Numpy and Matplotlib

Measuring executing time

time.time() returns the time in seconds from a reference instant in the past (the reference changes depending on the operative system)

```
import time

t_start = time.time()
[some code]
t_end = time_time()
print(t_end-t_start)
```

This code provides an estimate of the elapsed time (not of the computing time, e.g. the CPU might be do something else for most of the elapsed time)

```
%timeit [some code]
```

This provides a more accurate estimate of the computing time (in ipython or notebooks)

Why lists are not enough ?

- Elementary data types are objects, e.g. an integer is more than just an integer
- There's a massive overload when sequence of numbers are stored as lists (each item is an object)
- Multidimensional lists are possible (as lists of lists), but indexing is complicated

NUMPY ARRAYS – np.ndarray

- All the elements are of the same type
- Mathematical operations are optimized (C++ routines)
- More indexing options are available
- It is easier to deal with multidimensional data

Basic data types for numpy arrays

bool	True/False
int8, int16, int32, int64 (int)	1, 2, 4, 8 bytes integers
uint8, uint16, uint32, uint64 (uint)	1, 2, 4, 8 bytes unsigned integers
float16	sign bit, 5 bits exponent, 10 bits mantissa (half-precision)
float32 (float)	sign bit, 8 bits exponent, 23 bits mantissa (single-precision)
float64	sign bit, 11 bits exponent, 52 bits mantissa (double-precision)
complex64	complex numbers, made of 2 float32
complex128 (complex)	complex numbers, made of 2 float64

All the elements of an array are of the same type

Initialization of arrays from lists

```
x = np.array([0,1,2,3]) # 1D array of int (int64)
y = np.array([0.0, 1, 2, 3]) # 1D array of float (float64)
z = np.array([0, True]) # 1D array of int (int64)
w = np.array([0, True], dtype = np.bool) # 1D array of bool
q = np.array([0, True], dtype = np.float64) # 1D array of float64


A = np.array([[0,1,2],[3,4,5]]) # 2D array

B = np.array([ [ [0,1], [2,3] ], [ [4,5], [6,7] ] ]) # 3D array
```

Functions to create numpy arrays

np.zeros(shape [, dtype])	array of zeros • shape is an integer for 1-dimensional arrays, or a sequence of integers for multi-dimensional arrays • dtype is the type of the elements, defined as a string or using the corresponding numpy object np.zeros(10, dtype = np.int64)
np.ones(shape [, dtype])	array of ones
np.full(shape, fill_value, [, dtype])	Array with all values equal to fill_value
np.empty(shape [, dtype])	Array with uninitialized elements
np.eye(n [, dtype])	n x n identity matrix

Functions to create numpy arrays

np.arange([start,] stop [, step])	Evenly spaced numbers between start and stop (not included) with period step np.arange(5) → 0,1,2,3,4 np.arange(2,10) → 2,3,4,5,6,7,8,9 np.arange(2,10,2) → 2,4,6,8
np.linspace(start, stop, n)	n evenly spaced numbers between start and stop (included) np.linspace(0,5,6) → 0,1,2,3,4,5
np.logspace(start, stop, n)	n evenly spaced numbers in logarithmic scale between 10**start and 10**stop (included) np.logspace(-3,3,7) → 1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03

attributes of numpy arrays

- .ndim → number of dimensions
- .shape → a tuple with the number of elements along each dimension
- .size → total number of elements
- .dtype → type of the items
- .itemsize → size in bytes of each element
- .nbytes → size in bytes of the entire array

```
x = np.ones((4,5,2))
x.ndim → 3
x.shape → (4,5,2)
x.size → 40
x.dtype → float64
x.itemsize → 8
x.bytes → 320
```

Indexing & Slicing (1/3)

- As in lists, but with different dimensions separated by commas

```
x = array([[12,  3,  2,  9],
      [ 1,  5,  0,  7],
      [ 5,  3,  4,  1]])
x[1,2] → 0
x[::-1,2] → array([4, 0, 2])
x[:2,1:3]
→ array([[3, 2],
      [5, 0]])
```

```
x[:,2] → array([2, 0, 4])
x[1,:] → array([1, 5, 0, 7])
```

slicing return views (not copies)

```
x = array([[12,  3,  2,  9],
     [ 1,  5,  0,  7],
     [ 5,  3,  4,  1]])
col0 = x[:,0]
col0[0] = 13
x → array([[13,  3,  2,  9],
     [ 1,  5,  0,  7],
     [ 5,  3,  4,  1]])
```

- This is different from list, where slicing return copies

- It is possible to explicitly ask for a copy, with the attribute copy()

```
x = array([[12,  3,  2,  9],
     [ 1,  5,  0,  7],
     [ 5,  3,  4,  1]])
col0 = x[:,0].copy()
col0[0] = 13
x → array([[12,  3,  2,  9],
     [ 1,  5,  0,  7],
     [ 5,  3,  4,  1]])
```

Array reshaping

ndarray.reshape(new_shape) → Returns a new array with the requested shape, where new_shape is a tuple. Error if the number of elements in the array do not fit into the new shape.

x = np.arange(0,9).reshape((3,3))

→ 3 x 3 matrix with elements from 0 to 9 in row order

x = np.arange(0,9).reshape((-1,1))

→ it creates a 9 x 1 column matrix. -1 is the syntax for: how many elements are needed in order to use all the items in the array

x = np.arange(0,9).reshape((1,-1))

→ same as before for creating a row array

Concatenating arrays

np.concatenate(seq_of_arrays) → returns a new array with all the arrays in seq_of_arrays (tuple or list) concatenated

```
x = np.concatenate([np.arange(5), np.arange(2), np.arange(3)])
→ array([0, 1, 2, 3, 4, 0, 1,0,1,2])
```

It works also for multidimensional arrays. In that case, it is possible to choose the concatenation axis

```
x = np.concatenate([np.eye(3), np.eye(3)], axis = 0)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
x = np.concatenate([np.eye(3), np.eye(3)], axis = 1)
```

```
array([[1., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0., 1.]])
```

All the dimensions (apart from the concatenated axis) must be identical

np.vstack – vertical stack

All the dimensions apart from the concatenated one need to be the same

It concatenates arrays along the row axis (it's the same as np.concatenate along axis 0)

```
x = np.vstack([np.eye(3), np.eye(3)])
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

np.hstack – horizontal stack

It concatenates arrays along the column axis (it's the same as np.concatenate along axis 1)

```
x = np.hstack([np.eye(3), np.eye(3)])
```

```
array([[1., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 1., 0.],
       [0., 0., 1., 0., 0., 1.]])
```

“reducing” methods

np.sum	
np.prod	
np.min	
np.max	
np.mean	
np.median	
np.var	variance
np.std	standard deviation
np.any	True if any element is true
np.all	True if all elements are true

- All these functions can also be called as methods
- If no optional argument is provided, they reduce the array to one value (regardless of the number of dimensions)

np methods Vs built-in reducing functions (max, min, sum, any, all)

```
import numpy as np

x = np.random.uniform(size = 1000)
%timeit sum(x) → 85.1 µs ± 4.96 µs per loop
%timeit np.sum(x) → 3.11 µs ± 116 ns per loop
%timeit max(x) → 66 µs ± 6.34 µs per loop
%timeit np.max(x) →3.23 µs ± 96.5 ns per loop
```

numpy methods are much faster !

axis argument

- When the axis argument is defined, the reducing method reduces along that dimension

```
x = np.random.randint(0,10,size = (3,4))
np.sum(x, axis = 0)
→ array([22, 21, 18, 22])
np.sum(x, axis = 1)
→ array([32, 26, 25])
```

with axis = 0, the method works along the rows

