2018

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 2-3: NumPy array & linear algebra (I)

Kai-Feng Chen
National Taiwan University

MANY NUMBERS IN A SINGLE LINE?

- With the notation of matrices, one can do a complex operation in a single line, isn't it?

Lots of linear equations Matrices-lize(?)

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + \ldots + A_{1N}x_N &= b_1 \\ A_{21}x_1 + A_{22}x_2 + \ldots + A_{2N}x_N &= b_2 \\ &\ldots \\ A_{N1}x_1 + A_{N2}x_2 + \ldots + A_{NN}x_N &= b_N \end{aligned}$$

$$\begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1N} \\ A_{21} & A_{22} & \ldots & A_{2N} \\ \vdots & \vdots & & \vdots \\ A_{N1} & A_{N2} & \ldots & A_{NN} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

Doing such a calculation could be still difficult with a pen and papers.

Single elegant line?

$$Ax = b$$

MATRICES OPERATIONS IN COMPUTERS

- One of the most important feature of numerical calculation is the repetition; you can just decide what to calculate, the underlying "real work" can be done by the computers.

HERE COMES THE NUMPY

Quote from http://www.numpy.org

- NumPy is the fundamental package for scientific computing with Python. It contains among other things:
  - a powerful N-dimensional array object
  - sophisticated (broadcasting) functions
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra, Fourier transform, and random number capabilities
- Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

In short: NumPy provides you a convenient **array object + tools**

NUMPY ARRAYS

- In python, the classical idea of arrays has been replaced by a more powerful type: **list**. But sometimes you still need to operate on arrays with a higher efficiency for scientific computing.
- NumPy is acting as an extension to Python for multi-dimensional arrays, for example:

```
>>> import numpy as np
>>> a = np.array([1,2,3,4])
>>> a
array([1, 2, 3, 4])
```

Such an array can be used to store any potential data from your experimental/theoretical work!

ARRAY CREATION

- Manual construction of 1-dimensional arrays is very simple. For example, you can create an array from a regular Python list or tuple using the array() function.

```
>>> a = np.array([1,2,3,4])
>>> a
array([1, 2, 3, 4])
>>> a.ndim
1 ⇐ ID
>>> a.shape
(4,)
>>> len(a)
4
>>> a.dtype      ⇐The type of the resulting array is deduced from the
dtype('int64')     type of the elements in the sequences.
```

ARRAY CREATION (II)

- Arrays with higher dimensions (e.g. 2D, 3D, etc.) can be converted from sequences of sequences, or sequences of sequences of sequences, and so on:

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.ndim
2   ⇐ 2D
>>> a.shape
(2,3)
>>> len(a)
2   ⇐ Return the size of the 1st dimension.
```

ARRAY CREATING FUNCTIONS

- In practice it is not very convenient to enter the numbers one-by-one; NumPy provides several functions to create arrays with some specific contents, e.g. **arange()** and **linspace()**:

```
>>> a = np.arange(10) ⇐ NumPy version of the range() function.
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.linspace(0.,1.,6) ⇐ starting point, ending point, # of points
>>> b
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> c = np.linspace(0.,1.,5,endpoint=False)
>>> c
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

Unlike the nominal python indexing, the ending point is included by default.

ARRAY CREATING FUNCTIONS (II)

- Sometimes you just want an array of zeros, ones, or identity:

```
>>> np.zeros((3,3)) ⇐ note the argument here is a tuple, (3,3)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.ones((3,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.eye(3) ⇐ "eye" = I = Identity
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

ARRAY CREATING FUNCTIONS (III)

- Or, creating an array with <u>a defined function</u> or even <u>randomly</u>:

```
>>> def f(i,j): return i+j
...
>>> np.fromfunction(f,(3,3))
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.],
       [ 2.,  3.,  4.]])
>>>
>>> np.random.rand(3,3) ⇐ note: it's not a tuple here.
array([[ 0.19836756,  0.53617863,  0.79492192],
       [ 0.6160475 ,  0.59142948,  0.89777024],
       [ 0.11665536,  0.10973303,  0.04245277]])
```

Random numbers between 0 and 1.

DATA TYPE

- You may have noticed that in some of the cases, the array elements could be either integer or float. For example:

