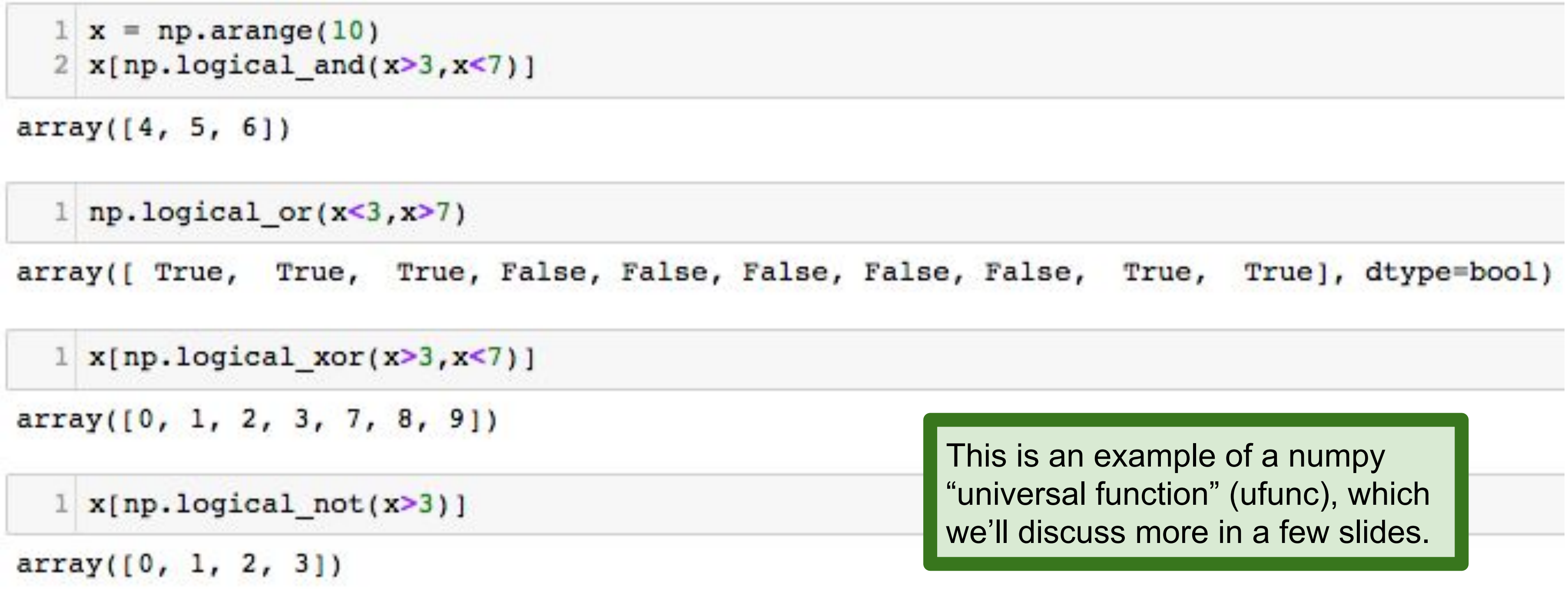
Setting `axis` to be the first (i.e., 0-th) axis yields the entrywise behavior we wanted.

```
np.any([x>7,x<2], axis=0)
```

```
array([ True,  True, False, False, False, False, False, False,  True,  True], dtype=bool)
```

Boolean operations: `np.logical_and()`

`numpy` also has built-in Boolean vector operations, which are simpler/clearer at the cost of the expressiveness of `np.any(), np.all()`.



```
x = np.arange(10)
x[np.logical_and(x>3,x<7)]
```

```
array([4, 5, 6])
```

```
np.logical_or(x<3,x>7)
```

```
array([ True,  True,  True, False, False, False, False, False,  True,  True], dtype=bool)
```

```
x[np.logical_xor(x>3,x<7)]
```

```
array([0, 1, 2, 3, 7, 8, 9])
```