```
array([[9, 8, 6, 9],
       [5, 7, 9, 5],
       [8, 6, 3, 8]])
```

```
x = np.random.randint(0,10,size = (3,4,5))
np.sum(x, axis = 0)
→ it gives back a (4,5) array by summing along
dimension 0
```

same for multidimensional arrays

memory layout

- When organizing n-dimensional data in memory, the order of the items need to be defined

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

0,0	0,1	0,2	1,0	1,1	1,2	2,0	2,1	2,2

row-major (the last index is the one changing the fastest)

0,0	1,0	2,0	0,1	1,1	2,1	0,2	1,2	2,2

column-major (the last index is the one changing the slowest)

- Row-major is the style adopted by C/C++

→ Numpy uses row-major by default, but it is possible to use column-major

- Column-major is the style adopted by Fortran

→ Matlab, R, Julia use column-major by default [LAPACK libraries are used for linear algebra]

```
x = np.array([[0,1,2],[3,4,5],[6,7,8]])
```

The default is to store arrays as in C, so here 0,1,2 are contiguous in memory

```
x = np.array([[0,1,2],[3,4,5],[6,7,8]], order = ‘F’)
```

Instead in this case 0,3,6 are contiguous in memory

Why should I care about memory layout ?

- If arrays are analysed along the fastest moving index, the code works on data that are contiguous in memory, so cache optimization and vectorization are possible

→Performances can dramatically change

- np.asfortranarray: converts an array into fortran-style

- np.ascontigousarray: converts an array into C-style

ufuncs – Universal functions

- Functions that operate on numpy arrays element-by-element in a vectorized form

https://docs.scipy.org/doc/numpy/reference/ufuncs.html

https://docs.scipy.org/doc/scipy/reference/special.html

np.abs, np.absolute	
np.sqrt, np.pow	
np.sin, np.cos, np.tan	
np.arcsin, np.arccos, np.arctan, np.arctan2	
np.deg2rad, np.rad2deg	convert degree to radians, or viceversa
np.sinh, np.cosh, np.tanh	
np.arcsinh, np.arccosh, np.arctanh	
np.exp	
np.exp2	2**
np.log, np.log2, np.log10	
np.expm1	exp(x)-1 useful to have better precision
np.log1p	log(1+x)

np.isfinite	Test arrays element-wise
np.isinf	
np.isnan	

ufuncs – vectorized functions for np.arrays

```
import time
import numpy as np

n = 100000
x = np.arange(n)

def temporize(func, n_repeats, *args):
    t_start = time.time()
    for i in range(n_repeats):
        func(*args)
    t_end = time.time()
    return (t_end-t_start)/n_repeats
```

```
def square_with_cycle(x):
    x2 = np.empty(x.shape)
    for i in range(x.size):
        x2[i] = x[i]**2.0
    return x2

def square_with_ufuncs(x):
    return x**2.0
```

```
print(temporize(square_with_cycle, 100, x)) → 0.23 s
print(temporize(square_with_ufuncs, 100, x)) → 0.00013 s
```

+	np.positive (unary)
-	np.negative (unary)
+	np.add
-	np.subtract
*	np.multiply
/	np.divide
//	np.floor_divide
%	np.mod
**	np.power

<table>
<tr><td>&</td><td>np.logical_and</td><td rowspan="4">These methods work element-wise
Do not use and, not, or operators with np.arrays</td></tr>
<tr><td>|</td><td>np.logical_or</td></tr>
<tr><td>~</td><td>np.logical_not</td></tr>
<tr><td>^</td><td>np.logical_xor</td></tr>
</table>

==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

np.all
np.any

Arithmetic operations

- Arithmetic operations on arrays are done element-wise

$$\begin{bmatrix} 1 & 2 & 1 \\ 4 & 1 & 3 \\ 6 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 2 & 1 & 1 \\ 1 & 4 & 5 \\ 3 & 1 & 7 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 1 \\ 4 & 4 & 15 \\ 18 & 2 & 7 \end{bmatrix}$$

Arithmetic operations with scalars

- Operations with scalars are applied to all elements

$$\begin{bmatrix} 1 & 2 & 1 \\ 4 & 1 & 3 \\ 6 & 2 & 1 \end{bmatrix} * 4 = \begin{bmatrix} 4 & 8 & 4 \\ 12 & 4 & 12 \\ 24 & 8 & 4 \end{bmatrix}$$

Broadcasting

- When arithmetic operations involve arrays of different shapes, the arrays are broadcasted to the same shape using the following rules

1. If the number of dimensions is different, the array with smaller number of dimensions is reshaped adding dimensions equal to 1 on the left

2. If the number of elements along any direction is different
   - if one of the two arrays have shape 1, that direction is expanded to fit with the bigger array
   - If both shapes are not 1 → ERROR

```
A = np.ones((3,4))
b = 2*np.ones(4)
C = A + b
```

- b is reshaped to (1,4)
- Now the shapes are (3,4) and (1,4)
- The first dimension is different, but one of the two shape is 1, so this is expanded to 3
- Now the arrays have the same shape and the operation can proceed

A

1	1	1	1
1	1	1	1
1	1	1	1

b

2	2	2	2

→

b

2	2	2	2
2	2	2	2
2	2	2	2

```
A = np.ones((3,4))
b = 2*np.ones(3)
C = A + b
```

- b is reshaped to (1,3)
- Now the shapes are (3,4) and (1,3)
- The first dimension is different, but one of the two shapes is 1, so this is expanded to 3
- The second dimension is different but both shapes are different than 1

→ The sum is not possible

```
a = np.ones((3,1))
b = np.ones(3)
C = a + b
```

- b is reshaped to (1,3)
- Now the shapes are (3,1) and (1,3)
- The first dimension is different, but the shape of b is 1, so b is expanded to (3,3)
- The second dimension is different, but the shape of a is 1, so a is expanded to (3,3)

→ The sum gives back a (3,3) array

```python
x1 = np.array([6,2,3])
x2 = np.array([4,1,9])
z = x1.reshape(3,1) – x2.reshape(1,3)
```

$$\begin{bmatrix} 6 \\ 2 \\ 3 \end{bmatrix} - \begin{bmatrix} 4 & 1 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 6 & 6 & 6 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} - \begin{bmatrix} 4 & 1 & 9 \\ 4 & 1 & 9 \\ 4 & 1 & 9 \end{bmatrix} = \begin{bmatrix} 2 & 5 & -3 \\ -2 & 1 & -7 \\ -1 & 2 & -6 \end{bmatrix}$$

It is the difference between all possible pairs of elements

```python
x1 = np.array([6,2,3])
x2 = np.array([4,1,9])
z = np.subtract.outer(x1, x2)
```

another strategy to do the same

Broadcasting Vs cycles

```
import numpy as np
X = np.random.uniform(size = 1000)
Y = np.random.uniform(size = 1000)
```

```
def pair_distance(X,Y):
    dist = np.empty((len(X),len(Y)))
    for i,x in enumerate(X):
        for j,y in enumerate(Y):
            dist[i,j] = abs(X[i]-Y[j])
    return dist
```

%timeit pair_distance(X,Y)

→ 513 ms ± 2.02 ms

%timeit np.abs(X.reshape(-1,1)-Y.reshape(1,-1))

→ 5.12 ms ± 101 µs

Broadcasting rules apply to any binary ufuncs

```
A = np.arange(12).reshape(3,4)
b = np.array([0,1,2,3])
```

```
np.logical_and(A < 5, b > 1)
```

A

0	1	2	3
4	5	6	7
8	9	10	11

b

0	1	2	3

A < 5

T	T	T	T
T	F	F	F
F	F	F	F

b > 1

F	F	T	T
F	F	T	T
F	F	T	T

Counting elements

```
A = np.array([ [2,5,-3,0], [-2,1,-7,4], [-1,2,-6,-1] ])
```

2	5	-3	0
-2	1	-7	4
-1	2	-6	-1

np.sum(A > 0) → 5

np.sum(A > 0, axis = 0) → 1, 3, 0, 1

np.sum((A > 0) & (A < 5)) → 4

Here, the bitwise and operator is used. As we're dealing with boolen values, it's the same as the logical and

&	np.logical_and	These methods work element-wise Do not use and, not, or operator with np.arrays
|	np.logical_or	
~	np.logical_not	
^	np.logical_xor	

Indexing arrays (2/3) using lists/arrays of indexes

1-dimensional arrays

