COMP 125 Programming with Python
Introduction to NumPy

NumPy

Mehmet Sayar

Koç University

Midterm 5

Sunday at 6.15 PM

The exam will focus on the following topics:

File I/O, String Formatting,

Functions Reloaded (multiple input, multiple output, default arguments, keyword arguments),

Exception Handling.

The exam will NOT include recursion and Numpy.

However, you may need everything else we have covered in class so far.

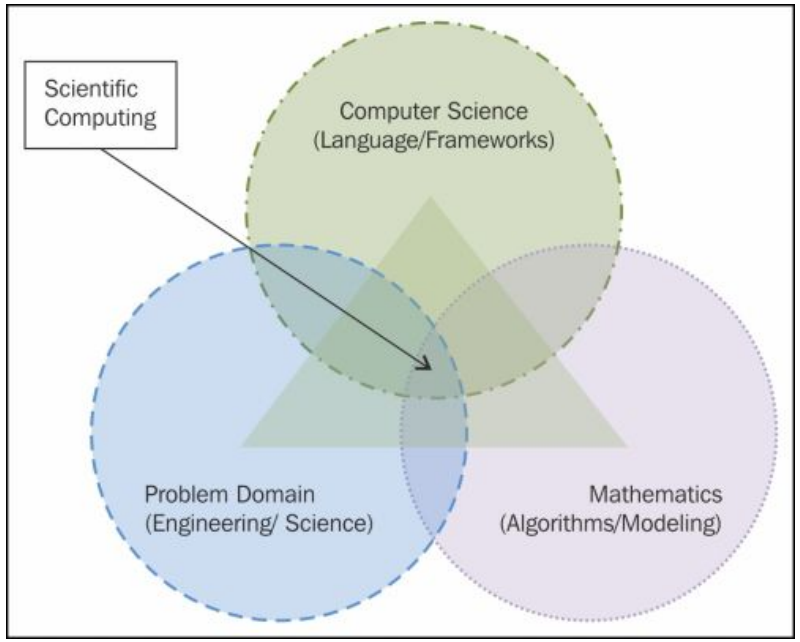
Homework 5 will be released soon, but it will include Recursion and Numpy on top of the Midterm 5 topics.

