**Linear Algebra**

Linear algebra is a branch of mathematics concerning vector spaces and linear mappings between such spaces. Simply, it explores line-like relationships. Practically every area of modern science approximates modeling equations with linear algebra. In particular, data science relies on linear algebra for machine learning, mathematical modeling, and dimensional distribution problem solving. Linear algebra concepts with examples include:

1. Vector spaces (vector points and vectors)
2. Vector math
3. Matrix math
4. Basic matrix transformations
5. Pandas matrix applications

*Vector Spaces*

A vector space is a collections of vectors. A vector is any quantity with magnitude and direction that determines the position of one point in space relative to another. Magnitude is the size of an object measures by movement, length, and/or velocity. Vectors can be added and multiplied (by scalars) to form new vectors. A scalar is any quantity with magnitude (size). In application, vectors are points in finite space.

Vector examples include breathing, walking, and displacement. Breathing requires diaphragm muscles to exert a force that has magnitude and direction. Walking requires movement in some direction. Displacement measures how far an object moves in a certain direction.

*Vector Math*

In vector math, a vector is depicted as a directed line segment whose length is its magnitude vector with an arrow indicating direction from tail to head. Tail is where the line segment begins and head is where it ends (the arrow). Vectors are the same if they have the same magnitude and direction.

To add two vectors a and b, start b where a finishes, and complete the triangle. Visually, start at some point of origin, draw a, start b from head of a, and the result is a line from tail of a to head of b. The 1st example illustrates vector addition as well as a graphic depiction of the process:

import matplotlib.pyplot as plt, numpy as np

def vector\_add(a, b):

return np.add(a, b)

def set\_up():

plt.figure()

plt.xlim(-.05, add\_vectors[0]+0.4)

plt.ylim(-1.1, add\_vectors[1]+0.4)

if \_\_name\_\_ == "\_\_main\_\_":

v1, v2 = np.array([3, -1]), np.array([2, 3])

add\_vectors = vector\_add(v1, v2)

set\_up()

ax = plt.axes()

ax.arrow(0, 0, 3, -1, head\_width=0.1, fc='b', ec='b')

ax.text(1.5, -0.35, 'a')

ax.set\_facecolor('honeydew')

set\_up()

ax = plt.axes()

ax.arrow(0, 0, 3, -1, head\_width=0.1, fc='b', ec='b')

ax.arrow(3, -1, 2, 3, head\_width=0.1, fc='crimson', ec='crimson')

ax.text(1.5, -0.35, 'a')

ax.text(4, -0.1, 'b')

ax.set\_facecolor('honeydew')

set\_up()

ax = plt.axes()

ax.arrow(0, 0, 3, -1, head\_width=0.1, fc='b', ec='b')

ax.arrow(3, -1, 2, 3, head\_width=0.1, fc='crimson', ec='crimson')

ax.arrow(0, 0, 5, 2, head\_width=0.1, fc='springgreen', ec='springgreen')

ax.text(1.5, -0.35, 'a')

ax.text(4, -0.1, 'b')

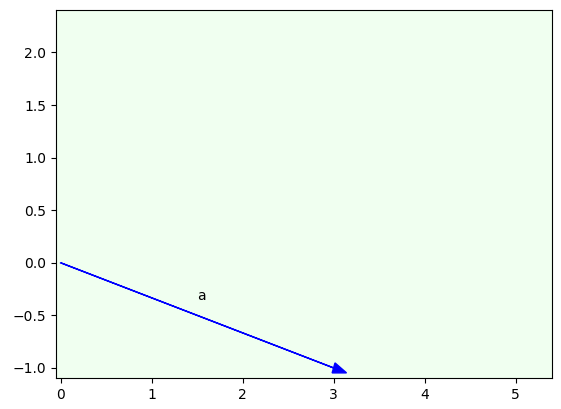
ax.text(2.3, 1.2, 'a + b')

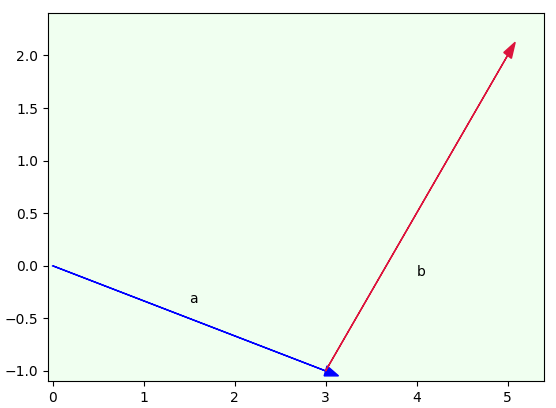
ax.text(4.5, 2.08, add\_vectors, color='fuchsia')