```
x = np.linspace(0,100,11)
→ array([  0.,  10.,  20.,  30.,  40.,  50.,  60.,  70.,  80.,  90., 100.])
i = [4, 2, 1] # i = np.array([4,2,1]) would give the same result
x[i] → array([40., 20., 10.])
i = np.array([[4,6],[2,3]])
x[i] → array([[40., 60.], [20., 30.]]) # the result has the shape of the indexes
```

```
A = np.array([ [2,5,-3,0], [-2,1,-7,4], [-1,8,-6,9] ])
```

2	5	-3	0
-2	1	-7	4
-1	8	-6	9

```
i = [1,2]
j = [0,1]
```

A[i,j]

```
array([-2,  8])
```

```
i = np.array([[0,1],[1,0]])
j = np.array([[2,2],[0,0]])
```

A[i,j]

```
array([[-3, -7],
       [-2,  2]])
```

As before, the shape is the shape of the indexes

If indexes have different shapes, broadcasting rules are applied

```
i = np.array([0,1]).reshape(2,1)
j = np.array([2,3]).reshape(1,2)
```

A[i,j]

```
array([[-3,  0],
       [-7,  4]])
```

Indexing arrays (3/3)
Boolean arrays as masks

```
A = np.array([ [2,5,-3,0], [-2,1,-7,4], [-1,2,-6,-1] ])
```

A[A > 0] → array([2, 5, 1, 4, 2])

A[(A > 0) & (A < 5)] → array([2, 1, 4, 2])

```
B = np.array([ [1,2,4,-1], [5,3,-2,-3], [-4,-1,6,1] ])
```

A[B > 2] → array([-3, -2, 1, -6])

A

2	5	-3	0
-2	1	-7	4
-1	2	-6	-1

B

1	2	4	-1
5	3	-2	-3
-4	-1	6	1

The shapes of the mask and of the array need to be the same

Different indexing schemes (lists/arrays, masks, normal slicing) can be combined

```
A = np.array([ [2,5,-3,0], [-2,1,-7,4], [-1,8,-6,9] ])
```

2	5	-3	0
-2	1	-7	4
-1	8	-6	9

```
i = [3,2,2,1]
```

A[:,i]

```
array([[ 0, -3, -3,  5],
       [ 4, -7, -7,  1],
       [ 9, -6, -6,  8]])
```

sorting

```
import numpy as np
x = np.random.normal(size = 1000)

y = np.sort(x)
i = np.argsort(x) # indexes of the sorted array
```

```
# in place sorting
x.sort()
```

```
%timeit y = np.sort(x)
→ 23.1 µs ± 23.8 ns per loop
%timeit y = sorted(x)
→ 231 µs ± 323 ns per loop
```

np.partion() and np.argpartition() can be used to sort only the first k smallest elements of the array

```
A = np.array([ [2,5,-3,0], [-2,1,-7,4], [-1,8,-6,9] ])
```

2	5	-3	0
-2	1	-7	4
-1	8	-6	9

```
B = np.sort(A, axis = 0)
```

-2	1	-7	0
-1	5	-6	4
2	8	-3	9

Matrix product

- Use the np.dot function for calculating the matrix product

$$\text{np.dot(}\quad \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix} \quad , \quad \begin{bmatrix} 3 & 2 \\ 1 & 1 \end{bmatrix} \quad \text{)=} \quad \begin{bmatrix} 5 & 4 \\ 8 & 6 \end{bmatrix}$$

- Don’t confuse it with the standard multiplication

$$\begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix} \quad * \quad \begin{bmatrix} 3 & 2 \\ 1 & 1 \end{bmatrix} \quad = \quad \begin{bmatrix} 3 & 4 \\ 2 & 2 \end{bmatrix}$$

Reading/writing numpy arrays

- np.save(FILE.npy, X) → save X to FILE.npy
- X = np.load(FILE.npy) → load data (single array) from from FILE.npy

- np.savez(FILE.npy, X0, X1, …) → save multiple arrays to FILE.npy
- data = np.load(FILE.npy) → load data (multiple arrays) from FILE.npy. The single arrays are accessible as data['arr_0'], data['arr_1'], …

- np.savez(FILE.npy, X0 = X0, X1 = X1, …) → save multiple arrays to FILE.npy
- data = np.load(FILE.npy) → load data (multiple arrays) from FILE.npy. The single arrays are accessible as data['X0'], data['X1'], …

- np.savez_compressed → as savez but compressing data

- np.savetxt(FILE.txt, X) → save X to text file FILE.txt
- X = np.genfromtxt(FILE.txt) → read array from FILE.txt

Matplotlib

- Module for plotting, part of the scipy ecosystem
- Based on numpy arrays
- Many output formats are available

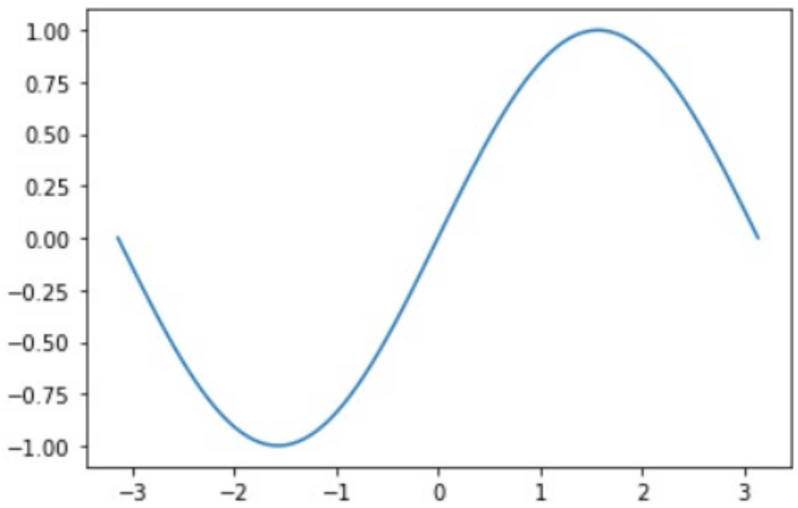
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

directive for jupyter notebook

```
%matplotlib inline
```

Usage in scripts

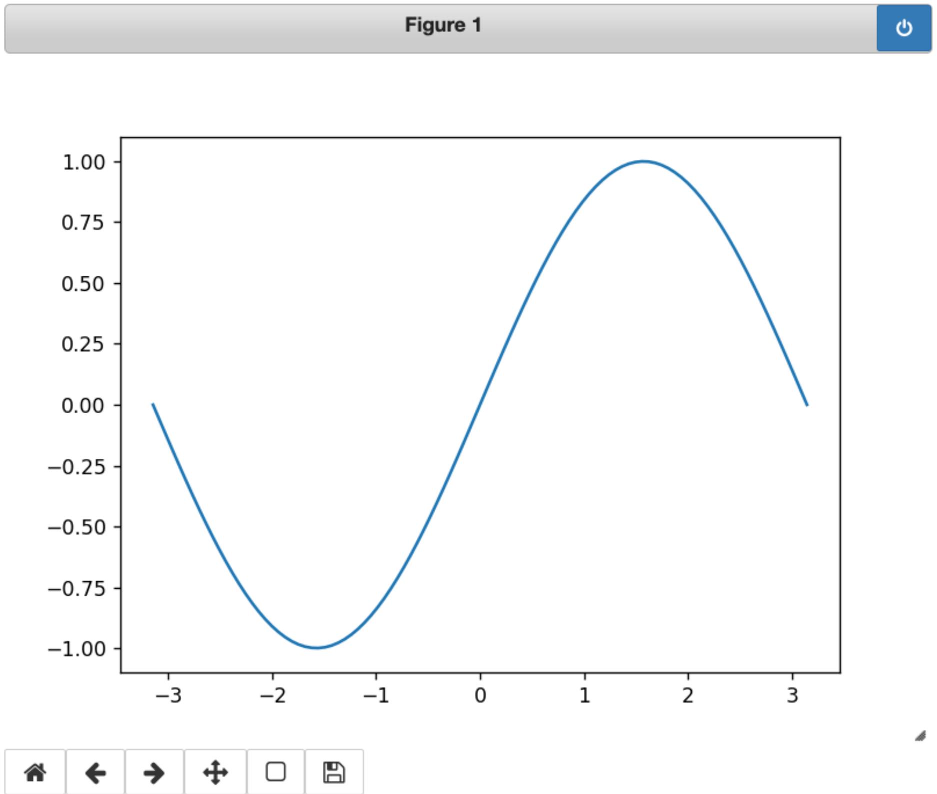
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-np.pi,np.pi,100)
y = np.sin(x)

f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,y)
plt.show()
```



