INTRODUCTION TO PYTHON

LECTURE 5: Numerical Python

Asem Elshimi

Final project roadmap

1^{st} evaluation: March 4^{th}

2^{nd} evaluation: March 11^{th}

March 12^{th}: Final report submission open!

Spring break: March 18^{th} to 23^{rd}

Last time to submit: March 21st

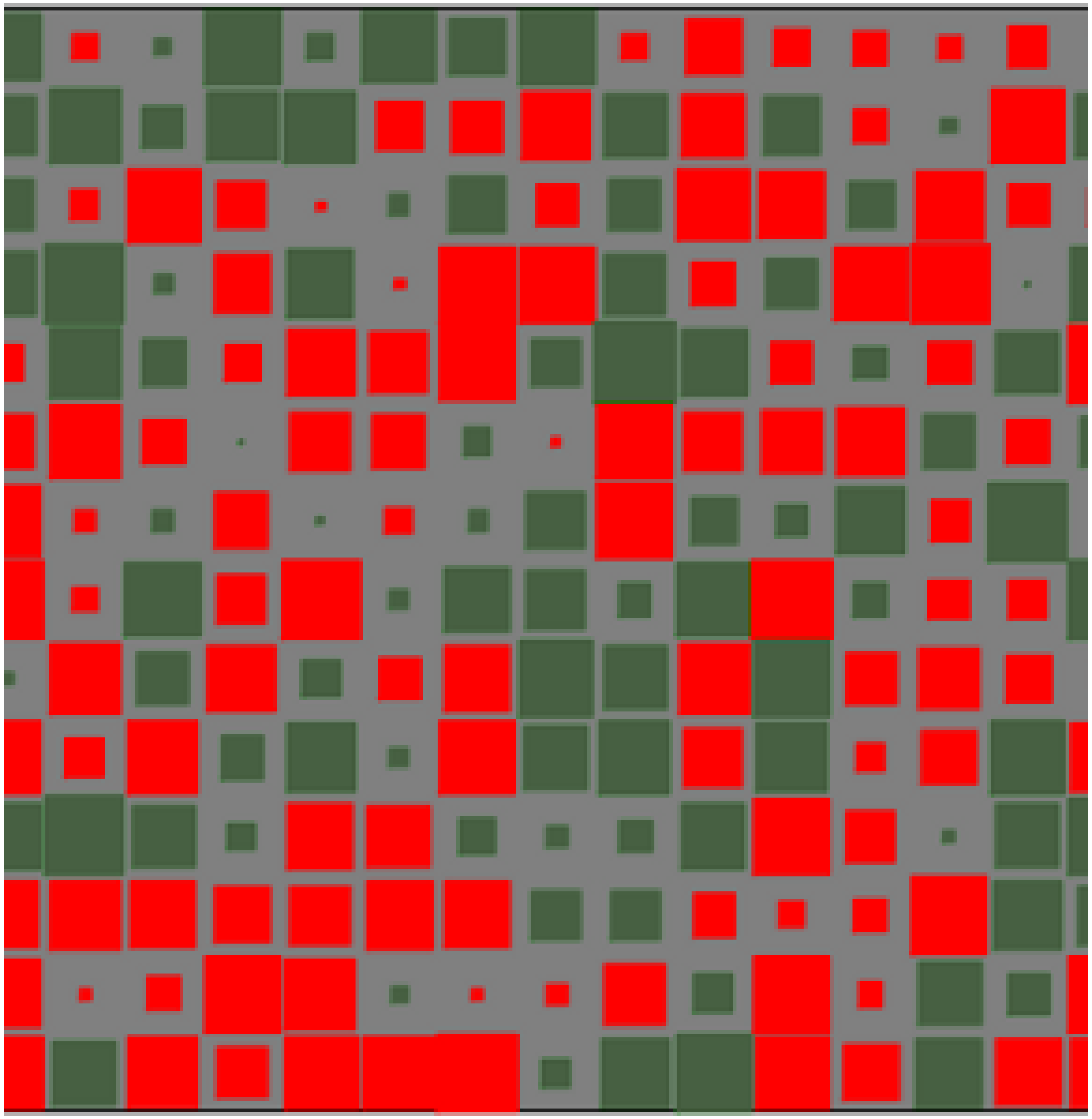
March 2019

Su	Mo	Tu	We	Th	Fr	Sa
24	25	26	27	28	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	1	2	3	4	5	6

Outline

Numpy

Matplotlib



numpy

extension package to Python for multidimensional arrays

closer to hardware (efficiency)

designed for scientific computation (convenience)

Importing libraries

```
from my_file1 import *

#my_file1 includes few functions:
#sum3 and Is_even etc

sum3(4,5)
Is_even(5)
```

Notice different syntax for **`import`**.

Semi-standardized import

```
import numpy as np
import matplotlib.pyplot as plt
```

