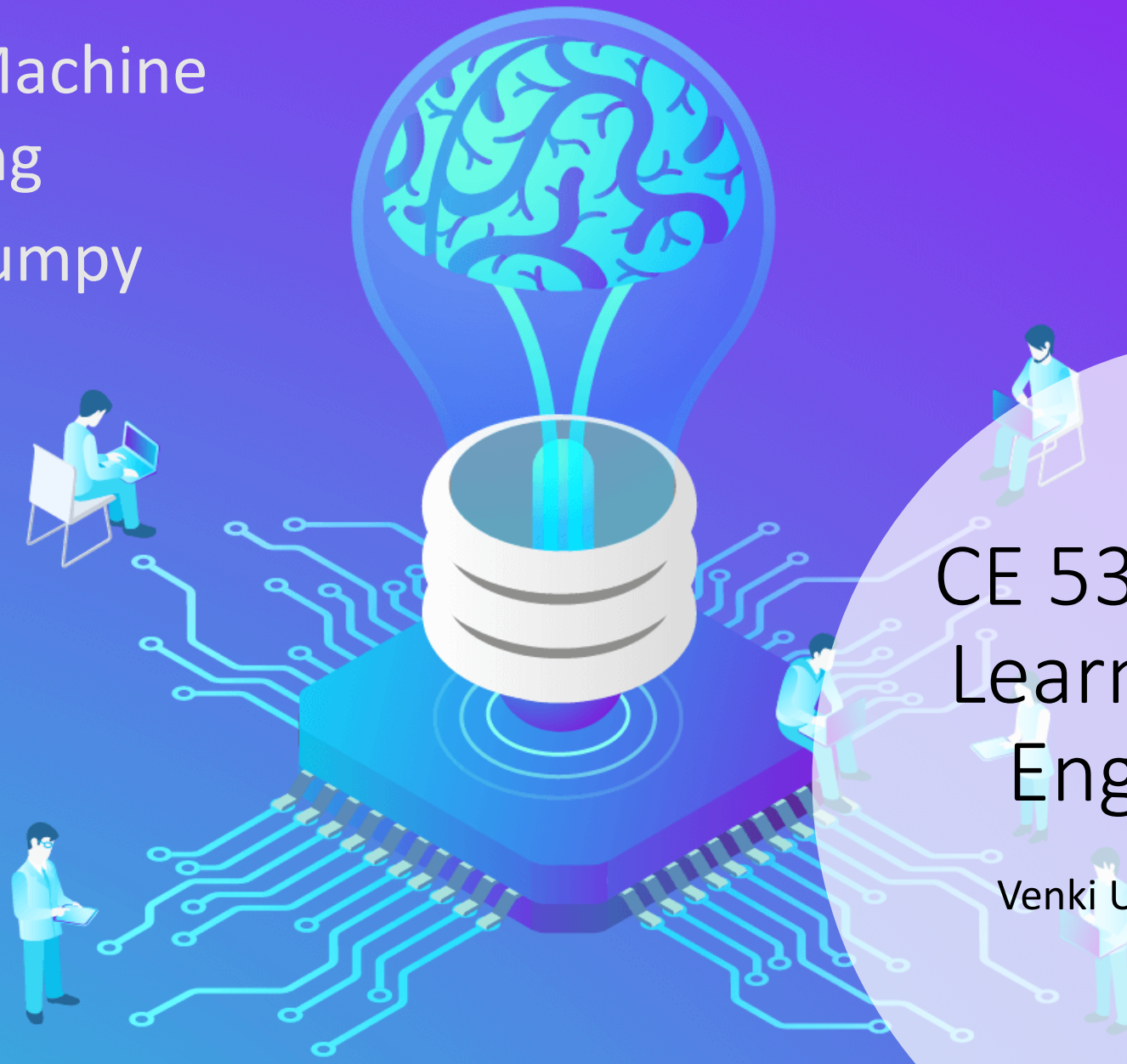
Python for Machine Learning

Arrays - Numpy



CE 5331 Machine Learning for Civil Engineers

Venki Uddameri, Ph.D. , P.E.