- plt.show() is the command actually showing all the figure objects that were created
- the program waits until all the graphical windows are closed

Usage in notebooks

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

x = np.linspace(-np.pi,np.pi,100)
y = np.sin(x)

f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,y)
```



- plt.show is not needed
- If *%matplotlib inline* is used, the figures are created as static png images

Saving figures

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

x = np.linspace(-np.pi,np.pi,100)
y = np.sin(x)

f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,y)
f.savefig('figure.png')
```

To check which formats are available on the system

```
f.canvas.get_supported_filetypes()
```

```
{'ps': 'Postscript', 'eps': 'Encapsulated Postscript', 'pdf': 'Portable Document Format', 'pgf': 'PGF code for LaTeX', 'png': 'Portable Network Graphics', 'raw': 'Raw RGBA bitmap', 'rgba': 'Raw RGBA bitmap', 'svg': 'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics', 'jpg': 'Joint Photographic Experts Group', 'jpeg': 'Joint Photographic Experts Group', 'tif': 'Tagged Image File Format', 'tiff': 'Tagged Image File Format'}
```

Closing figures

- The figure objects take some memory… it’s a bad idea to have many figure objects open at the same time if those objects are not needed anymore

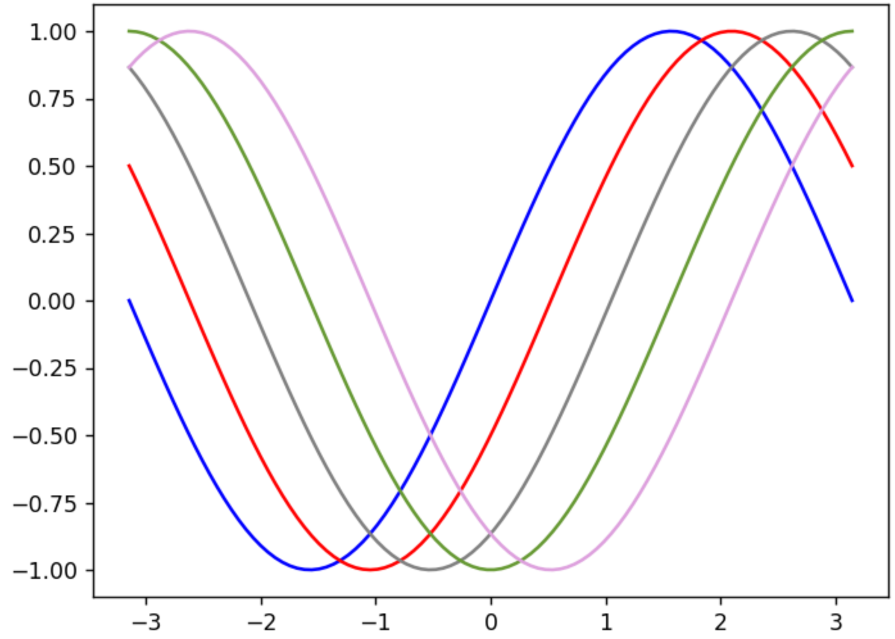
→ Once you’re done with a figure, close it with the close method

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

x = np.linspace(-np.pi,np.pi,100)
y = np.sin(x)

f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,y)
f.savefig(‘figure.png’)
plt.close()
```

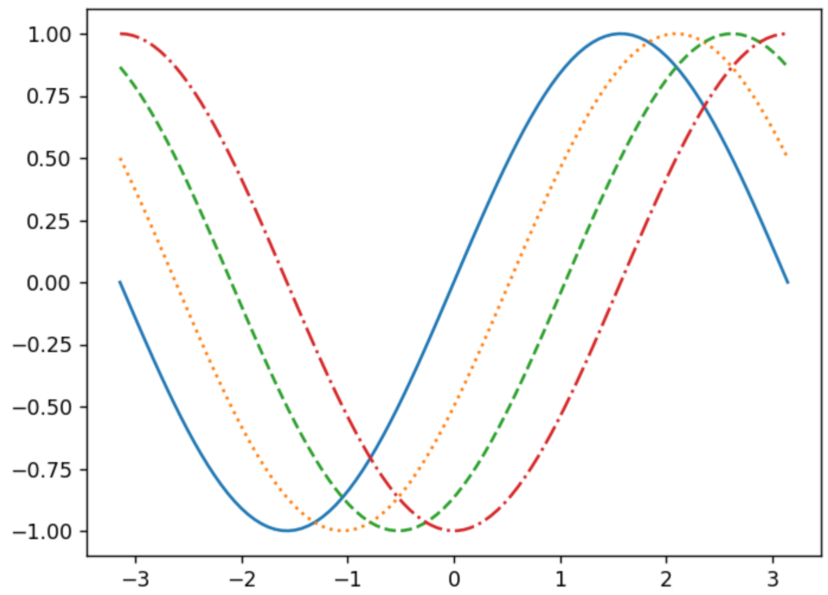
```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,100)
f = plt.figure()
ax = f.add_subplot(1,1,1)
# any color from the link below
ax.plot(x,np.sin(x), color = 'blue’)
# any of r,g,b,c,m,y,k
ax.plot(x,np.sin(x-np.pi/6), color = ‘r’)
# gray scale from 0 to 1, but using string !
ax.plot(x,np.sin(x-np.pi/3), color = '0.5’)
# RGB code
ax.plot(x,np.sin(x-np.pi/2), color = (0.4,0.6,0.2))
# hex RGB code
ax.plot(x,np.sin(x-4*np.pi/6), color = '#DDA0DD')
```

https://matplotlib.org/2.0.2/examples/color/named_colors.html

If no color is defined, matplotlib cycles automatically

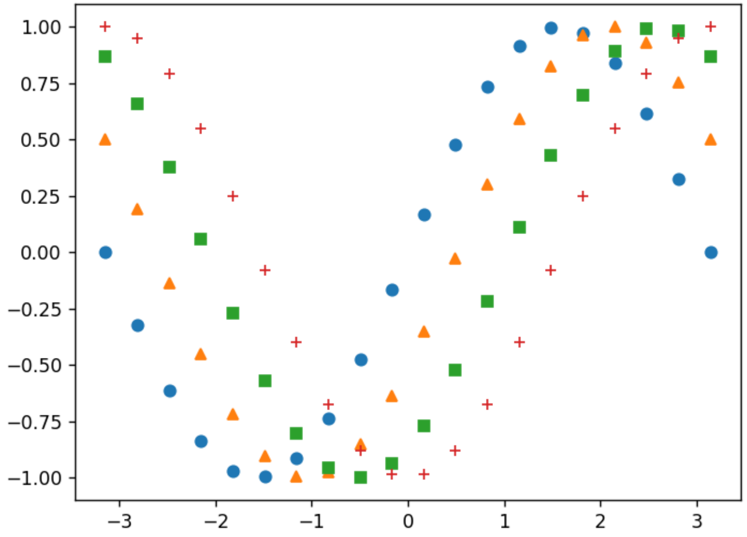


