Assignment 2 (Data Science Tools and Techniqes, AM609)

Reg. No: 24-14-20

Name: Sunandan Sharma

Course: M.Tech Modelling & Simulation

Q1: a) Create a variable named var1 that stores an array of numbers from 0 to 30, inclusive. Print var1 and its shape.

In [2]:
```
import numpy as np
var1 = np.arange(31)
print(var1)
print("The shape is:",var1.shape)
```

```
[ 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]
The shape is: (31,)
```

b) Change var1 to a validly-shaped two dimensional matrix and store it in a new variable called var2. Print var2 and its shape.

In [3]:
```
var2 = np.delete(var1,0).reshape(6,-1)
print(var2)
print("The shape is:",var2.shape)
```

```
[[ 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]]
The shape is: (6, 5)
```

c) Create a third variable, var3 that reshapes var2 into a valid three dimensional shape. Print var3 and its shape.

In [4]:
```
var3 = np.delete(var1,0).reshape(2,3,-1)
print(var3)
print("The shape is:",var3.shape)
```

```
[[[ 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]]]
The shape is: (2, 3, 5)
```

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?)

In [5]:
```
var2[1,0]=-1
print("Updated var2 is:")
print(var2)
print("Array var1 after modifying var2:")
print(var1)
print("Array var3 after modifying var2:")
print(var3)
```

```
Updated var2 is:
[[ 1  2  3  4  5]
 [-1  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]
 [21 22 23 24 25]
 [26 27 28 29 30]]
Array var1 after modifying var2:
[ 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]
Array var3 after modifying var2:
[[[ 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]]]
```

The var1 and var3 has not changed after updating var2. The numpy function rehape here is returning a view.

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.

In [7]:
```
sum_var3_dim2 = var3.sum(axis=1)
print("Sum of var3 over its second dimension:\n", sum_var3_dim2)
sum_var3_dim3 = var3.sum(axis=2)
print("Sum of var3 over its third dimension:\n", sum_var3_dim3)
sum_var3_dim12=var3.sum(axis=(0,2))
print("Sum of var3 over its first and third dimension:\n", sum_var3_dim12)
```

```
Sum of var3 over its second dimension:
 [[18 21 24 27 30]
 [63 66 69 72 75]]
Sum of var3 over its third dimension:
 [[ 15  40  65]
 [ 90 115 140]]
Sum of var3 over its first and third dimension:
 [105 155 205]
```

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.

In [10]:
```
row2_of_var2=var2[1,:]
print("The second row of var2 is: ")
print(row2_of_var2)
col_last_var2=var2[:,-1:]
print("The last column of var2 is: ")
print(col_last_var2)
mat=var2[0:2,-2:]
print("The top right 2x2 submatrix of var2 is: ")
print(mat)
```

```
The second row of var2 is: 
[-1  7  8  9 10]
The last column of var2 is: 
[[ 5]
 [10]
 [15]
 [20]
 [25]
 [30]]
The top right 2x2 submatrix of var2 is: 
[[ 4  5]
 [ 9 10]]
```

Q2: 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 the operation with the scalar for every element of the array. To try this out, create a vector 1,2,3,...,10 by adding 1 to the result of the arange function.

In [3]:
```
import numpy as np

Arr1 = np.arange(10)+1
print(Arr1)
```

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

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

In [29]:
```
Arr2 = Arr1.reshape(10,1)
Arr3 = Arr1+Arr2
print(Arr3)
```

```
[[ 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]
 [10 11 12 13 14 15 16 17 18 19]
 [11 12 13 14 15 16 17 18 19 20]]
```

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.