```
>>> a = np.array([1,2,3])
>>> a.dtype
dtype('int64')
>>>
>>> b = np.array([1.,2.,3.])
>>> b.dtype
dtype('float64')
>>>
>>> b = np.zeros((2,2))
>>> b.dtype
dtype('float64') ⇐ Default type is 'float64'.
```

Now you see the difference between NumPy array and regular python list!
The NumPy array should have a uniform data type.

DATA TYPE (II)

- At the creation of array, you may explicitly specify which data-type you want to use, for example:

```
>>> a = np.array([1,2,3],dtype='float64')
>>> a.dtype
dtype('float64')
>>> a
array([ 1.,  2.,  3.])
>>> b = np.array([1,2,3],dtype='complex128')
>>> b.dtype
dtype('complex128')
>>> b
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Other data types are also available, for example: 'bool', 'int32', 'int64', etc.

BASIC OPERATIONS

- In NumPy, arithmetic operations on arrays are **"element-wise"**, ie. one element by one element:

```
>>> a = np.array([1,2,3])
>>> b = np.array([4,5,6])
>>> a**3
array([ 1,  8, 27])
>>> a-b
array([-3, -3, -3])
>>> c = np.array([0,np.pi*0.5,np.pi,np.pi*1.5,np.pi*2])
>>> d = np.sin(c) ⇐ NumPy has all the basic functions you need!
>>> d
array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16,
        -1.00000000e+00,  -2.44929360e-16])
>>> d>0.5
array([False,  True, False, False, False], dtype=bool)
```

BASIC OPERATIONS (II)

- The * operator is applying on the elements one-by-one as well:

```
>>> a = np.arange(4).reshape(2,2) ⇐ reshape() helps you to
>>> a                                rearrange the shape of array!
array([[0, 1],
       [2, 3]])
>>> b = np.array([[1,1],[2,2]])
>>> a*b
array([[0, 1],
       [4, 6]])
>>> a.dot(b)       ⇐ If you want to do matrix product, call dot() function
array([[2, 2],
       [8, 8]])
>>> b.dot(a)       ⇐ Non-commute: dot(a,b) ≠ dot(b,a)
array([[2, 4],
       [4, 8]])
```

INTERMISSION

- What will happen if you operate on two different data types of array? For example:

```
>>> a = np.ones(4,dtype='int64')
>>> a
array([1, 1, 1, 1])
>>> a+1.5
```

```
>>> a = np.ones(4,dtype='int64')
>>> a
array([1, 1, 1, 1])
>>> b = np.arange(4,dtype='complex128')
>>> b
array([ 0.+0.j,  1.+0.j,  2.+0.j,  3.+0.j])
>>> a+b
```

INDEXING AND SLICING

- The indexing of NumPy array is very close to standard python list:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0], a[1], a[-1]
(0, 1, 9)
>>> b = np.arange(9).reshape((3,3))
>>> b[2]
array([6, 7, 8])
>>> b[2,2] ⇐ In 2D, the first dimension corresponds to rows,
8            the second to columns.
>>> b[2,2] = 100
>>> b[2]
array([  6,   7, 100])
```

INDEXING AND SLICING (II)

- Just like nominal python list, it can also be sliced:

```
>>> a = np.arange(10)
>>> a[2:9]
array([2, 3, 4, 5, 6, 7, 8])
>>> a[2:9:2]
array([2, 4, 6, 8])
>>> a[5:] = 0  ⇐ combining slicing & assignment
>>> a
array([0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
>>> b = np.arange(5)
>>> a[5:] = b
>>> a
array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```

Full syntax:
array[start:end:step]

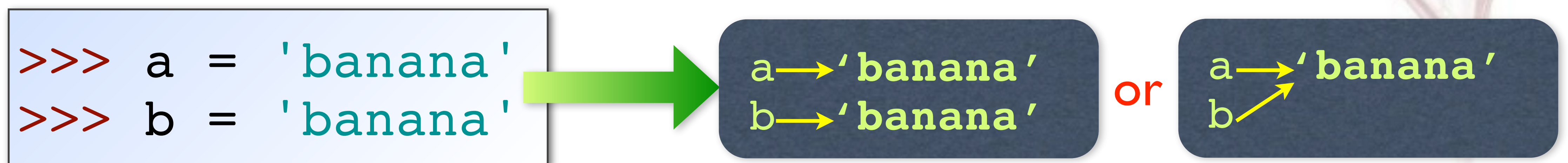
FANCY INDEXING

- NumPy arrays can be indexed with slices, but also with **boolean** or **integer** arrays (so-called "masks"). This method is called fancy indexing. For example:

```
>>> a = np.arange(10)
>>> a % 2 == 0
array([ True, False,  True, False,  True, False,
True, False,  True, False], dtype=bool)
>>> a[a % 2 == 0]
array([0, 2, 4, 6, 8])
>>> list(range(0,10,2))
[0, 2, 4, 6, 8]
>>> a[range(0,10,2)] = 99
>>> a
array([99,  1, 99,  3, 99,  5, 99,  7, 99,  9])
```

