Lecture 11 - More NumPy

Week 5 Wednesday

Miles Chen, PhD

Based on Python Data Science Handbook by Jake VanderPlas

In [1]:
```
import numpy as np
```

Concatenating Arrays

In [2]:
```
x = np.arange(4)
y = np.arange(100, 104)
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [3]:
```
np.concatenate([x,y])
```

Out[3]:
```
array([  0,   1,   2,   3, 100, 101, 102, 103])
```

`np.concatenate` has an argument for axis. The axes are 0-indexed.

In [4]:
```
np.concatenate([x,y], axis = 0)
```

Out[4]:
```
array([  0,   1,   2,   3, 100, 101, 102, 103])
```

Let's try to concatenate in the other direction. We specify axis = 1

In [5]:
```
np.concatenate([x,y], axis = 1) # throws an error
```

```
---------------------------------------------------------------------------
AxisError                                 Traceback (most recent call last)
<ipython-input-5-2cd32e4cabd5> in <module>
----> 1 np.concatenate([x,y], axis = 1) # throws an error

<__array_function__ internals> in concatenate(*args, **kwargs)

AxisError: axis 1 is out of bounds for array of dimension 1
```

In [6]:
```
x.shape # you can't use axis with index 1, because axis index 1 does not exist
```

Out[6]:
```
(4,)
```

In [7]:
```
np.vstack([x,y])   # vstack will vertically stack unidimensional arrays
```

Out[7]:
```
array([[  0,   1,   2,   3],
       [100, 101, 102, 103]])
```

In [8]:
```
x.reshape(1,4)
```
Out[8]:
```
array([[0, 1, 2, 3]])
```
In [9]:
```
y.reshape(1,4)
```
Out[9]:
```
array([[100, 101, 102, 103]])
```
In [10]:
```
np.concatenate([x.reshape(1,4), y.reshape(1,4)], axis = 0)
```
Out[10]:
```
array([[  0,   1,   2,   3],
       [100, 101, 102, 103]])
```
note that when I concatenate along axis 0 for a 2-dimensional array, it concatenates by rows. In a 2D array, index 0 is for rows, and index 1 is for columns.

In [11]:
```
np.concatenate([x.reshape(1,4), y.reshape(1,4)], axis = 1)
```
Out[11]:
```
array([[  0,   1,   2,   3, 100, 101, 102, 103]])
```

In [12]:

```
xm = np.arange(6).reshape([2,3])
ym = np.arange(100,106,1).reshape([2,3])
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [13]:

```
xm.shape
```

Out[13]:

```
(2, 3)
```

In [14]:

```
ym.shape
```

Out[14]:

```
(2, 3)
```

In [15]:
```
print(np.concatenate([xm,ym])) # default behavior concatenates on axis 0
```

```
[[  0   1   2]
 [  3   4   5]
 [100 101 102]
 [103 104 105]]
```

In [16]:
```
print(np.concatenate([xm,ym], axis = 0))
# axes are reported as rows, then columns.
# concatenating along axis 0 will concatenate along rows
```

```
[[  0   1   2]
 [  3   4   5]
 [100 101 102]
 [103 104 105]]
```

In [17]:
```
print(np.concatenate([xm,ym], axis = 1))
# concatenating along axis 1 will concatenate along columns
```

```
[[  0   1   2 100 101 102]
 [  3   4   5 103 104 105]]
```

In [18]:
```
np.vstack([xm, ym])
```

Out[18]:
```
array([[  0,   1,   2],
       [  3,   4,   5],
       [100, 101, 102],
       [103, 104, 105]])
```

In [19]:
```
np.hstack([xm, ym])
```

Out[19]:
```
array([[  0,   1,   2, 100, 101, 102],
       [  3,   4,   5, 103, 104, 105]])
```

You can always use vstack and hstack for 2D arrays.