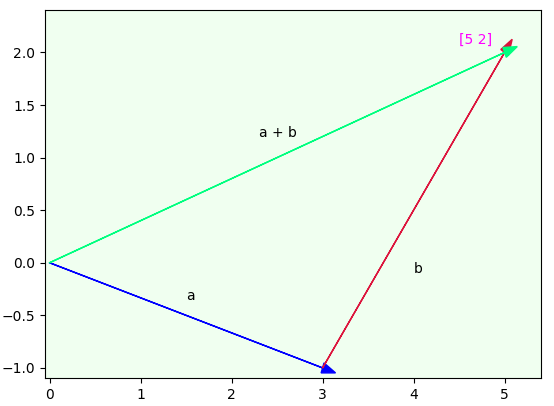
ax.set\_facecolor('honeydew')

plt.show()

Output:







The code begins by importing matplotlib and numpy libraries. Library matplotlib is a plotting library used for high quality visualization. Library numpy is the fundamental package for scientific computing. It is a wonderful library for working with vectors and matrices. The code continues with two functions – vector\_add() and set\_up(). Function vector\_add() adds two vectors. Function set\_up() sets up the figure for plotting. The main block begins by creating two vectors and adding them. The remainder of the code demonstrates graphically how vector addition works. First, it creates an axes() object with an arrow representing vector a beginning at origin (0, 0) and ending at (3, -1). It continues by adding text and a background color. Next, it creates a 2nd axes() object with the same arrow a, but adds arrow b (vector b) starting at (3, -1) and continuing to (2, 3). Finally, it creates a 3rd axes() object with the same arrows a and b, but adds arrow c (a + b) starting at (0, 0) and ending at (5, 2).

The 2nd example modifies the previous example by using subplots. Subplots divide a figure into a m x n grid for a different visualization experience.

import matplotlib.pyplot as plt, numpy as np

def vector\_add(a, b):

return np.add(a, b)

if \_\_name\_\_ == "\_\_main\_\_":

v1, v2 = np.array([3, -1]), np.array([2, 3])

add\_vectors = vector\_add(v1, v2)

f, ax = plt.subplots(3)

x, y = [0, 3], [0, -1]

ax[0].set\_xlim([-0.05, 3.1])

ax[0].set\_ylim([-1.1, 0.1])

ax[0].scatter(x,y,s=1)

ax[0].arrow(0, 0, 3, -1, head\_width=0.1, head\_length=0.07,

fc='b', ec='b')

ax[0].text(1.5, -0.35, 'a')

ax[0].set\_facecolor('honeydew')

x, y = ([0, 3, 5]), ([0, -1, 2])

ax[1].set\_xlim([-0.05, 5.1])

ax[1].set\_ylim([-1.2, 2.2])

ax[1].scatter(x,y,s=0.5)

ax[1].arrow(0, 0, 3, -1, head\_width=0.2, head\_length=0.1,

fc='b', ec='b')

ax[1].arrow(3, -1, 2, 3, head\_width=0.16, head\_length=0.1,

fc='crimson', ec='crimson')

ax[1].text(1.5, -0.35, 'a')

ax[1].text(4, -0.1, 'b')

ax[1].set\_facecolor('honeydew')

x, y = ([0, 3, 5]), ([0, -1, 2])

ax[2].set\_xlim([-0.05, 5.25])

ax[2].set\_ylim([-1.2, 2.3])

ax[2].scatter(x,y,s=0.5)

ax[2].arrow(0, 0, 3, -1, head\_width=0.15, head\_length=0.1,

fc='b', ec='b')

ax[2].arrow(3, -1, 2, 3, head\_width=0.15, head\_length=0.1,

fc='crimson', ec='crimson')

ax[2].arrow(0, 0, 5, 2, head\_width=0.1, head\_length=0.1,

fc='springgreen', ec='springgreen')

ax[2].text(1.5, -0.35, 'a')

ax[2].text(4, -0.1, 'b')

ax[2].text(2.3, 1.2, 'a + b')

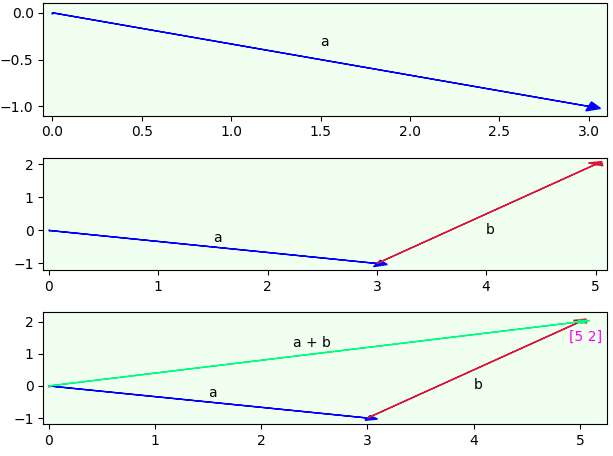
ax[2].text(4.9, 1.4, add\_vectors, color='fuchsia')

ax[2].set\_facecolor('honeydew')

plt.tight\_layout()

plt.show()

