numpy continued

Ben Bolker

07 November 2019

## operations along axes

* array axes are numbered
  + 0 = rows
  + 1 = columns
  + 2 = “slices”

From [here](https://www.sharpsightlabs.com/blog/numpy-axes-explained/):

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 . 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](http://www.labri.fr/perso/nrougier/teaching/numpy.100/index.html) (or [here](http://mybinder.org/repo/rougier/numpy-100/notebooks/100_Numpy_exercises.ipynb)), 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 or 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

## 0.198

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](http://www.columbia.edu/~ks20/FE-Notes/4700-07-Notes-GR.pdf) for the derivation of the analytical solution:

where =starting value; =winning probability; ; =upper bound

# numerics

## 

* In Python, numbers are stored as binary digits (bits).
* If n bits are available to store a **signed** integer, we use one bit to indicate the sign; this gives room to store **signed** values between and
* So, 64 bits can be used to store any integer between -9223372036854775808 and 9223372036854775807 (since = 9223372036854775807). Fortunately, base Python automatically uses as many bits as necessary to store arbitrary-length integers

a = 2 \*\* 63 - 1  
b = a \* 100000  
print("a = ",a, ", b = ",b)

## a = 9223372036854775807 , b = 922337203685477580700000

* In other languages, and with numpy arrays, you need to be careful!
* The default type for integers within numpy is int32 or int64 but this might depend on your hardware/operating system

a = np.array([2 \*\* 63 - 1])  
b = np.array([2 \*\* 31 - 1])  
print(a.dtype, b.dtype)

## int64 int64

* If you’re not using huge integers (i.e. > ), you don’t need to worry
* You have [lots of choices](https://docs.scipy.org/doc/numpy/user/basics.types.html), including
  + int8, int16, int32, int64
  + **unsigned** values: uint8, uint16, uint32 uint64
* for small sizes, or huge numbers, you can get **overflow**

a = np.array([1], dtype="int8") ## 8-bit integer (-127 to 128)  
print(bin(a[0]))

## 0b1

a[0] = 127  
print(bin(a[0]))

## 0b1111111

a[0] += 1  
print(bin(a[0]))

## -0b10000000

print(a)

## [-128]

**be careful** ([obligatory xkcd](https://xkcd.com/571/))

## floats

* Floating point numbers are represented in computer hardware as **binary fractions** plus
* Many decimal fractions cannot be represented exactly as binary fractions
* This can lead to unexpected or suprising results.

print("2/3 = ",2 / 3," 2/3 + 1 =",2/3 + 1, "\n",  
" 5/3 =", 5/3)

## 2/3 = 0.6666666666666666 2/3 + 1 = 1.6666666666666665   
## 5/3 = 1.6666666666666667

print("1.13 - 1.1 =", 1.13 - 1.1, "\n3.13 - 1.1 =", 3.13 - 1.1)

## 1.13 - 1.1 = 0.029999999999999805   
## 3.13 - 1.1 = 2.03

print("1+1e-15 =",1+1e-15, "\n1+1e-16 =",1+1e-16)

## 1+1e-15 = 1.000000000000001   
## 1+1e-16 = 1.0

a = float(1234567890123456)  
print("a=",a,"\na\*10=",a\*10)

## a= 1234567890123456.0   
## a\*10= 1.234567890123456e+16

* None of these results are errors: they are an inevitable outcome of finite precision
* Small differences **might** not matter, but they can accumulate, and

sqrt2 = np.sqrt(2)  
sqrt2\*\*2==2.0

## False

np.isclose(sqrt2\*\*2,2.0)

## True

* floating point values are stored as a **mantissa** (digits) and an **exponent**

import sys  
sys.float\_info()

* max=1.7976931348623157e+308 (the largest float that can be stored)
* max\_exp=1024 (so 11 bits are needed to store the signed exponent)
* max\_10\_exp=308
* min=2.2250738585072014e-308 (closest to zero [almost])
* min\_10\_exp=-307
* dig=15 (number of decimal digits)
* mant\_dig=53 (bits in mantissa)
* epsilon=2.220446049250313e-16 (smallest number such that 1+x > x)

## overflow and underflow

x = 1e308  
small\_x = 2e-323  
print("x\*1000=",x\*1000,  
 "\nx\*1000-x\*1000=",x\*1000-x\*1000,  
 "\nsmall\_x/1000",small\_x/1000)

## x\*1000= inf   
## x\*1000-x\*1000= nan   
## small\_x/1000 0.0

inf means “infinity” and nan means “not a number”

## What should you do instead?

* devise a more stable algorithm (e.g. one that adds items in increasing order)
* work on the log scale (i.e. add log values rather than multiplying values)
* use extended/arbitrary precision floats: decimal module (built in), or mpmath
* **always be careful comparing floating point**

## higher precision

* temptation is just to increase precision
  + float128 in numpy
  + mpmath module for **arbitrary-precision** numbers (but infinite precision!)

import mpmath  
print(+1\*mpmath.pi)

## 3.14159265358979

mpmath.mp.dps=1000  
print(+1\*mpmath.pi)

## 3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117067982148086513282306647093844609550582231725359408128481117450284102701938521105559644622948954930381964428810975665933446128475648233786783165271201909145648566923460348610454326648213393607260249141273724587006606315588174881520920962829254091715364367892590360011330530548820466521384146951941511609433057270365759591953092186117381932611793105118548074462379962749567351885752724891227938183011949129833673362440656643086021394946395224737190702179860943702770539217176293176752384674818467669405132000568127145263560827785771342757789609173637178721468440901224953430146549585371050792279689258923542019956112129021960864034418159813629774771309960518707211349999998372978049951059731732816096318595024459455346908302642522308253344685035261931188171010003137838752886587533208381420617177669147303598253490428755468731159562863882353787593751957781857780532171226806613001927876611195909216420198

but you will often be disappointed