Math Operators with numpy arrays

In [20]:
```
print(x)
print(y)
```

```
[0 1 2 3]
[100 101 102 103]
```

In [21]:
```
x + 5
```

Out[21]:
```
array([5, 6, 7, 8])
```

In [22]:
```
x + y   # elementwise addition
```

Out[22]:
```
array([100, 102, 104, 106])
```

In [23]:
```
x * y
```

Out[23]:
```
array([  0, 101, 204, 309])
```

In [24]:
```
np.sum(x * y)
```

Out[24]:
```
614
```

In [25]:
```
np.dot(x,y)    # 0 * 100 + 1 * 101 + 2 * 102 + 3 * 103
```

Out[25]:
```
614
```

In [26]:
```python
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [27]:
```python
xm + 5
```

Out[27]:
```
array([[ 5,  6,  7],
       [ 8,  9, 10]])
```

In [28]:
```python
xm + ym  # elementwise addition
```

Out[28]:
```
array([[100, 102, 104],
       [106, 108, 110]])
```

In [29]:
```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [30]:
```
xm * ym
```

Out[30]:
```
array([[  0, 101, 204],
       [309, 416, 525]])
```

In [31]:
```
np.multiply(xm, ym)
```

Out[31]:
```
array([[  0, 101, 204],
       [309, 416, 525]])
```

In [32]:
```
np.dot(xm, ym.T)
```

Out[32]:
```
array([[ 305,  314],
       [1214, 1250]])
```

In [33]:
```
xm.dot(ym.T)
```

Out[33]:
```
array([[ 305,  314],
       [1214, 1250]])
```

Basic Math

In [34]:
```
x = np.arange(4)
print(x)
```

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

In [35]:
```
print(x + 4)
```

```
[4 5 6 7]
```

In [36]:
```
print(x - 5)
```

```
[-5 -4 -3 -2]
```

In [37]:
```
print(x * 2)
```

```
[0 2 4 6]
```

In [38]:
```
print(x / 2)
```

```
[0.  0.5 1.  1.5]
```

In [39]:
```
print(-x)
```

```
[ 0 -1 -2 -3]
```

In [40]:
```
print(x ** 2)
```

```
[0 1 4 9]
```

In [41]:
```
print(x % 2) # modulo division
```

```
[0 1 0 1]
```

In [42]:
```
print(abs(x)) # abs
```

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

Trig functions

note that the functions are preceeded by np.

In [43]:
```
theta = np.linspace(0, np.pi, 5)
print(theta)
```

```
[0.         0.78539816 1.57079633 2.35619449 3.14159265]
```

In [44]:
```
print(np.sin(theta))
```

```
[0.00000000e+00 7.07106781e-01 1.00000000e+00 7.07106781e-01
 1.22464680e-16]
```

In [45]:
```
print(np.cos(theta))
```

```
[ 1.00000000e+00  7.07106781e-01  6.12323400e-17 -7.07106781e-01
 -1.00000000e+00]
```

In [46]:
```
print(np.tan(theta))
```

```
[ 0.00000000e+00  1.00000000e+00  1.63312394e+16 -1.00000000e+00
 -1.22464680e-16]
```

Log and Exp

In [47]:
```
x = np.array([1, 10, 100])
print(np.log(x))    # natural log
print(np.log10(x)) # common log
```

```
[0.         2.30258509 4.60517019]
[0. 1. 2.]
```

In [48]:
```
y = np.arange(3)
print(np.exp(y))   # e^y
```

```
[1.         2.71828183 7.3890561 ]
```

In [49]:
```
print(np.exp2(y))   # 2^y
```

```
[1. 2. 4.]
```

In [50]:
```
print(np.power(3, y)) # power ^ y
```

```
[1 3 9]
```

Aggregates

you can use `sum()`

or `np.sum()`

`np.sum()` is faster than sum, but doesn't always behave the same way

In [51]:
```
x = np.arange(100)
print(x)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
```