Interface for matplotlib to make it similar to MATLAB

Array based operations [without loops]

```
#create a list with temperature values
cvalues = list(range(-40,100,20))
#convert to a numpy array
C = np.array(cvalues)
print(C)
type(C)
#convert to F
F= C * 9 / 5 + 32
print(F)
#Lists have to iterate!
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

[-40 -20 0 20 40 60 80]

numpy.ndarray

[-40. -4. 32. 68. 104. 140. 176.]

[-40.0, -4.0, 32.0, 68.0, 104.0, 140.0, 176.0]

Numpy is faster

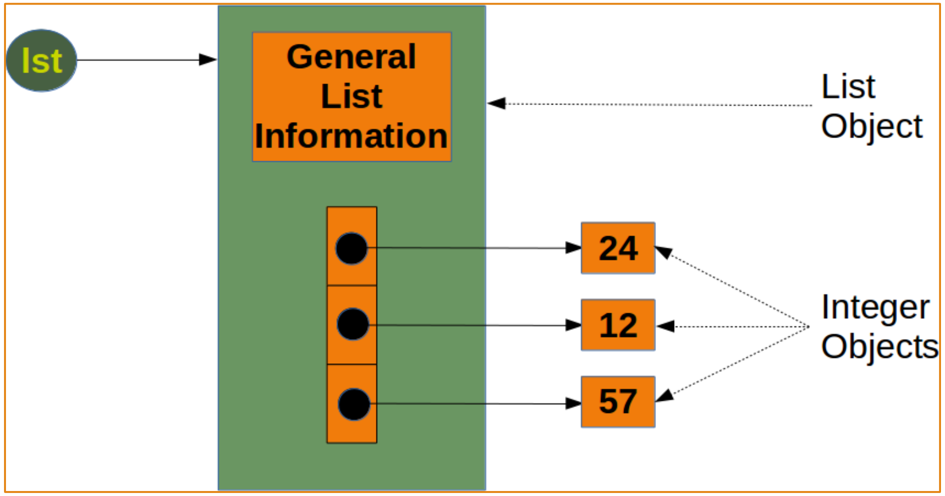
```python
import numpy as np
my_arr = np.arange(1000000)
my_list = list(range(1000000))
#output
```

```python
%time for _ in range(10): my_arr2 = my_arr * 2
%time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

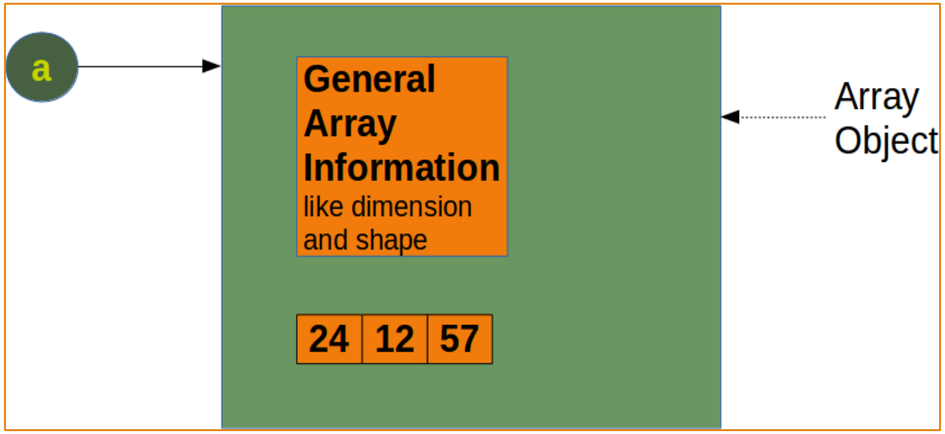
```
Wall time: 26.1 ms
Wall time: 1.07 s
```

Memory usage

Python list:
- List info
- n*pointer size
- n*object size



numpy arrays:
- Array info
- n*object size



Numpy syntax

Object

Creating arrays

Indexing & slicing

Use cases

The NumPy ndarray: A Multidimensional Array Object

```
# Generate some random data
data = np.random.randn(2, 3)
data
```

```
array([[-0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])
```

```
data + data
```

```
array([[-0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

```
print(data.shape)
```

```
(2, 3)
```

```
print(data.dtype)
```

```
float64
```

Creating arrays: from lists

```
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
print(data1)
print(arr1)
```

```
[6, 7.5, 8, 0, 1]
```

```
[6.  7.5 8.  0.  1. ]
```

Creating arrays: 2D -- from lists

```
data2 = [[1, 2, 3, 4], [5, 6, 7,8]]
arr2 = np.array(data2)
Arr2
```

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

```
print(arr2.ndim)
print(arr2.shape)
```

```
2
```

```
(2, 4)
```

Creating arrays: zeros and ones

```
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
np.ones((3, 6))
```

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

```
np.ones_like(arr2)
```

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1]])
```

```
np.full((2, 3, 2),5)
```

```
array([[[5, 5],
        [5, 5],
        [5, 5]],

       [[5, 5],
        [5, 5],
        [5, 5]]])
```

Ones and zeros

`empty`(shape[, dtype, order])	Return a new array of given shape and type, without initializing entries.
`empty_like`(a[, dtype, order, subok])	Return a new array with the same shape and type as a given array.
`eye`(N[, M, k, dtype])	Return a 2-D array with ones on the diagonal and zeros elsewhere.
`identity`(n[, dtype])	Return the identity array.
`ones`(shape[, dtype, order])	Return a new array of given shape and type, filled with ones.
`ones_like`(a[, dtype, order, subok])	Return an array of ones with the same shape and type as a given array.
`zeros`(shape[, dtype, order])	Return a new array of given shape and type, filled with zeros.
`zeros_like`(a[, dtype, order, subok])	Return an array of zeros with the same shape and type as a given array.
`full`(shape, fill_value[, dtype, order])	Return a new array of given shape and type, filled with *fill_value*.
`full_like`(a, fill_value[, dtype, order, subok])	Return a full array with the same shape and type as a given array.

Creating arrays: arrange and linspace

```
print( np.arange(1, 100, 20))
```

[1 21 41 61 81]

```
print(np.linspace(1, 100, 7))
```

[1.   17.5 34.   50.5 67.   83.5 100.]

arange: step.
linspace: number of steps.k

Q: endpoint of linspace?

Q: Multidimensional arange?