Output:



The code begins by importing matplotlib and numpy libraries. It continues with the same vector\_add() function. The main block creates three subplots with plt.subplots(3) and assigns to f and ax, where f represents the figure and ax represents each subplot (ax[0], ax[1], and ax[2]). Instead of working with one figure, the code builds each subplot by indexing ax. The code uses plt.tight\_layout() to automatically align each subplot.

The 3rd example adds vector subtraction. Subtracting two vectors is addition with the opposite (negation) of a vector. So, vector a minus vector b is the same as a + (-b). The code example demonstrates vector addition and subtraction for both 2 and 3 dimensional vectors.

import numpy as np

def vector\_add(a, b):

return np.add(a, b)

def vector\_sub(a, b):

return np.subtract(a, b)

if \_\_name\_\_ == "\_\_main\_\_":

v1, v2 = np.array([3, -1]), np.array([2, 3])

add = vector\_add(v1, v2)

sub = vector\_sub(v1, v2)

print ('2D vectors:')

print (v1, '+', v2, '=', add)

print (v1, '-', v2, '=', sub)

v1 = np.array([1, 3, -5])

v2 = np.array([2, -1, 3])

add = vector\_add(v1, v2)

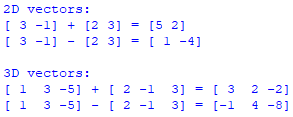
sub = vector\_sub(v1, v2)

print ('\n3D vectors:')

print (v1, '+', v2, '=', add)

print (v1, '-', v2, '=', sub)

Output:



The code begins by importing the numpy library. It continues with functions vector\_add() and vector\_subtract(), which add and subtract vectors respectively. The main block begins by creating two 2 dimensional vectors, and adding and subtracting them. It continues by adding and subtracting two 3 dimensional vectors. Any n-dimensional can be added and subtracted in the same manner.

Magnitude is measured by the distance formula. Magnitude of a single vector is measured from the origin (0, 0) to the vector. Magnitude between two vectors is measured from the 1st vector to the 2nd vector. The distance formula is the square root of ((the 1st value from the 2nd vector minus the 1st value from the 1st vector squared) plus (the 2nd value from the 2nd vector minus the 2nd value from the 1st vector squared)).

*Matrix Math*

A matrix is an array of numbers. Many operations can be performed on a matrix such as addition, subtraction, negation, multiplication, and division. The dimension of a matrix is its size in number of rows and columns in that order. That is, a 2 x 3 matrix has 2 rows and 3 columns. Generally, a m x n matrix has m rows and n columns. An element is an entry in a matrix. Specifically, an element in rowi and columnj of matrix A is denoted as ai,j. Finally, a vector in a matrix is typically viewed as a column. So, a 2 x 3 matrix has 3 vectors (columns) each with 2 elements. This is a very important concept to understand when performing matrix multiplication and/or using matrices in data science algorithms.

The 1st code example creates a numpy matrix, multiplies it by a scalar, calculates means row- and column-wise, creates a numpy matrix from numpy arrays, and displays it by row and element.

import numpy as np

def mult\_scalar(m, s):

matrix = np.empty(m.shape)

m\_shape = m.shape

for i, v in enumerate(range(m\_shape[0])):

result = [x \* s for x in m[v]]

x = np.array(result[0])

matrix[i] = x

return matrix

def display(m):

s = np.shape(m)

cols = s[1]

for i, row in enumerate(m):

print ('row', str(i) + ':', row, 'elements:', end=' ')

for col in range(cols):

print (row[col], end=' ')

print ()

if \_\_name\_\_ == "\_\_main\_\_":

v1, v2, v3 = [1, 7, -4], [2, -3, 10], [3, 5, 6]

A = np.matrix([v1, v2, v3])

print ('matrix A:\n', A)

scalar = 0.5

B = mult\_scalar(A, scalar)

print ('\nmatrix B:\n', B)

mu\_col = np.mean(A, axis=0, dtype=np.float64)

print ('\nmean A (column-wise):\n', mu\_col)

mu\_row = np.mean(A, axis=1, dtype=np.float64)

print ('\nmean A (row-wise):\n', mu\_row)

print ('\nmatrix C:')