```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,100)
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,np.sin(x), linestyle = '-')
ax.plot(x,np.sin(x-np.pi/6), linestyle = ':')
ax.plot(x,np.sin(x-np.pi/3), linestyle = '--')
ax.plot(x,np.sin(x-np.pi/2), linestyle = '-.')
```

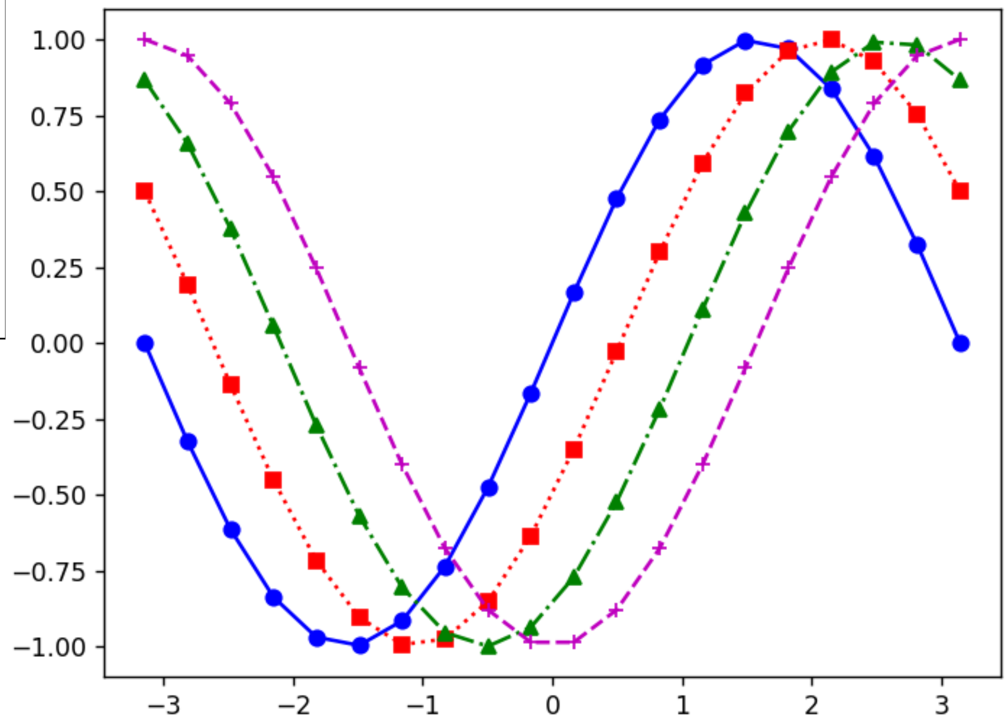


```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,20)
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,np.sin(x), marker = 'o', linestyle = 'None')
ax.plot(x,np.sin(x-np.pi/6), marker = '^', linestyle = 'None')
ax.plot(x,np.sin(x-np.pi/3), marker = 's', linestyle = 'None')
ax.plot(x,np.sin(x-np.pi/2), marker = '+', linestyle = 'None')
```

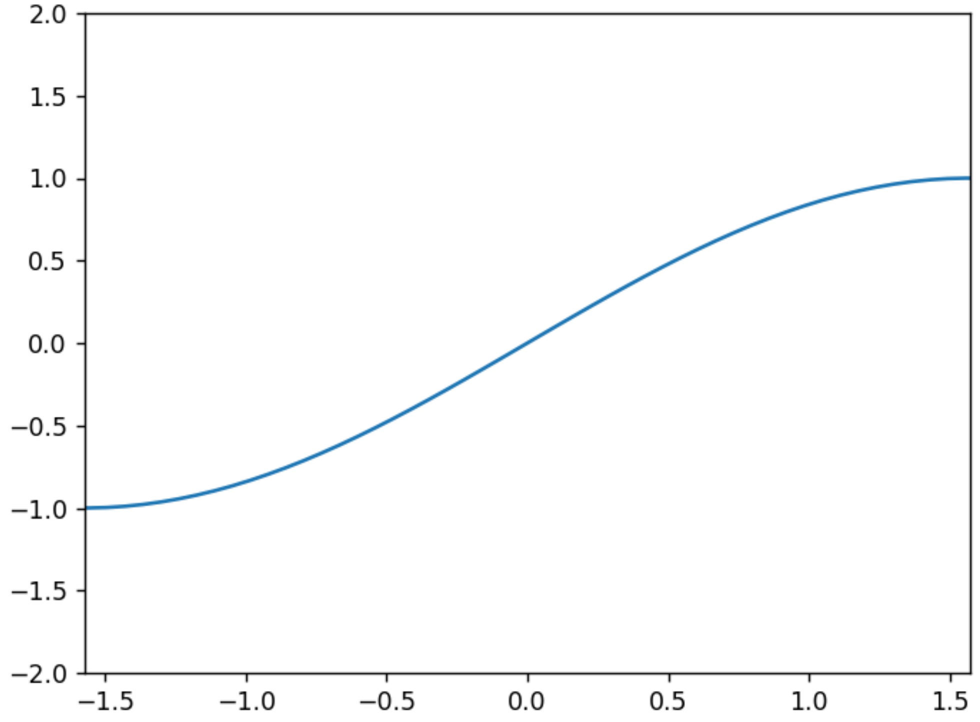
https://matplotlib.org/3.1.1/api/markers_api.html



```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,20)
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,np.sin(x), 'o-b')
ax.plot(x,np.sin(x-np.pi/6), 's:r')
ax.plot(x,np.sin(x-np.pi/3), '^-.g')
ax.plot(x,np.sin(x-np.pi/2), '+--m')
```



```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,100)
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,np.sin(x))
plt.xlim([-0.5*np.pi,0.5*np.pi])
plt.ylim([-2,2])
```



plt.xlim and plt.ylim (as other plt methods) work on the current figure, that by default is the last figure that was opened

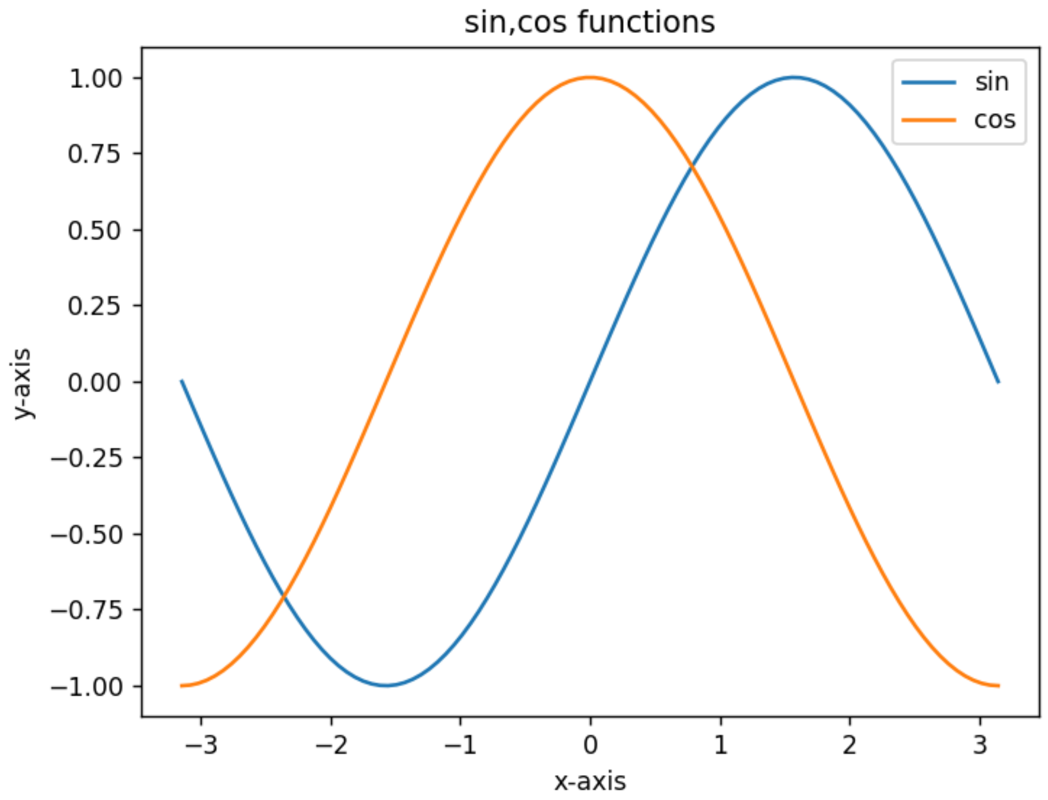
```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-np.pi,np.pi,100)

