Assn2_Manish_Raut_24-27-21

October 6, 2024

0.0.1 DSTT Assignment2

0.0.2 Manish Raut

0.0.3 Reg No.: 24-27-21

0.0.4 Branch : Mtech(Data Science)

[88]:
```python
import numpy as np
```

0.0.5 Question 1

a) Create a variable named var1 that stores an array of numbers from 0 to 30, inclusive. Print var1 and its shape. Hint : arange

[89]:
```python
var1 = np.arange(31)
var1
```

[89]:
```
array([ 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, 25, 26, 27, 28, 29, 30])
```

b) Change var2 to a validly-shaped two-dimensional matrix and store it in a new variable called var2. Print var2 and its shape. Hint: Use the reshape function

[90]:
```python
var2 = np.arange(30)
var2
```

[90]:
```
array([ 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, 25, 26, 27, 28, 29])
```

[91]:
```python
var2 = var2.reshape(3,10)
var2
```

[91]:
```
array([[ 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, 25, 26, 27, 28, 29]])
```

c) Create a third variable, var3 that reshapes it into a valid three-dimensional shape. Print var3 and its shape.

[92]:
```
var3 = np.arange(27)
var3
```

[92]:
```
array([ 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, 25, 26])
```

[93]:
```
var3 = var3.reshape(3,3,3)
var3
```

[93]:
```
array([[[ 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, 25, 26]]])
```

d) Use two-dimensional array indexing to set the first value in the second row of var2 to -1. Now look at var1 and var3. Did they change? Explain what's going on. (Hint: does reshape return a view or a copy?)

[94]:
```
var2[1,0]=-1
var2
```

[94]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [-1, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])
```

[95]:
```
print(var1)
print(var3)
```

```
[ 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 25 26 27 28 29 30]
[[[ 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 25 26]]]
```

e) Another thing that comes up a lot with array shapes is thinking about how to aggregate over specific dimensions. Figure out how the NumPy sum function works (and the axis argument in particular) and do the following:
   (i) Sum var3 over its second dimension and print the result.
   (ii) Sum var3 over its third dimension and print the result.
   (iii) Sum var3 over both its first and third dimensions and print the result.

[96]:
```
sumvar3 = np.sum(var3,axis=1)
print(sumvar3)
```

```
[[ 9 12 15]
 [36 39 42]
 [63 66 69]]
```

[97]:
```
 sum_over_third=np.sum(var3,axis=2)
 print(sum_over_third)
```

```
[[ 3 12 21]
 [30 39 48]
 [57 66 75]]
```

[98]:
```
sum_over_first_second=np.sum(var3,axis=(0,2))
print(sum_over_first_second)
```

```
[ 90 117 144]
```

f) Write code to do the following:
   (i) Slice out the second row of var2 and print it.
   (ii) Slice out the last column of var2 using the -1 notation and print it.
   (iii) Slice out the top right 2×2 submatrix of var2 and print it.

[99]:
```
 print(var2[1:2])
 print(var2[0:,-1])
 print(var2[:2,-2:])
```

```
[[-1 11 12 13 14 15 16 17 18 19]]
[ 9 19 29]
[[ 8  9]
 [18 19]]
```

0.0.6 Question 2

a) The most basic kind of broadcast is with a scalar, in which you can perform a binary operation (e.g., add, multiply, . . .) on an array and a scalar, the effect is to perform that operation with the scalar for every element of the array. To try this out, create a vector 1, 2, . . . , 10 by adding 1 to the result of the arange function.

[100]:
```
res = np.arange(0,10,1)
res_broadcasted = arr+1
print(arr_broadcasted)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

b) Now, create a 10 × 10 matrix A in which Ai j = i + j. You'll be able to do this using the vector you just created, and adding it to a reshaped version of itself.

[101]:
```
A = res+res.reshape(10,1)
print(A)
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [ 1  2  3  4  5  6  7  8  9 10]
 [ 2  3  4  5  6  7  8  9 10 11]
 [ 3  4  5  6  7  8  9 10 11 12]
 [ 4  5  6  7  8  9 10 11 12 13]
 [ 5  6  7  8  9 10 11 12 13 14]
 [ 6  7  8  9 10 11 12 13 14 15]
 [ 7  8  9 10 11 12 13 14 15 16]
 [ 8  9 10 11 12 13 14 15 16 17]
 [ 9 10 11 12 13 14 15 16 17 18]]
```

c) A very common use of broadcasting is to standardize data, i.e., to make it have zero mean and unit variance. First, create a fake "data set" with 50 examples, each with five dimensions. import numpy.random as npr data = np.exp(npr.randn (50 , 5))

[102]:
```
data = np.exp(np.random.randn(50,5))
```

d) You don't worry too much about what this code is doing at this stage of the course, but for completeness: it imports the NumPy random number generation library, then generates a 50 × 5 matrix of standard normal M.Tech. Data Science and Modelling & Simulation random variates and exponentiates them. The effect of this is to have a pretend data set of 50 independent and identically-distributed vectors from a log-normal distribution.

