Python for Scientific Data Analysis

NumPy/SciPy

Section 4: More Array Operations: Concatenating, Stacking and Splitting, Repeating, and Meshgrid; Array Broadcasting

Concatenation

concatenating vectors and matrices

NumPy allows a fairly straightforward way to concatenate arrays with the `np.concatenate` function. The function accepts a list of arrays (i.e. enclosed by [ ]). Here's an example:

```
arr = np.random.randn(5, 4)
#array([[-0.68343715,  0.42697112,  1.55040793, -0.17410807],
#       [-1.00085308, -0.41885304,  0.38639884,  0.41435839],
#       [-2.08405897, -1.12942072,  0.44574679,  0.16589074],
#       [ 0.63881343, -0.43989698, -0.34659523, -0.97224899],
#       [ 0.98016071, -1.12799699, -1.48254024,  0.59625527]])

arr2=arr+np.random.randn(arr.shape[0],arr.shape[1])*0.1  #create another
array that is just slightly different from the first one

arr2
#array([[-0.73568365,  0.46549071,  1.53651663, -0.11438495],
#       [-0.83757223, -0.26504489,  0.33562877,  0.26723342],
#       [-1.85794528, -1.11542342,  0.38957999,  0.29106831],
#       [ 0.70057992, -0.54014221, -0.5489486 , -0.84330599],
#       [ 1.02714068, -1.24619816, -1.30797856,  0.38491189]])

arr3=np.concatenate([arr,arr2])
#array([[-0.68343715,  0.42697112,  1.55040793, -0.17410807],
       [-1.00085308, -0.41885304,  0.38639884,  0.41435839],
       [-2.08405897, -1.12942072,  0.44574679,  0.16589074],
       [ 0.63881343, -0.43989698, -0.34659523, -0.97224899],
       [ 0.98016071, -1.12799699, -1.48254024,  0.59625527],
       [-0.73568365,  0.46549071,  1.53651663, -0.11438495],
       [-0.83757223, -0.26504489,  0.33562877,  0.26723342],
       [-1.85794528, -1.11542342,  0.38957999,  0.29106831],
       [ 0.70057992, -0.54014221, -0.5489486 , -0.84330599],
       [ 1.02714068, -1.24619816, -1.30797856,  0.38491189]])
```

Note that the `shape` of the two input arrays -- `arr` and `arr2` -- are each (5,8). And the shape of the concatenated array is (10,4) (ten rows and four columns). Now we can invoke the `axis` keyword to control *how* these arrays are concatenated. I.e.

```
arr4=np.concatenate([arr,arr2],axis=1)
 #array([[-0.68343715,  0.42697112,  1.55040793, -0.17410807, -0.73568365
,
 #        0.46549071,  1.53651663, -0.11438495],
 #      [-1.00085308, -0.41885304,  0.38639884,  0.41435839, -0.83757223,
 #       -0.26504489,  0.33562877,  0.26723342],
 #      [-2.08405897, -1.12942072,  0.44574679,  0.16589074, -1.85794528,
 #       -1.11542342,  0.38957999,  0.29106831],
 #      [ 0.63881343, -0.43989698, -0.34659523, -0.97224899,  0.70057992,
 #       -0.54014221, -0.5489486 , -0.84330599],
 #      [ 0.98016071, -1.12799699, -1.48254024,  0.59625527,  1.02714068,
 #      -1.24619816, -1.30797856,  0.38491189]])
```

This yields an array `arr4` with a shape of (5,8) (i.e now we have five rows and eight columns).

Stacking and Splitting

array stacking

In the previous sections you found how you can create NumPy arrays as 1-D vectors and then reshape these 1-D vectors into 2-D arrays (matrices). And you also saw how to combine arrays together via concatenation. Now if, for whatever reason, you just are philsophically opposed to concatenation never fear: there is another NumPy operation at your disposal: *stacking*. There are a couple of ways to stack arrays.