Recap and Goals

- Installed Python and Anaconda Environments
- Introduction to Python
  - Setting working directory
  - Adding comment lines
    - Docstrings
- Introduction to Pandas
  - Reading a csv
  - Extracting columns (attributes)
  - Extracting rows
  - Obtaining summary measures
- Control Statements
  - If, if-elif-else, if-else
  - For loop
  - While loop
  - Use of Boolean operators
- Functions
  - Passing inputs
  - Lambda functions
  - Pass by object reference

Goal of this module is to explore Array Functionality of Numpy Library

What is Numpy

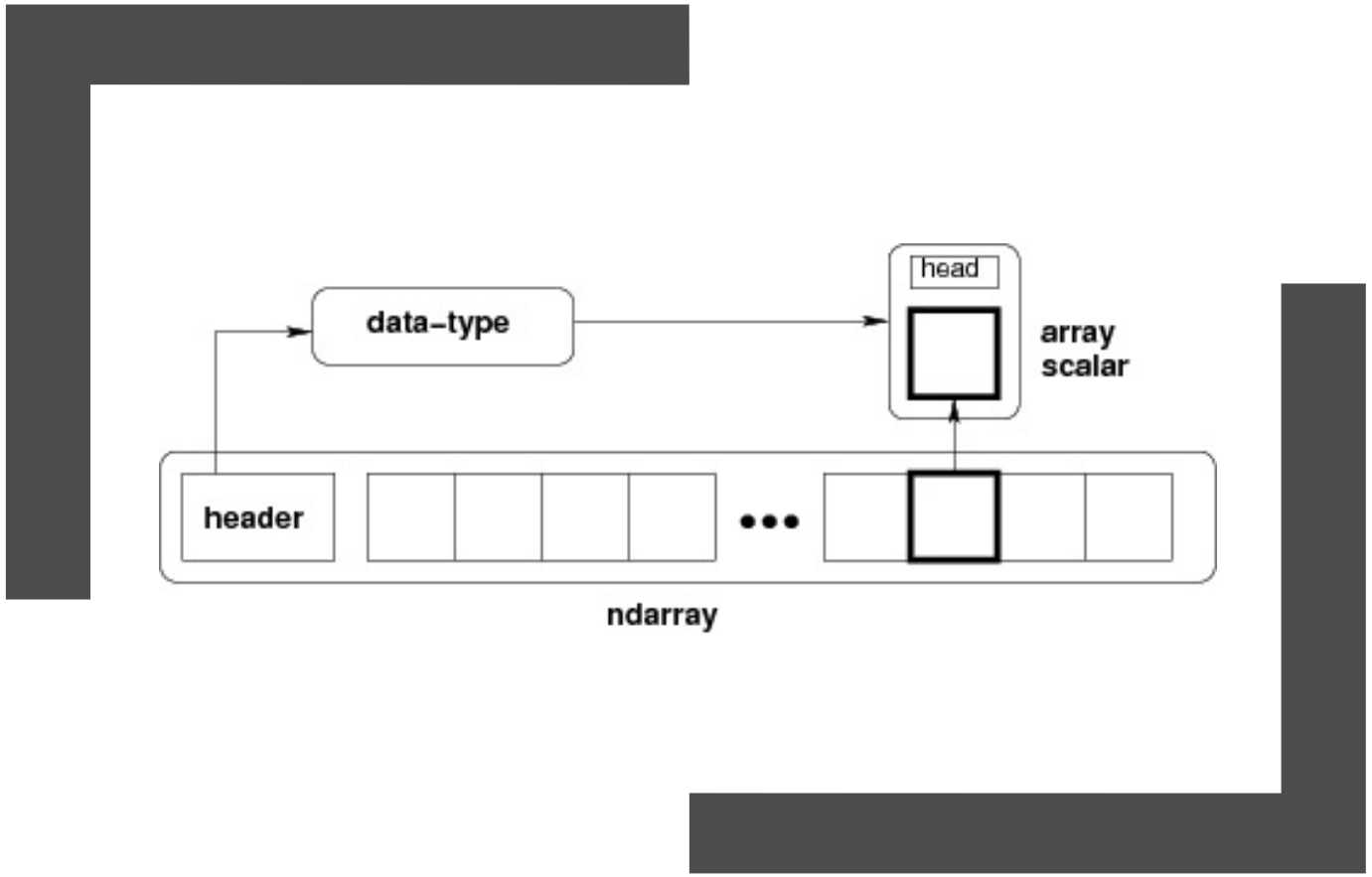
Numpy

- Numpy stands for ‘Numerical Python’
- Numpy is the most important foundational package for numerical and data analysis
  - Many other packages (including pandas) build on Numpy
- Numpy in itself does not provide any modeling or scientific functionality
  - Understanding it is still important as libraries that do build on Numpy
- Numpy allows vectorized operations
- Numpy allows data munging and cleaning operations
  - Data munging or data wrangling refers to the process of transforming raw data into a format that is useful for analysis

Numpy

- Numpy aggregated several array functions and libraries prior to 2005 and has since 2006 become the de facto standard for matrix computations
  - Many functions are written in C or Fortran and execute fast
  - Allows vectorization which speeds up the calculations as compared to for loops
- Numpy has better memory management
  - Significantly less memory compared to native Python counterparts

Numpy - ndarray



- Numpy's most important object is the **ndarray object**
  - **'ndarray'** stands for n-dimensional array
  - Collection of objects of the **same type**
    - In contrast pandas data_frame can have objects of different type
  - Numpy's ndarray can be viewed as a collection of scalars

It is important to note that all elements of a **ndarray** must be of the same data type