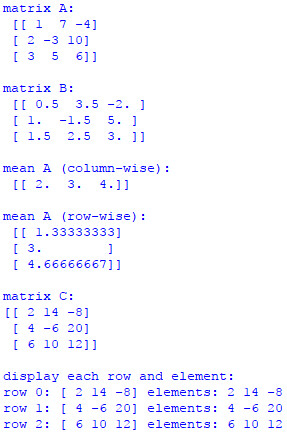
C = np.array([[2, 14, -8], [4, -6, 20], [6, 10, 12]])

print (C)

print ('\ndisplay each row and element:')

display(C)

Output:



The code begins by importing numpy. It continues with two functions – mult\_scalar() and display(). Function mult\_scalar() multiplies a matrix by a scalar. Function display() displays a matrix by row and each element of a row. The main block creates three vectors and adds them to numpy matrix A. B is created by multiplying scalar 0.5 by A. Next, means for A are calculated by column and row. Finally, numpy matrix C is created from three numpy arrays and displayed by row and element.

The 2nd code example creates a numpy matrix A, sums its columns and rows, calculates the dot product of two vectors, and calculates the dot product of two matrices. Dot product multiplies two vectors to get magnitude that can be used to compute lengths of vectors and angles between vectors. Specifically, the dot product of two vectors a and b is ax x bx + ay x by.

For matrix multiplication, dot product produces matrix C from two matrices A and B. However, two vectors cannot be multiplied when both are viewed as column matrices. To rectify this problem, transpose the 1st vector from A turning it into a 1 x n row matrix so it can be multiplied by the 1st vector from B and summed. The product is now well defined because the product of a 1 x n matrix with an n x 1 matrix is a 1 x 1 matrix (a scalar). To get the dot product, repeat this process for the remaining vectors from A and B. Numpy includes a handy function that calculates dot product for you, which greatly simplifies matrix multiplication.

import numpy as np

def sum\_cols(matrix):

return np.sum(matrix, axis=0)

def sum\_rows(matrix):

return np.sum(matrix, axis=1)

def dot(v, w):

return np.dot(v, w)

if \_\_name\_\_ == "\_\_main\_\_":

v1, v2, v3 = [1, 7, -4], [2, -3, 10], [3, 5, 6]

A = np.matrix([v1, v2, v3])

print ('matrix A:\n', A)

v\_cols = sum\_cols(A)

print ('\nsum A by column:\n', v\_cols)

v\_rows = sum\_rows(A)

print ('\nsum A by row:\n', v\_rows)

dot\_product = dot(v1, v2)

print ('\nvector 1:', v1)

print ('vector 2:', v2)

print ('\ndot product v1 and v2:')

print (dot\_product)

v1, v2, v3 = [-2, 5, 4], [1, 2, 9], [10, -9, 3]

B = np.matrix([v1, v2, v3])

print ('\nmatrix B:\n', B)

C = A.dot(B)

print ('\nmatrix C (dot product A and B):\n', C)

print ('\nC by row:')

for i, row in enumerate(C):

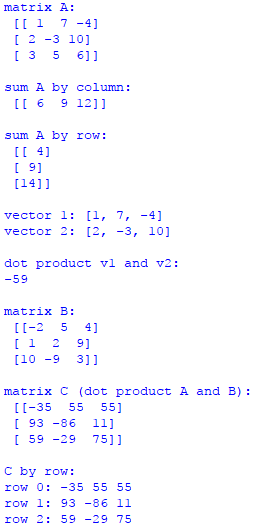
print ('row', str(i) + ': ', end='')

for v in np.nditer(row):

print (v, end=' ')

print()

Output:



The code begins by importing numpy. It continues with three functions – sum\_cols(), sum\_rows(), and dot(). Function sum\_cols() sums each column and returns a row with these values. Function sum\_rows() sums each row and returns a column with these values. Function dot() calculates the dot product. The main block begins by creating three vectors that are then used to create matrix A. Columns and rows are summed for A. Dot product is then calculated for two vectors (v1 and v2). Next, three new vectors are created that are then used to create matrix B. Matrix C is created by calculating the dot product for A and B. Finally, each row of C is displayed.