- *hstack* - stacks arrays horizontally
- *vstack* - stacks arrays vertically
- *columnstack* stacks vectors in columns

A simple example:

```
newarr=np.array([1,2,3,4,5])
newarr2=np.array([6,7,8,9,10])

np.vstack([newarr,newarr2])
#array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])

np.hstack([newarr,newarr2])
#array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

np.column_stack([newarr,newarr2])
#array([[ 1,  6],
       [ 2,  7],
       [ 3,  8],
       [ 4,  9],
       [ 5, 10]])
```

A slightly more involved example with a matrix instead of a 1-D vector.

```
arr5=np.vstack([arr,arr2])
arr5
#array([[-0.68343715,  0.42697112,  1.55040793, -0.17410807],
       [-1.00085308, -0.41885304,  0.38639884,  0.41435839],
       [-2.08405897, -1.12942072,  0.44574679,  0.16589074],
       [ 0.63881343, -0.43989698, -0.34659523, -0.97224899],
       [ 0.98016071, -1.12799699, -1.48254024,  0.59625527],
       [-0.73568365,  0.46549071,  1.53651663, -0.11438495],
       [-0.83757223, -0.26504489,  0.33562877,  0.26723342],
       [-1.85794528, -1.11542342,  0.38957999,  0.29106831],
       [ 0.70057992, -0.54014221, -0.5489486 , -0.84330599],
       [ 1.02714068, -1.24619816, -1.30797856,  0.38491189]])

arr6=np.hstack([arr,arr2])
arr6
#array([[-0.68343715,  0.42697112,  1.55040793, -0.17410807, -0.73568365,
         0.46549071,  1.53651663, -0.11438495],
       [-1.00085308, -0.41885304,  0.38639884,  0.41435839, -0.83757223,
        -0.26504489,  0.33562877,  0.26723342],
       [-2.08405897, -1.12942072,  0.44574679,  0.16589074, -1.85794528,
        -1.11542342,  0.38957999,  0.29106831],
       [ 0.63881343, -0.43989698, -0.34659523, -0.97224899,  0.70057992,
        -0.54014221, -0.5489486 , -0.84330599],
       [ 0.98016071, -1.12799699, -1.48254024,  0.59625527,  1.02714068,
        -1.24619816, -1.30797856,  0.38491189]])
```

You'll notice that in these cases the resulting arrays `arr5` and `arr6` are identical to `arr3` and `arr4`, respectively.

array splitting

`split` slices apart an array into multiple arrays along an axis . For example, starting with `arr3` from before

```
arr3
##array([[-0.68343715,  0.42697112,  1.55040793, -0.17410807],
       [-1.00085308, -0.41885304,  0.38639884,  0.41435839],
       [-2.08405897, -1.12942072,  0.44574679,  0.16589074],
       [ 0.63881343, -0.43989698, -0.34659523, -0.97224899],
       [ 0.98016071, -1.12799699, -1.48254024,  0.59625527],
       [-0.73568365,  0.46549071,  1.53651663, -0.11438495],
       [-0.83757223, -0.26504489,  0.33562877,  0.26723342],
       [-1.85794528, -1.11542342,  0.38957999,  0.29106831],
       [ 0.70057992, -0.54014221, -0.5489486 , -0.84330599],
       [ 1.02714068, -1.24619816, -1.30797856,  0.38491189]])


first,second=np.split(arr3,[1])

first
#array([[-0.68343715,  0.42697112,  1.55040793, -0.17410807]])

second
#array([[-1.00085308, -0.41885304,  0.38639884,  0.41435839],
       [-2.08405897, -1.12942072,  0.44574679,  0.16589074],
       [ 0.63881343, -0.43989698, -0.34659523, -0.97224899],
       [ 0.98016071, -1.12799699, -1.48254024,  0.59625527],
       [-0.73568365,  0.46549071,  1.53651663, -0.11438495],
       [-0.83757223, -0.26504489,  0.33562877,  0.26723342],
       [-1.85794528, -1.11542342,  0.38957999,  0.29106831],
       [ 0.70057992, -0.54014221, -0.5489486 , -0.84330599],
       [ 1.02714068, -1.24619816, -1.30797856,  0.38491189]])


first,second,third=np.split(arr3,[2,4])

first
#array([[-0.68343715,  0.42697112,  1.55040793, -0.17410807],
       [-1.00085308, -0.41885304,  0.38639884,  0.41435839]])
second
```

```
#array([[-2.08405897, -1.12942072,  0.44574679,  0.16589074],
       [ 0.63881343, -0.43989698, -0.34659523, -0.97224899]])
third
#array([[ 0.98016071, -1.12799699, -1.48254024,  0.59625527],
       [-0.73568365,  0.46549071,  1.53651663, -0.11438495],
       [-0.83757223, -0.26504489,  0.33562877,  0.26723342],
       [-1.85794528, -1.11542342,  0.38957999,  0.29106831],
       [ 0.70057992, -0.54014221, -0.5489486 , -0.84330599],
       [ 1.02714068, -1.24619816, -1.30797856,  0.38491189]])

first,second=np.split(arr3,[1],axis=1)

first
#array([[-0.68343715],
       [-1.00085308],
       [-2.08405897],
       [ 0.63881343],
       [ 0.98016071],
       [-0.73568365],
       [-0.83757223],
       [-1.85794528],
       [ 0.70057992],
       [ 1.02714068]])
second
#array([[ 0.42697112,  1.55040793, -0.17410807],
       [-0.41885304,  0.38639884,  0.41435839],
       [-1.12942072,  0.44574679,  0.16589074],
       [-0.43989698, -0.34659523, -0.97224899],
       [-1.12799699, -1.48254024,  0.59625527],
       [ 0.46549071,  1.53651663, -0.11438495],
       [-0.26504489,  0.33562877,  0.26723342],
       [-1.11542342,  0.38957999,  0.29106831],
       [-0.54014221, -0.5489486 , -0.84330599],
       [-1.24619816, -1.30797856,  0.38491189]])
```