Creating Arrays

- Arrays can be created using arange method
- However in most instances you will likely read the data using pandas and then convert it into an array object
  - This is shown in the example

Numpy makes informed guesses on the data type

```
# Import libraries
import os
import pandas as pd
import numpy as np

# Set working directory - Needs to be present
os.chdir('D:\\Dropbox\\000CE5333Machine Learning\\Module5\\Code')

# Read Ogallaladata.csv file using pandas
a = pd.read_csv('Ogallaladata.csv')
a.head(4) # Write first 4 lines
a.columns # Write the list of columns to the console

# Extract Depth to WT and WellDepth
DWT = a.DWT
WD = a.WellDepth

# Create numpy array of DWT
DWTnp = np.array(DWT)

#look at some properties
DWTnp.dtype # Data type
DWTnp.shape # Shape
DWTnp.size # Size
```

Console

```
DWTnp.dtype # Data type
Out[54]: dtype('float64')

DWTnp.shape # Shape
Out[55]: (101,)

DWTnp.size # Size
Out[56]: 101
```

The simplest form is the one-dimensional array (a vector of objects)

Creating Arrays

- Pandas Dataframe can also be converted into a two (multi) dimensional array
  - All variables must be of the same type
  - May need to conversion of some

```
# Create a Pandas Dataframe
zz = pd.concat([WD,DWT], axis=1)  # Contatenate columns (axis=1)
# Create a numpy array 101 rows x 2 columns
zznp = np.array(zz,dtype='float64') # Make all float
```

Here WD (well depth) is ‘int64’ and DWT is ‘float64’
We will convert WD to ‘float64’

Creating Special Matrices

- Special matrices are sometimes used in matrix algebra
  - A matrix of zeros
  - A matrix of ones
  - An identity matrix
- Numpy provides methods to build these arrays

Note the Parenthesis

```
zerom = np.zeros((3,3))
```

```
zerom
Out[77]:
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

```
np.ones((3,3))
```

```
Out[78]:
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

No Parenthesis (rows, cols)