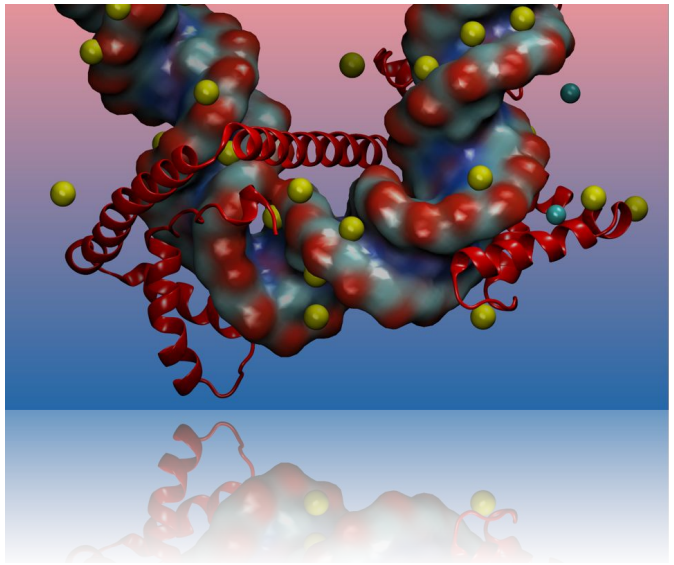
Previously on Comp125

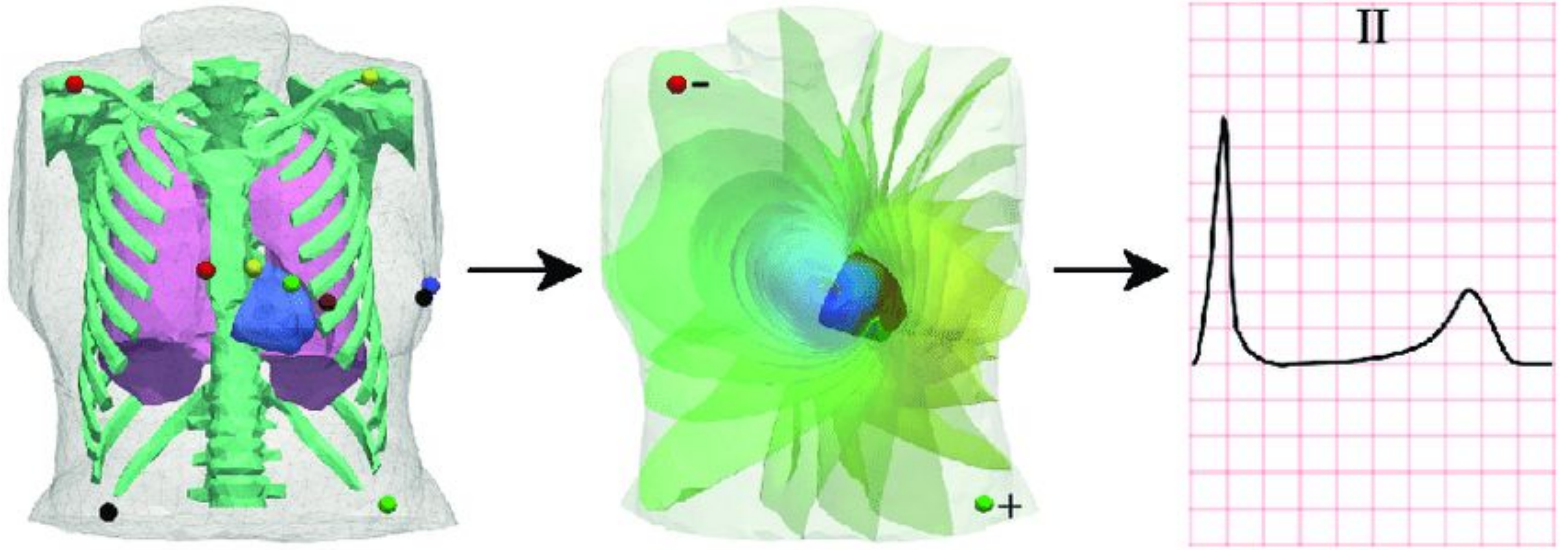
- Karel
- Conditionals
- Loops
- Variables
- Functions
- Scope
- Strings Lists
- Tuples and Dictionaries
- File I/O
- String Formatting
- Exception Handling
- Recursion

Now – Scientific Computing

- Scientific Computing is the collection of tools, techniques, and theories required to solve mathematical models of problems in Science and Engineering on a computer.