The 3rd code example illuminates a realistic scenario. Suppose a company sells three types of pies – beef, chicken, and vegetable. Beef pies cost $3 each, chicken pies cost $4 dollars each, and vegetable pies cost $2 dollars each. The vector representation for pie cost is [3, 4, 2]. You also know sales by pie for Monday through Thursday. Beef sales are 13 for Monday, 9 for Tuesday, 7 for Wednesday, and 15 for Thursday. The vector for beef sales is thereby [13, 9, 7, 15]. Using the same logic, the vectors for chicken sales are [8, 7, 4, 6] and [6, 4, 0, 3] respectively. The goal is to calculate total sales for 4 days (Monday-Thursday).

import numpy as np

def dot(v, w):

return np.dot(v, w)

def display(m):

for i, row in enumerate(m):

print ('total sales by day:\n', end='')

for v in np.nditer(row):

print (v, end=' ')

print()

if \_\_name\_\_ == "\_\_main\_\_":

a = [3, 4, 2]

A = np.matrix([a])

print ('cost matrix A:\n', A)

v1, v2, v3 = [13, 9, 7, 15], [8, 7, 4, 6], [6, 4, 0, 3]

B = np.matrix([v1, v2, v3])

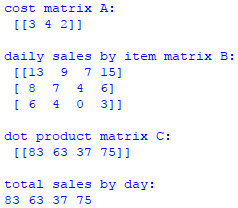
print ('\ndaily sales by item matrix B:\n', B)

C = A.dot(B)

print ('\ndot product matrix C:\n', C, '\n')

display(C)

Output:



The code begins by importing numpy. It continues with function dot() that calculates the dot product, and function display() that displays the elements of a matrix, row by row. The main block begins by creating a vector that holds the cost of each type of pie. It continues by converting the vector into matrix A. Next, three vectors are created that represent the sales for each type of pie for Monday through Friday. The code continues by converting the three vectors into matrix B. Matrix C is created by finding the dot product of A and B. This scenario demonstrates how dot product can be used for solving business problems.

The 4th code example calculates the magnitude (distance) and direction (angle) with a single vector and between two vectors:

import math, numpy as np

def sqrt\_sum\_squares(ls):

return math.sqrt(sum(map(lambda x:x\*x,ls)))

def mag(v):

return np.linalg.norm(v)

def a\_tang(v):

return math.degrees(math.atan(v[1]/v[0]))

def dist(v, w):

return math.sqrt(((w[0]-v[0])\*\* 2) + ((w[1]-v[1])\*\* 2))

def mags(v, w):

return np.linalg.norm(v - w)

def a\_tangs(v, w):

val = (w[1] - v[1]) / (w[0] - v[0])

return math.degrees(math.atan(val))

if \_\_name\_\_ == "\_\_main\_\_":

v = np.array([3, 4])

print ('single vector', str(v) + ':')

print ('magnitude:', sqrt\_sum\_squares(v))

print ('NumPY magnitude:', mag(v))

print ('direction:', round(a\_tang(v)), 'degrees\n')

v1, v2 = np.array([2, 3]), np.array([5, 8])

print ('two vectors', str(v1) + ' and ' + str(v2) + ':')

print ('magnitude', round(dist(v1, v2),2))

print ('NumPY magnitude:', round(mags(v1, v2),2))

print ('direction:', round(a\_tangs(v1, v2)), 'degrees\n')

v1, v2 = np.array([0, 0]), np.array([3, 4])

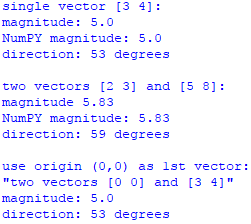
print ('use origin (0,0) as 1st vector:')

print ('"two vectors', str(v1) + ' and ' + str(v2) + '"')

print ('magnitude:', round(mags(v1, v2),2))

print ('direction:', round(a\_tangs(v1, v2)), 'degrees')

Output:



The code begins by importing math and numpy libraries. It continues with 6 functions. Function sqrt\_sum\_squares() calculates magnitude for one vector from scratch. Function mag() does the same but uses numpy. Function a\_tang() calculates the arctangent of a vector, which is the direction (angle) of a vector from the origin (0,0). Function dist() calculates magnitude between two vectors from scratch. Function mags() does the same but uses numpy. Function a\_tangs() calculates the arctangent of two vectors. The main block creates a vector, calculates magnitude and direction and displays. Next, magnitude and direction are calculated and displayed for two vectors. Finally, magnitude and direction for a single vector are calculated using the two vector formulas. This is accomplished by using the origin (0,0) as the 1st vector. So, functions that calculate magnitude and direction for a single vector are not needed because any single vector always begins from the origin (0,0). Therefore, a vector is simply a point in space measured either from the origin (0,0) or in relation to another vector by magnitude and direction.