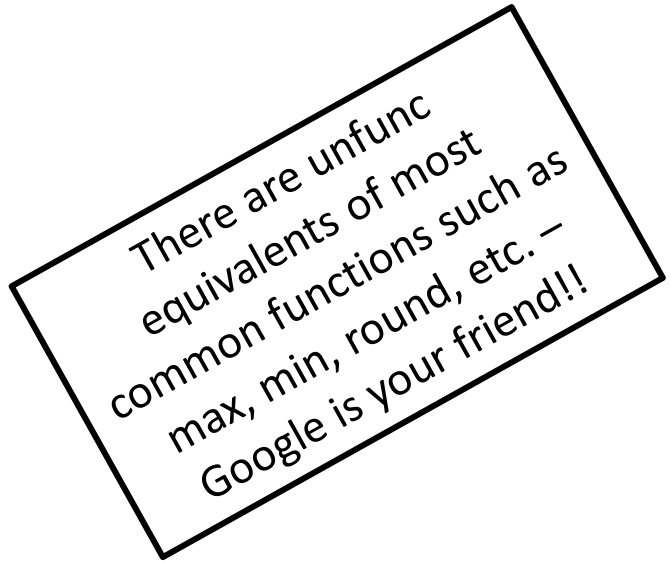
Identity Matrix

```
np.eye(3,4)
```

```
Out[86]:
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

Numpy Universal Functions (ufunc)

- Universal functions or ufunc for short perform cell by cell operations on Numpy arrays
  - Helps avoid using nested for loops
  - Vectorized operations
- These ufuncs are typically written in c and are faster
- Use ufunc counterparts of Numpy instead of built-in functions in Python when operating on arrays
  - There will be an error flag thrown otherwise
- You can write your own ufuncs
  - Convert python functions to ufuncs using numpy.frompyfunc
  - See - https://docs.scipy.org/doc/numpy/reference/generated/numpy.frompyfunc.html#numpy.frompyfunc



For More Info on ufuncs: https://docs.scipy.org/doc/numpy/reference/ufuncs.html

Vectorization in Python – Cell by cell Calculations

- Native python offers the **map** function
  - Map can be used to perform calculations over a tuple or a list
- Numpy offers **numpy.vectorize** function
  - Essentially a for loop so may not lead to significant time savings
  - Follows the broadcast rules of Numpy instead of native python

A = [0.7,0.29,0.23] — A Python List

```
sumz = lambda x,y=2: x + y #Function mapped
A = [0.7,0.29,0.23]
# Use * to unpack and notice the end comma,
# Need to send in values of y for each x
amap = *map(sumz,A,(3,3,3)),
```

amap = (2.7, 2.29, 2.23)

dum =

0.7	0.29	0.23
0.55	0.72	0.42
0.98	0.68	0.48

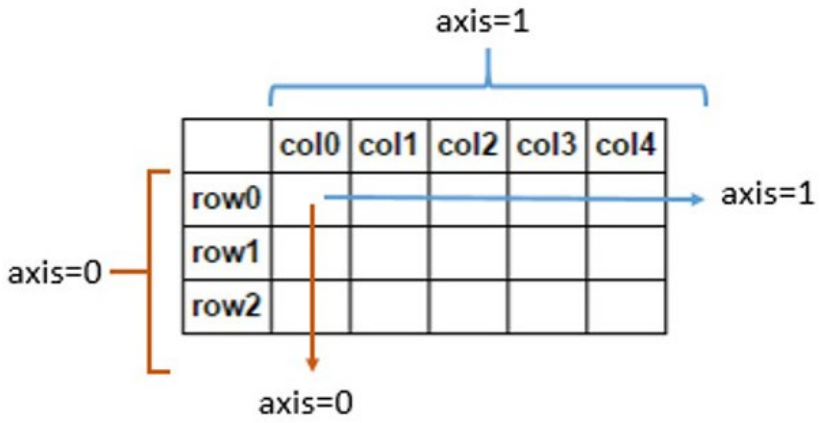
Numpy Array

```
sumz = lambda x,y=2: x + y
vfunc = np.vectorize(sumz) #Create vectorized function
vfunc(dum,3) # Perform cell by cell operations
```

```
array([[3.7 , 3.29, 3.23],
       [3.55, 3.72, 3.42],
       [3.98, 3.68, 3.48]])
```

Vectorization – Operations or Rows/Columns

- Dimensions of an array are referred to as axis in numpy and python
  - In a two-dimensional matrix
    - Axis 0 is along rows
    - Axis 1 is along columns
- Numpy provides apply_along_axis function to vectorize



dum

0.7	0.29	0.23
0.55	0.72	0.42
0.98	0.68	0.48

```
def sumx(x,y):
# Sums each element of a list and adds y
    suz = sum(x) + y
    return suz
```

```
# All rows in a column (column by column)
B = np.apply_along_axis(sumx,0,dum,3)