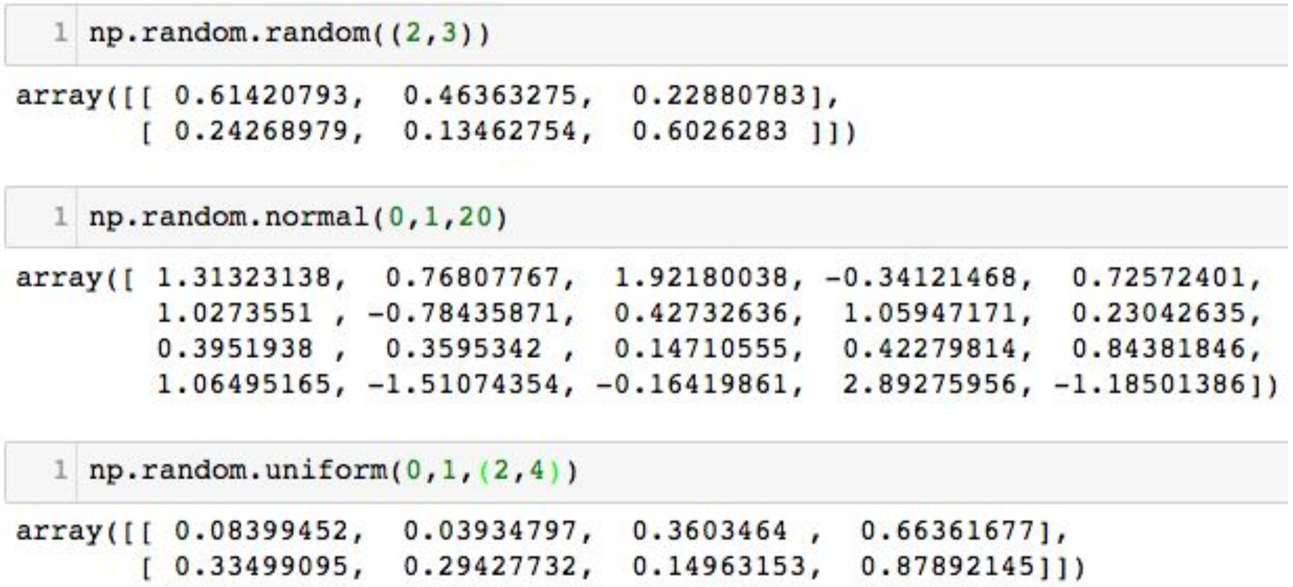
```
x[np.logical_not(x>3)]
```

```
array([0, 1, 2, 3])
```

This is an example of a numpy "universal function" (ufunc), which we'll discuss more in a few slides.

Random numbers in `numpy`

`np.random` contains methods for generating random numbers



```
np.random.random((2,3))
```

```
array([[ 0.61420793,  0.46363275,  0.22880783],
       [ 0.24268979,  0.13462754,  0.6026283 ]])
```

```
np.random.normal(0,1,20)
```

```
array([ 1.31323138,  0.76807767,  1.92180038, -0.34121468,  0.72572401,
        1.0273551 , -0.78435871,  0.42732636,  1.05947171,  0.23042635,
        0.3951938 ,  0.3595342 ,  0.14710555,  0.42279814,  0.84381846,
        1.06495165, -1.51074354, -0.16419861,  2.89275956, -1.18501386])
```

```
np.random.uniform(0,1,(2,4))
```

```
array([[ 0.08399452,  0.03934797,  0.3603464 ,  0.66361677],
       [ 0.33499095,  0.29427732,  0.14963153,  0.87892145]])
```

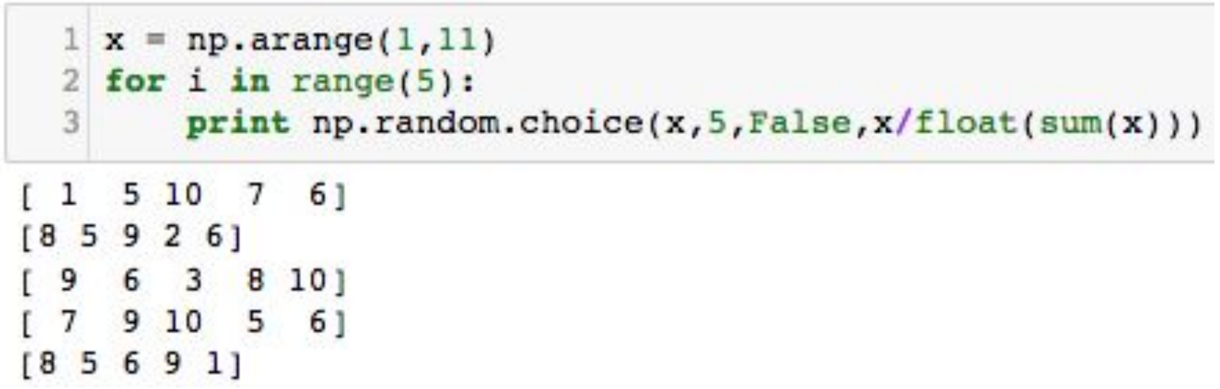
Lots more distributions:

https://docs.scipy.org/doc/numpy/reference/routines.random.html#distributions

`np.random.choice()`: random samples from data

`np.random.choice(x,[size,replace,p])`

Generates a sample of `size` elements from the array `x`, drawn with (`replace=True`) or without (`replace=False`) replacement, with element probabilities given by vector `p`.



```
x = np.arange(1,11)
for i in range(5):
    print np.random.choice(x,5,False,x/float(sum(x)))
```

```
[ 1  5 10  7  6]
[8 5 9 2 6]
[ 9  6  3  8 10]
[ 7  9 10  5  6]
[8 5 6 9 1]
```