f1 = plt.figure()
ax = f1.add_subplot(1,1,1)
ax.plot(x,np.sin(x))

f2 = plt.figure()
ax = f2.add_subplot(1,1,1)
ax.plot(x,np.cos(x))

plt.figure(f1.number)
plt.xlim([-0.5*np.pi,0.5*np.pi])
plt.ylim([-2,2])
```

- f1.number is a unique identifier of the object f1
- plt.figure(f1.number) set the current figure to f1
- So, plt.xlim and plt.ylim are now working on f1

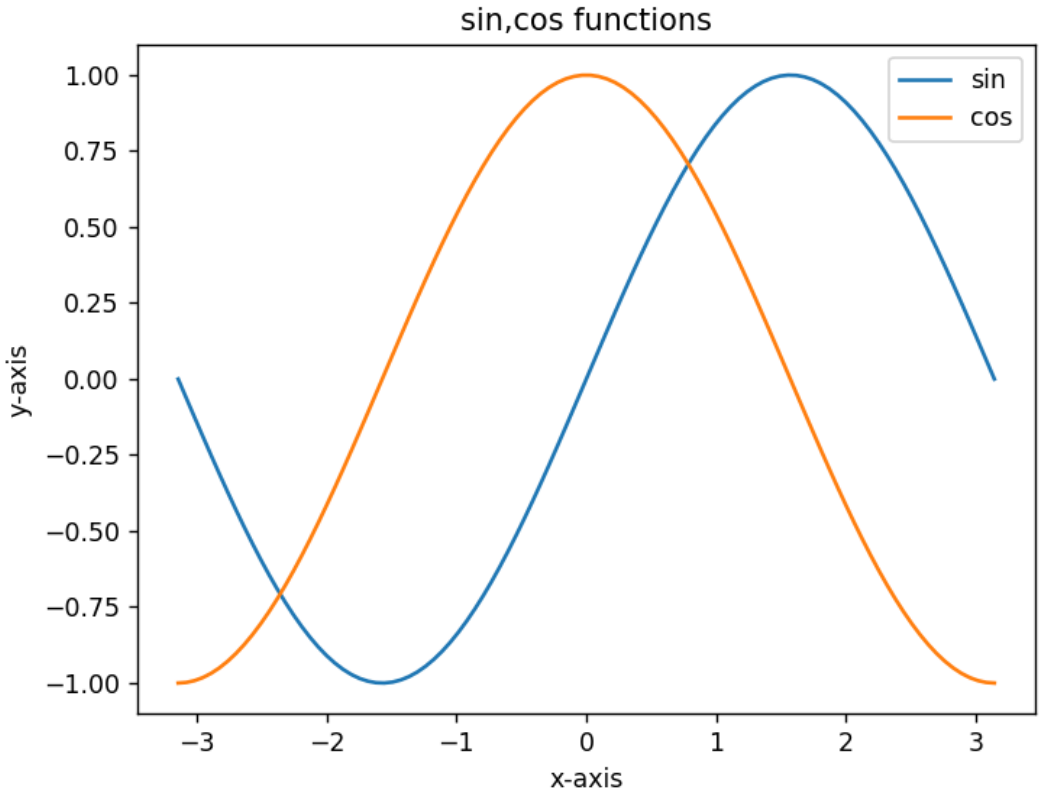
```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-np.pi,np.pi,100)

f1 = plt.figure()
ax1 = f1.add_subplot(1,1,1)
ax1.plot(x,np.sin(x))

f2 = plt.figure()
ax2 = f2.add_subplot(1,1,1)
ax2.plot(x,np.cos(x))

ax1.set_xlim(-0.5*np.pi,0.5*np.pi])
ax1.set_ylim([-2,2])
```

Same as previous slide, but now using methods of the axis object (so no need to change the current figure)

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,100)
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,np.sin(x),label = 'sin')
ax.plot(x,np.cos(x),label = 'cos')
plt.title('sin,cos functions')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend()
```

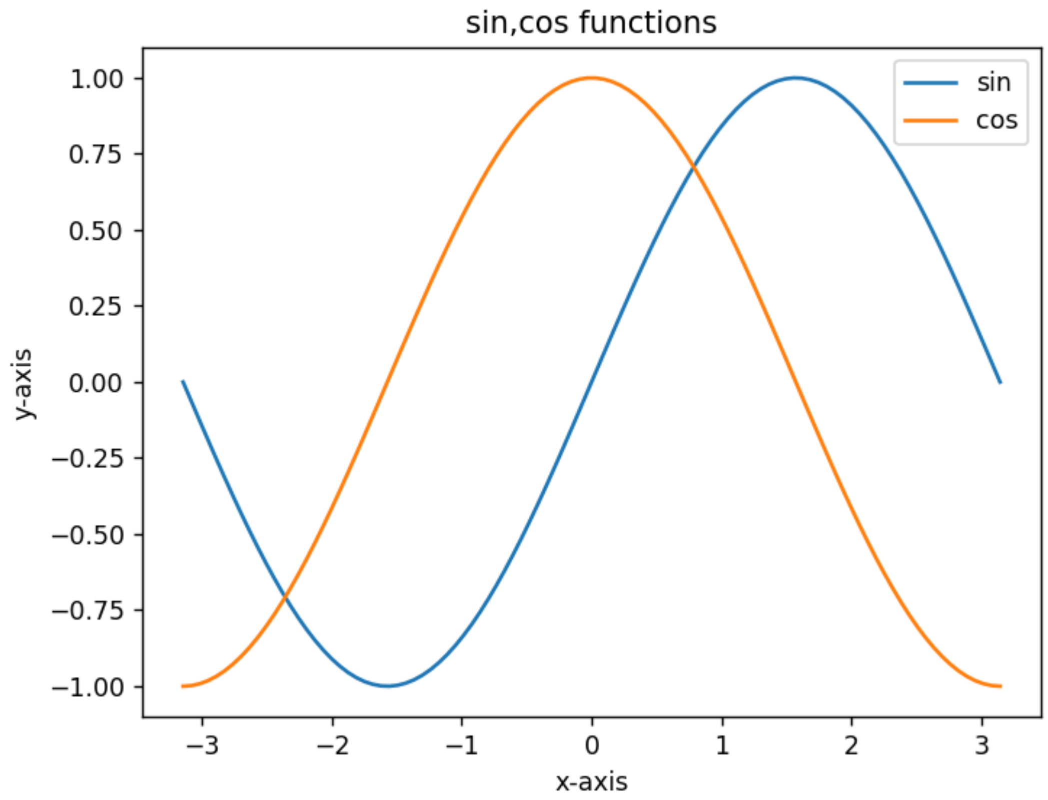


```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,100)
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,np.sin(x),label = 'sin')
ax.plot(x,np.cos(x),label = 'cos')
ax.set_title('sin,cos functions')
ax.set_xlabel('x-axis')
ax.set_ylabel('y-axis')
ax.legend()
```