*Basic Matrix Transformations*

The 1st code example introduces the identity matrix, which is a square matrix with ones on the main diagonal and zeros elsewhere. The product of matrix A and its identity matrix is A, which is important mathematically because the identity property of multiplication states that any number multiplied by 1 is equal to itself.

import numpy as np

def slice\_row(M, i):

return M[i,:]

def slice\_col(M, j):

return M[:, j]

def to\_int(M):

return M.astype(np.int64)

if \_\_name\_\_ == "\_\_main\_\_":

A = [[1, 9, 3, 6, 7],

[4, 8, 6, 2, 1],

[9, 8, 7, 1, 2],

[1, 1, 9, 2, 4],

[9, 1, 1, 3, 5]]

A = np.matrix(A)

print ('A:\n', A)

print ('\n1st row: ', slice\_row(A, 0))

print ('\n3rd column:\n', slice\_col(A, 2))

shapeA = np.shape(A)

I = np.identity(np.shape(A)[0])

I = to\_int(I)

print ('\nI:\n', I)

dot\_product = np.dot(A, I)

print ('\nA \* I = A:\n', dot\_product)

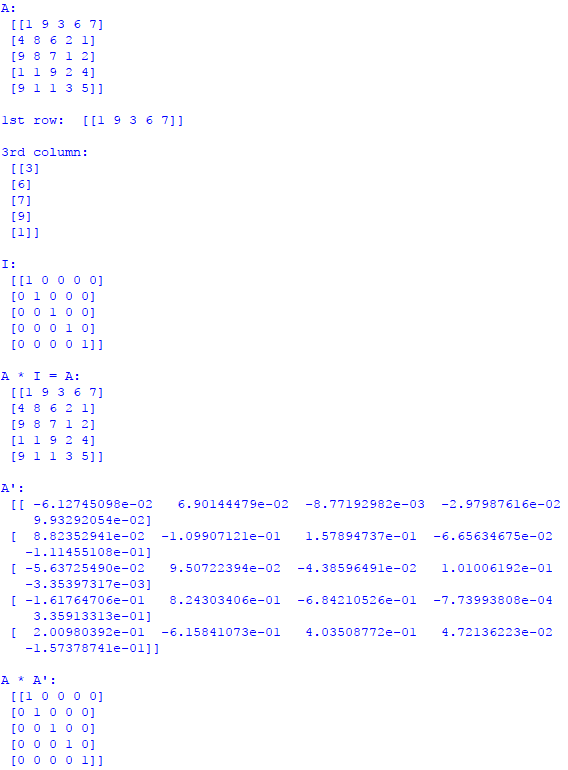
print ('\nA\':\n', A.I)

A\_by\_Ainv = np.round(np.dot(A, A.I), decimals=0, out=None)

A\_by\_Ainv = to\_int(A\_by\_Ainv)

print ('\nA \* A\':\n', A\_by\_Ainv)

Output:



The code begins by importing numpy. It continues with three functions. Function slice\_row() slices a row from a matrix. Function slice\_col() slices a column from a matrix. Function to\_int() converts matrix elements to integers. The main block begins by creating matrix A. It continues by creating the identity matrix for A. Finally, it creates the identity matrix for A by using the dot product of A with A' (inverse of A).

The 2nd code example converts a list of lists into a numpy matrix and traverses it:

import numpy as np

if \_\_name\_\_ == "\_\_main\_\_":

data = [

[41, 72, 180], [27, 66, 140],

[18, 59, 101], [57, 72, 160],

[21, 59, 112], [29, 77, 250],

[55, 60, 120], [28, 72, 110],

[19, 59, 99], [32, 68, 125],

[31, 79, 322], [36, 69, 111]

]

A = np.matrix(data)

print ('manual traversal:')

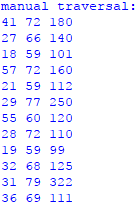
for p in range(A.shape[0]):

for q in range(A.shape[1]):

print (A[p,q], end=' ')

print ()

Output:



The code begins by importing numpy. The main block begins by creating a list of lists, converting it into numpy matrix A, and traversing A. Although I have demonstrated several methods for traversing a numpy matrix, this is my favorite method.

The 3rd code example converts a list of lists into numpy matrix A. It then slices and dices A:

import numpy as np

if \_\_name\_\_ == "\_\_main\_\_":

points\_3D\_space = [

[0, 0, 0],

[1, 2, 3],

[2, 2, 2],

[9, 9, 9] ]

A = np.matrix(points\_3D\_space)

print ('slice entire A:')

print (A[:])

print ('\nslice 2nd column:')

print (A[0:4, 1])

print ('\nslice 2nd column (alt method):')

print (A[:, 1])

print ('\nslice 2nd & 3rd value 3rd column:')

print (A[1:3, 2])

print ('\nslice last row:')

print (A[-1])

print ('\nslice last row (alt method):')

print (A[3])

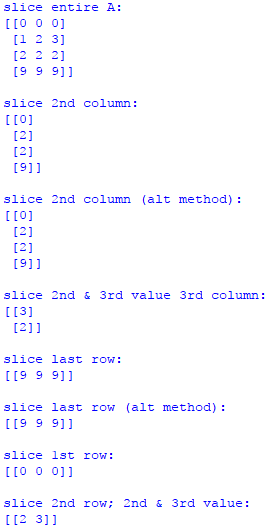
print ('\nslice 1st row:')

print (A[0, :])

print ('\nslice 2nd row; 2nd & 3rd value:')

print (A[1, 1:3])

Output:



The code begins by importing numpy. The main block begins by creating a list of lists and converting it into numpy matrix A. The code continues by slicing and dicing the matrix.