`shuffle()` **vs** `permutation()`

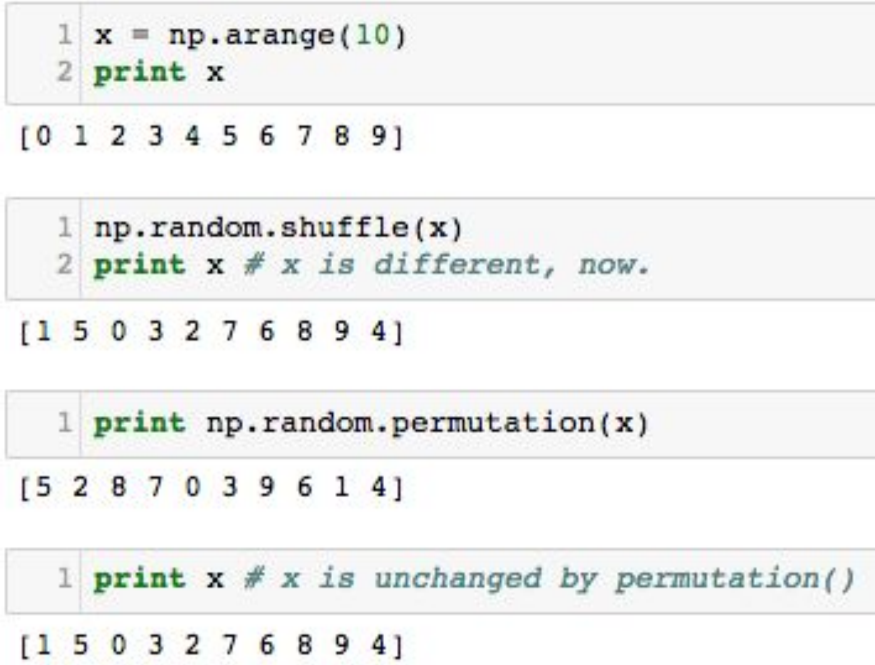
`np.random.shuffle(x)`

randomly permutes entries of `x` in place
so `x` itself is changed by this operation!

`np.random.permutation(x)`

returns a random permutation of `x`
and `x` remains unchanged.

Compare with the Python `list.sort()` and `sorted()` functions.



```
x = np.arange(10)
print x
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
np.random.shuffle(x)
print x # x is different, now.
```

```
[1 5 0 3 2 7 6 8 9 4]
```

```
print np.random.permutation(x)
```

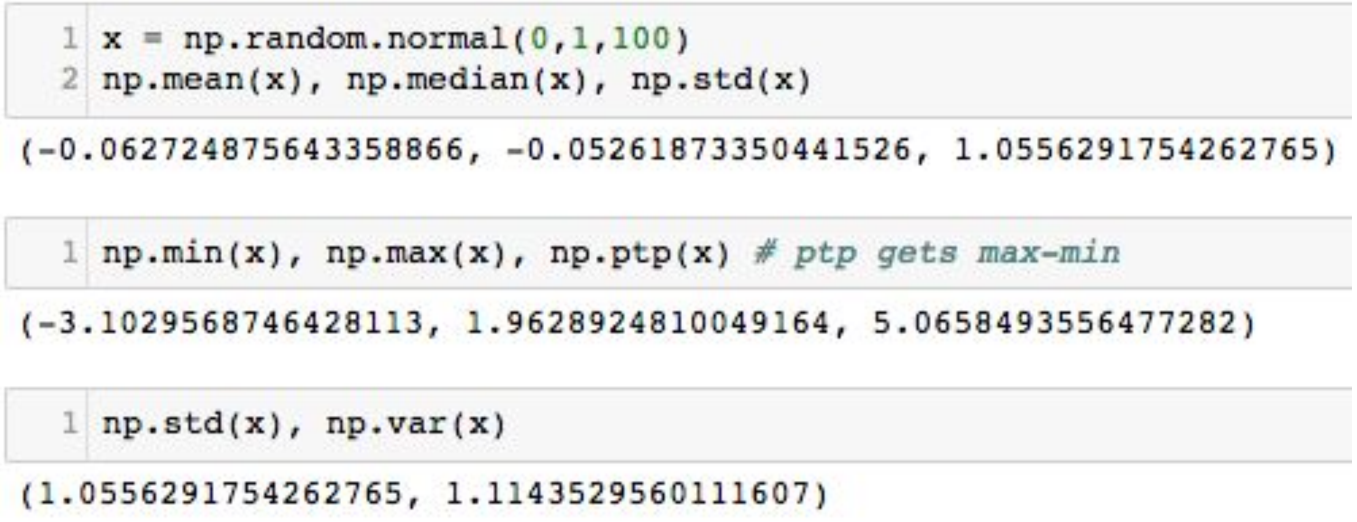
```
[5 2 8 7 0 3 9 6 1 4]
```

```
print x # x is unchanged by permutation()
```

```
[1 5 0 3 2 7 6 8 9 4]
```