Repeating Array Elements

It's often useful to create an array that is a repeat or replication of another array some number of times. E.g. IDL's replicate function does this for scalars: with some trickery you can make it work for vectors and arrays. Python has stand-alone functions for exactly this purpose.

Two useful tools for repeating or replicating arrays to produce larger arrays are the `repeat` and `tile` functions.

`repeat` replicates each element in an array some number of times, producing a larger array. E.g.

```
a=np.arange(3)
#array([0,1,2])

a.repeat(3)
#array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

By default, if you pass an integer, each element will be repeated that number of times. If you pass an array of integers, each element can be repeated a different number of times:

```
a.repeat([2,3,4])
array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

Multidimensional arrays can have their elements repeated along a particular axis.

```
arr = np.random.randn(2, 2)
#array([[-0.36605212,  1.17756764],
       [-0.06350767, -0.86832577]])
arr.repeat(2,axis=0)
#array([[-0.36605212,  1.17756764],
       [-0.36605212,  1.17756764],
       [-0.06350767, -0.86832577],
       [-0.06350767, -0.86832577]])

arr.repeat(2,axis=1)
#array([[-0.36605212, -0.36605212,  1.17756764,  1.17756764],
       [-0.06350767, -0.06350767, -0.86832577, -0.86832577]])
```

Note that if no axis is passed, the array will be flattened first, which is likely not what you want. Similarly, you can pass an array of integers when repeating a multidimensional array to repeat a given slice a different number of times:

```
arr.repeat([3,2], axis=0)

#array([[-0.36605212,  1.17756764],
       [-0.36605212,  1.17756764],
       [-0.36605212,  1.17756764],
       [-0.06350767, -0.86832577],
       [-0.06350767, -0.86832577]])
```

`tile` is different. tile, on the other hand, is a shortcut for stacking copies of an array along an axis. Visually you can think of it as being akin to “laying down tiles”:

```
arr
#array([[-0.36605212,  1.17756764],
       [-0.06350767, -0.86832577]])

np.tile(arr,2)
#array([[-0.36605212,  1.17756764, -0.36605212,  1.17756764],
       [-0.06350767, -0.86832577, -0.06350767, -0.86832577]])
```

The second argument is the number of tiles; with a scalar, the tiling is made row by row, rather than column by column. The second argument to tile can be a tuple indicating the layout of the “tiling”:

```
np.tile(arr,(2,1))
#array([[-0.36605212,  1.17756764],
       [-0.06350767, -0.86832577],
       [-0.36605212,  1.17756764],
       [-0.06350767, -0.86832577]])

np.tile(arr,(1,2))
#array([[-0.36605212,  1.17756764, -0.36605212,  1.17756764],
       [-0.06350767, -0.86832577, -0.06350767, -0.86832577]])

np.tile(arr,(3,2))
#array([[-0.36605212,  1.17756764, -0.36605212,  1.17756764],
       [-0.06350767, -0.86832577, -0.06350767, -0.86832577],
       [-0.36605212,  1.17756764, -0.36605212,  1.17756764],
       [-0.06350767, -0.86832577, -0.06350767, -0.86832577],
       [-0.36605212,  1.17756764, -0.36605212,  1.17756764],
       [-0.06350767, -0.86832577, -0.06350767, -0.86832577]])
```

Meshgrid

NumPy arrays allow you to vectorize array expressions that might otherwise require writing loops (which Python is a lot slower at than C, Fortran, and Julia).