REVIEW: OBJECTS AND VALUES

- If we execute these assignments and following statements:



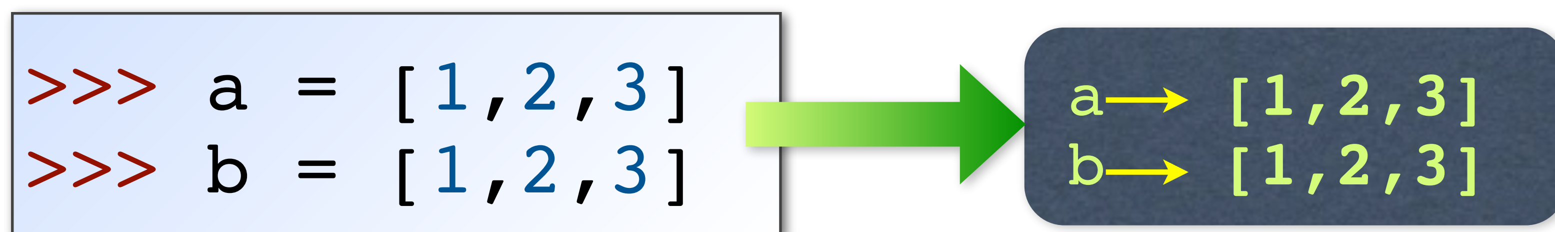
To check whether two variables (a,b) refer to the **SAME** object, one can use the **is** operator (while the regular == operator check the contents).

```
>>> a is b    ⇐ same object
True
>>> a == b    ⇐ same content
True
```

Python creates only one 'banana' string in this example.

REVIEW: OBJECTS AND VALUES

- But when you create two lists, you actually get two objects:



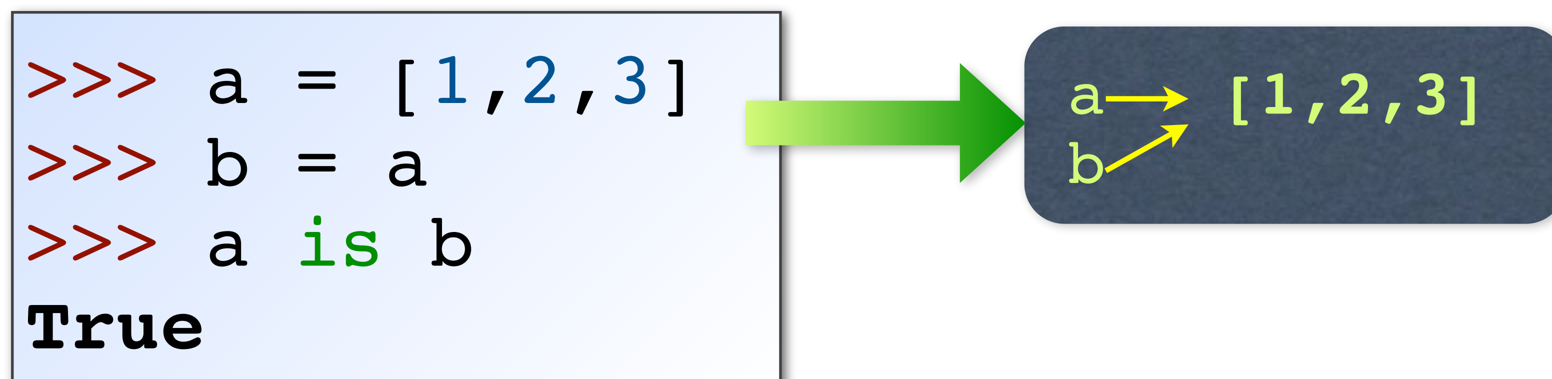
- In this case we would say that the two lists are equivalent, but not identical, because they are not the same object.
- **"a == b" does not imply "a is b":**

```
>>> a is b
False
>>> a == b
True
```

Python can create two separate lists with the same elements.