Same as previous slide, but now we’re using methods of the axis object

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,100)
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.plot(x,np.sin(x),label = 'sin')
ax.plot(x,np.cos(x),label = 'cos')
ax.set(title = 'sin,cos functions',
      xlabel = 'x-axis',
      ylabel = 'y-axis')
ax.legend()
```

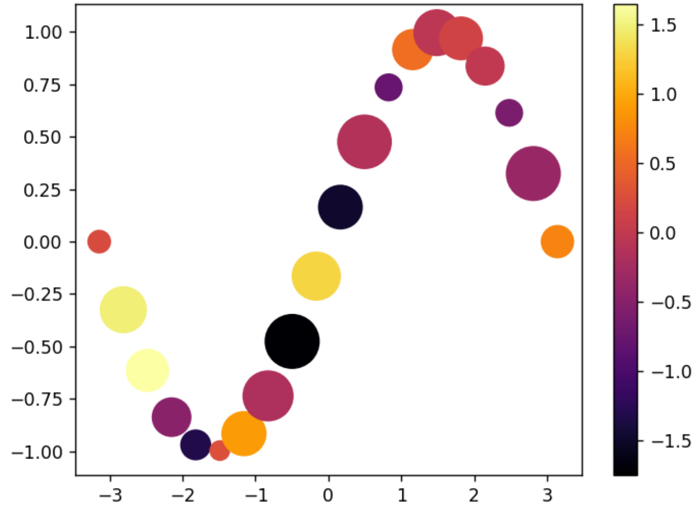


Still another way to do the same…

scatter = use it when you want to control the colors/sizes of individual points

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,20)
y = np.sin(x)
f = plt.figure()
ax = f.add_subplot(1,1,1)
sc_plot = ax.scatter(x,y, marker = 'o',
        c = np.random.normal(size = len(x)),
        s = np.random.uniform(10,1000,size = len(x)),
        cmap='inferno')
f.colorbar(sc_plot)
```

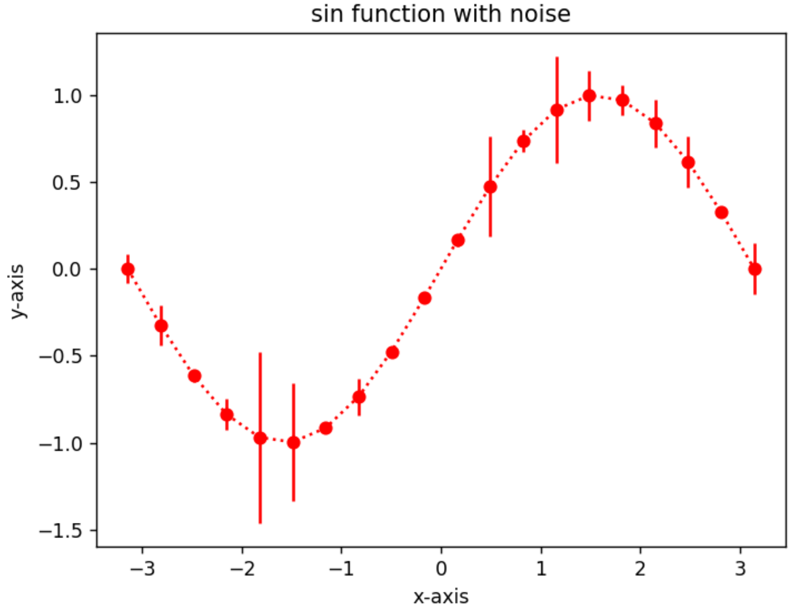
https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html



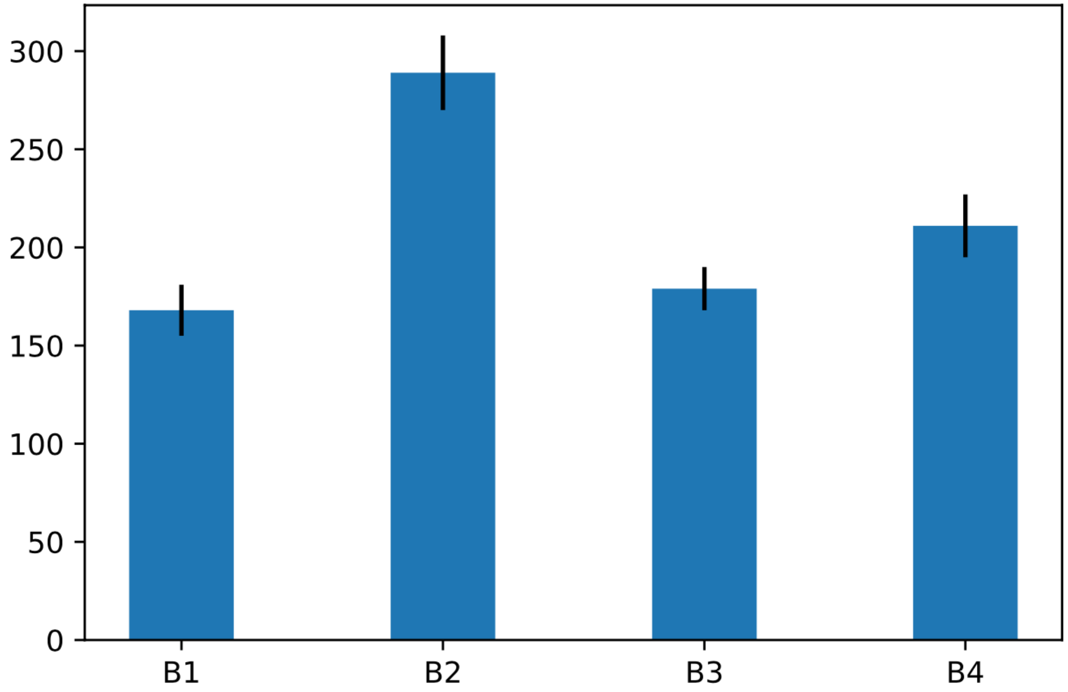
the computational cost is higher for scatter than from plot

errorbar = use it to show uncertain ranges

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-np.pi,np.pi,20)
y = np.sin(x)
y_std = np.random.normal(0,0.2, size = len(x))
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.errorbar(x, y, yerr = y_std,
        marker = 'o',
        color = 'r',
        linestyle = ':')
ax.set(title = 'sin function with noise',
     xlabel = 'x-axis',
     ylabel = 'y-axis')
```



bar plot




```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([0,1,2,3])
y = np.random.randint(100,300, 4)
ye = np.random.randint(10 ,30, 4)
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.bar(x, y, yerr = ye, width = 0.4)
ax.set_xticks([0,1,2,3])
ax.set_xticklabels([‘B1', ‘B2', ‘B3', ‘B4'])
plt.show()
plt.close()
```

2D plots – prepare the grid

```
x = np.array([0,1,2])
y = np.array([7,8])
X, Y = np.meshgrid(x,y)
```

X

0	1	2
0	1	2

Y

7	7	7
8	8	8

- np.meshgrid return two 2-dimensional arrays
- The number of rows is equal to the length of the second array
- The number of columns is equal to the length of the first array

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0,1,2])
y = np.array([7,8])
X,Y = np.meshgrid(x,y)
Z = X + Y
f = plt.figure()
ax = f.add_subplot(1,1,1)
im = ax.contourf(X,Y,Z,
            levels = 10,
            cmap = 'inferno'
            interpolation)
f.colorbar(im)
```

