Lecture 4-2

NumPy Basics

Week 4 Wednesday

Miles Chen, PhD

Based on Python Data Science Handbook by Jake VanderPlas

ALWAYS do: `import numpy as np`

This is a convention that everyone follows. If you do not do this, other people will have a hard time reading your code

In [1]:

```
import numpy as np
```

In [2]:

```
np.__version__
```

Out[2]:

```
'1.21.2'
```

Numpy arrays

- like lists, arrays are mutable
- unlike lists, arrays can only contain data of the same data type

Making Arrays

- direct creation with `np.array()`
- Create a list with square brackets, and put that inside `np.array()`

In [3]:

```
np.array( [1,2,3] )
```

Out[3]:

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

In [4]:

```
a = np.array([1, 2, 3])
print(a) # printing an array appears different from the array([]) in ipython
```

```
[1 2 3]
```

In [5]:

```
print([1,2,3]) # a printed list has commas
```

```
[1, 2, 3]
```

A printed array has no commas. A printed list has commas.

In [6]:

```
type(a)
```

Out[6]:

```
numpy.ndarray
```

Upcasting

If you mix data types in an array, the values of the more restrictive types will get upcast to the value of the less restrictive type.

In [7]:

```
b = np.array([1, 2, 3.0, False, True])
print(b) # the 3.0 is a float and will upcast (coerce) other values to floats
```

```
[1. 2. 3. 0. 1.]
```

In [8]:

```
c = np.array([1, 2, "3", True, False]) # upcast (coerced) to strings
print(c)
```

```
['1' '2' '3' 'True' 'False']
```

Arrays in Higher dimensions

If you provide a list of lists, you can create a multi-dimensional array. (Like a matrix)

In [9]:

```
d = np.array( [ [1,2,3] , [4,5,6] ] )
print(d)
```

```
[[1 2 3]
 [4 5 6]]
```

When you print a multidimensional array, the number of opening square brackets is the number of dimensions. The above array is 2 dimensional. but if the dimensions don't match, you'll get an array of lists... which is not as useful.

In [10]:

```
e = np.array([ [1,2,3],[4,5] ])
print(e)
```

```
[list([1, 2, 3]) list([4, 5])]

C:\Users\miles\anaconda3\lib\site-packages\i
pykernel_launcher.py:1: VisibleDeprecationWa
rning: Creating an ndarray from ragged neste
d sequences (which is a list-or-tuple of lis
ts-or-tuples-or ndarrays with different leng
ths or shapes) is deprecated. If you meant t
o do this, you must specify 'dtype=object' w
hen creating the ndarray.
  """Entry point for launching an IPython ke
rnel.
```

Other ways to make arrays

In [11]:

```
np.zeros(5) # makes an array of 0s. similar to rep(0, 5)
```

Out[11]:

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

In [12]:

```
np.zeros(5, dtype = int)  # default is to make floats, you can specify ints
```

Out[12]:

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

In [13]:

```
np.zeros((2,4))  # give dimensions as a tuple: makes an array 2x4
```

Out[13]:

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

In [14]:

```
np.zeros((2,3,4)) # 3 dimensional array 2 x 3 x 4...
# notice the order of creation: 2 'sheets' of 3 rows by 4 columns
```

Out[14]:

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

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

In [15]:

```
np.zeros((2,3,4,5))
# make 2 'blocks', each with 3 'sheets', of 4 rows, and 5 columns
```

Out[15]:

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

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0.],
```

```
   [0., 0., 0., 0., 0.]]],


 [[[0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.]],

  [[0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.]],

  [[0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.],
   [0., 0., 0., 0., 0.]]]])
```

In addition to `np.zeros` there is `np.ones` and `np.full` which can create new arrays.

In [16]:

```
np.ones(5)  # similar, but inserts ones
```

Out[16]:

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

In [17]:

```
np.full((2,3), 1.2)  # similar, but you specify one value that gets repeated
```

Out[17]:

```
array([[1.2, 1.2, 1.2],
       [1.2, 1.2, 1.2]])
```

Making arrays of random numbers

numpy uses the Mersenne Twister

- All random generator functions begin with `np.random.`

In [18]:

```
np.random.seed(1)  # seed the generator for reproducibility
```

In [19]:

```
np.random.random(5)  # random.random for random values on the interval [0,1)
```

Out[19]:

```
array([4.17022005e-01, 7.20324493e-01, 1.14374817e-04, 3.02332573e-01,
       1.46755891e-01])
```

In [20]:

```
np.random.randn(5)
# random.randn for random normal from standard normal
# this command will produce 5 values
```

Out[20]:

```
array([-1.10593508, -1.65451545, -2.3634686
,  1.13534535, -1.01701414])
```

In [21]:

```
np.random.normal(10, 3, (2, 4))
# random.randn for random normal from normal with mean 10 and sd 3
# arranged in a 2 x 4 matrix
```

Out[21]:

```
array([[11.91208544,  7.42028018, 15.3178228
9,  6.66891084],
       [10.5436428 , 11.6930346 ,  8.3004693
1, 12.18992679]])
```

In [22]:

```
np.random.randint(0, 10, 20)
# select random integers from 0 inclusive to 10 exclusive
# and return 20 values
```

Out[22]:

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

In [23]:

```
# simulate dice rolls
np.random.randint(1,7, 50)
```

Out[23]:

```
array([3, 1, 5, 2, 3, 3, 2, 1, 2, 4, 6, 5,
4, 6, 2, 4, 1, 1, 3, 3, 2, 4,
       5, 3, 1, 1, 2, 2, 6, 4, 1, 1, 6, 6,
5, 6, 3, 5, 4, 6, 4, 6, 1, 4,
       5, 4, 5, 5, 6, 5])
```

More random generation at:

https://numpy.org/doc/stable/reference/random/index.html

Array sequences

make sequences with

- `np.arange(start, stop, step)`
- makes an **a**rray **range** from start (inclusive) to stop (exclusive), by step

In [24]:

```
range(0, 10, 2) # range object in regular python
```

Out[24]:

```
range(0, 10, 2)
```

In [25]:

```
list(range(0, 10, 2))
```

Out[25]:

```
[0, 2, 4, 6, 8]
```

In [26]:

```
np.arange(0, 10, 2)  # numpy's arange function
```

Out[26]:

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

In [27]:

```
np.array(range(0,10,2)) # equivalent 'manual' creation
```

Out[27]:

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

In [28]:

```
np.arange(0, 100, 5)
```

Out[28]:

```
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 4
5, 50, 55, 60, 65, 70, 75, 80,
       85, 90, 95])
```

In [29]:

```python
np.arange(20) # quickest
```

Out[29]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

- `np.linspace(start, stop, num)`
- makes an array of **lin**ear **space**d values beginning with start, ending with stop (inclusive), with a length of num

In [30]:

```
np.linspace(0, 100, 11)
```

Out[30]:

```
array([  0.,  10.,  20.,  30.,  40.,  50.,
60.,  70.,  80.,  90., 100.])
```

In [31]:

```
np.linspace(0, 100, 10)
```

Out[31]:

```
array([  0.        ,  11.11111111,  22.22222
222,  33.33333333,
        44.44444444,  55.55555556,  66.66666
```

```
667,  77.77777778,
        88.88888889, 100.        ])
```

In [32]:

```
np.linspace(0, 100, 10, endpoint = False)  # optional parameter endpoint to exclude the stop value
```

Out[32]:

```
array([ 0., 10., 20., 30., 40., 50., 60., 7
0., 80., 90.])
```

In [33]:

```
np.linspace(0, 100, 9, endpoint = False)
# if you use the endpoint argument, the last number in the array will depend on the output length
```

Out[33]:

```
array([ 0.        , 11.11111111, 22.2222222
2, 33.33333333, 44.44444444,
       55.55555556, 66.66666667, 77.7777777
8, 88.88888889])
```

Array Attributes

- `array.ndim` for number of dimensions
- `array.shape` for the size of each dimension
- `array.dtype` for the data type

In [34]:

```
x = np.ones((3,4))
print(x)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

In [35]:

```
x.ndim
```

Out[35]:

```
2
```

In [36]:

```
x.shape
```

Out[36]:

```
(3, 4)
```

In [37]:

```
x.dtype
```

Out[37]:

```
dtype('float64')
```

In [38]:

```
y = np.arange(0, 12, 1)
print(y)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

In [39]:

```
y.ndim
```

Out[39]:

```
1
```

In [40]:

```
y.shape # a one dimensional array. Note that there's no second dimension.
```

Out[40]:

```
(12,)
```

Reshaping Arrays

- `np.reshape(array, [new shape])` returns a new array that is reshaped
  - you can also use the method `array.reshape(shape)`
- `array.T` is the transpose method, but leaves the original array unaffected

In [41]:

```
j = np.arange(0,12,1)
print(j) # j is one dimensional
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

In [42]:

```
k = np.reshape(j, (3,4))  # note that it fills row-wise unlike R
print(k)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [43]:

```
j # j is left unchanged
```

Out[43]:

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

In [44]:

```
j.reshape(4,3)  # you can also call the method reshape() on the array j
```

Out[44]:

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

In [45]:

```
j # j is left unchanged here as well
```

Out[45]:

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

In [46]:

```
print(k)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [47]:

```
print(k.T)  # the transpose of k
```

```
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

In [48]:

```
print(k) # calling k.T does not modify the original k array
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [49]:

```
# can combine the above methods and steps into one:
l = np.arange(0,12,1).reshape((3,4)).T
# create a-range >> reshape >> transpose
print(l)
```

```
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

In [50]:

```
j
```

Out[50]:

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

In [51]:

```
j.reshape((3, -1))  # using -1 for a dimension will ask python to figure out the number to use for that dimensior
```

Out[51]:

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

In [52]:

```
j.reshape((-1, 4))
```

Out[52]:

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

In [53]:

```
j.reshape((2, -1, 2)) # two sheets, unknown number of rows, 2 columns
```

Out[53]:

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

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

In [54]:

```
y = np.arange(0,12, 1)
print(y)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

In [55]:

```
y.shape
```

Out[55]:

```
(12,)
```

In [56]:

```
print(y.T) # the transpose of a one dimensional array doesn't suddenly give it a second dimension
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

In [57]:

```
y.T.shape
```

Out[57]:

```
(12,)
```

In [58]:

```
z = np.reshape(y, (1,12)) # the array now has two dimensions
print(z)
```

```
[[ 0  1  2  3  4  5  6  7  8  9 10 11]]
```

In [59]:

```
z.shape
```

Out[59]:

```
(1, 12)
```

In [60]:

```
print(z.T)  # with two dimensions, the transpose become a column
```

```
[[ 0]
 [ 1]
 [ 2]
 [ 3]
 [ 4]
```

```
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]
 [11]]
```

In [61]:

```
z.T.shape
```

Out[61]:

```
(12, 1)
```

Subsetting and Slicing Arrays

- very similar to subsetting and slicing lists

In [63]:

```
y[4]
```

Out[63]:

```
4
```

In [64]:

```
y.shape
```

Out[64]:

```
(12,)
```

In [65]:

```
y[4:6]
```

Out[65]:

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

you can slice with a second colon. The array gets subset with `array[start:stop:step]`

In [66]:

```
y[1:8:3]
```

Out[66]:

```
array([1, 4, 7])
```

In [67]:

```
np.arange(100)[:100:2] # to get even values
```

Out[67]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
       34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66,
       68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98])
```

