numpy continued

Ben Bolker

06 November 2019

operations along axes

- array axes are numbered
  - 0 = rows
  - 1 = columns
  - 2 = "slices"

  From here:

  > When you use the NumPy sum function with the axis parameter, the axis that you specify is the axis that gets collapsed.

examples

```
import numpy as np
a = np.arange(25).reshape((5,5))
print(a)
```

```
## [[ 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]]
```

```
print(a.sum())       ## axis=None, collapse everything
```

```
## 300
```

```
print(a.sum(axis=0)) ## sum *across* rows, collapse rows
```

```
## [50 55 60 65 70]
```

```
print(a.sum(axis=1)) ## sum *across* columns, collapse columns
```

```
## [ 10  35  60  85 110]
```

try a 3-D array

```
b = np.arange(24).reshape((2,3,4))
print(b)  ## 2 slices, 3 rows, 4 columns
```

```
## [[[ 0  1  2  3]
##   [ 4  5  6  7]
```

```
##   [ 8  9 10 11]]
##
##  [[12 13 14 15]
##   [16 17 18 19]
##   [20 21 22 23]]]
```

```
print(b.sum())
```

```
## 276
```

```
print(b.sum(axis=0))
```

```
## [[12 14 16 18]
##  [20 22 24 26]
##  [28 30 32 34]]
```

```
print(b.sum(axis=1))
```

```
## [[12 15 18 21]
##  [48 51 54 57]]
```

```
print(b.sum(axis=2))
```

```
## [[ 6 22 38]
##  [54 70 86]]
```

broadcasting

- **broadcasting** means matching up dimensions when doing operations on two non-matching arrays.
- errors may be thrown if arrays do not match in size, e.g.

```
np.array([1, 2, 3]) + np.array([4, 5])
## ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

- arrays that do not match in the number of **dimensions** will be broadcast (to perform mathematical operations)
- the smaller array will be repeated as necessary

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

```
## [[0. 5.]
##  [2. 7.]
##  [4. 9.]]
```

- sometimes it doesn't work

```
c = np.arange(3)
```

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

- you could reshape it:

```
a + c.reshape(3,1)
```

```
## array([[1., 2.],
##        [4., 5.],
##        [7., 8.]])
```

- or use slicing with `np.newaxis`

```
print(c)
```

```
## [0 1 2]
```

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

```
## [0 1 2]
```

```
print(c[np.newaxis,:])
```

```
## [[0 1 2]]
```

```
print(c[:,np.newaxis])
```

```
## [[0]
##  [1]
##  [2]]
```

```
a + c[:,np.newaxis]
```

```
## array([[1., 2.],
##        [4., 5.],
##        [7., 8.]])
```

- think of `np.newaxis` as adding a new, *length-one* dimension

matrix and vector math

- dot products: use the `np.dot()` function

```
c = np.arange(4,7)
d = np.arange(-1,-4,-1)
print(np.dot(c,d))
```

```
## -32
```

- `.dot()` also works for matrix multiplication
- here we multiply a = (3x2) x e = (2x4) to get a 3x4 matrix

```
e = np.array([[1, 0, 2, -1], [0, 1, 2, -3]])
print(np.dot(a,e))
```

```
## [[  1.   2.   6.  -7.]
##  [  3.   4.  14. -15.]
##  [  5.   6.  22. -23.]]
```

more matrix math

- get transposes with `a.T` or `np.transpose(a)`
- the `linalg` submodule does non-trivial linear algebra: determinants, inverses, eigenvalues and eigenvectors

```
a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]])
print(np.linalg.det(a))
```

```
## -48.00000000000003
```

```
import numpy.linalg as npl  ## shortcut
npl.det(a)
```

```
## -48.00000000000003
```

inverses

```
print(npl.inv(a))
```

```
## [[ 0.22916667  0.04166667 -0.29166667]
##  [ 0.04166667 -0.08333333  0.58333333]
##  [-0.3125      0.125       0.125     ]]
```

```
m = np.dot(a,npl.inv(a))
print(m)
```

```
## [[1.00000000e+00 5.55111512e-17 0.00000000e+00]
##  [0.00000000e+00 1.00000000e+00 2.22044605e-16]
##  [0.00000000e+00 1.38777878e-17 1.00000000e+00]]
```

```
print(m.round())
```

```
## [[1. 0. 0.]
##  [0. 1. 0.]
##  [0. 0. 1.]]
```

eigenstuff

```
vals, vecs = npl.eig(a) ## unpack
print(vals)
```

```
## [ 8.85591316  1.9391628  -2.79507597]
```

```
print(vecs)
```

```
## [[-0.3663565  -0.54736745  0.25928158]
##  [-0.88949768  0.5640176  -0.88091903]
##  [-0.27308752  0.61828231  0.39592263]]
```

testing eigenstuff

We expect $Ae_0 = \lambda_a e_0$. Does it work?