One example of this is the `np.meshgrid` function, which takes two one-dimensinoalarrays and produces two two-dimensional matrices corresponding to all pairs of (x,y) in the two arrays.

E.g.

```
points=np.arange(-5,5,0.01)
xs,ys= np.meshgrid(points,points) #returns a two-element list of NumPy ar
rays
```

Here, both xs and ys are arrays of dimension(1000,1000).

The use of `meshgrid` is that it allows an easy evaluation of functions of x and y (i.e. f(x,y)).vvv

Array Broadcasting

Broadcasting refers to how arithmetic works between NumPy arrays of different shapes. E.g. a vector array and a scalar.

```
arr=np.arange(5)
#array([0,1,2,3,4])
arr*4
#array([0,4,8,12,16])
```

Here, the array `arr` is not repeated four times but instead each element of the array is multiplied by a scalar value of 4. We say that four is *broadcast* to all of the array elements.

For example, we can demean each column of an array by subtracting the column means. In this case, it is very simple:

```
arr = np.random.randn(4, 3)
arr.mean(0)
array([-0.3928, -0.3824, -0.8768])
demeaned = arr - arr.mean(0)
demeaned
#array([[ 0.3937, 1.7263, 0.1633],
[-0.4384, -1.9878, -0.9839],
[-0.468 , 0.9426, -0.3891],
[ 0.5126, -0.6811, 1.2097]])
demeaned.mean(0)
#array([-0., 0., -0.])
```

Improper array broadcasting (or miscasting) is the source of many Python coding errors. Which brings us to the broadcasting "rule":

Two arrays are compatible for broadcasting if for each trailing dimension (i.e., starting from the end) the axis lengths match or if either of the lengths is 1. Broadcasting is then

performed over the missing or length 1 dimensions.

For example, for a (4,3) NumPy array and a (3,) array, the result is a (4,3) array where the (3,) array operates on each row of the first array. For a (4,3) array and a (4,1) array, the result is a (4,3) array where the (4,1) array operates on every column.

Note that if you ask Python to do an impossible broadcast, you will get an error that looks something like this:

```
ValueError Traceback (most recent call last)
<ipython-input-93-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4,3) (4
,)
```

A common problem, therefore, is needing to add a new axis with length 1 specifically for broadcasting purposes. Using reshape is one option, but inserting an axis requires constructing a tuple indicating the new shape. This can often be a tedious exercise. Thus, NumPy arrays offer a special syntax for inserting new axes by indexing. We use the special np.newaxis attribute along with “full” slices to insert the new axis:

```
arr=np.zeros((4,4))

arr_3d=arr[:,np.newaxis,:]
arr_3d.shape
#(4,1,4)
```

Another example:

```
arr_1d=np.random.normal(size=3)
array([-0.18062971,  0.2243808 , -0.08067678])
arr_1d[:,np.newaxis]
#array([[-0.18062971],
        [ 0.2243808],
        [-0.08067678]])

arr_1d[np.newaxis,:]
#array([[-0.18062971,  0.2243808 , -0.08067678]])
```

The same broadcasting rule governing arithmetic operations also applies to setting values via array indexing. In a simple case, we can do things like:

```
arr = np.zeros((4, 3))
arr[:] = 5
arr

#array([[ 5., 5., 5.],
[ 5., 5., 5.],
[ 5., 5., 5.],
[ 5., 5., 5.]])
```

However, if we had a one-dimensional array of values we wanted to set into the columns of the array, we can do that as long as the shape is compatible:

```
col = np.array([1.28, -0.42, 0.44, 1.6])
arr[:] = col[:, np.newaxis]
arr

#array([[ 1.28, 1.28, 1.28],
[-0.42, -0.42, -0.42],
[ 0.44, 0.44, 0.44],
[ 1.6 , 1.6 , 1.6 ]])

arr[:2] = [[-1.37], [0.509]]
arr

#array([[-1.37 , -1.37 , -1.37 ],
[ 0.509, 0.509, 0.509],
[ 0.44 , 0.44 , 0.44 ],
[ 1.6 , 1.6 , 1.6 ]])
```