Lecture 8
Numpy

(Some examples adapted from Keith Levin @ Madison, Ambuj Tewari @ Umich)

Numerical computing in Python: `numpy`

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

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

https://ebooks.mobibootcamp.com/py-libs/index.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/

Everything (in Python datasci) is based on Numpy

SciPy

pandas

TensorFlow

PyTorch

`numpy` data types

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

```
1 import numpy as np
2
3 x = np.float32(3.1415)
4 type(x)
```

```
numpy.float32
```

Five basic numerical data types:

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

```
1 x
```

```
3.1415
```

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

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

```
8675309
```

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

`numpy` data types

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

```
3.1415
```

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

```
numpy.float32
```

```
1 x==y
```

```
False
```

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

```
False
```

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

```
numpy.int64
```

32-bit and 64-bit representations are distinct!

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

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.

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

Can be created from a Python list…

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

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

...by "shaping" an array…

```
1 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"...

```
1 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:

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

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

2-dimensional arrays (matrices):

```
1 x.shape = (3,4) # now x is a 3-by-4 matrix
2 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”):

```
1 x.shape = (2,3,2)
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.

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

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

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

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

```
1 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

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

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

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

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

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

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

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

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

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

```
import numpy as np

x = np.reshape(np.arange(0,12), (3,4))
x
```

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

Another nd.array type

```
x[1]
```

```
array([4, 5, 6, 7])
```

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

```
array([ 1, 11])
```

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

```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
```

More methods on iterating over arrays
https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html#integer-array-indexing

More array indexing

Numpy allows MATLAB/R-like indexing by Booleans

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

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

```
1 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()`

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

```
False
```

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

```
1 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.

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

```
True
```

```
1 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.

```
1 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()`.

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

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

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

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

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

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

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

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

Random numbers in `numpy`

`np.random` contains methods for generating random numbers

```
rng = np.random.default_rng()
rng.random(2)
```

```
array([0.40502669, 0.95842369])
```

```
np.random.default_rng().integers(2,10)
```

```
8
```

```
x = np.random.default_rng().standard_normal(10)
x.mean(), x.std(), x
```

```
(0.09520535654735485,
 1.0013922683960477,
 array([-0.1497099 , -1.44187331, -0.5844645 , -0.68035121,  0.18213075,
         0.19120056,  2.59328871,  0.62570092,  0.0700956 ,  0.14603595]))
```

Lots more distributions: https://numpy.org/doc/stable/reference/random/index.html
Why use default_rng: https://albertcthomas.github.io/good-practices-random-number-generators/

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

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

Generates a sample of `size` elements from the array `x`, drawn with (`replace=True`) or without (`replace=False`) replacement

```
x = np.arange(1,10)
for i in range(5):
  print(np.random.default_rng().choice(x, 5, False))
```

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

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

`np.random.default_rng().shuffle(x)`

randomly permutes entries of `x` in place

so `x` itself is changed by this operation!

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

returns a random permutation of `x`

and `x` remains unchanged

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

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

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

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

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

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

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

```
1 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.default_rng().random(20)
x.min(), x.max(), x.std(), x.var(), x.ptp() # ptp is max-min
```

```
(0.09477568483516696,
 0.8408443530123654,
 0.20736327727409185,
 0.042999528761851896,
 0.7460686681771984)
```

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:

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

```
nan
```

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

```
(-3.1029568746428113,
 1.9628924810049164,
 1.04394791581027O7,
 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.default_rng().standard_normal(20)
scipy.stats.kstest(x, 'norm')
```

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

```
KstestResult(statistic=0.149511002574903, pvalue=0.708144180892844)
```

Kolmogorov-Smirnov test

Matrix-vector operations in `numpy`

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

```
1 y = np.ones(3)
2 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,)
```

```
1 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`

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

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

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

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

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

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

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

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

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

```
1 x = range(10)
2 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'
```

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

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

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

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

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

Sorting

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

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

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

```
1 np.sort(charray)
```

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

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

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

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

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

To treat the array as a single vector, `axis` must be set to `None`.

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

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

```
1 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 some with same and others 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!