In [9]:
```
import numpy.random as npr

data = np.exp(npr.randn ( 50 , 5 ) )
print("Data:")
print(data)
```

```
Data:
[[ 6.658129    0.81286557  1.20562622  1.92308062  0.45908714]
 [ 0.21494963  0.23059312  4.72971593  2.30865767  0.22681681]
 [ 0.43017036  0.17009394  0.52952579  3.00374149  0.29100251]
 [ 1.1679981   1.00640953  0.86539284  2.23946193  0.39208725]
 [ 0.83884504  0.80618404  0.328907    1.65149385  2.90637805]
 [ 0.56529767  0.71623637  3.51864261  0.82676241  0.15530989]
 [ 0.16588978  0.92562513  2.04869506  1.27808106  0.8029798 ]
 [ 0.86978291  2.3474971   1.6696924   1.62032216  1.57280157]
 [ 3.531185    1.01012076  0.66467236  0.12405558  0.4635516 ]
 [ 0.92937834  5.40242772  0.660536    0.8053234   0.93716791]
 [ 1.23243755  2.45538066  1.04737914  3.80175968  1.44004563]
 [ 0.98013738  1.9285947   2.3769456   0.29755941  1.84706769]
 [ 0.17401144  3.51049175  0.90673919  0.39887571  2.06573931]
 [ 0.79059013  3.08445158  0.67196243  0.86572085  4.81301686]
 [ 1.59553969  1.11356624  0.12868344  0.34739798  0.76348703]
 [ 0.93187842  4.38540243  1.20887573  2.3833126   1.08586096]
 [ 0.6827438   1.98265365  1.16805837  0.84496788  0.60834087]
 [ 0.55957774  2.90706499  1.35759841  1.36075919  0.77512909]
 [ 0.18180046  0.52440571  0.47516888  0.9189283   1.10898862]
 [ 1.61861803  1.23911938  1.22018377  1.04709446  1.47595755]
 [ 2.22114189  0.8877723   0.47543329  0.27556778  0.96012745]
 [ 0.41036835  0.73584131  2.74246234  0.38909225  0.98580177]
 [ 1.38611622  2.30228031  0.3138773   1.02149675  5.37498437]
 [ 4.99605827  1.29668903  0.61046639  1.27373425  2.12810873]
 [ 0.4519309   1.19525286  1.67140513  1.99492757  1.12615634]
 [ 0.76095274  0.18193105  0.0790993   1.36634653  0.57485493]
 [ 0.29734415  1.17137901  1.37995345  0.60582566  0.77224597]
 [ 0.17303381  1.8269292   0.24886477  0.38216044  0.31695639]
 [ 1.27976503  0.86295548  1.29685365  0.77727236  2.69310388]
 [ 2.12648366  0.35441278  0.19479202  0.10155898  1.44907417]
 [ 1.67705434  0.48264529  0.83154553  1.00077676  0.90454065]
 [ 0.47335391  1.05527999  0.97064094  0.97520389  5.09775759]
 [ 0.69750887  3.07081483  2.05292241  0.96282521  0.96053926]
 [ 4.25719113  0.4460186   0.41083734  2.74081974  0.81531606]
 [ 1.43712809  1.02614912  1.32638883  0.42534875  0.50693456]
 [ 0.84576915  0.74427439  1.534292    4.04429421  1.42087457]
 [ 0.39076964  0.10218259 11.255169    2.02156343  2.00649677]
 [ 3.32847005  0.70301355  0.69688979  1.65995481  1.82149378]
 [ 2.83176517  3.40385165  0.74010899  1.57406034  0.67368066]
 [ 0.78787175  0.57054792  0.33506092  0.42439224  0.60647303]
 [ 3.38398946  0.78709911  4.95141499  0.44824234  0.26716938]
 [ 4.28860309  1.70611754  2.1673744   0.38445634  0.37366841]
 [ 0.76837933  0.89926117  0.28618128  0.20764896  0.8023045 ]
 [ 0.15845823  0.18744739  0.23253158  0.5702493   0.50542567]
 [ 0.34492402  0.46505404  0.58172609  0.21540414  1.32811904]
 [ 2.6418767   0.45341703  2.46327671  1.68718772  1.24913323]
 [ 0.67530825  3.05629992  2.12627775  0.56865168  0.21736735]
 [ 0.131135    1.34348085  0.22450105  0.50639784  8.07269799]
 [ 1.67495544  1.17507959  0.6840257  14.69694624  1.11456589]
 [ 0.37719799  0.69903784  1.17262499  4.2182766   0.80347802]]
```

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 x 5 matrix of standard normal 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 distributions.