REVIEW: ALIASING

- If **a** refers to an object and you assign **b = a**, then both variables refer to the same object:



- The association of a variable with an object is called a **reference**.
- If the aliased object is mutable (such as list!), changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print(a)
[17, 2, 3]
```

Be careful about this when you are developing your code!

VIEW AND COPY

- Remember there are some differences between alias, shallow copy and deep copy. Similar issue with NumPy array:

```
>>> a = np.arange(10)
>>> b = a   ⇐ b is an alias of a
>>> b is a
True
>>> c = a[2:8]   ⇐ c is a view/shallow copy of a
>>> c is a
False
>>> np.may_share_memory(a,c)
True
>>> d = a.copy()   ⇐ d is a deep copy of a (call the copy() function)
>>> d is a
False
>>> np.may_share_memory(a,d)
False
```

`may_share_memory()` could tell if two arrays are sharing the same block of memory.

VIEW AND COPY (II)

- Remark: the behavior of slicing is different between nominal python list and NumPy array!

```
>>> a = np.arange(10)
>>> b = a[2:8]
>>> b[0] = 99
>>> a
array([ 0,  1, 99,  3,  4,  5,  6,  7,  8,  9])
```

NumPy array:
*slicing will create a **view***

```
>>> x = [0,1,2,3,4,5,6,7,8,9]
>>> y = x[2:8]
>>> y[0] = 99
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python list:
slicing will make a copy
(unless the elements are also sequence!)

VIEW AND COPY (III)

- However, the operation of fancy indexing will always create a copy rather than a view:

```
>>> a = np.arange(10)
>>> b = a[2:8]
>>> b[0] = 99
>>> a
array([ 0,  1, 99,  3,  4,  5,  6,  7,  8,  9])
```

Standard slicing will create a view

```
>>> a = np.arange(10)
>>> b = a[range(2,8)]
>>> b[0] = 99
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Fancy indexing will result a copy

REDUCTIONS

- There are several very useful reductions available with NumPy array. For example:

```
>>> a = np.arange(10)
>>> a.sum()
45
>>> a.min(), a.max()
(0, 9)
>>> a.mean()
4.5
>>> a.std()  ⇐ standard deviation = root-mean-square
2.8722813232690143
>>> r = np.random.randn(100000) ⇐ create an array filled with
>>> r.mean(), r.std()              Gaussian random numbers.
(0.0019795901122359044, 0.9988282230495326 7)
```

REDUCTIONS (II)

- Reduction with boolean arrays is also useful in many cases:

```
>>> a = np.array([True,False,False,False])
>>> a.any()
True
>>> a.all()
False
>>> b = np.array([1, 2, 3, 4])
>>> c = np.array([3, 2, 4, 5])
>>> b == c
array([ False, True, False, False], dtype=bool)
>>> np.any(b == c)
True
>>> np.all(b <= c)  ⇐ you can also do (b <= c).all()
True
```

INTERMISSION

- Remember the full syntax of slicing is **array[start:end:step]**. What will you get if you do this?

```
>>> a = np.arange(10)
>>> a[::-1]
```

- There is another special indexing operator "…" (very human readable!). See what do you get from a[…,0] and a[0,…]?