Statistics in `numpy`

`numpy` implements all the standard statistics functions you've come to expect



```
x = np.random.normal(0,1,100)
np.mean(x), np.median(x), np.std(x)
```

```
(-0.062724875643358866, -0.05261873350441526, 1.0556291754262765)
```

```
np.min(x), np.max(x), np.ptp(x) # ptp gets max-min
```

```
(-3.1029568746428113, 1.9628924810049164, 5.0658493556477282)
```

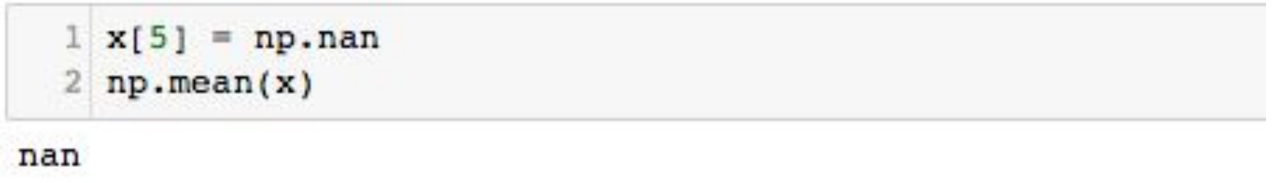
```
np.std(x), np.var(x)
```

```
(1.0556291754262765, 1.114352956011607)
```

Statistics in `numpy`

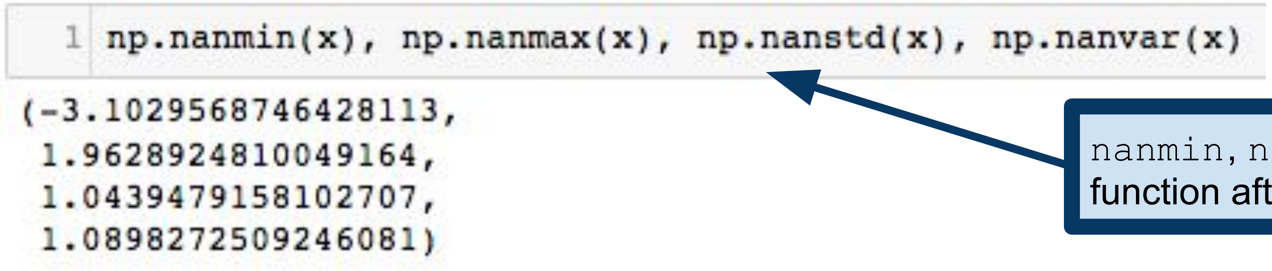
NaN is short for “not a number”. NaNs typically arise either because or improper mathematical operations (e.g., dividing by zero) or to represent missing data.

Numpy deals with NaNs more gracefully than MATLAB/R:



```
x[5] = np.nan
np.mean(x)
```

```
nan
```



```
np.nanmin(x), np.nanmax(x), np.nanstd(x), np.nanvar(x)
```

```
(-3.102956874642811З,
 1.9628924810049164,
 1.0439479158102707,
 1.0898272509246081)
```

`nanmin`, `nanvar`, etc compute function after dropping NaNs.

For more statistical functions, see:
https://docs.scipy.org/doc/numpy-1.8.1/reference/routines.statistics.html

Probability and statistics in `scipy`

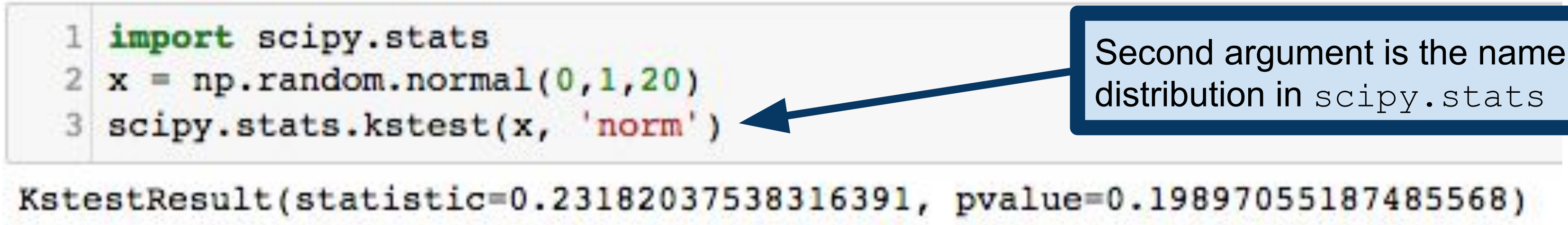
`scipy` is a distinct Python package, part of the `numpy` ecosystem.