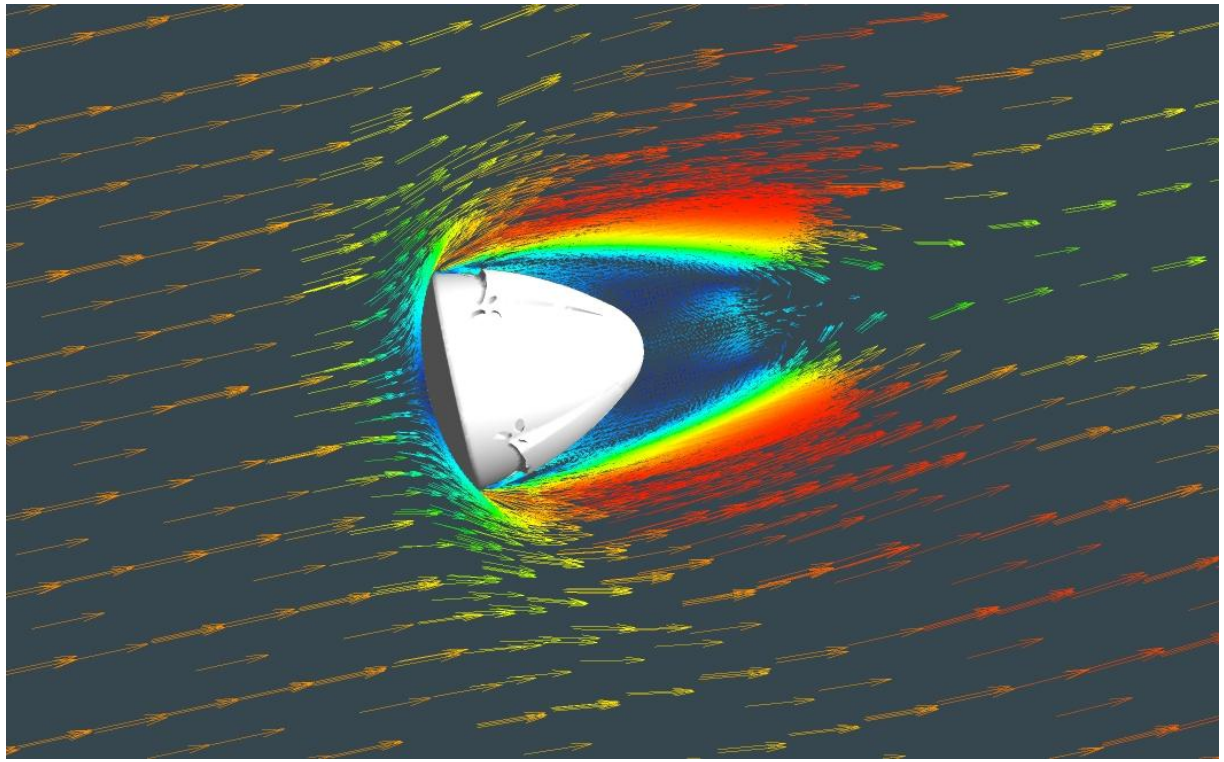


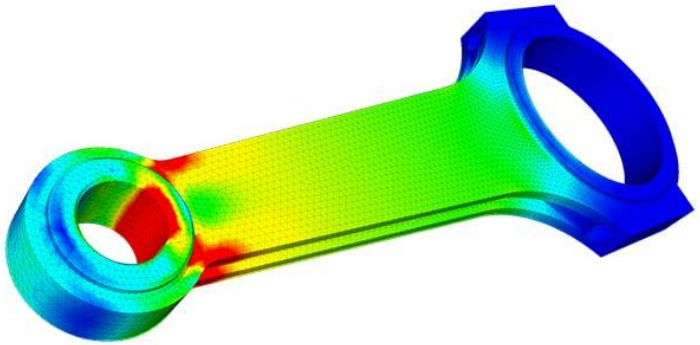
Matrix Operations

- Linear Algebra, Probability and Statistics, and Calculus show up almost always in scientific and engineering problems
- Need computers for the required math operations
- Matrix operations are a frequent subset

Example Problems

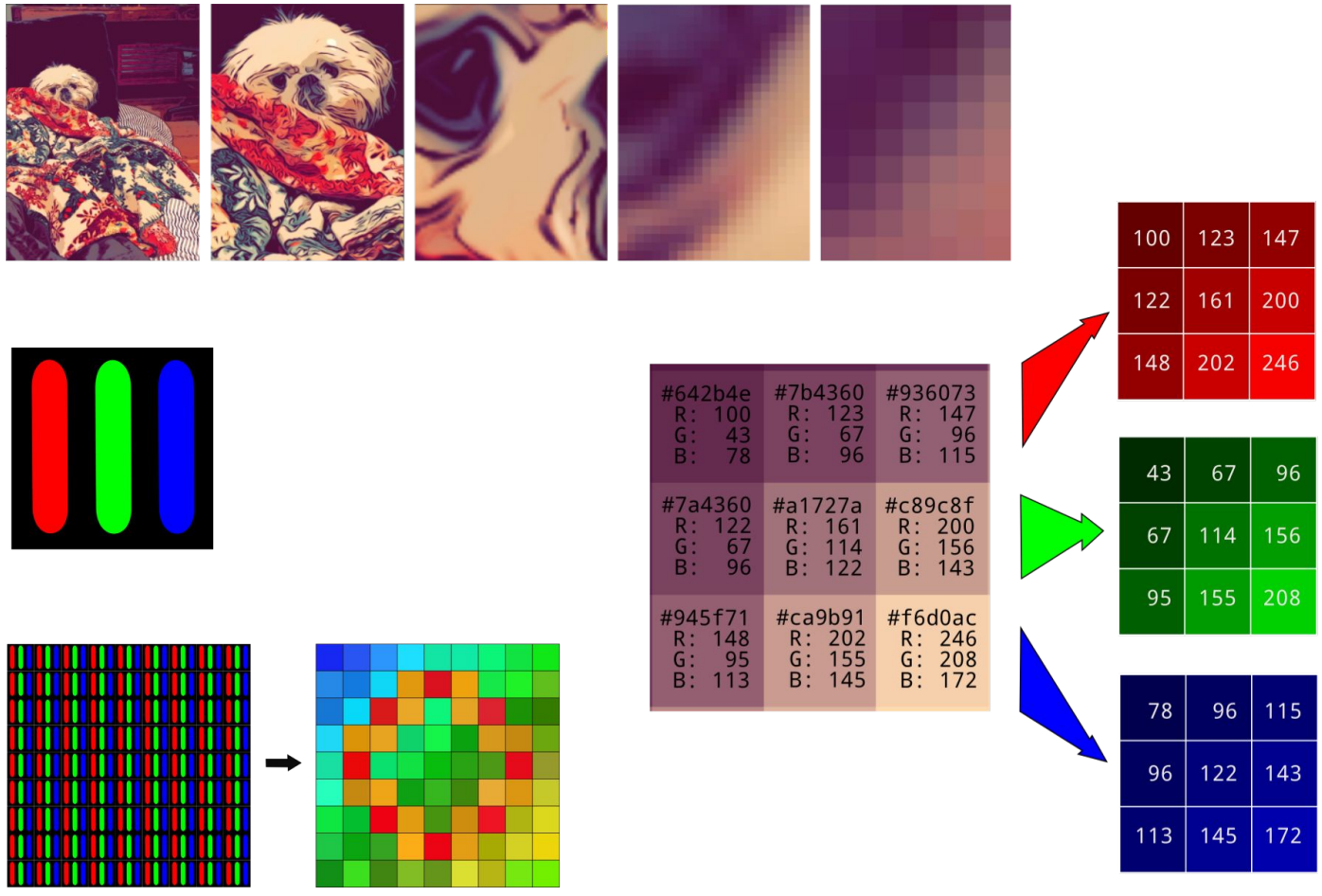
- Calculating air flow, heat flow, stress-strain etc. (Finite Element/Volume Methods)





Example Problems

- Representing and working on Images



Example Problems

- Least Squares Curve Fitting (e.g. to get a parametric Voltage-Current Curve)