# All columns in a row (row by row)
C = np.apply_along_axis(sumx,1,dum,3)
```

B = array([5.23, 4.69, 4.13]) ← Sum each column and add 3

C = array([4.22, 4.69, 5.14]) ← Sum each row and add 3

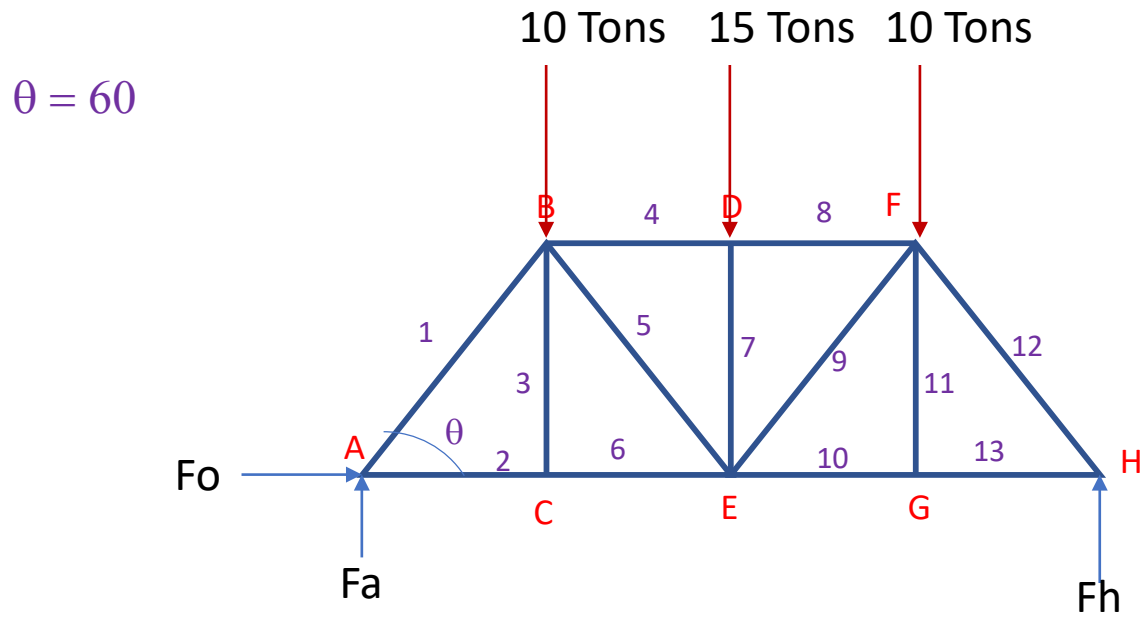
Illustrative Application

- Numpy can read array data from a csv file
  - Also can read and write binary files
  - You can always read pandas to read a csv file
- Numpy.linalg (linear algebra) module has several matrix manipulation functions
  - Inverse, determinant, QR decomposition, solve linear system of equations, compute least square solution to Ax=b, eigen values and eigen vectors, singular value decomposition, etc.
- Numpy.linalg uses industry-standard linear algebra engines (libraries) such as BLAS, LAPACK that are used by other languages such as R and MATLAB
  - Written in C or Fortran and optimized to work with Python

I will illustrate matrix manipulations in Python using some examples

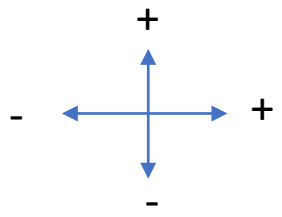
Illustrative Example

- Consider a plane truss shown in the Figure below. Calculate the forces in each member by setting up and solving a system of linear equations (Truss members form isosceles triangles with 60^o angle)



Initially assume all members are in tension

Force Direction Convention

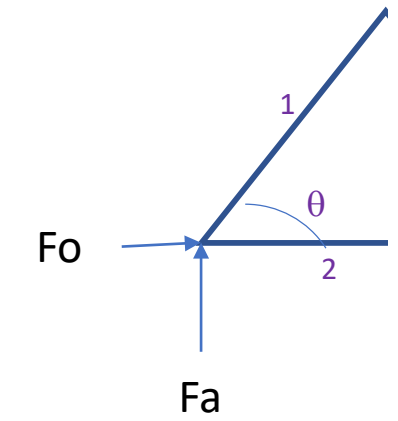


8 Joints (A - H) and 13 members (statically determinate)
A is fixed support and H is simple (rolling) support

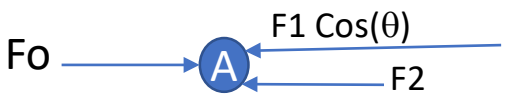
Illustrative Example

- At each joint the sum of the forces should be equal to zero
  - Equilibrium condition
  - Sum of the resolved forces in the horizontal and vertical directions should be equal to zero
- Vertical and Horizontal Force summation at each joint gives us two equations
  - One horizontal
  - One vertical

Illustrative example (section A)

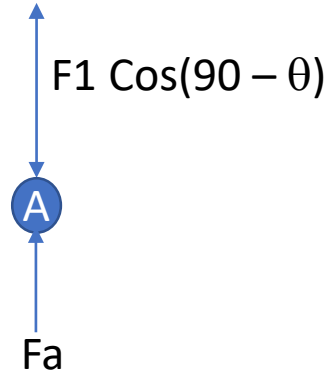


Horizontal



$$F_{H,A} = F_o - F_2 - F_1 cos(\theta) = 0$$

Vertical



$$F_{V,A} = F_a - F_1 cos(90 - \theta) = 0$$

One can proceed in a similar fashion at all joints and write a system of linear equations (16 equations)

Matrix Method For a System of Linear Equations (Detour)

- A system of linear equations can be represented in a matrix form
  - Coefficient matrix, unknowns matrix, RHS matrix

Coeff. Matrix