(Almost) all the distributions you could possibly ever want:

- https://docs.scipy.org/doc/scipy/reference/stats.html#continuous-distributions
- https://docs.scipy.org/doc/scipy/reference/stats.html#multivariate-distributions
- https://docs.scipy.org/doc/scipy/reference/stats.html#discrete-distributions

More statistical functions (moments, kurtosis, statistical tests):

- https://docs.scipy.org/doc/scipy/reference/stats.html#statistical-functions



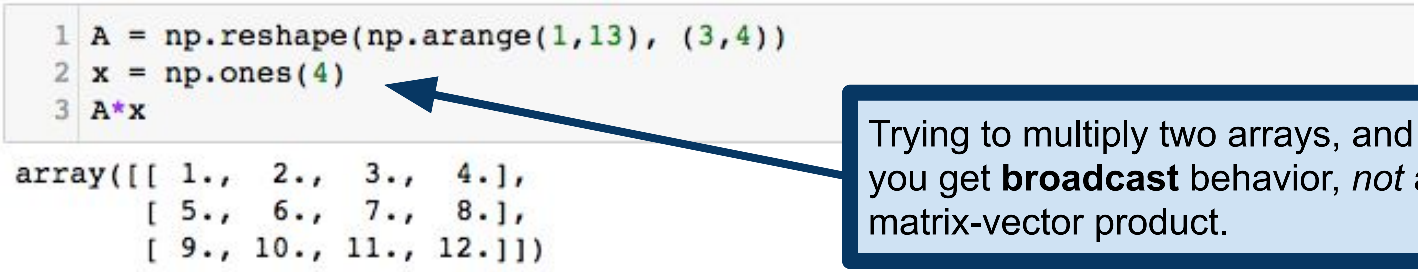
```
import scipy.stats
x = np.random.normal(0,1,20)
scipy.stats.kstest(x, 'norm')
```

```
KstestResult(statistic=0.23182037538316391, pvalue=0.19897055187485568)
```

Second argument is the name of a distribution in `scipy.stats`

Kolmogorov-Smirnov test

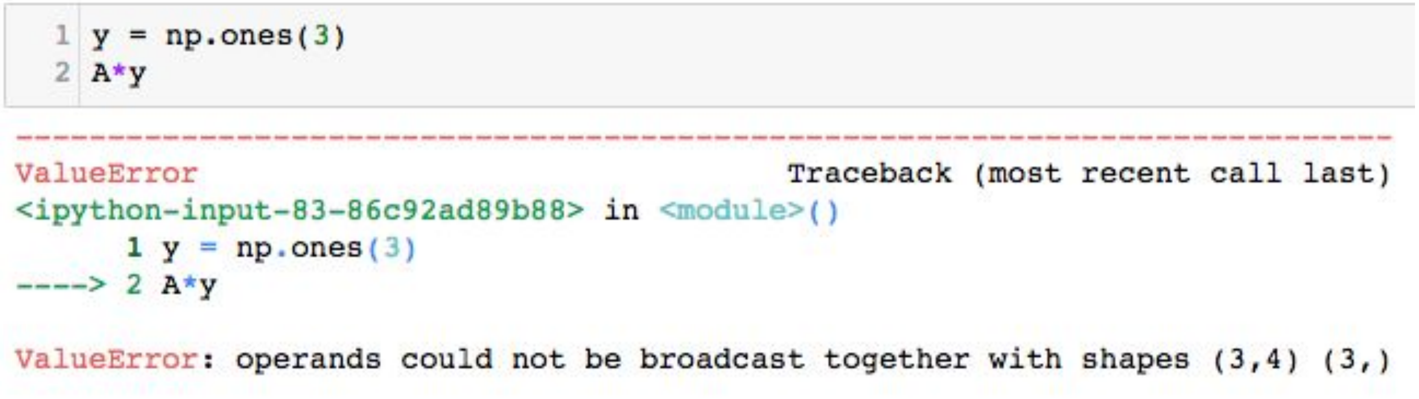
Matrix-vector operations in `numpy`



```
A = np.reshape(np.arange(1,13), (3,4))
x = np.ones(4)
A*x
```

```
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
```