Example 3.2 Linear Least Squares Data Fitting. We illustrate linear least squares by fitting a quadratic polynomial to the following five data points:

t	-1.0	-0.5	0.0	0.5	1.0
y	1.0	0.5	0.0	0.5	2.0

The overdetermined 5×3 linear system is therefore

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} 1 & -1.0 & 1.0 \\ 1 & -0.5 & 0.25 \\ 1 & 0.0 & 0.0 \\ 1 & 0.5 & 0.25 \\ 1 & 1.0 & 1.0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \approx \begin{bmatrix} 1.0 \\ 0.5 \\ 0.0 \\ 0.5 \\ 2.0 \end{bmatrix} = \boldsymbol{b}.$$

The solution to this system, which we will see later how to compute, turns out to be $\boldsymbol{x} = [0.086 \quad 0.40 \quad 1.4]^T$, which means that the approximating polynomial is

$$p(t) = 0.086 + 0.4t + 1.4t^2.$$

The resulting curve and the original data points are shown in Fig. 3.1. The least squares solution minimizes the sum of squares of vertical distances between the data points and the curve over all possible quadratic polynomials.

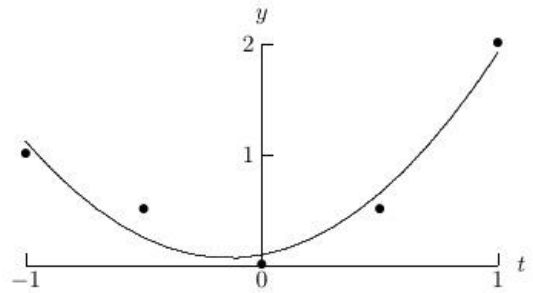
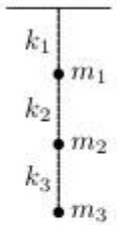


Figure 3.1: Least squares fit of a quadratic polynomial to the given data.

Example Problems

- Solving or Simulating Dynamical Systems

4.12 Consider the spring-mass system



with three masses m_1, m_2, and m_3 at vertical locations y_1, y_2, and y_3 connected by three springs having spring constants k_1, k_2, and k_3. According to Newton's Second Law, the motion of the system is governed by the system of ordinary differential equations

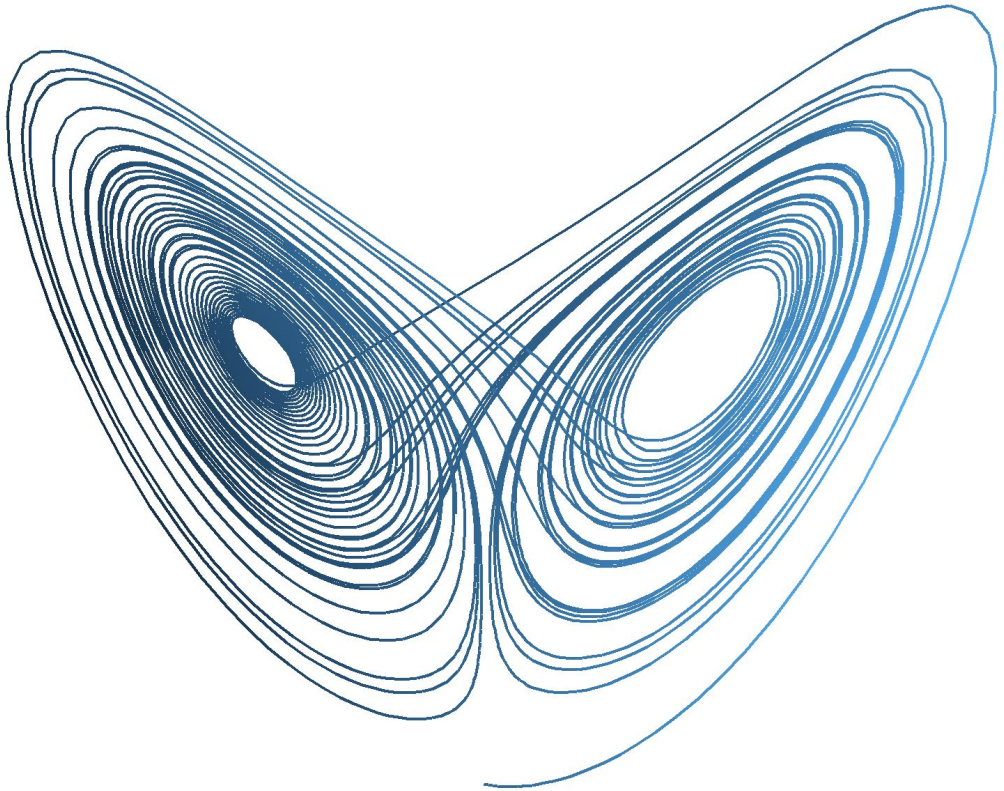
$$\boldsymbol{M}\boldsymbol{y}'' + \boldsymbol{K}\boldsymbol{y} = 0,$$