```
e0 = vecs[:,0]
print(np.isclose(np.dot(a,e0),vals[0]*e0))
```

```
## [ True  True  True]
```

array iteration

- arrays can be iterated over in a similar way to lists
- the statement `for x in a:` will iterate over the *first* (0) axis of `a`

```
c = np.arange(2, 10, 3, dtype=float)
for x in c:
   print(x)
```

```
for x in a:
    print(a)
```

```
## [[4 2 0]
##  [9 3 7]
##  [1 2 1]]
## [[4 2 0]
##  [9 3 7]
##  [1 2 1]]
## [[4 2 0]
##  [9 3 7]
##  [1 2 1]]
```

logical arrays

- vectorized logical comparisons
- e.g. `a>0` gives an array of `bool`

```
a = np.array([2, 4, 6], float)
b = np.array([4, 2, 6], float)
result1 = (a > b)
result2 = (a == b)
print(result1, result2)
```

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

more examples

```
## compare with scalar
print(a>3)
```

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

- any and all and logical expressions work:

```
c = np.array([True, False, False])
d = np.array([False, False, True])
print(any(c), all(c))
```

```
## True False
```

```
print(np.logical_and(c,d))
```

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

```
print(np.logical_or(a>4,a<3))
```

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

selecting based on logical values

```
print(a[a >= 6])
```

```
## [6.]
```

```
sel = np.logical_and(a>5, a<9)
print(a[sel])
```

```
## [6.]
```

Set all elements of a that are >4 to 0:

```
a[a>4] = 0
print(a)
```

```
## [2. 4. 0.]
```

examples

Many examples here (or here), e.g.

-calculate the mean of the squares of the natural numbers up to 7 - create a 5 x 5 array with row values ranging from 0 to 1 by 0.2 - create a 3 x 7 array containing the values 0 to 20 and a 7 x 3 array containing the values 0 to 20 and matrix-multiply them: the result should be

```
## [[ 273  294  315]
##  [ 714  784  854]
##  [1155 1274 1393]]
```

gambler's ruin revisited

A slightly more compact version of the "gambler's ruin" code (i.e., a Markov chain starting at a particular value and going up or down by one unit at each step with a probability of p or $1 - p$ respectively.

```
import numpy as np
import numpy.random as npr
def gamblers_ruin(start=10,max=50,prob=0.5):
    ## iterate until you get to zero or max
    ## return tuple: (0 = lost, 1 = won,
    ##   [number of steps]
    i = 0
    x = start
    while x>0 and x<max:
        r = npr.uniform()
        x += np.sign(prob-r) ## +/- 1
    result = int(x>0)
    return(np.array((result, i)))
```

Simulate 1000 games:

```
sim = np.zeros((1000,2))
for i in range(1000):
    sim[i,:] = gamblers_ruin()
```

Evaluate results:

```
sim[:,0].mean()  ## prob of winning
```

```
## 0.196
```

```
sim[:,1].max()   ## max number of steps
```

```
## 0.0
```

```
sim[:,1].min()   ## min number of steps
```

```
## 0.0
```

```
lost = sim[:,0]==0
sim[lost,1].mean()
```

```
## 0.0
```

```
sim[np.logical_not(lost),1].mean()
```

```
## 0.0
```

We can try this for different starting values, upper bounds, probabilities of winning, etc.: see e.g. here for the derivation of the analytical solution:

$$P_i = \begin{cases} \frac{1-\left(\frac{q}{p}\right)^i}{1-\left(\frac{q}{p}\right)^N} \quad , & \text{if } p \neq q \\ \frac{i}{N} \quad , & \text{if } p = q = 0.5 \end{cases}$$

where i=starting value; p=winning probability; $q = 1 - p$; N=upper bound

coming soon: Mandelbrot set example