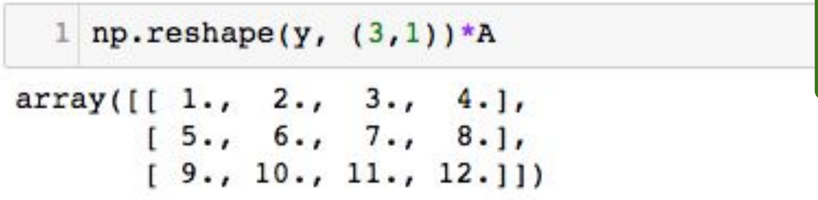
Trying to multiply two arrays, and you get **broadcast** behavior, *not* a matrix-vector product.



```
y = np.ones(3)
A*y
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-83-86c92ad89b88> in <module>()
      1 y = np.ones(3)
----> 2 A*y

ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

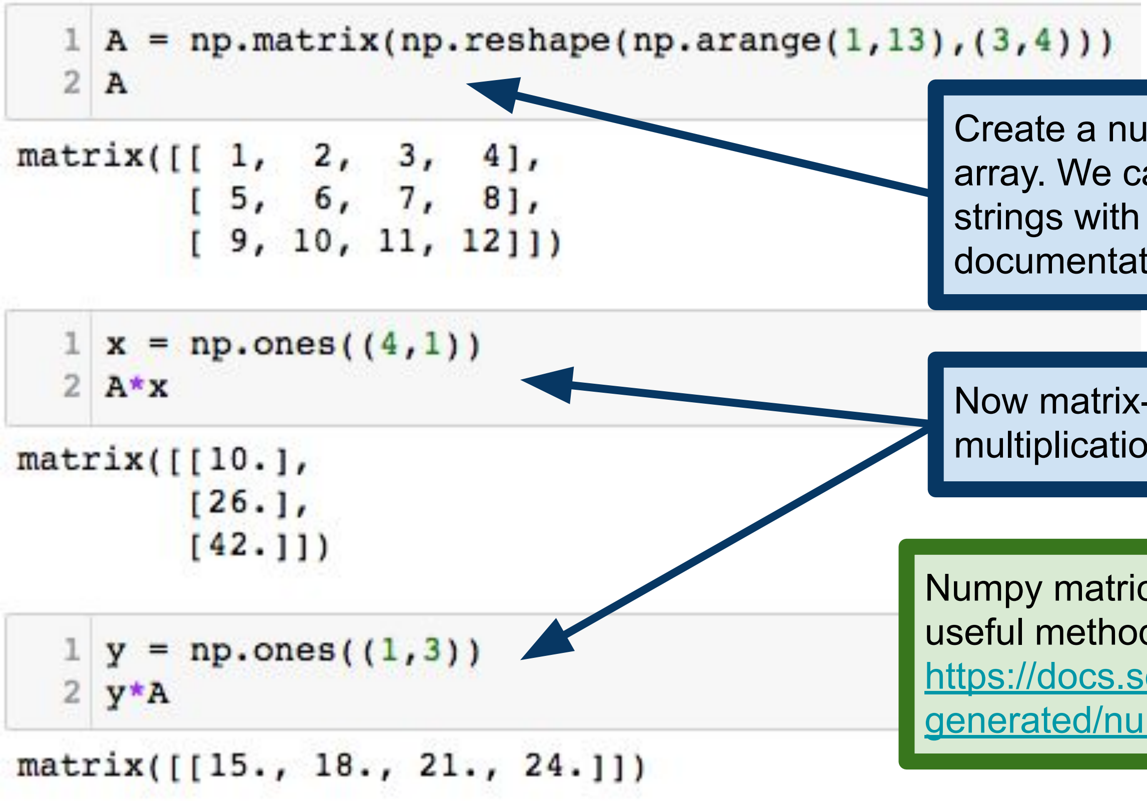


```
np.reshape(y, (3,1))*A
```

```
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.]])
```

Broadcast multiplication still requires that dimensions agree and all that.

Matrix-vector operations in `numpy`