Arithmetic [element wise]

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(1 / arr)
print(arr ** 0.5)
arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
print(arr2 > arr)
```

```
[[1.         0.5        0.33333333]
 [0.25       0.2        0.16666667]]

[[1.         1.41421356 1.73205081]
 [2.         2.23606798 2.44948974]]

[[False  True False]
 [ True False  True]]
```

Indexing

ndarray

```
b = np.diag(np.arange(3))

print(b)
```

```
[[0 0 0]
 [0 1 0]
 [0 0 2]]
```

```
b[1,1]
```

```
1
```

list

```
ls1=[[0,0,0],[0,1,0],[0,0,2]]

print(ls1)
```

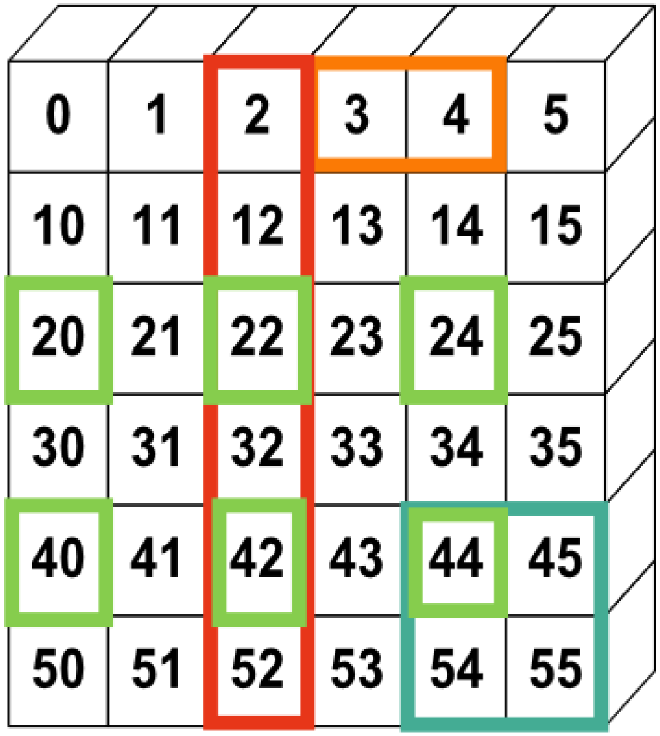
```
[[0, 0, 0], [0, 1, 0], [0, 0, 2]]
```

```
ls1[1][1]
```

```
1
```

Slicing

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



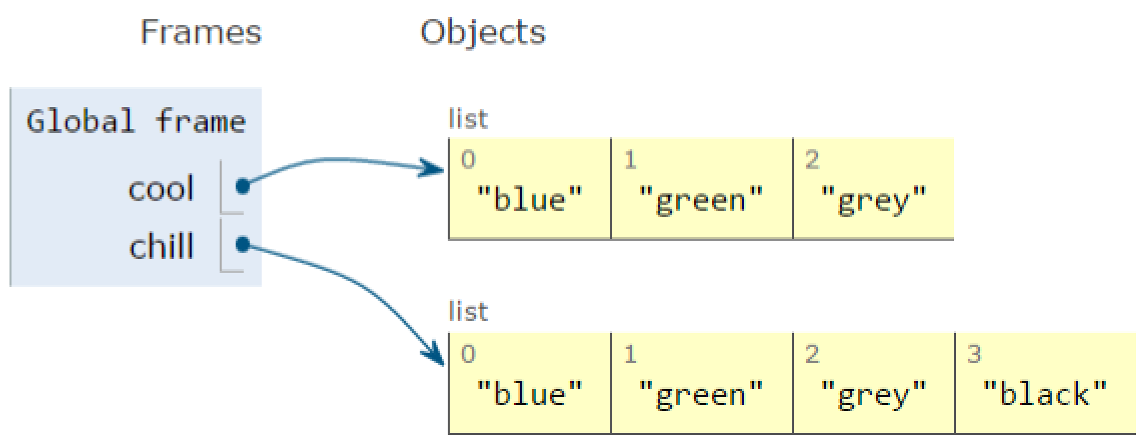
Recall cloning from lists

Create a new list and copy every element using:

chill = cool[:]

```python
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```



Mutability

list

```
lst = [0, 1, 2, 3, 4,
5, 6, 7, 8, 9]
lst2 = lst[2:6]
lst2[0] = 22
lst2[1] = 23
print(lst)
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

ndarray

```
A = np.array([0, 1, 2,
3, 4, 5, 6, 7, 8, 9])
S = A[2:6].copy()
S[0] = 22
S[1] = 23
print(A)
```

[0 1 2 3 4 5 6 7 8 9]

Boolean indexing

```
data = np.random.randn(7, 4)
print(data)
```

```
[[-1.66242323  0.97689833  1.63151977 -0.30667844]
 [ 2.15915191  0.49681718  0.39358763 -2.15413683]
 [ 0.464344    2.21659482 -2.59028366  0.59281529]
 [ 0.2132683  -0.65344045 -0.1768013   1.05902399]
 [-0.48605123  1.39963788  0.39012119 -1.23126857]
 [ 1.23357431  0.45750964  1.77420339 -0.06870388]
 [-2.2654817   0.63990272  0.54798449  3.26197398]]
```

```
boolmask=data<0
print(boolmask)
```

```
[[ True False False  True]
 [False False False  True]
 [False False  True False]
 [False  True  True False]
 [ True False False  True]
 [False False False  True]
 [ True False False False]]
```