RHS Matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 2 & 5 \\ 2 & 5 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ -4 \\ 27 \end{bmatrix}$$

Unk. Matrix

System of Linear Equations

$$\begin{array}{rcrcrcr} x & + & y & + & z & = & 6 \\ & & 2y & + & 5z & = & -4 \\ 2x & + & 5y & - & z & = & 27 \end{array}$$

In compact form this is generally expressed as

$$AX = B$$

The solution to the unknown X can be obtained as

$$X = A^{-1}B$$

Force Balance Equations – Matrix Form

	Coeffient Matrix																Unknown		RHS
Rowid	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	Fo	FA	FH	Matrix		Matrix
AH	**-0.94496**	**-1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	F1		**0**
AV	**-0.98614**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	F2		**0**
BH	**0.944957**	**0**	**0**	**-1**	**-0.94496**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	F3		**0**
BV	**0.986143**	**0**	**1**	**0**	**0.986143**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	F4		**10**
CH	**0**	**1**	**0**	**0**	**0**	**-1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	F5		**0**
CV	**0**	**0**	**-1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	F6		**0**
DH	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**-1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	F7		**0**
DV	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	F8		**15**
EH	**0**	**0**	**0**	**0**	**0.944957**	**1**	**0**	**0**	**-0.94496**	**-1**	**0**	**0**	**0**	**0**	**0**	**0**	F9	=	**0**
EV	**0**	**0**	**0**	**0**	**-0.98614**	**0**	**-1**	**0**	**-0.98614**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	F10		**0**
FH	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0.944957**	**0**	**0**	**-0.94496**	**0**	**0**	**0**	**0**	F11		**0**
FV	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0.986143**	**0**	**1**	**0.986143**	**0**	**0**	**0**	**0**	F12		**10**
GH	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**-1**	**0**	**0**	**0**	F13		**0**
GV	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-1**	**0**	**0**	**0**	**0**	**0**	Fo		**0**
HH	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0.944957**	**1**	**0**	**0**	**0**	FA		**0**
HV	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**-0.98614**	**0**	**0**	**0**	**1**	FH		**0**

Solving the System of Linear Equation in Python using Numpy