```
A = np.matrix(np.reshape(np.arange(1,13),(3,4)))
A
```

```
matrix([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
```

```
x = np.ones((4,1))
A*x
```

```
matrix([[10.],
        [26.],
        [42.]])
```

```
y = np.ones((1,3))
y*A
```

```
matrix([[15., 18., 21., 24.]])
```

Create a numpy matrix from a numpy array. We can also create matrices from strings with MATLAB-like syntax. See documentation.

Now matrix-vector and vector-matrix multiplication work as we want.

Numpy matrices support a whole bunch of useful methods. See documentation: https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html

`numpy`/`scipy` universal functions (ufuncs)

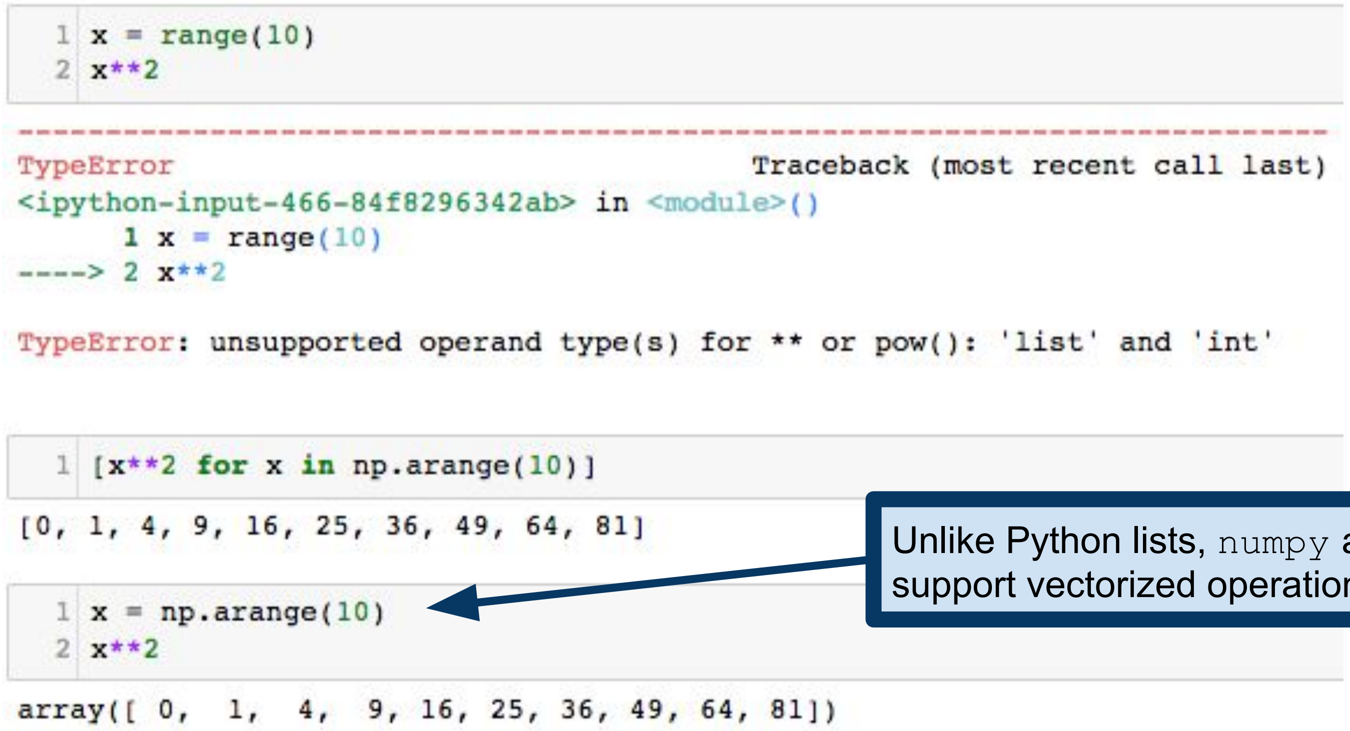
From the documentation:

> A universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a “vectorized” wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.
> https://docs.scipy.org/doc/numpy/reference/ufuncs.html

So ufuncs are vectorized operations, just like in R and MATLAB

ufuncs in action

List comprehensions are great, but they’re not well-suited to numerical computing



```
x = range(10)
x**2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-466-84f8296342ab> in <module>()
      1 x = range(10)
----> 2 x**2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

```
[x**2 for x in np.arange(10)]
```

