Python for Scientific Data Analysis

NumPy/SciPy

Section 3: Array Slicing and Reshaping

Array Slicing

Basic Slicing and Caveats

We already covered array slicing in the **Data Structures** section. But because this is so important, we will review what we said before. We will also describe in more detail "boolean slicing".

Let's start with a NumPy array created from one of the functions described in the previous section: `arr = np.arange(10)`. This equals `array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])`.

And do some slicing. `a[4]=4`; `a[3:7] = array([3, 4, 5, 6])`.

We can also *redefine* array elements from slicing.

E.g. `a[2:5]=12`. After that operation, `a= array([ 0, 1, 12, 12, 12, 5, 6, 7, 8, 9])`. As you can see, if you assign a scalar value to a slice, as in arr[2:5] = 12, the value is propagated (or broadcasted henceforth) to the entire selection. An important first distinction from Python's built-in lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

To give an example of this, I first create a slice of arr:

`arr_slice=a[2:5]`, which equals `array([12,12,12])`

Now, when I change values in arr_slice, the mutations are reflected in the original array arr: `arr_slice[1] = 12345`

then `a=array([ 0, 1, 12, 12345, 12, 5, 6, 7, 8, 9])`.

CAUTION !!! Now, this is a bit different than the form in other languages (e.g. IDL). If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example, `arr[2:5].copy()`.

Higher Dimensional Slicing

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d[2]=array([7, 8, 9]
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
arr2d[0][2]
arr2d[0,2]
#both equal 3
```

Indexing with Slicing

Another example shows how indexing is treated with slicing for NumPy arrays.

```
arr2d = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
arr2d[:2]=array([7,8,9])
#array([[1, 2, 3],
#[4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read the expression arr2d[:2] as “select the first two rows of arr2d.” You can pass multiple slices just like you can pass multiple indexes:

`arr2d[:2, 1:]` equals `array([[2, 3], [5, 6]])`.

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices. For example, I can select the second row but only the first two columns like so: `arr2d[1, :2]` equals `array([4, 5])`.

Similarly, I can select the third column but only the first two rows like so: `arr2d[:2, 2]` equals `array([3, 6])`.

Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

`arr2d[:, :1]` which equals

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

assigning to a slice expression assigns to the whole selection:

arr2d[:2, 1:] = 0

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

Slicing with Boolean Indexing

One Way

Let’s consider an example where we have some data in a NumPy array and an array of names with duplicates. We use the randn function in `numpy.random` to generate some random normally distributed data:

`names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])`

`data = np.random.randn(7, 4)`, which yields

```
array([[-1.67943596, -0.22358419,  0.22288571, -0.08834986],
       [ 0.46251834,  0.7372516 ,  0.24279069,  1.3422404 ],
       [ 0.08926212, -0.59101301, -1.71152208, -0.27493075],
       [ 1.36745826,  0.67940994,  0.6952116 ,  0.18046883],
       [ 1.2781309 , -0.26204851,  1.14487204,  1.49803399],
       [ 1.12997958, -1.7684264 ,  0.96524459,  0.40287942],
       [-0.91130181, -0.68538892, -0.49644622, -1.33955245]])
```

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

`names == 'Bob'; array([ True, False, False, True, False, False, False])`

This boolean array can be passed when indexing the array: `data[names == 'Bob']` yields

```
array([[-1.67943596, -0.22358419,  0.22288571, -0.08834986],
       [ 1.36745826,  0.67940994,  0.6952116 ,  0.18046883]])
```

Note: the boolean array must be of the same length as the array axis it’s indexing.

In these examples, I select from the rows where names == 'Bob' and index the columns, too:

```
data[names == 'Bob', 2:]
array([[ 0.22288571, -0.08834986],
       [ 0.6952116 ,  0.18046883]])

data[names == 'Bob', 3]
array([-0.008834986,0.18046883])
```

To select everything but 'Bob', you can either use != or negate the condition using ~: e.g. `data[names != 'Bob', 2:]` or `data[~(names == 'Bob'),2:]`.

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like

`&` (and) and `|` (or): `mask = (names == 'Bob') | (names == 'Will')`

`data[mask]`

```
array([[-1.67943596, -0.22358419,  0.22288571, -0.08834986],
       [ 0.08926212, -0.59101301, -1.71152208, -0.27493075],
       [ 1.36745826,  0.67940994,  0.6952116 ,  0.18046883],
       [ 1.2781309 , -0.26204851,  1.14487204,  1.49803399]])
```

Setting values with boolean arrays works in a common-sense way. To set all of the negative values in data to 0 we need only do:

`data[data<0] = 0` yields

```
array([[0.        , 0.        , 0.22288571, 0.        ],
       [0.46251834, 0.7372516 , 0.24279069, 1.3422404 ],
       [0.08926212, 0.        , 0.        , 0.        ],
       [1.36745826, 0.67940994, 0.6952116 , 0.18046883],
       [1.2781309 , 0.        , 1.14487204, 1.49803399],
       [1.12997958, 0.        , 0.96524459, 0.40287942],
       [0.        , 0.        , 0.        , 0.        ]])
```