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.

In [11]:
```
mean = np.mean(data,axis=0)
sd = np.std(data,axis=0)
print("Mean:")
print(mean)
print("Standard deviation:")
print(sd)
```

```
Mean:
[1.3878773  1.395034   1.41679998 1.51136079 1.40240533]
Standard deviation:
[1.40754872 1.15412403 1.75550062 2.13179293 1.49156636]
```

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 the normalized and print it.

In [12]:
```
z_data = (data-mean)/sd
z_mean = np.mean(z_data,axis=0)
z_sd = np.std(z_data,axis=0)
print("Normalized Mean:")
print(z_mean)
print("Normalized Standard Deviation:")
print(z_sd)
```

```
Normalized Mean:
[-9.99200722e-17 -4.66293670e-17 -8.82627305e-17  7.54951657e-17
  1.33226763e-17]
Normalized Standard Deviation:
[1. 1. 1. 1. 1.]
```

Q3: a) A Vandermode matrix is a matrix generated from a vector in which each column of the matrix is an integer power starting from zero. 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 any loop. Do it without using a loop.Use your function for N = 12, store it in variable named vander, and print the result.

In [13]:
```
import numpy as np
def vandermonde (N):
    vec = (np.arange (N) +1).reshape(N,1)
```

```
    pow_vander = np.arange(N)
    return vec**pow_vander.astype(dtype="int64")

N = int(input("Enter the Number: "))
vander=vandermonde(N)
print("The Vandermonde Matrix is:")
print(vander)
```

```
Enter the Number: 12
The Vandermonde Matrix is:
[[           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.

In [14]:
```
x = np.ones(N)
b = np.matmul(vander,x).astype(dtype="int64").reshape(N,1)
print("b:")
print(b)
```

```
b:
[[          12]
 [        4095]
 [      265720]
 [     5592405]
 [    61035156]
 [   435356467]
 [  2306881200]
 [  9817068105]
 [ 35303692060]
 [111111111111]
 [313842837672]
 [810554586205]]
```

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.

In [15]:
```
print("Solving by multipling b with inverse of vander:")
x_1 = np.matmul(np.linalg.inv(vander),b)
print(x_1)
```

```
Solving by multipling b with inverse of vander:
[[-134.109375  ]
 [ 165.4453125 ]
 [ -26.921875  ]
 [  -1.171875  ]
 [   1.84765625]
 [   0.91162109]
 [   1.00439453]
 [   1.00006104]
 [   0.99997139]
 [   1.00000191]
 [   0.99999994]
 [   1.        ]]
```

The answer is different from what is expected because as the elements in Vandermonde Matrix is very big during the inversion of matrix process errors can be introduced (floating point issues). This may result in such errors.

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?

In [16]:
```
x_2 = np.linalg.solve(vander,b)
print("Solving by Numpy linear algebra function (solve):")
print(x_2)
```

```
Solving by Numpy linear algebra function (solve):
[[0.98080503]
 [1.05139768]
 [0.94513807]
 [1.03151945]
 [0.98899166]
 [1.00248816]
 [0.99962407]
 [1.0000384 ]
 [0.99999737]
 [1.00000012]
 [1.        ]
 [1.        ]]
```

This is more in line with the expectation.

Github Link:

https://github.com/Sunandanshrm/Sunandan_Sharma_24-14-20/blob/main/Assn2_Sunandan_Sharma_24-14-20.ipynb