In [68]:

```python
np.arange(0,100,2)
```

Out[68]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 1
8, 20, 22, 24, 26, 28, 30, 32,
       34, 36, 38, 40, 42, 44, 46, 48, 50, 5
2, 54, 56, 58, 60, 62, 64, 66,
       68, 70, 72, 74, 76, 78, 80, 82, 84, 8
6, 88, 90, 92, 94, 96, 98])
```

Subsetting and slicing higher dimensional arrays is similar, and uses a comma to separate subsetting instructions for each dimension.

In [69]:

```
z = np.reshape(y, [3,4])
print(z)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [70]:

```
print(z[1,2]) # returns what is at row index 1, col index 2
```

```
6
```

In [71]:

```
type(z[1,2]) # with only one value, the type is the integer. It is no longer an array.
```

Out[71]:

```
numpy.int32
```

In [72]:

```
print(z)
z[0:2, 0:2] # note the type remains a numpy array
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Out[72]:

```
array([[0, 1],
       [4, 5]])
```

In [73]:

```
print(z[2, :]) # returns row at index 2
```

```
[ 8  9 10 11]
```

In [74]:

```
z[2, :].shape  # the shape is one dimensional
```

Out[74]:

```
(4,)
```

In [75]:

```
print(z[:,2]) # returns column at index 2
```

```
[ 2  6 10]
```

In [76]:

```
z[:,2].shape # shape is one dimensional
```

Out[76]:

```
(3,)
```

Slices of numpy arrays are view objects, and automatically update if the original array is updated.

In [77]:

```
z = np.arange(12).reshape([3,4])
print(z)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [78]:

```
# we use numpy array slicing to create z_sub, the top left corner of z
z_sub = z[:2, :2]
print(z_sub)
```

```
[[0 1]
 [4 5]]
```

In [79]:

```python
# I modify the first element of z to be 99.
z[0,0] = 99
```

In [80]:

```python
print(z_sub)  # z_sub is updated, even though we never redefined it
```

```
[[99  1]
 [ 4  5]]
```

In [81]:

```python
z
```

Out[81]:

```
array([[99,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [82]:

```
z = np.arange(15).reshape([3,5]) # here z gets redefined to an entirely new object
# we are not modifying the object that used to be called z
# we created a new object, and the name z now points to the new object
```

In [83]:

```
z
```

Out[83]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [84]:

```
print(z_sub)  # the view z_sub still points to the object formerly known as z, which was not modified
```

```
[[99  1]
 [ 4  5]]
```

If you want a copy that will not update if the original is updated, use

`array.copy()`

In [85]:

```
print(z)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

In [86]:

```
z_sub_copy = z[:2, :2].copy()
print(z_sub_copy)
```

```
[[0 1]
 [5 6]]
```

In [87]:

```
z[0,0] = 55 # modify the first element of z
```

In [88]:

```
print(z_sub_copy) # the copy remains unaffected by the change
```

```
[[0 1]
 [5 6]]
```

In [89]:

```
print(z)
```

```
[[55  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

Modifying the view object modifies the underlying array

In [90]:

```
z = np.arange(12).reshape((3,4))
print(z)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [91]:

```
view = z[:2,:2]
```

In [92]:

```
view[0,0] = 99
```

In [93]:

```
view
```

Out[93]:

```
array([[99,  1],
       [ 4,  5]])
```

In [94]:

```
z
```

Out[94]:

```
array([[99,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [95]:

```
type(view) # view objects themselves are arrays and have all the same methods and attributes
```

Out[95]:

```
numpy.ndarray
```

In [96]:

```
view.T
```

Out[96]:

```
array([[99,  4],
       [ 1,  5]])
```

In [97]:

```
view.T.reshape((4,))
```

Out[97]:

```
array([99,  4,  1,  5])
```

In [98]:

```
view # attributes like .T do not affect the orignal array
```

Out[98]:

```
array([[99,  1],
       [ 4,  5]])
```