Setting whole rows or columns using a one-dimensional boolean array is also easy:

`data[names != 'Joe']` yields

```
array([[7.        , 7.        , 7.        , 7.        ],
       [0.46251834, 0.7372516 , 0.24279069, 1.3422404 ],
       [7.        , 7.        , 7.        , 7.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [1.12997958, 0.        , 0.96524459, 0.40287942],
       [0.        , 0.        , 0.        , 0.        ]])
```

Boolean Slicing with np.where

Another way to do very complex slicing is with the `where` function (e.g. `np.where[set of boolean conditions]`). E.g. you can write the previous slicing as:

```
bad=np.where(names != 'Joe')
data[bad]=7
```

The `where` function allows for highly complex slicing. E.g.

```
data2=data.copy()
data2
#array([[-1.67943596, -0.22358419,  0.22288571, -0.08834986],
#       [ 0.46251834,  0.7372516 ,  0.24279069,  1.3422404 ],
#       [ 0.08926212, -0.59101301, -1.71152208, -0.27493075],
#       [ 1.36745826,  0.67940994,  0.6952116 ,  0.18046883],
#       [ 1.2781309 , -0.26204851,  1.14487204,  1.49803399],
#       [ 1.12997958, -1.7684264 ,  0.96524459,  0.40287942],
#       [-0.91130181, -0.68538892, -0.49644622, -1.33955245]])
bad=np.where( (names == 'Will') | ( (data2[:,0] > 0) & (names == 'Bob')))
data2[bad]=9
data2
#array([[-1.67943596, -0.22358419,  0.22288571, -0.08834986],
       [ 0.46251834,  0.7372516 ,  0.24279069,  1.3422404 ],
       [ 9.        ,  9.        ,  9.        ,  9.        ],
       [ 9.        ,  9.        ,  9.        ,  9.        ],
       [ 9.        ,  9.        ,  9.        ,  9.        ],
       [ 1.12997958, -1.7684264 ,  0.96524459,  0.40287942],
       [-0.91130181, -0.68538892, -0.49644622, -1.33955245]])
```

here, `data2[:,0]` originally equals `array([-1.67943596, 0.46251834, 0.08926212, 1.36745826, 1.2781309 , 1.12997958, -0.91130181])` (i.e. along a column).

Expressing Conditional Logic as Array Operations with np.where

Where can do even more complex operations. Essentially it is a vectorized version of the ternary expression `x if condition else y`. Before, we were just using it as `x if condition` just to find indexes of an array. Now we can use it to do more complex operations.

E.g. say we have two arrays of values and a boolean array:

```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from xarr whenever the corresponding value in cond is True, and otherwise take the value from yarr. A list comprehension doing this might look like:

```
result = [(x if c else y)
 for x, y, c in zip(xarr, yarr, cond)]
```

For our simple example, this is fine but it is slow for large arrays and hard to pull off for multi-dimensional arrays. `np.where` is the solution:

```
result = np.where(cond, xarr, yarr)
#array([ 1.1, 2.2, 1.3, 1.4, 2.5])
```

Another example:

```
data3=data.copy()
good=data3 > 0
data3c=np.where(good,data3,-1*data3)
data3c
#array([[1.67943596, 0.22358419, 0.22288571, 0.08834986],
       [0.46251834, 0.7372516 , 0.24279069, 1.3422404 ],
       [0.08926212, 0.59101301, 1.71152208, 0.27493075],
       [1.36745826, 0.67940994, 0.6952116 , 0.18046883],
       [1.2781309 , 0.26204851, 1.14487204, 1.49803399],
       [1.12997958, 1.7684264 , 0.96524459, 0.40287942],
       [0.91130181, 0.68538892, 0.49644622, 1.33955245]])
```

Array Shapes and Reshaping

NumPy arrays have given dimensions: the `shape` of a NumPy array distinguishes vectors or matrices of different sizes. The attribute `.shape` for a NumPy array returns its dimensionality in the form of a tuple of its dimensions. E.g. for the matrix `a=np.array([[3,4,5],[6,7,8]])` the shape can be returned from `a.shape #yields (2,3)`. A vector returns `([number],)`: e.g. `a=np.array([3,4,5,6,7,8])` returns as `a.shape #(6,)`.

reshape

We can give a view of the array where we change the dimensions of the array while keeping the same number of elements with the `.reshape` function.

E.g. for `arr=np.array([8,6,7,5,3,0])` we can reshape it as `arr.reshape(2,3)`, which returns `array([[8, 6, 7], [5, 3, 0]])`. Figure 4.2 in Fuhrer gives a nice overview of what happens when you `reshape` an array different ways. E.g. for `arr` in the above example, we get the following results from different reshapings:

```
arr=np.array([8,6,7,5,3,0])

arr.reshape(1,6)
#yields array([[8, 6, 7, 5, 3, 0]]) ... i.e. the same as before

arr.reshape(6,1)
#yields a column matrix
#array([[8],
#       [6],
#       [7],
#       [5],
#       [3],
#       [0]])

arr.reshape(2,3)
#yields
#array([[8, 6, 7],
#       [5, 3, 0]])

arr.reshape(3,2)
#yields
#array([[8, 6],
#       [7, 5],
#       [3, 0]])
```

Now say you have an array of some dimensions and you want to reshape it so it has, say, two columns and "some" number of rows but that you don't know the number of rows beforehand, or vice versa. I.e. you specify one shape but want Python to specify (er, *compute*) the other one. You can let Python determine this by setting the second paramter to `-1`. E.g.