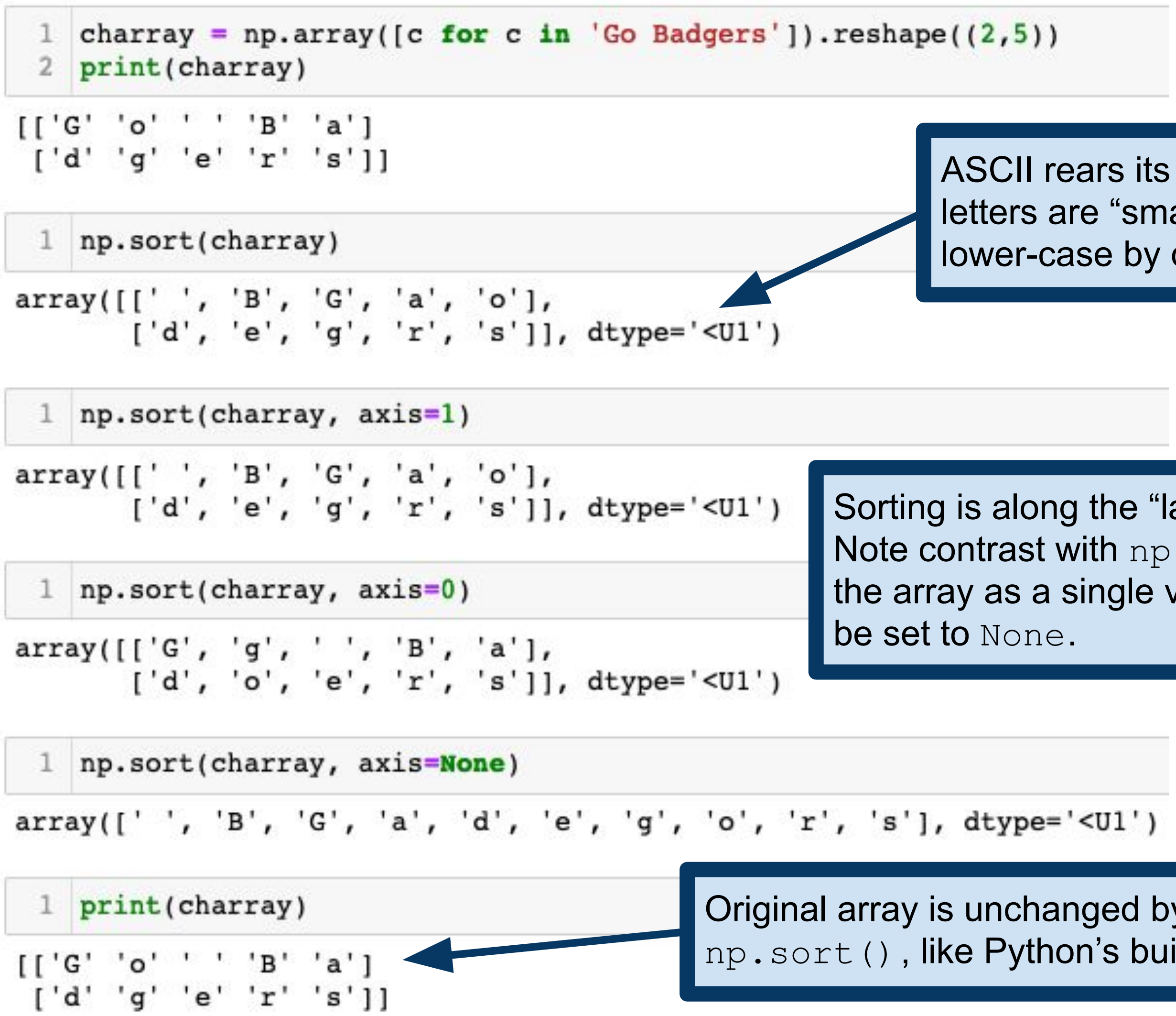
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
x = np.arange(10)
x**2
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Unlike Python lists, `numpy` arrays support vectorized operations.

Sorting



```python
charray = np.array([c for c in 'Go Badgers']).reshape((2,5))
print(charray)
```

```
[['G' 'o' ' ' 'B' 'a']
 ['d' 'g' 'e' 'r' 's']]
```

```python
np.sort(charray)
```

```
array([[' ', 'B', 'G', 'a', 'o'],
       ['d', 'e', 'g', 'r', 's']], dtype='<U1')
```

ASCII rears its head-- capital letters are “smaller” than all lower-case by default.

```python
np.sort(charray, axis=1)
```

```
array([[' ', 'B', 'G', 'a', 'o'],
       ['d', 'e', 'g', 'r', 's']], dtype='<U1')
```

```python
np.sort(charray, axis=0)
```

```
array([['G', 'g', ' ', 'B', 'a'],
       ['d', 'o', 'e', 'r', 's']], dtype='<U1')
```

Sorting is along the “last” axis by default. Note contrast with `np.any()`. To treat the array as a single vector, `axis` must be set to `None`.

```python
np.sort(charray, axis=None)
```

```
array([' ', 'B', 'G', 'a', 'd', 'e', 'g', 'o', 'r', 's'], dtype='<U1')
```

```python
print(charray)
```

```
[['G' 'o' ' ' 'B' 'a']
 ['d' 'g' 'e' 'r' 's']]
```

Original array is unchanged by use of `np.sort()`, like Python’s built-in `sorted()`

A cautionary note

`numpy`/`scipy` have several similarly-named functions with different behaviors!

Example: `np.amax`, `np.ndarray.max`, `np.maximum`

The best way to avoid these confusions is to

1) Read the documentation carefully
2) Test your code!