where

$$\boldsymbol{M} = \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix}$$

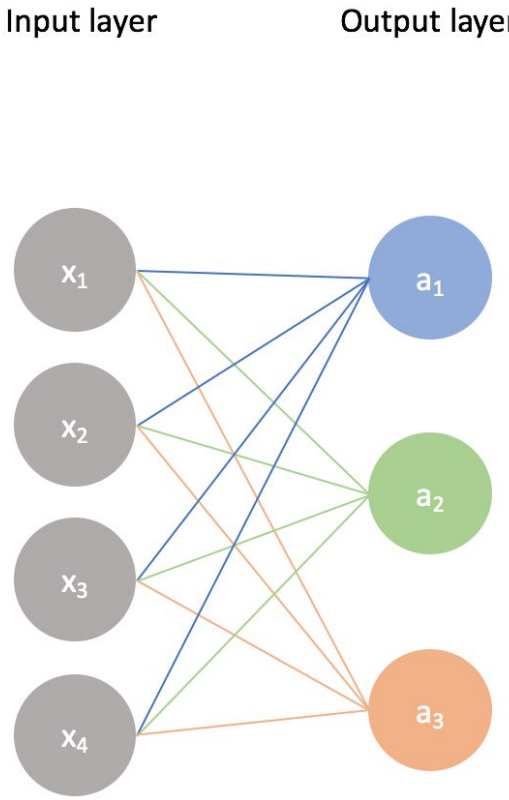
is the *mass matrix* and

$$\boldsymbol{K} = \begin{bmatrix} k_1 + k_2 & -k_2 & 0 \\ -k_2 & k_2 + k_3 & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix}$$



Example Problems

- Machine Learning



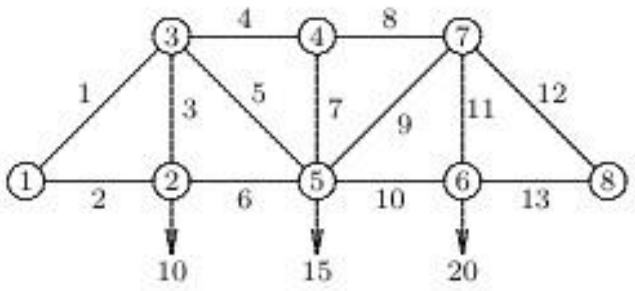
A simple neural network

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \\ w_1 & w_2 & w_3 & w_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix} = \begin{bmatrix} w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \\ w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b \end{bmatrix} \xrightarrow[activation]{} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Example Problems

- Linear System of Equations

2.3 The following diagram depicts a plane truss having 13 members (the numbered lines) connected by 10 joints (the numbered circles). The indicated loads, in tons, are applied at joints 2, 5, and 6, and we wish to determine the resulting force on each member of the truss.



For the truss to be in static equilibrium, there must be no net force, horizontally or vertically, at any joint. Thus, we can determine the member forces by equating the horizontal forces to the left and right at each joint, and similarly equating the vertical forces upward and downward at each joint. For the eight joints, this would give 16 equations, which is more than

$$
\begin{aligned}
&\text{Joint 2}: \begin{cases} f_2 = f_6 \\ f_3 = 10 \end{cases} \\
&\text{Joint 3}: \begin{cases} \alpha f_1 = f_4 + \alpha f_5 \\ \alpha f_1 + f_3 + \alpha f_5 = 0 \end{cases} \\
&\text{Joint 4}: \begin{cases} f_4 = f_8 \\ f_7 = 0 \end{cases} \\
&\text{Joint 5}: \begin{cases} \alpha f_5 + f_6 = \alpha f_9 + f_{10} \\ \alpha f_5 + f_7 + \alpha f_9 = 15 \end{cases} \\
&\text{Joint 6}: \begin{cases} f_{10} = f_{13} \\ f_{11} = 20 \end{cases} \\
&\text{Joint 7}: \begin{cases} f_8 + \alpha f_9 = \alpha f_{12} \\ \alpha f_9 + f_{11} + \alpha f_{12} = 0 \end{cases} \\
&\text{Joint 8}: \begin{cases} f_{13} + \alpha f_{12} = 0 \end{cases}
\end{aligned}
$$

How to convert this to matrix formulation?

Matrix Operations

- Linear Algebra, Probability and Statistics, and Calculus show up almost always in scientific and engineering problems
- Need computers for the required math operations
- Matrix operations are a frequent subset

- So far, we have seen a few simple examples with list of lists (dense matrices) and dictionaries (sparse matrices)
- However, these are inefficient

Matrix Operations

- So far, we have seen a few simple examples with list of lists (dense matrices) and dictionaries (sparse matrices)
- However, these are inefficient – why?
  - Too general, causing overheads
  - All the axes are not equal (remember getting a column from a list of lists?)
  - Gets worse when generalizing to Tensors
  - Looping over items, difficult to parallelize
- Need for a specialized implementation - NumPy

NumPy

The fundamental package for scientific computing with Python

GET STARTED

NumPy v1.19.0 First Python 3 only release - Cython interface to numpy.random complete

POWERFUL N-DIMENSIONAL ARRAYS

Fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are the de-facto standards of array computing today.