```
data[data<0]=0
print(data)
```

```
[[0.         0.97689833 1.63151977 0.        ]
 [2.15915191 0.49681718 0.39358763 0.        ]
 [0.464344   2.21659482 0.         0.59281529]
 [0.2132683  0.         0.         1.05902399]
 [0.         1.39963788 0.39012119 0.        ]
 [1.23357431 0.45750964 1.77420339 0.        ]
 [0.         0.63990272 0.54798449 3.26197398]]
```

Using Numpy

So many applications.

Linear algebra:

- Specially to solve optimization problems.

Loop vectorization.

Control systems and signal processing.

Data analysis [Pandas is much better.]

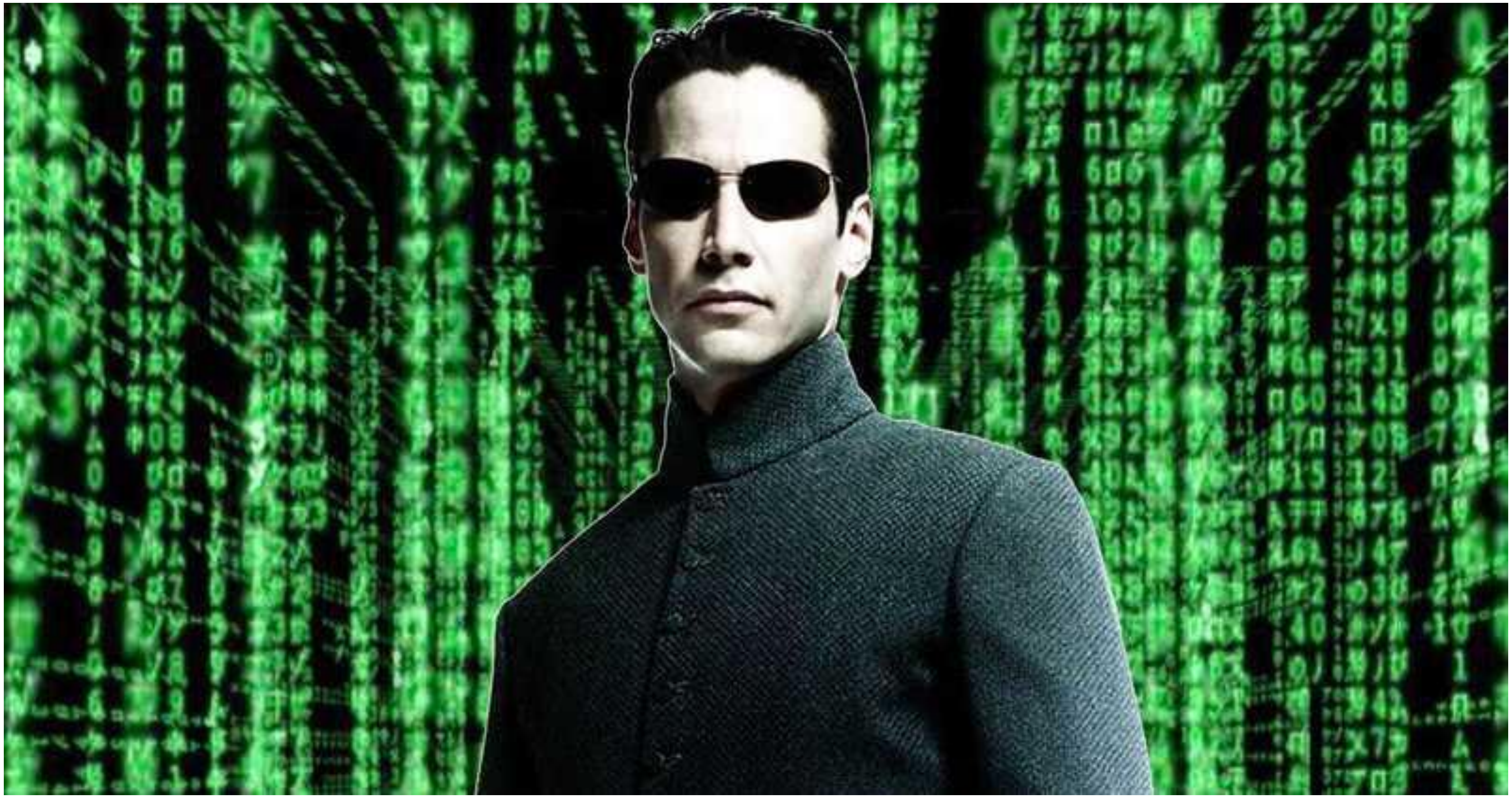
Linear algebra: scalar product

```
x = np.array([1,2,3])

y = np.array([-7,8,9])


dot = np.vdot(x,y)
```

36

$$\mathbf{a}\cdot\mathbf{b}=\sum a_i\overline{b_i}$$

Matrix operations



Matrix operations

FUNCTION	DESCRIPTION
matmul()	Matrix product of two arrays.
inner()	Inner product of two arrays.
outer()	Compute the outer product of two vectors.
linalg.multi_dot()	Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.
tensordot()	Compute tensor dot product along specified axes for arrays >= 1-D.
einsum()	Evaluates the Einstein summation convention on the operands.
einsum_path()	Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays.
linalg.matrix_power()	Raise a square matrix to the (integer) power n.
kron()	Kronecker product of two arrays.

np.linalg

```
A = np.array([[6, 1, 1],
              [4, -2, 5],
              [2, 8, 7]])
print(np.linalg.matrix_rank(A))
print(np.trace(A))
print(np.linalg.det(A))
print(np.linalg.inv(A))
print(np.linalg.matrix_power(A, 3))
```

```
3

11

-306.0

[[ 0.17647059 -0.00326797 -0.02287582]
 [ 0.05882353 -0.13071895  0.08496732]
 [-0.11764706  0.1503268   0.05228758]]