In [52]:
```
print(sum(x))
```

```
4950
```

In [53]:
```
print(np.sum(x))
```

```
4950
```

In [54]:

```
big_array = np.random.rand(10000)
%timeit sum(big_array)
%timeit np.sum(big_array)  # the np version is much faster
```

```
2.29 ms ± 508 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
9.13 µs ± 301 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

min and max

In [55]:
```
print(min(big_array))
print(max(big_array))
```

```
1.8117769764680602e-05
0.9999571744622143
```

In [56]:
```
print(np.min(big_array))
print(np.max(big_array))
```

```
1.8117769764680602e-05
0.9999571744622143
```

In [57]:
```
%timeit min(big_array)
%timeit np.min(big_array)  # the np version is much faster
```

```
1.24 ms ± 138 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
7.82 µs ± 861 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

summaries for matrices

In [58]:
```
np.random.seed(1)
# M = np.random.random((3, 4))
M = np.arange(12)
np.random.shuffle(M)
M = np.reshape(M, [3,4])
print(M)
```

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

In [59]:
```
sum(M) # regular sum function
```

Out[59]:
```
array([14, 18, 12, 22])
```

In [60]:
```
np.sum(M) # np.sum function
```

Out[60]:
```
66
```

In [61]:
```
print(M)
```
```
[[ 2  3  4 10]
 [ 1  6  0  7]
 [11  9  8  5]]
```

In [62]:
```
np.sum(M, axis = 0)   # np.sum function with axis specified
# matrices have two dimensions
# 0 is rows, 1 is columns
# np.sum axis = 0, will sum over rows, so you end up getting column totals
```
Out[62]:
```
array([14, 18, 12, 22])
```

In [63]:
```
np.sum(M, axis = 1)
```
Out[63]:
```
array([19, 14, 33])
```

In [64]:
```
np.min(M, axis = 0)
```
Out[64]:
```
array([1, 3, 0, 5])
```

In [65]:
```
print(M)
```
```
[[ 2  3  4 10]
 [ 1  6  0  7]
 [11  9  8  5]]
```

In [66]:
```
print(xm)
print(ym)
```

```
[[0 1 2]
 [3 4 5]]
[[100 101 102]
 [103 104 105]]
```

In [67]:
```
np.std(M)
```

Out[67]:
```
3.452052529534663
```

In [68]:
```
np.std(M, axis = 0)
```

Out[68]:
```
array([4.49691252, 2.44948974, 3.26598632, 2.05480467])
```

dealing with nan

nan is the float value for something that is not a number. We often use it in the place of a missing value. nan only exists in float type.

In [69]:
```
x = float("nan")   # direct creation of nan
print(x)
print(type(x))
```

```
nan
<class 'float'>
```

In [70]:
```
y = float("inf")   # y is the float representation of infinity
print(y / y)   # these calculations will yield a nan result
print(y - y)
```

```
nan
nan
```

In [71]:
```
np.sum([x, 2])
```

Out[71]:
```
nan
```

In [72]:
```
np.nansum([x, 2])    # in R you have the option na.rm = TRUE
```

Out[72]:
```
2.0
```

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	**NaN-safe Version**	**Description**
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Broadcasting

This is a similar concept to recyling values in R, but only works when the dimensions are compatible

In [73]:
```
a = np.array([1,2,3])
b = np.array([4,5,6])
print(a + b)
```

```
[5 7 9]
```

In [74]:
```
c = np.array([7,8])
print(a + c)   # doesn't work
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-74-7215ac8feb02> in <module>
      1 c = np.array([7,8])
----> 2 print(a + c)   # doesn't work

ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

In [75]:
```
print(a)
```

```
[1 2 3]
```

In [76]:
```
e = np.ones([3,3])
print(e)
```

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

In [77]:
```
print(e + a)   # the array a gets 'broadcast' across all three rows
```