NUMERICAL COMPUTING TOOLS

NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.

INTEROPERABLE

NumPy supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries.

PERFORMANT

The core of NumPy is well-optimized C code. Enjoy the flexibility of Python with the speed of compiled code.

EASY TO USE

NumPy's high level syntax makes it accessible and productive for programmers from any background or experience level.

OPEN SOURCE

Distributed under a liberal BSD license, NumPy is developed and maintained publicly on GitHub by a vibrant, responsive, and diverse community.

https://numpy.org/devdocs/user/absolute_beginners.html

NumPy (Numerical Python)

- Implemented in C (means fast)
- NumPy `ndarrays`: Efficient storage of a single type of data
- These arrays can be of any dimension (e.g. vector 1, matrix 2, tensor n)
- Efficient mathematical and logical operations on arrays
- Efficient linear algebra operations on arrays
- Random number (array) generation

- Foundation of the Python scientific computation stack

NumPy Module

- Typically called as

```
import numpy as np
```

- Its main data structure, `ndarray`, is created as:

```
x = np.array(array_like)
```

- For example:

```
x = np.array((3,4)) #1D
x = np.array([[3.3,4.9],[-1,3.2],[7.1,-5.7]]) #2D
```

NumPy Array – Range Creating Functions

- Better and improved range: `arange(start, stop, step)`
- Start optional, step optional, each can be floats! Stop non-inclusive

```
x = np.arange(5)
x = np.arange(2.2,8.3)
x = np.arange(6.7,-2.3,-0.1)
```

- This creates an `ndarray` object (as opposed to `range` not creating a `list`)

NumPy Array – Range Creating Functions

- Controlling number of points: `linspace(start, stop, number)`
- Stop inclusive, number optional, default 50
- Can go both in positive and negative directions

```
x = np.linspace(3, 6)
x = np.linspace(2.2, 8.3, 10)
x = np.linspace(-5, 5, 101)
```

- This creates an `ndarray` object as well
- See also: `logspace`

NumPy Array – Creating with Homogeneous Data

- Homogeneous data: `zeros`, `ones`, `full`
- Usage: `np.zeros(shape), np.ones(shape), np.full(shape,value)`
- Shape is a tuple of integers:
  - Length denotes the number of dimensions
  - Individual integers denote the size of their corresponding dimension
- Examples

```
x = np.zeros(10), x = np.zeros((10,))
x = np.ones((3,5))
x = np.full((3,2,5),-1.5)
```

NumPy Array – Creating with Empty Arrays

- Create an entry without initializing the entries: `empty(shape)`
- Useful to pre-allocate space (you can use the previous functions as well)

```
x = np.empty(10), x = np.empty((10,))
x = np.empty((3,5))
x = np.empty((3,2,5))
```

NumPy Array – Creating Diagonal Arrays

- Creating with diagonal elements (all else 0): `diag`, `eye`
- Identity Matrix (main diagonal all 1s): `np.eye(dim)`

```
x = np.eye(4)
```

- Creating a diagonal matrix from 1-D inputs: `np.diag(array_like)`

```
x = np.diag([4,-8,2])
```

- Getting the diagonal of a matrix: `np.eye(matrix)`

```
y = np.array([[1,-2,2],[3,-4,7],[9,6,-8]])
x = np.diag(y)
```

NumPy Array – Creating Random Arrays

- Uniformly random in [0.0,1.0):

```
np.random.random(shape)
```

- Random integer in range [a,b):

```
np.random.randint(a,b,shape)
```

- Gaussian with mean: m and standard deviation s:

```
np.random.normal(m,s,shape)
```

NumPy Arrays – Some Number Types

- Integer: `int8`, `int16`, `int32`, `int64`, uint8
- Float: `float16`, `float32`, `float64`
- Complex: `complex64`, `complex128`
- Boolean: `bool8`
- Unicode string (not a number but still)
- Default type: `float64`
- Can directly create arrays with desired types (applies to most functions):

```
x=np.zeros(10, dtype='int16')
y=np.zeros(10, dtype='float32')
```