X

0	1	2
0	1	2

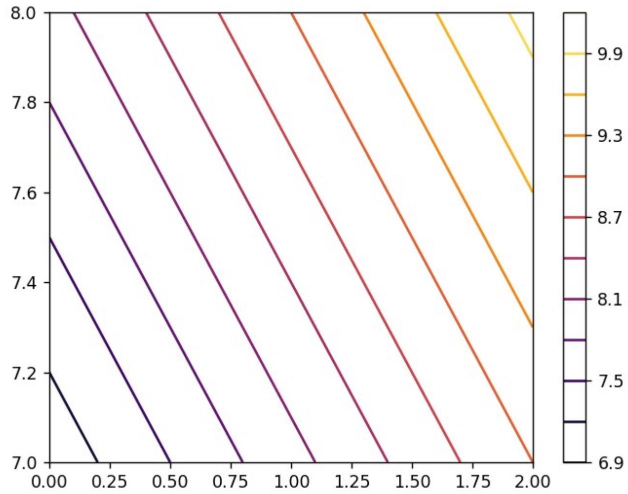
Y

7	7	7
8	8	8

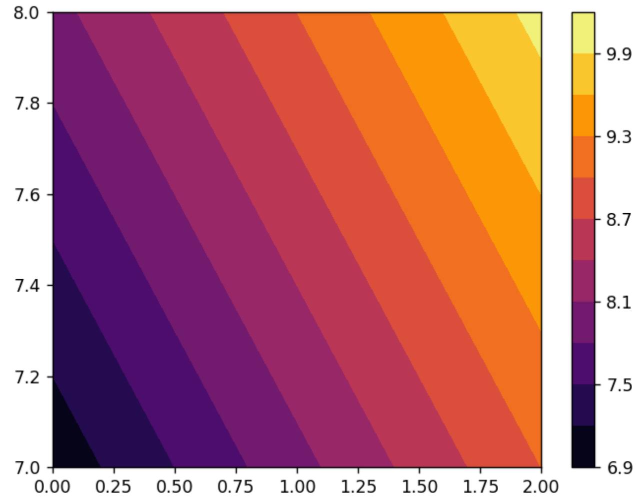
Z

7	8	9
8	9	10

ax.contour



ax.contourf



imshow: use it to see the actual values in a matrix

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0,1,2])
y = np.array([7,8])
X,Y = np.meshgrid(x,y)
Z = X + Y
f = plt.figure()
ax = f.add_subplot(1,1,1)
im = ax.imshow(Z, cmap = 'inferno')
f.colorbar(im)
```

Beware: The order of the elements is the one of the matrix (not of increasing X or Y)

X

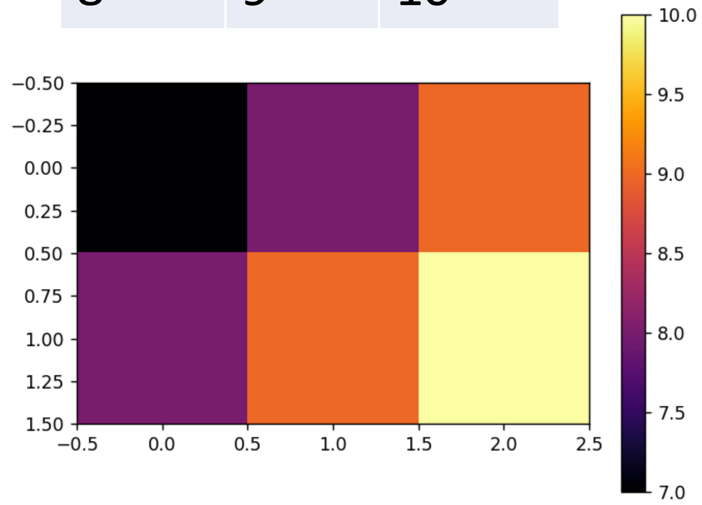
0	1	2
0	1	2

Y

7	7	7
8	8	8

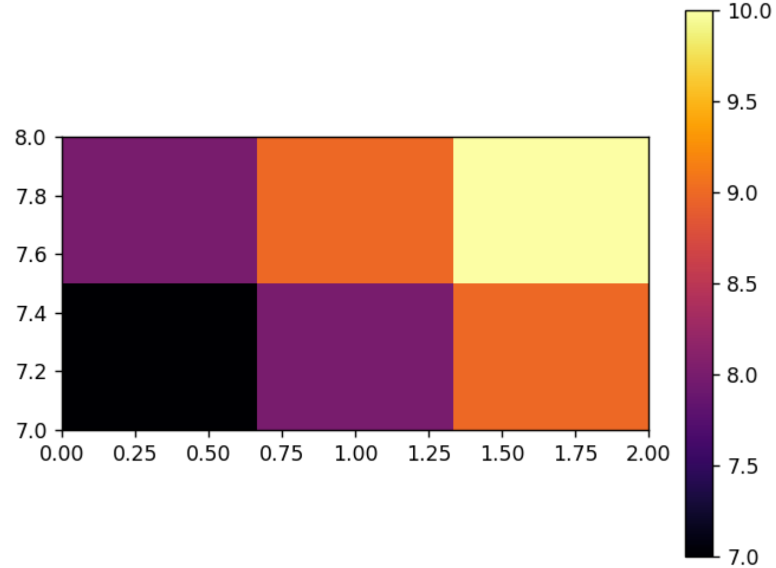
Z

7	8	9
8	9	10

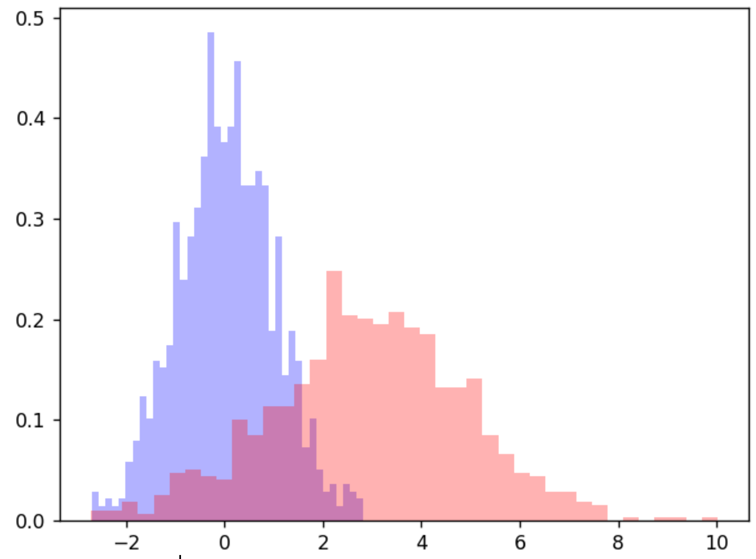


A simple way to fix the axes

```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0,1,2])
y = np.array([7,8])
X,Y = np.meshgrid(x,y)
Z = X + Y
f = plt.figure()
ax = f.add_subplot(1,1,1)
im = ax.imshow(Z,
            origin = 'lower',
            extent = (0,2,7,8),
            cmap = 'inferno')
f.colorbar(im)
```



```
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np
x1 = np.random.normal(0,1,size = 1000)
h1, e1 = np.histogram(x1, bins = 40, density = True)
b1 = 0.5*(e1[1:]+e1[:-1])
x2 = np.random.normal(3,2,size = 1000)
h2, e2 = np.histogram(x2, bins = 40, density = True)
b2 = 0.5*(e2[1:]+e2[:-1])
f = plt.figure()
ax = f.add_subplot(1,1,1)
ax.bar(b1, h1, width = (b1[1]-b1[0]), color = 'blue', alpha = 0.3)
ax.bar(b2, h2, width = (b2[1]-b2[0]), color = 'red', alpha = 0.3)
```



There's also a method plt.hist for directly calculating and showing the histogram (and also one for 2d histograms, plt.hist2d)

- A gallery of images done with matplolib (and corresponding codes) : https://matplotlib.org/2.0.2/gallery.html

- seaborn - https://seaborn.pydata.org/

“Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.”

- plotly - https://plot.ly/python/

“plotly.py enables Python users to create beautiful interactive web-based visualizations that can be displayed in Jupyter notebooks, saved to standalone HTML files […]”