```
[[2. 3. 4.]
 [2. 3. 4.]
 [2. 3. 4.]]
```

In [78]:
```
print(a.reshape([3,1]))   # we reshape a to be a 3x1 array
```

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

In [79]:
```
print(e + a.reshape([3,1])) # the reshaped array is broadcast across columns
```

```
[[2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
```

In [80]:
```
d = np.vstack([a,b])  # we stack the arrays a and b vertically
print(d)
```

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

In [81]:
```
a
```

Out[81]:
```
array([1, 2, 3])
```

In [82]:
```
print(d + a)  # a is broadcast across row
```

```
[[2 4 6]
 [5 7 9]]
```

In [83]:
```
print(c)
```

```
[7 8]
```

In [84]:
```
print(d)
```

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

In [85]:
```
print(d + c)  # c does not have compatible dimensions
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-85-8c651d5d46fc> in <module>
----> 1 print(d + c)  # c does not have compatible dimensions

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

In [86]:
```
print(d + c.reshape([2,1]))  # after we reshape c to be a column, we can broadcast it
```

```
[[ 8  9 10]
 [12 13 14]]
```

In [87]:

```
e = np.arange(10).reshape((10, 1))
f = np.arange(11)
print(e)
print(f)
```

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

In [88]:

```
print(e + f)  ## e and f are broadcast into compatible matrices and then added
```

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

In [89]:
```python
print(e * f)  ## e and f are broadcast into compatible matrices and then multiplied element-wise
```

```
[[ 0  0  0  0  0  0  0  0  0  0  0]
 [ 0  1  2  3  4  5  6  7  8  9 10]
 [ 0  2  4  6  8 10 12 14 16 18 20]
 [ 0  3  6  9 12 15 18 21 24 27 30]
 [ 0  4  8 12 16 20 24 28 32 36 40]
 [ 0  5 10 15 20 25 30 35 40 45 50]
 [ 0  6 12 18 24 30 36 42 48 54 60]
 [ 0  7 14 21 28 35 42 49 56 63 70]
 [ 0  8 16 24 32 40 48 56 64 72 80]
 [ 0  9 18 27 36 45 54 63 72 81 90]]
```

In [90]:
```python
print(d)
```

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

In [91]:
```python
d.reshape((1,6)) + d.reshape((6,1))
```

Out[91]:
```
array([[ 2,  3,  4,  5,  6,  7],
       [ 3,  4,  5,  6,  7,  8],
       [ 4,  5,  6,  7,  8,  9],
       [ 5,  6,  7,  8,  9, 10],
       [ 6,  7,  8,  9, 10, 11],
       [ 7,  8,  9, 10, 11, 12]])
```

Boolean Operators in NumPy

In [92]:
```
x = np.arange(6)
print(x)
```

```
[0 1 2 3 4 5]
```

In [93]:
```
print(x < 3)
```

```
[ True  True  True False False False]
```

In [94]:
```
print(x >= 3)
```

```
[False False False  True  True  True]
```

In [95]:
```
print(x == 3)
```

```
[False False False  True False False]
```

In [96]:
```
# the results can then be used to subset
print(x[x >= 3])
```

```
[3 4 5]
```

In [97]:
```
np.sum(x >= 3) # True = 1, False = 0, so sum counts how many are true
```

Out[97]:
```
3
```

In [98]:
```
np.mean(x >= 3)  # finds the proportion that is True
```

Out[98]:
```
0.5
```

In [99]:
```
print(~(x == 3)) # use the tilde for negation of boolean values
```

```
[ True  True  True False  True  True]
```

In [100]:
```
print(~x == 3) # be careful if you leave off parenthesis
```

```
[False False False False False False]
```

In [101]:
```
~x
```

Out[101]:
```
array([-1, -2, -3, -4, -5, -6], dtype=int32)
```

Working with matrices

In [102]:
```
y = np.arange(12).reshape([3,4])
print(y)
```

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

In [103]:
```
print(y >= 6)
```

```
[[False False False False]
 [False False  True  True]
 [ True  True  True  True]]
```