*Pandas Matrix Applications*

The pandas library provides high-performance, easy-to-use data structure and analysis tools. The most commonly used pandas object is a DataFrame (df). A df is a 2 dimensional structure with labeled axes (row and column) of potentially different types. Math operations align on both row and column labels. A df can be conceptualized by column or row. To view by column, use axis = 0 or axis= 'index'. To view by row, use axis=1 or axis='columns'. This may seem counterintuitive when working with rows, but this is the way pandas implemented this feature.

A pandas df is much easier to work with than a numpy matrix, but it is also less efficient. That is, it takes a lot more resources to process a pandas df. The numpy library is optimized for processing large amounts of data and numerical calculations.

The 1st example creates a list of lists, places it into a pandas df, and displays some data:

import pandas as pd

if \_\_name\_\_ == "\_\_main\_\_":

data = [

[41, 72, 180], [27, 66, 140],

[18, 59, 101], [57, 72, 160],

[21, 59, 112], [29, 77, 250],

[55, 60, 120], [28, 72, 110],

[19, 59, 99], [32, 68, 125],

[31, 79, 322], [36, 69, 111]

]

headers = ['age', 'height', 'weight']

df = pd.DataFrame(data, columns=headers)

n = 3

print ('First', n, '"df" rows:\n', df.head(n))

print ('\nFirst "df" row:')

print (df[0:1])

print ('\nRows 2 through 4')

print (df[2:5])

print ('\nFirst', n, 'rows "age" column')

print (df[['age']].head(n))

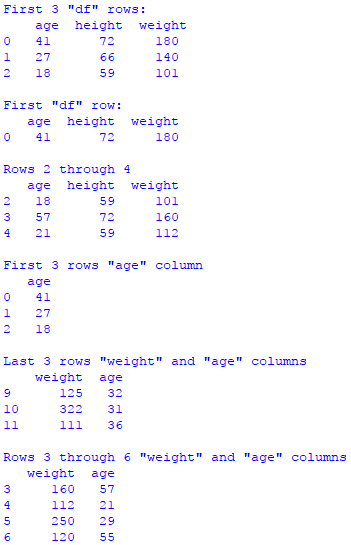
print ('\nLast', n, 'rows "weight" and "age" columns')

print (df[['weight', 'age']].tail(n))

print ('\nRows 3 through 6 "weight" and "age" columns')

print (df.ix[3:6, ['weight', 'age']])

Output:



The code begins by importing pandas. The main block begins by creating a list of lists and adding it to a pandas df. It is a good idea to create your own headers as we do here. Method head() and tail() automatically display the 1st five records and last five records respectively unless a value is included. In this case, we display the 1st and last three records. Using head() and tail() are very useful, especially with a large df. Notice how easy it is to slice and dice the df. Also, notice how easy it is to display column data of your choice.

The 2nd example creates a list of lists, places it into numpy matrix A, and puts A into a pandas df. This ability is very important because it shows how easy it is to create a df from a numpy matrix. So, you can be working with numpy matrices for precision and performance, and then convert to pandas for slicing, dicing, and other operations.

import pandas as pd, numpy as np

if \_\_name\_\_ == "\_\_main\_\_":

data = [

[41, 72, 180], [27, 66, 140],

[18, 59, 101], [57, 72, 160],

[21, 59, 112], [29, 77, 250],

[55, 60, 120], [28, 72, 110],

[19, 59, 99], [32, 68, 125],

[31, 79, 322], [36, 69, 111]

]

A = np.matrix(data)

headers = ['age', 'height', 'weight']

df = pd.DataFrame(A, columns=headers)

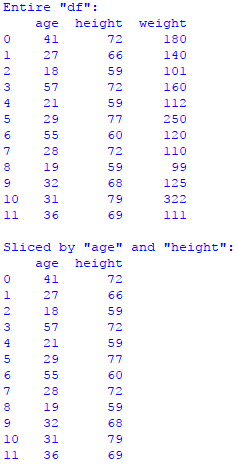
print ('Entire "df":')

print (df, '\n')

print ('Sliced by "age" and "height":')

print (df[['age', 'height']])

Output:



The code begins by importing pandas and numpy. The main block begins by creating a list of lists, converting it to numpy matrix A, and then adding A to a pandas df.