```
v=np.array([1,2,3,4,5,6,7,8])
M=v.reshape(2,-1)
M.shape
#returns (2,4)
M=v.reshape(-1,2)
M.shape
#returns (4,2)
```

Transpose

A special form of reshaping used often in linear algebra is ***transposing***, which shwithces the two shape elements of the matrix (e.g. columns--> rows; rows--> columns). The transpose of a matrix ***A*** is a matrix ***B*** such that: $B_{ij} = B_{ji}$.

The simplest to do a transpose with a NumPy array is ... `array.T`, e.g. for a matrix A, the transpose B is `B= A.T`. An alternative way is with the `np.transpose()` operation: e.g. `B=np.transpose(A)`.

Note that transposing just returns a view of the array: it does not copy.

Array transposing becomes very useful later when we try to do matrix operations. E.g. when computing the inner matrix product using `np.dot`:

```
a=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
result1=np.dot(a,a.T)
result1
#array([[ 14,  32,  50,  68],
       [ 32,  77, 122, 167],
       [ 50, 122, 194, 266],
       [ 68, 167, 266, 365]])
result1.shape
#(4,4)

result2=np.dot(a.T,a)
result2
#array([[166, 188, 210],
       [188, 214, 240],
       [210, 240, 270]])
result2.shape
#(3, 3)
```

flatten **and** ***ravel***

With `reshape` we have found a way to convert a 1-D vector into 2-D matrix: i.e. reshape-ing an array to a higher dimension. We can also do the opposite -- converting a 2+D array into a 1-D vector -- using the `flatten` or `ravel` function. These two functions largely do the same thing (as described above). The only difference is that `ravel` does NOT produce a copy of the underlying values if the values in the result were contiguous in the original array, while `flatten` always returns a copy of the data.

Here are examples of `flatten` and `ravel` in action:

```
arr = np.arange(15).reshape((5, 3))
arr
#array([[ 0, 1, 2],
#[ 3, 4, 5],
#[ 6, 7, 8],
#[ 9, 10, 11],
#[12, 13, 14]])

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

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

A short caveat ... Note that data can be reshaped or raveled in different orders. One order is "row-major" order, where values within each row of data are stored in adjacent memory locations (you traverse higher dimensions first). The alternative to row major ordering is column major order, which means that values within each column of data are stored in adjacent memory locations (you traverse higher dimensions last). "Row major" order is traditionally known as "C order" and "column major" order is traditionally known as "Fortran order".

By default, NumPy arrays are created in *row major* order. But both `ravel` and `flatten` have an order keyword that allows you to switch to *column major* order. E.g.

```
arr = np.arange(12).reshape((3, 4))
arr
#array([[ 0, 1, 2, 3],
#[ 4, 5, 6, 7],
#[ 8, 9, 10, 11]])

arr.ravel() #default ordering
#array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

arr.ravel('F')
#array([ 0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11])
```

The Axis Keyword

Before proceeding to the next section, it's worth highlighting a particularly important keyword for many NumPy functions, since it crops up everwhere: `axis`. Usually this is invoked as `np.[functionname](array,axis=[some number])`. Basically it means "apply the vectorized NumPy function only along a given axis instead of for the array as a whole"

You will immediately see that this is extremely powerful.

For 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=np.sum(arr)
#-4.255006232632961

arr2=np.sum(arr,axis=0)
#array([-2.14937506, -2.6891966 ,  0.55341809,  0.03014734])

arr2=np.sum(arr,axis=1)
#array([ 1.11983383, -0.61894889, -2.60184216, -1.11992777, -1.03412125])
```

In this case, for "axis=0", NumPy performs a sum over *each row in a column*. Since there are 4 columns, the shape and len of the resulting array `arr2` is (4,) and 4, respectively. For "axis=1", NumPy performs a sum over *each column in a row*. Here's another example with the same array

```
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=np.median(arr)
#-0.2603516489786638

arr2=np.median(arr,axis=0)
#array([-0.68343715, -0.43989698,  0.38639884,  0.16589074])

arr2=np.median(arr,axis=1)
array([ 0.12643153, -0.0162271 , -0.48176499, -0.3932461 , -0.26587086])
```

You can also call combinations of axes: e.g. `np.median(array,axis=[0,1]`.