In [104]:
```
np.sum(y >= 6)
```

Out[104]:
```
6
```

In [105]:
```
np.sum(y >= 6, axis = 0)   # you can perform sums and other aggregate functions axis-wise on the boolean matrix
```

Out[105]:
```
array([1, 1, 2, 2])
```

In [106]:
```
np.sum(y >= 6, axis = 1)
```

Out[106]:
```
array([0, 2, 4])
```

Bitwise (element-wise) Boolean operators

In [107]:
```
a = np.array([True, True, False, False])
b = np.array([True, False, True, False])
print(a)
print(b)
```

```
[ True  True False False]
[ True False  True False]
```

In [108]:
```
print(a & b) # bitwise and
```

```
[ True False False False]
```

In [109]:
```
print(a | b) # bitwise or
```

```
[ True  True  True False]
```

In [110]:
```
print(a ^ b) # bitwise xor (exclusive or)
```

```
[False  True  True False]
```

In [111]:
```
print(~a)  # bitwise not
```

```
[False False  True  True]
```

In [112]:
```
np.any(a)
```

Out[112]:
```
True
```

In [113]:
```
np.all(a)
```

Out[113]:
```
False
```

fancy indexing

Regular lists in python do not support fancy indexing, but NumPy does!

In [114]:
```
np.random.seed(1)
x = np.random.randint(100, size = 10)
print(x)
```

```
[37 12 72  9 75  5 79 64 16  1]
```

In [115]:
```
index = [0, 1, 5]
print(x[index])
```

```
[37 12  5]
```

In [116]:
```
a = [1, 4, 7]
b = [2, 3, 8]
ind = np.vstack([a,b])
print(ind)
```

```
[[1 4 7]
 [2 3 8]]
```

In [117]:
```
print(x[ind])
```

```
[[12 75 64]
 [72  9 16]]
```

In [118]:
```
X = np.arange(12).reshape((3, 4))
print(X)
```

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

In [119]:
```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
```

Out[119]:
```
array([ 2,  5, 11])
```

sorting

- `np.sort()`
- `np.argsort()` gives the indexes of the values to have the proper sorting

In [120]:
```
np.random.seed(2)
x = np.arange(5)
np.random.shuffle(x)
print(x)
```

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

In [121]:
```
x.sort() # sorts x in place
print(x)
```

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

In [122]:
```
y = np.array([5, 2, 1, 4])
print(y)
print(y.argsort())
```

```
[5 2 1 4]
[2 1 3 0]
```

In [123]:
```
d = y.argsort()
y[d]
```

Out[123]:
```
array([1, 2, 4, 5])
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:

In [124]:
```
np.random.seed(1)
X = np.random.randint(0, 10, (4, 6))
print(X)
```

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

In [125]:
```
# sort each column of X
# np.sort returns a copy of X after sorted. It does not modify X
np.sort(X, axis=0)
```

Out[125]:
```
array([[1, 2, 1, 2, 0, 0],
       [5, 7, 4, 5, 0, 4],
       [5, 8, 6, 7, 2, 6],
       [7, 9, 9, 9, 4, 7]])
```

In [126]:
```
# sort each row of X
np.sort(X, axis=1)
```

Out[126]:
```
array([[0, 0, 5, 5, 8, 9],
       [1, 2, 4, 6, 7, 9],
       [2, 2, 4, 4, 5, 7],
       [0, 1, 6, 7, 7, 9]])
```

other things

In [127]:
```
X[0,:] # selecting a row
```

Out[127]:
```
array([5, 8, 9, 5, 0, 0])
```

In [128]:
```
print(X)
```

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

In [129]:
```
X[:,1].argsort()  # the argsort for the column index 1
```

Out[129]:
```
array([2, 1, 0, 3], dtype=int64)
```

In [130]:
```
print(X[ X[:,1].argsort() , : ])  # 'subset' X by the argsort to arrange X by the c
olumn
```

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