The 3rd example creates a list of lists, places it into a list of dictionary elements, and puts it into a pandas df. This ability is also very important because dictionaries are very efficient data structures when working with data science applications.

import pandas as pd

if \_\_name\_\_ == "\_\_main\_\_":

data = [

[41, 72, 180], [27, 66, 140],

[18, 59, 101], [57, 72, 160],

[21, 59, 112], [29, 77, 250],

[55, 60, 120], [28, 72, 110],

[19, 59, 99], [32, 68, 125],

[31, 79, 322], [36, 69, 111]

]

d = {}

dls = []

key = ['age', 'height', 'weight']

for row in data:

for i, num in enumerate(row):

d[key[i]] = num

dls.append(d)

d = {}

df = pd.DataFrame(dls)

print ('dict elements from list:')

for row in dls:

print (row)

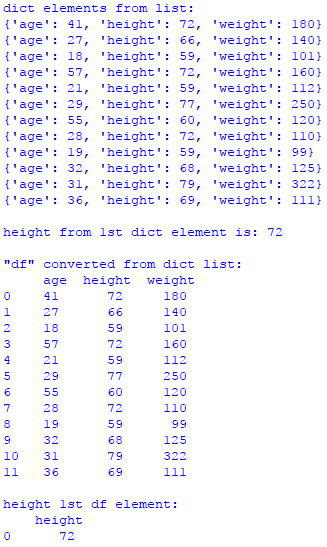
print ('\nheight from 1st dict element is:', end=' ')

print (dls[0]['height'])

print ('\n"df" converted from dict list:\n', df)

print ('\nheight 1st df element:\n', df[['height']].head(1))

Output:



The 4th code example creates two lists of lists – data and scores. The data list holds ages, heights, and weights for 12 athletes. The scores list holds three exam scores for 12 students. The data list is put directly into df1, and the scores list is put directly into df2. Averages are computed and displayed.

import pandas as pd, numpy as np

if \_\_name\_\_ == "\_\_main\_\_":

data = [

[41, 72, 180], [27, 66, 140],

[18, 59, 101], [57, 72, 160],

[21, 59, 112], [29, 77, 250],

[55, 60, 120], [28, 72, 110],

[19, 59, 99], [32, 68, 125],

[31, 79, 322], [36, 69, 111]

]

scores = [

[99, 90, 88], [77, 66, 81], [78, 77, 83],

[75, 72, 79], [88, 77, 93], [88, 77, 94],

[100, 99, 93], [94, 74, 90], [98, 97, 99],

[73, 68, 77], [55, 50, 68], [36, 77, 90]

]

n = 3

key1 = ['age', 'height', 'weight']

df1 = pd.DataFrame(data, columns=key1)

print ('df1 slice:\n', df1.head(n))

avg\_cols = df1.apply(np.mean, axis=0)

print ('\naverage by columns:')

print (avg\_cols)

avg\_wt = df1[['weight']].apply(np.mean, axis='index')

print ('\naverage weight')

print (avg\_wt)

key2 = ['exam1', 'exam2', 'exam3']

df2 = pd.DataFrame(scores, columns=key2)

print ('\ndf2 slice:\n', df2.head(n))

avg\_scores = df2.apply(np.mean, axis=1)

print ('\naverage scores for 1st', n, 'students (rows):')

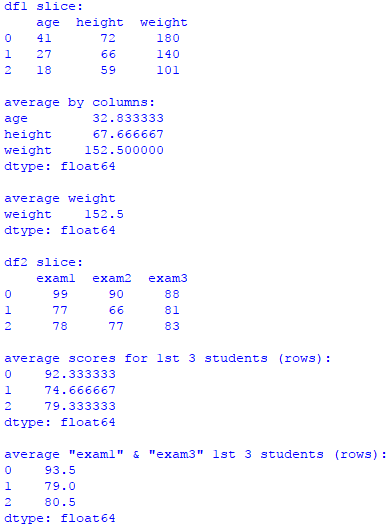
print (avg\_scores.head(n))

avg\_slice = df2[['exam1','exam3']].apply(np.mean, axis='columns')

print ('\naverage "exam1" & "exam3" 1st', n, 'students (rows):')

print (avg\_slice[0:n])

Output:



The code begins by importing pandas and numpy. The main block creates the data and scores lists and puts them in df1 and df2 respectively. With df1 (data), we average by column because our goal is to return the average age, height, and weight for all athletes. With df2 (scores), we average by row because our goal is to return the average overall exam score for each student. We could average by column for df2 if the goal is to calculate the average overall score for one of the exams. Try this if you wish.