[103]:
```
means = np.mean(data,axis=0)
sd = np.std(data,axis=0)
```

e) Now, compute the mean and standard deviation of each column. This should result in two vectors of length 5. You'll need to think a little bit about how to use the axis argument to mean and std. Store these vectors into variables and print both of them.

[104]:
```
print('Mean= '+str(means))
print('Standard Diviation= '+str(sd))
```

```
Mean= [1.51667502 1.72754954 1.8706754  1.47229786 1.7419575 ]
Standard Diviation= [2.24328134 3.26736521 2.18162767 1.38118514 1.83073208]
```

f) Now standardize the data matrix by
   1) subtracting the mean off of each column, and
   2) dividing each column by its standard deviation. Do this via broadcasting, and store the result in a matrix called normalized. To verify that you successfully did it, compute the mean and standard deviation of the columns of normalized and print them out.

[105]:
```
std_data=(data-means)/sd
```

[106]:
```
new_means = np.mean(std_data,axis=0)
new_sd = np.std(std_data,axis=0)
print(new_means)
print(new_sd)
```

```
[ 9.54791801e-17  6.07847106e-17  2.19824159e-16 -2.44249065e-16
  1.97619698e-16]
[1. 1. 1. 1. 1.]
```

0.0.7 Question 3

a) A Vandermonde matrix is a matrix generated from a vector in which each column of the matrix is an integer power starting from zero. So, if I have a column vector [x1, x2, . . . , xN] T, then the associated (square) Vandermonde matrix would be Use what you learned about broadcasting in the previous problem to write a function that will produce a Vandermonde matrix for a vector [1, 2, . . . ,N]T for any N. Do it without using a loop. Here's a stub to get you started: def vandermonde (N): vec = np.arange (N) +1 # write your code here . Use your function for N = 12, store it in variable named vander, and print the result.

[107]:
```
def vander_Function(N):
    vec=np.arange(N)+1
    vander=vec.reshape(N,1)
    vander=vander**np.arange(N)
    return vander

vander=vander_Function(12)
print(vander)
```

```
[[          1           1           1           1           1
            1           1           1           1           1
            1           1]
 [          1           2           4           8          16
           32          64         128         256         512
         1024        2048]
 [          1           3           9          27          81
          243         729        2187        6561       19683
        59049      177147]
 [          1           4          16          64         256
         1024        4096       16384       65536      262144
      1048576     4194304]
 [          1           5          25         125         625
         3125       15625       78125      390625     1953125
      9765625    48828125]
 [          1           6          36         216        1296
         7776       46656      279936     1679616    10077696
     60466176   362797056]
 [          1           7          49         343        2401
        16807      117649      823543     5764801    40353607
    282475249  1977326743]
```

```
 [           1            8           64          512         4096
         32768       262144      2097152     16777216    134217728
    1073741824   8589934592]
 [           1            9           81          729         6561
         59049       531441      4782969     43046721    387420489
    3486784401  31381059609]
 [           1           10          100         1000        10000
        100000      1000000     10000000    100000000   1000000000
   10000000000 100000000000]
 [           1           11          121         1331        14641
        161051      1771561     19487171    214358881   2357947691
   25937424601 285311670611]
 [           1           12          144         1728        20736
        248832      2985984     35831808    429981696   5159780352
   61917364224 743008370688]]
```

b) Now, let's make a pretend linear system problem with this matrix. Create a vector of all ones, of length 12 and call it x. Perform a matrix-vector multiplication of vander with the vector you just created and store that in a new vector and call it b. Print the vector b.

[108]:
```python
x = np.ones(12)
x
```

[108]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

[109]:
```python
b = np.dot(vander,x)
b
```

[109]:
```
array([1.20000000e+01, 4.09500000e+03, 2.65720000e+05, 5.59240500e+06,
       6.10351560e+07, 4.35356467e+08, 2.30688120e+09, 9.81706810e+09,
       3.53036921e+10, 1.11111111e+11, 3.13842838e+11, 8.10554586e+11])
```

c) First, solve the linear system the naïve way, pretending like you don't know x. Import numpy.linalg, invert V and multiply it by b. Print out your result. What should you get for your answer? If the answer is different than what you expected, write a sentence about that difference.

[110]:
```python
inv_v = np.linalg.inv(vander)
res = np.dot(inv_v,b)
print(res)
```

```
[-12.26953125   5.90625      15.421875    -6.03515625   2.05859375
   0.92675781   1.00244141   1.00010681   0.99997711   1.00000131
   0.99999996   1.        ]
```

d) Now, solve the same linear system using solve. Print out the result. Does it seem more or less in line with what you'd expect?

[111]:
```python
res2 = np.linalg.solve(vander,b)
print(res2)
```

```
[0.98080503 1.05139768 0.94513807 1.03151945 0.98899166 1.00248816
 0.99962407 1.0000384  0.99999737 1.00000012 1.         1.        ]
```

```
[ ]:
```

Github Profile: https://github.com/Prosedus007

Github Repository :https://github.com/Prosedus007/Manish$_2$4 − 27 − 21