[[336 162 228]
 [406 162 469]
 [698 702 905]]
```

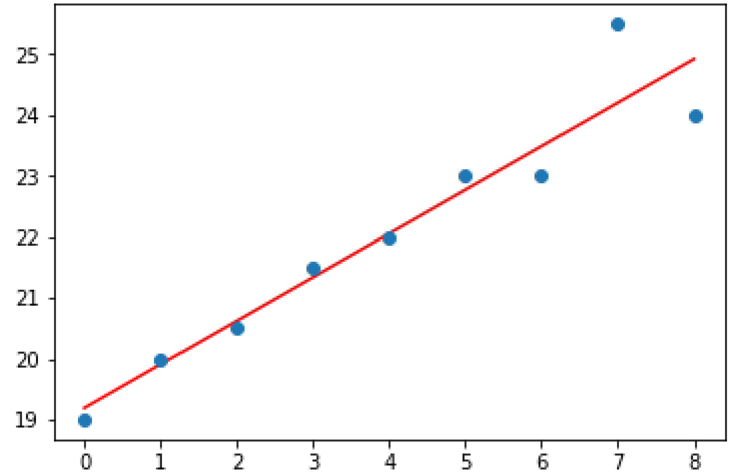
Eigen values

```
a = np.diag((1, 2, 3))
print(a)
vals, vects = np.linalg.eig(a)


print(vals)
print(vects)
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]

[1. 2. 3.]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Least squares

```
# x co-ordinates

x = np.arange(0, 9)

# linearly generated sequence

y = [19, 20, 20.5, 21.5, 22, 23, 23, 25.5, 24]

plt.plot(x, y, 'o')

plt.show()

# obtaining the parameters of regression
line
A = np.array([x, np.ones(9)])
w = np.linalg.lstsq(A.T, y)[0]
 # plotting the line
line = w[0]*x + w[1] # regression line
plt.plot(x, line, 'r-')
plt.plot(x, y, 'o')
plt.show()
```

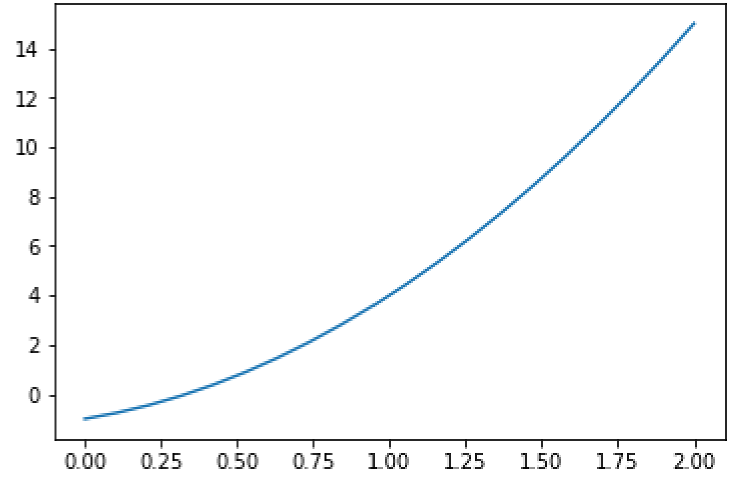


Polynomials

$$3x^2 + 2x - 1$$

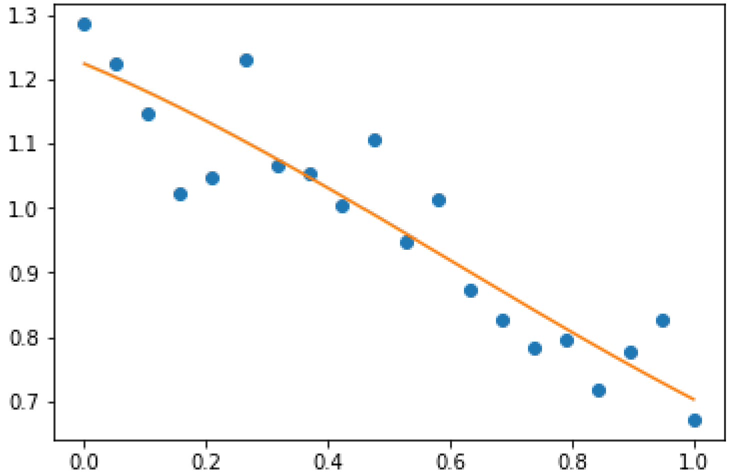
```
p = np.poly1d([3, 2, -1])
p
p(0)
p.roots
p.order
x = np.linspace(0, 2, 20)
y=p(x)
plt.plot(x,y)
```

```
poly1d([ 3,  2, -1])
 -1
array([-1.        ,  0.33333333])
```



Polynomials

```
x = np.linspace(0, 1, 20)
y = np.cos(x) + 0.3*np.random.rand(20)
p = np.poly1d(np.polyfit(x, y, 3))

t = np.linspace(0, 1, 200)
plt.plot(x, y, 'o', t, p(t), '-')
plt.show()
```



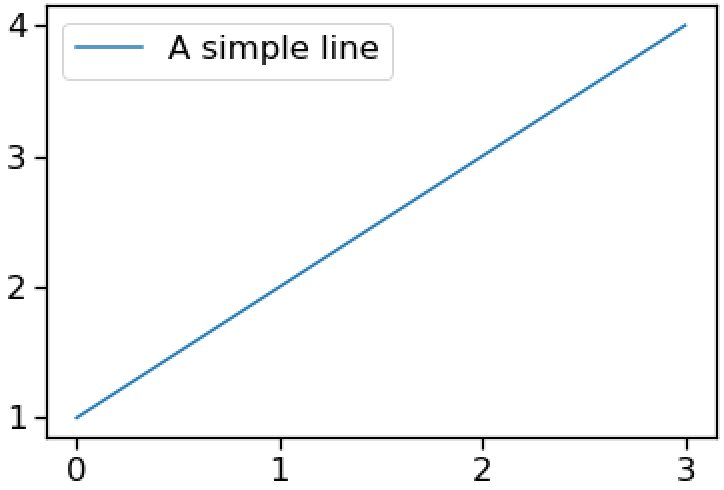
Matplotlib

Replaces matlab

Generate scientific quality plots (more on this next lecture)

A simple example

```
import numpy as np
import matplotlib.pyplot as plt
ax = plt.gca()
ax.plot([1, 2, 3, 4])
ax.legend(['A simple line’])
plt.show()
```