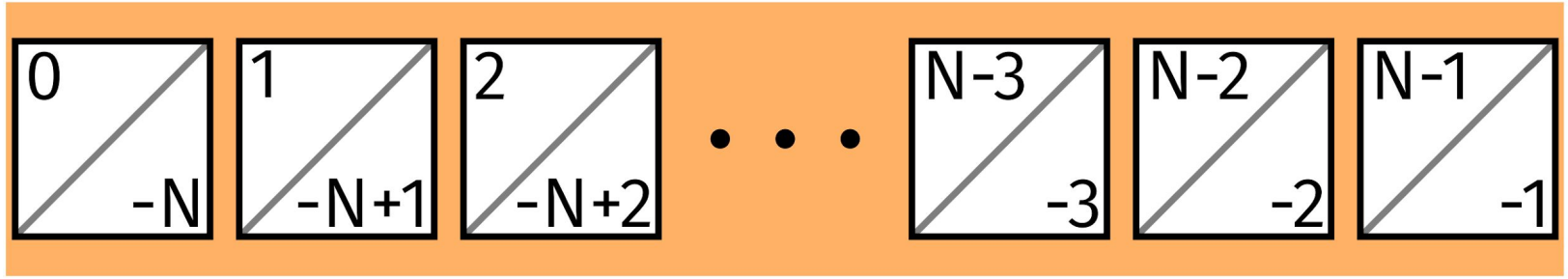
NumPy Arrays – Main Properties

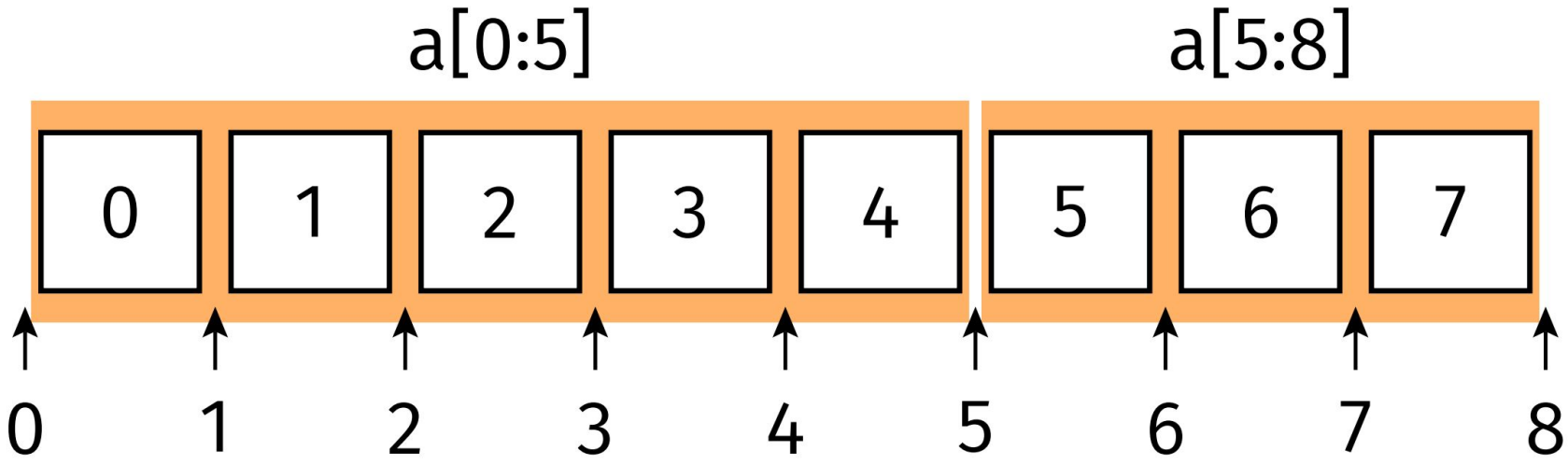
- Number of dimensions: `ndim`
- Size of each dimension: `shape`
- Total number of elements: `size`
- Type of stored data: `dtype`

```
x = np.random.randint(10, size=(3, 4, 5))
print("x ndim:", x.ndim)
print("x shape:", x.shape)
print("x size:", x.size)
print("x dtype:", x.dtype)
```

Outputs?

