NumPy Snippets

Applications and examples...

Published - 2016-04-xx
Dan Patterson

The code to the right provides a simple example of constructing arrays which contain nodata values... masked arrays.

```
# coding: utf-8
"""
Script:  masked_array_notes.py
"""
import numpy as np
np.set_printoptions(edgeitems=3,linewidth=80,
                    precision=2,suppress=True,threshold=5)

a = np.array([0,1,-99,3,4,5,6,-99,8,9,10,11]).reshape(3,4)
m = (np.where(a==-99,True,False)).astype(int)
b0 = np.ma.MaskedArray(a, mask=m, fill_value=-99)
b1 = b0*2
b2 = b0*3
c = np.ma.vstack((b0,b1,b2)).reshape(3,3,4)
c.fill_value = -99
d = np.array([0,-99,2,3,4,5,6,7,8,9,10,-99]).reshape(3,4)
m0 = (np.where(d==-99,True,False))#.astype(int)
e0 = np.ma.MaskedArray(d, mask=m0, fill_value=-99)

args = [a, b0.filled(),
        np.ma.mean(b0), np.ma.mean(b1),
        np.ma.mean(b2), np.ma.mean([b0,b1,b2]),
        c.filled(), np.ma.mean(c,axis=0),
        m.astype(bool), m.astype(int),
        b0.filled(),e0.filled(),
       (b0+e0).filled()]
print(frmt.format(*args))
```

Array **a,** is a 3x4 array with nodata values denoted by -99. A nodata value must be a number that cannot be a valid value in a dataset. Often it is made ridiculously large or small when used with the representation of physical properties (ie. elevation, temperature)

There are a couple of ways to create masked arrays, but they all use the numpy ma (np.ma) module. In these examples, I specifically use the MaskedArray class which uses a data set, a mask and a fill value. Array b0 exemplifies this.

The data can be obtained in any fashion that can be employed in conventional arrays. The mask can the conditions in another array or derived from the input data. In the examples, a value of -99 was used and this could represent anything, perhaps a water body in a digital elevation model.

If you create an array from a masked array, as is the case for arrays b1 and b2, they inherent the parent array properties. So far the arrays I have created are all 2D arrays. By adding the third dimension, you can model the changes in space over time. Array c employ's the masked equivalent of the ndarray vstack function to stack the three arrays. Each array has a shape of (3,4) so the vstack is simply reshaped to incorporate the 3rd dimension (3,3,4). All the input arrays are nicely stacked in space like a pad of paper. Too finish off the possibilities, I created a 4th array which has different

positions for the nodata values. When working with stacked arrays with different positional nodata values, they can both be preserved or treated in different ways.

So to continue, some of the properties of the arrays are presented. Array **a** is a standard ndarray from outward appearance and this is confirmed by its type.

Array **b0**, is a masked array representation of the previous array. A masked array consists of data, a nodata value and a boolean mask indicating which array positions are occupied by nodata values and which are not.

The array mask (b0.mask) is a boolean array. Boolean values can be converted to an integer counterpart by using a type conversion. This particular array uses a fill value of -99. The fill value is used to occupy the -- positions in the string representation of the array. It is a good practice to make the nodata and the fill value the same, however, there is no requirement to do this.

In brief, you can:

- view the array with or without its mask
- view the array in string or filled form
- show the mask as boolean or integer values.

By default, masked arrays only perform operations on valid array positions, ignoring the values of nodata but accounting for the reduction in the number of valid cells.

```
>>> a
array([[  0,   1, -99,   3],
       [  4,   5,   6, -99],
       [  8,   9,  10,  11]])

>>> type(a)
<type 'numpy.ndarray'>

>>> b0
masked_array(data =
 [[0 1 -- 3]
 [4 5 6 --]
 [8 9 10 11]],
             mask =
 [[False False  True False]
 [False False False  True]
 [False False False False]],
       fill_value = -99)

>>> b0.mask
array([[False, False,  True, False],
       [False, False, False,  True],
        [False, False, False, False]],
dtype=bool)

>>> b0.mask.astype(int)
array([[0, 0, 1, 0],
       [0, 0, 0, 1],
       [0, 0, 0, 0]])

>>> b0.fill_value
-99

>>> b0.filled()
```

Some results for array operations are shown to the right. The simple means obviously don't include the nodata value of -99 since the derived mean values would be significantly impacted.

By performing the operation all at once, the overall mean can be determined.

The mean on a locational basis is obtained from the stacked array and calculated along the 3rd dimension (axis=0).

```
Single means
 np.ma.mean(b0) =  5.7
 np.ma.mean(b1) = 11.4
 np.ma.mean(b2) = 17.1

Mean for all
 np.ma.mean([b0,b1,b2]) = 11.4

Mean on a cell-by-cell basis
 np.ma.mean(c,axis=0)
[[0.0 2.0 -- 6.0]
 [8.0 10.0 12.0 --]
 [16.0 18.0 20.0 22.0]]
```

When the input arrays have different nodata locations, the output array will retain their respective positional locations. In the example, to the right, a simple addition shows this.

So the rest of masked arrays is more fancy stuff with most methods and functions just being a counterpart of the ndarray. It is akin to learning two languages for the price of one.

That is all for now.

```
Now, two masked arrays with different
mask locations
b0...
[[  0   1 -99   3]
 [  4   5   6 -99]
 [  8   9  10  11]]

e0....
[[  0 -99   2   3]
 [  4   5   6   7]
 [  8   9  10 -99]]

add b0+e0, note the mask locations
[[  0 -99 -99   6]
 [  8  10  12 -99]
 [ 16  18  20 -99]]
```