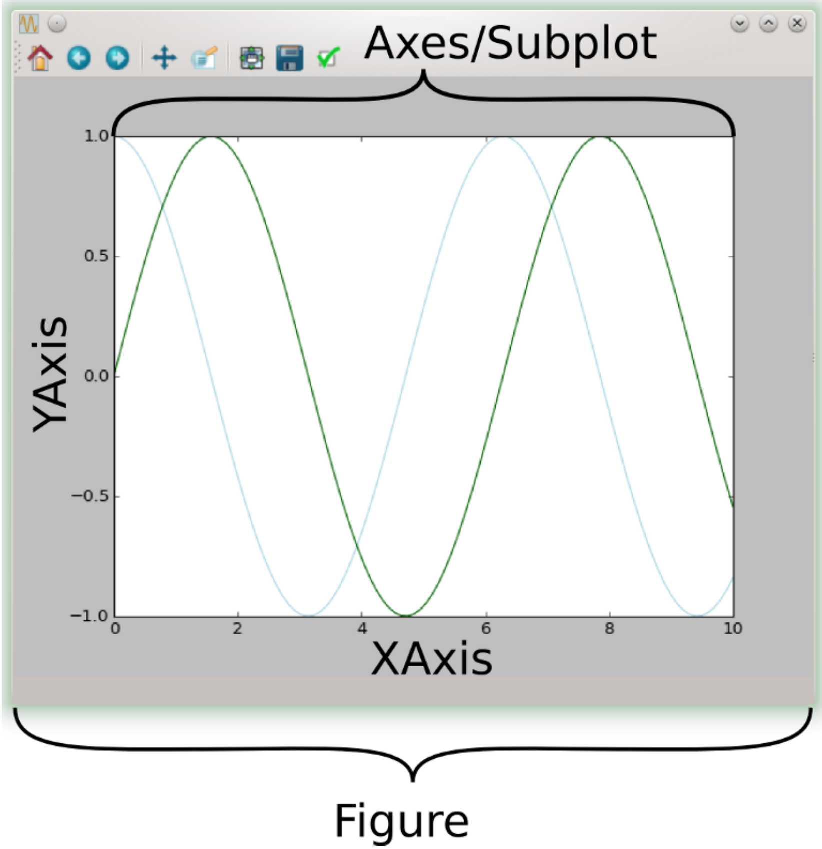


Anatomy of a plot

The Figure is the top-level container in this hierarchy.

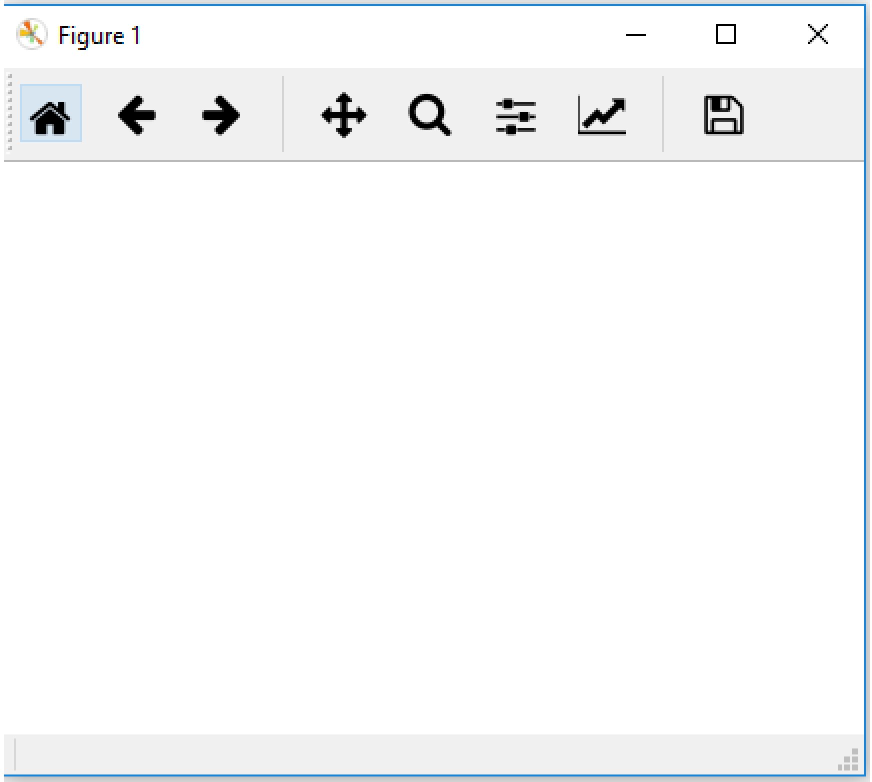
Most plotting occurs on an Axes.

Axes and Subplot are almost synonymous.



Figures

```
fig = plt.figure()
plt.show()
```



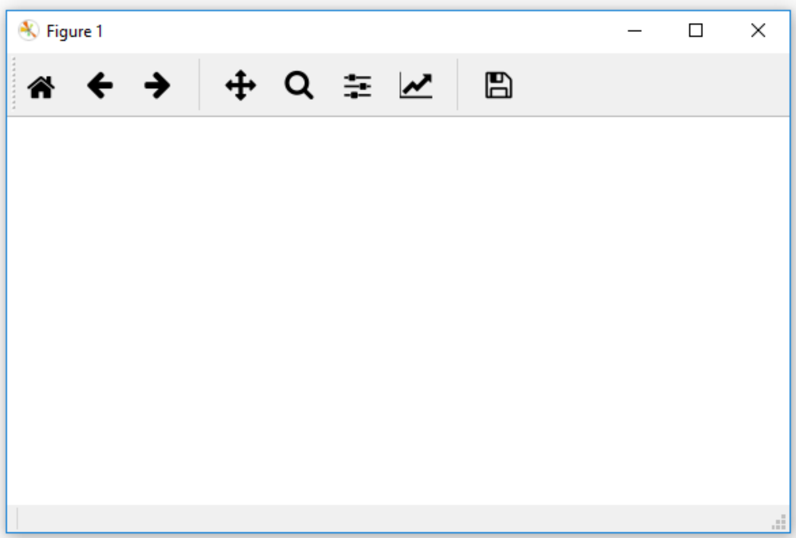
You might need:

%matplotlib [Spyder]

Or %matplotlib inline [Jupyter]

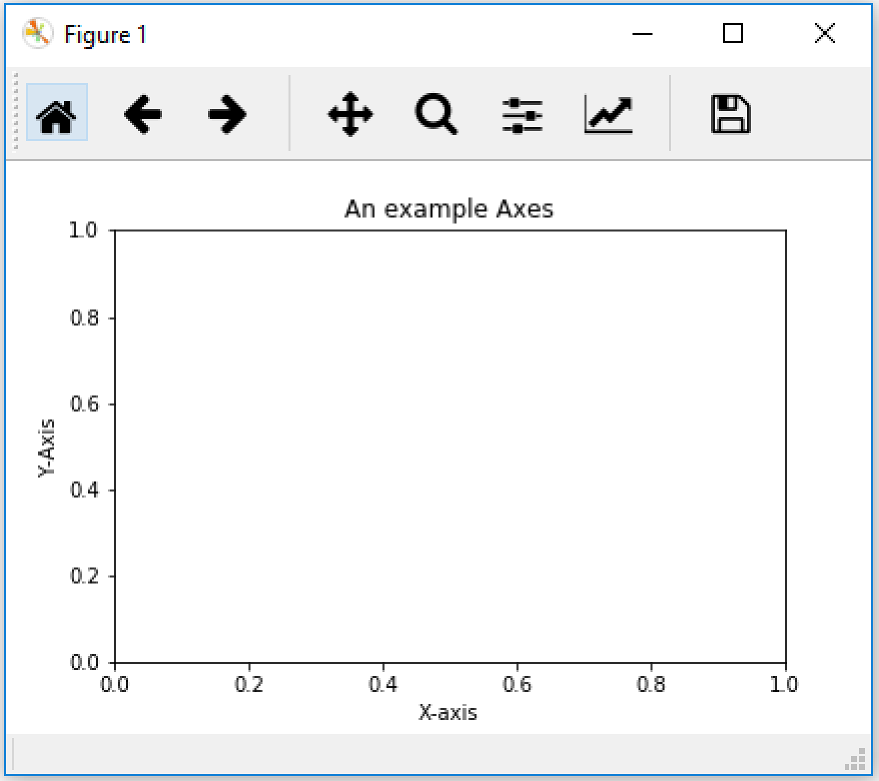
figsize=(width, height)

```
# Twice as wide as it is tall:
fig = plt.figure(figsize=plt.figaspect(0.5))
plt.show()
```



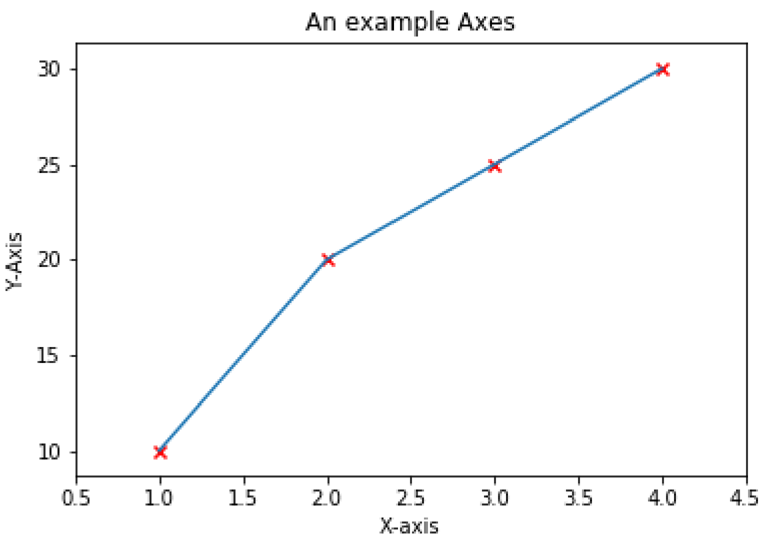
axes

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-Axis')
ax.set_title('An example Axes')
plt.show()
```



Exercise: explore tab completion with ax.set_

Basic plotting