Recall Indexing and Slicing





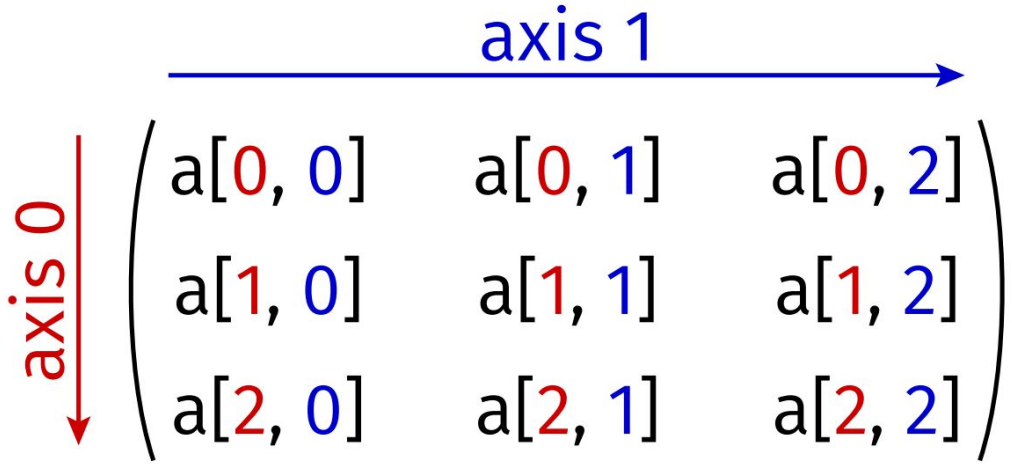
General Slicing: `[start:stop:step]`

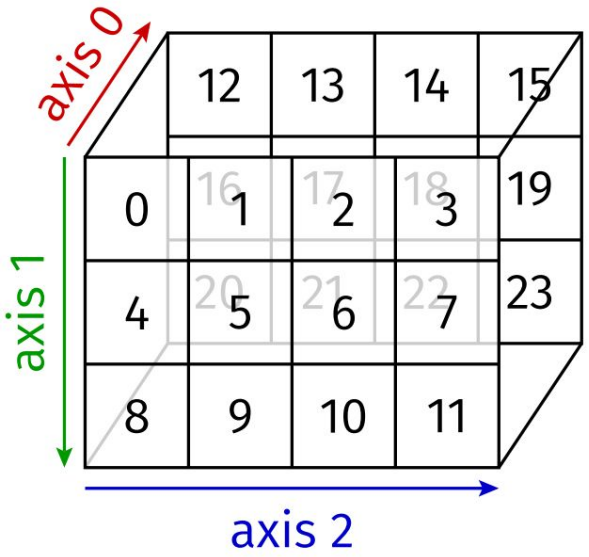
NumPy Arrays – Indexing and Slicing

- In 1D, both are exactly the same as lists, strings etc.
- What about in higher dimensions?
- Indexing:
  - `array[index_tuple]`, where `index_tuple = (i1, i2, ..., iN)`
  - In certain contexts `(i1, i2, ..., iN)` is equivalent to `i1, i2, ..., iN`
  - Thus intuitively: `array[i1, i2, ..., iN]`

```
x = np.random.random((3, 4, 5))
print(x[0,2,1])
```

NumPy Array - Axes






```
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]]])
```

NumPy Arrays – Indexing and Slicing

- Slicing: Syntax is the same as 1D but for each dimension
- The combination is taken:

a[:3, :5]

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	31
32	33	34	35	36	37	38	39

a[-3:, -3:]

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	31
32	33	34	35	36	37	38	39

More Examples

a[:, 3]

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	31
32	33	34	35	36	37	38	39

a[1, 3:6]

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	31
32	33	34	35	36	37	38	39

a[1::2, ::3]

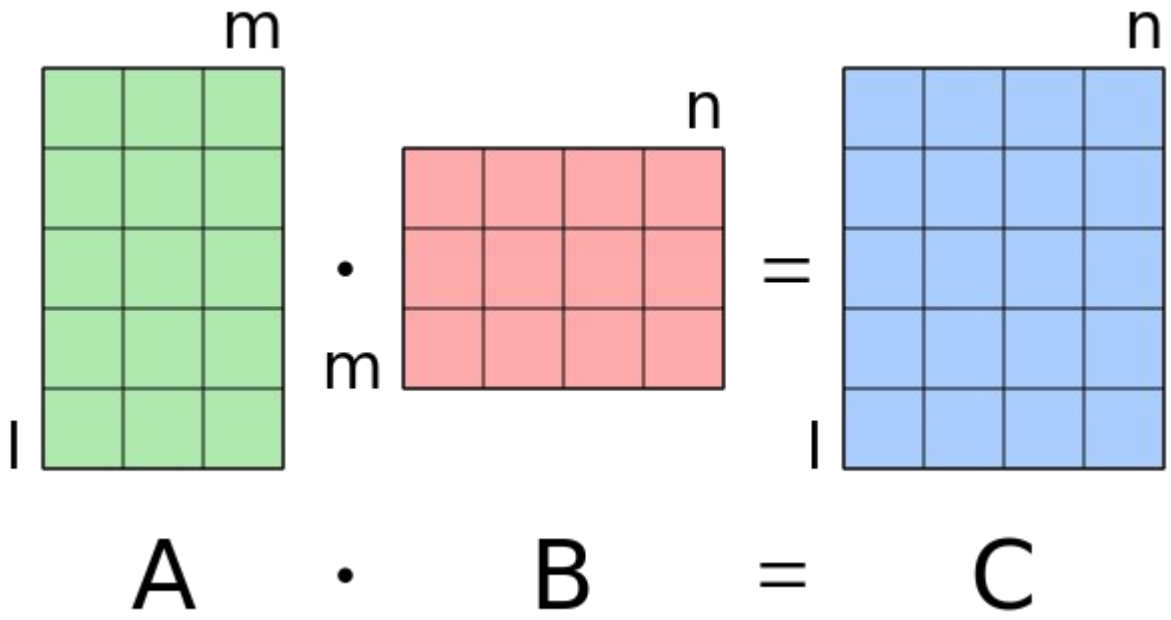
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Important Note:

- The examples show retrieval, but you can do assignment as well!
- For example:

```
a[:,3] = np.zeros(3)
a[:3,:5] = np.zeros((3,5))
```

Matrix Multiplication



$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}\begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} \boxed{6} & 7 \\ 26 & 31 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}\begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 6 & \boxed{7} \\ 26 & 31 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}\begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 6 & 7 \\ \boxed{26} & 31 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}\begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 6 & 7 \\ 26 & \boxed{31} \end{pmatrix}$$

NumPy:

- `np.dot(A,B)`
- `A.dot(B)`
- `A@B`