- Basic Strategy
  - Read the coefficient matrix and RHS from a csv file
  - Separate out the coefficient and RHS matrices
  - Invert the coefficient matrix
  - Multiply inverted coefficient matrix and RHS matrix
    - Order in which these matrices are multiplied matters

Dimensions of the Resultant Matrix

$$[M \mathbf{X} N][N \mathbf{X} K] = [M \mathbf{X} K]$$

Cols of A must Match the Rows of B

Code

trussmatrixnumpy.csv' – File Format

-0.94496	-1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
-0.98614	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0.944957	0	0	-1	-0.94496	0	0	0	0	0	0	0	0	0	0	0	0
0.986143	0	1	0	0.986143	0	0	0	0	0	0	0	0	0	0	0	10
0	1	0	0	0	-1	0	0	0	0	0	0	0	0	0	0	0
0	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	-1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	15
0	0	0	0	0.944957	1	0	0	-0.94496	-1	0	0	0	0	0	0	0
0	0	0	0	-0.98614	0	-1	0	-0.98614	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0.944957	0	0	-0.94496	0	0	0	0	0
0	0	0	0	0	0	0	0	0.986143	0	1	0.986143	0	0	0	0	10
0	0	0	0	0	0	0	0	0	1	0	0	-1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0.944957	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	-0.98614	0	0	0	1	0

Red Coefficient Matrix rows 0:16 in Python Notation; Green – RHS matrix row 16 in Python Notation

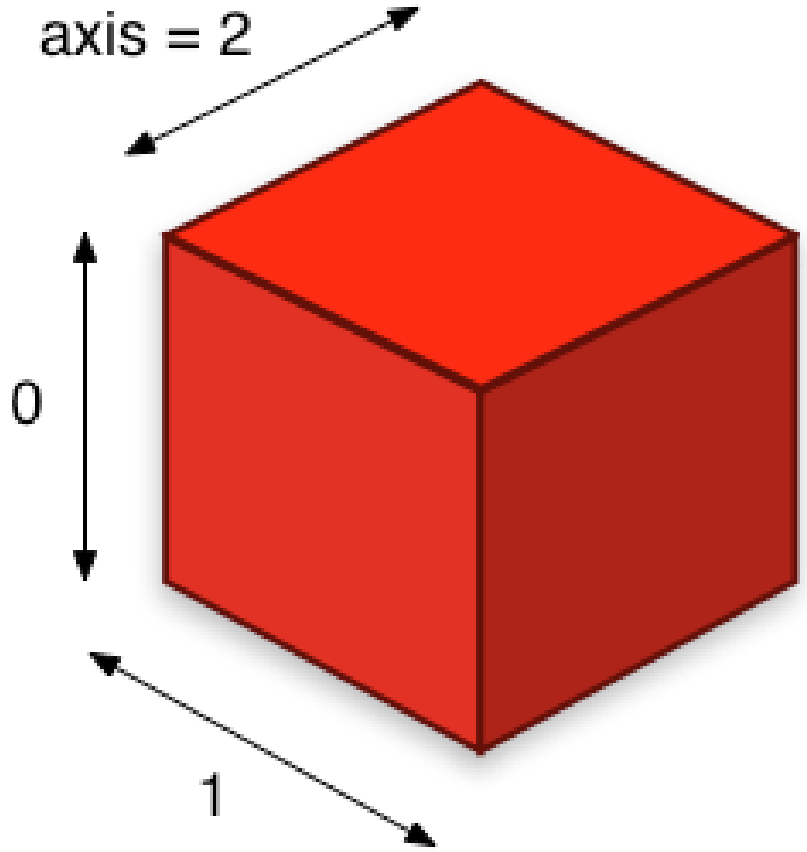
```
# Read the matrix file
mat = np.genfromtxt('trussmatrixnumpy.csv',delimiter=',')

# Extract Coefficient and RH Matrix
coeff = mat[0:16,0:16]
RHS = mat[0:16,16]

# Perform Necessary Matrix Calculations
coeffinv = np.linalg.inv(coeff) # Inverse of coeff matrix using linalg
Fbeam = np.matmul(coeffinv,RHS) # Matrix multiplication
np.round(Fbeam,2) # Use Numpy ufunc round to round numbers
```

Results (All Forces in Tons)

```
array([ 17.75, -16.77,   0. ,  23.96,  -7.61, -16.77,  15. ,
23.96, -7.61, -16.77,  -0. ,  17.75, -16.77,   0. ,  17.5 ,  17.5 ])
```



Arrays and can be read and written as both text and binary files using Numpy

You should Know

- What is Numpy
  - Numpy array structures
- Performing calculations using Numpy
  - Matrix calculations using Numpy
  - Matrix algebra using linalg
- Vectorized numpy built-in functions
- Vectorize user-defined functions
  - Cell-by-cell operations
- Vectorize along an axis
  - Row and column operations