```
...
xdata=[1, 2, 3, 4]
ydata=[10, 20, 25, 30]
ax.plot(xdata, ydata)
ax.scatter(xdata,ydata, marker='x', color='r')
ax.set_xlim(0.5, 4.5)
plt.show()
```

Axes methods vs. pyplot

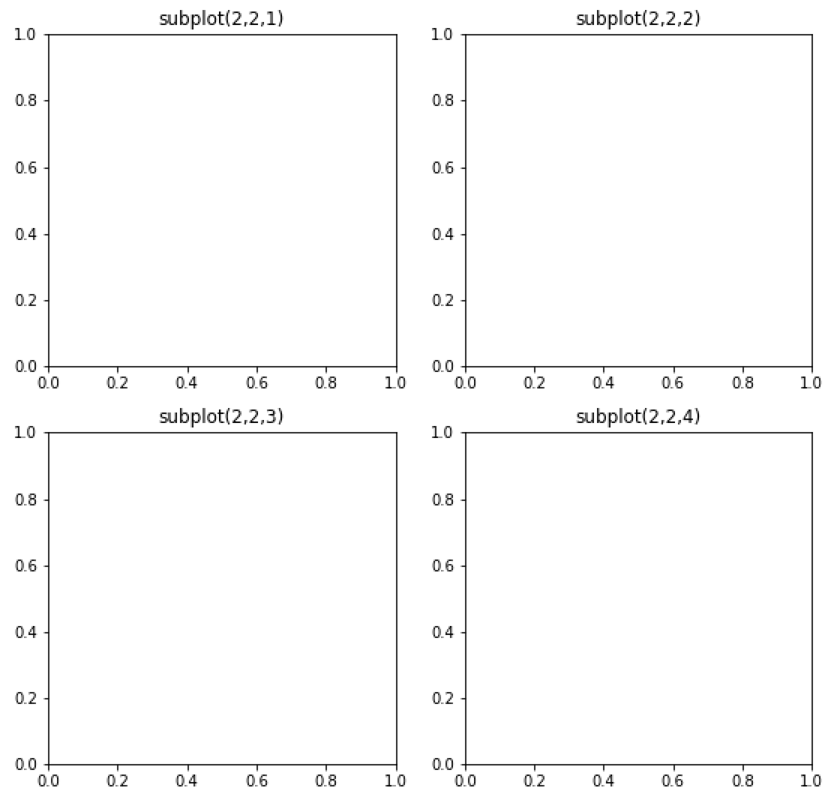
```
...
plt.plot(xdata, ydata)
            =[implicitly calls]=
                          ax.plot(xdata, ydata)
```

"The Zen of Python" says:

"Explicit is better than implicit"

Multiple axes in one figure: subplots

```
fig=plt.figure()
ax=fig.add_subplot(2,2,1)
ax.set_title('subplot(2,2,

ax=fig.add_subplot(2,2,2)
ax.set_title('subplot(2,2,

ax=fig.add_subplot(2,2,3)
ax.set_title('subplot(2,2,

ax=fig.add_subplot(2,2,4)
ax.set_title('subplot(2,2,
plt.show()
```



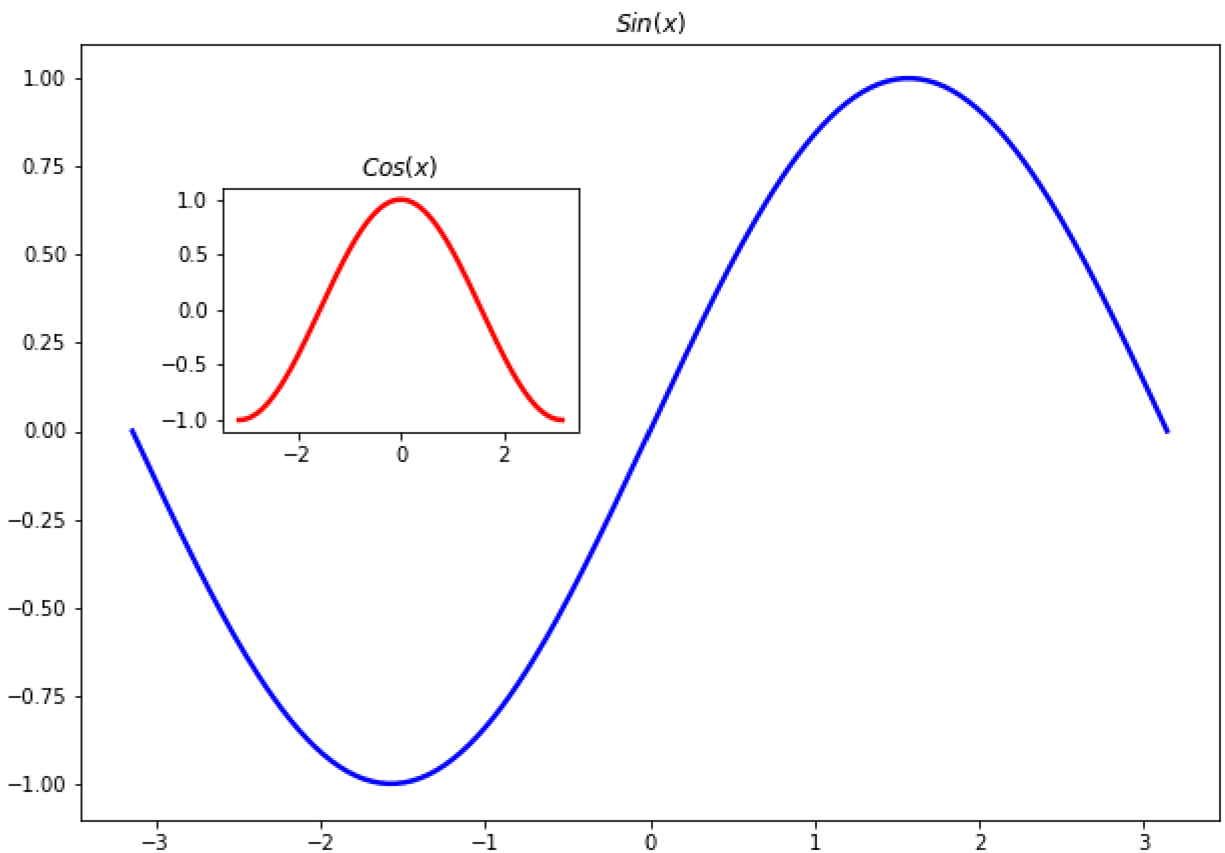
More advanced

Advanced is: https://matplotlib.org/users/gridspec.html

```
ax = plt.subplot2grid((2, 2), (0, 0))
```

Axes inside axes

```
X =
C,S

fig=

#[le
ax1=
ax1
ax1

ax2=
ax2
ax2

plt
```



Next lecture:

This lecture: matplotlib anatomy

Next lecture: the art of creating beautiful graphs

Questions?

THANK YOU!