```
>>> a = np.arange(81).reshape(9,9)
>>> a[...,0]
>>> a[0,...]
```

BASIC VISUALIZATION

- We all know that a single plot can beat thousands of words!
- Before moving to next topic, let's discuss how to visualize your results (assuming your data is already stored in the format of NumPy array)!

Surely the ASCII art is also a way to display the results; but nowadays we can do much fancier.

BASIC VISUALIZATION WITH MATPLOTLIB

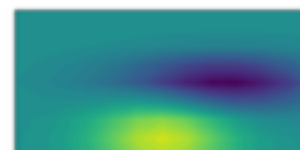
- Visualization of your results — here comes the **matplotlib**.
- The matplotlib is a python 2D plotting library, which can produce good quality figures in a variety of formats and interactive environments across platforms.

home | examples | tutorials | pyplot | docs »

http://matplotlib.org

Introduction

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits.

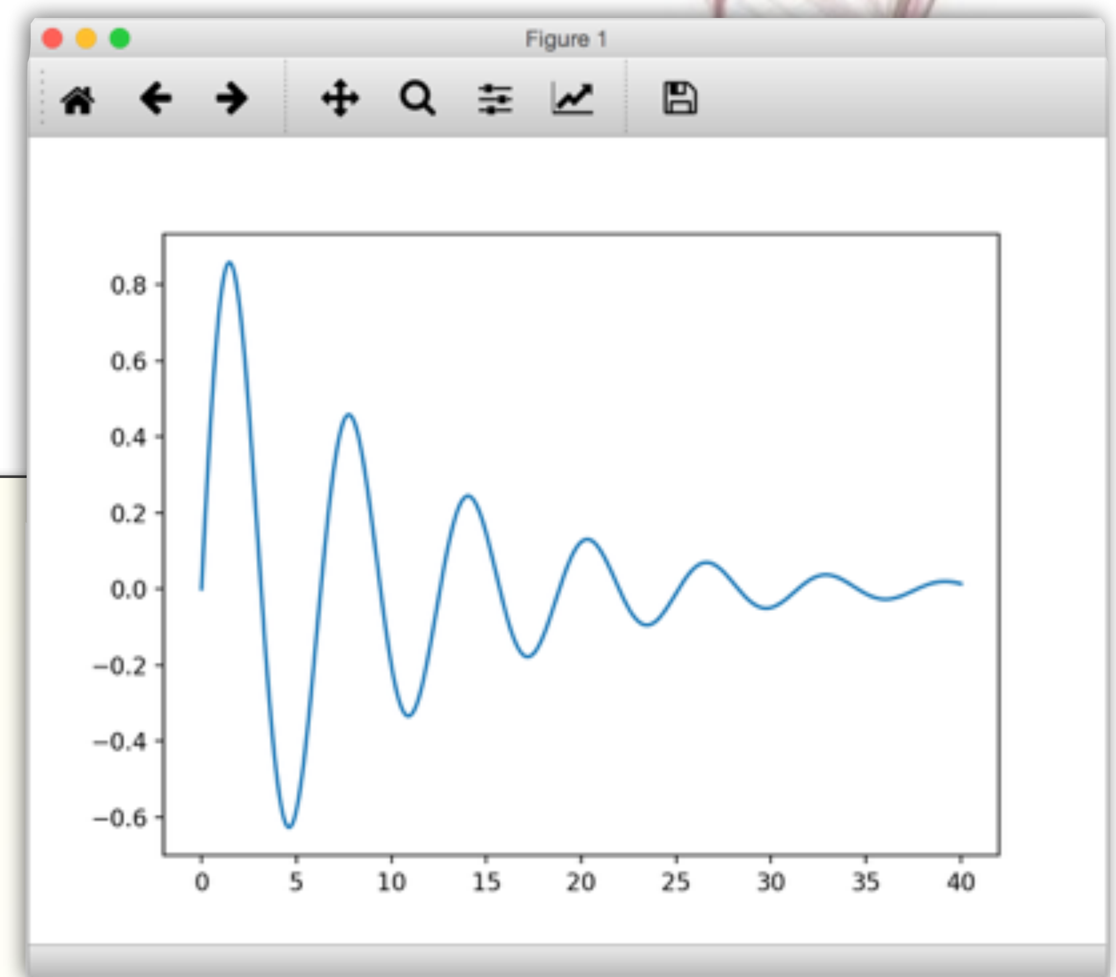


MAKE YOUR VERY FIRST PLOT

- Try to run the following code and see if you can get the same plot!
- Note your screen display does depend on your system!

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

x = np.linspace(0.,40.,500)
y = np.sin(x)*np.exp(-x*0.1)

plt.plot(x,y)
plt.show()
```

l203-example-01.py



A LITTLE BIT ON THE PLOTTING STYLE

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

plt.figure(figsize=(8, 8), dpi=80) ⇐ initializing a fig with 8x8 inches

x = np.linspace(0.,40.,1000)
y1 = np.sin(x)*np.exp(-x*0.10)
y2 = np.cos(x)*np.exp(-x*0.15)
y3 = np.sin(x*2.0)*np.sin(x*0.2)*np.random.rand(x.size)

plt.subplot(2, 1, 1) ⇐ initial a subplot, from grid of 1x2
plt.plot(x, y1, color = 'blue', linewidth = 2, linestyle = '-')
plt.plot(x, y2, color = 'red', linewidth = 2, linestyle = '--')
plt.ylim(-1.,+1.)

plt.subplot(2, 1, 2) ⇐ initial another subplot
plt.plot(x, y3, color = 'green', linewidth = 1)

plt.show()
```

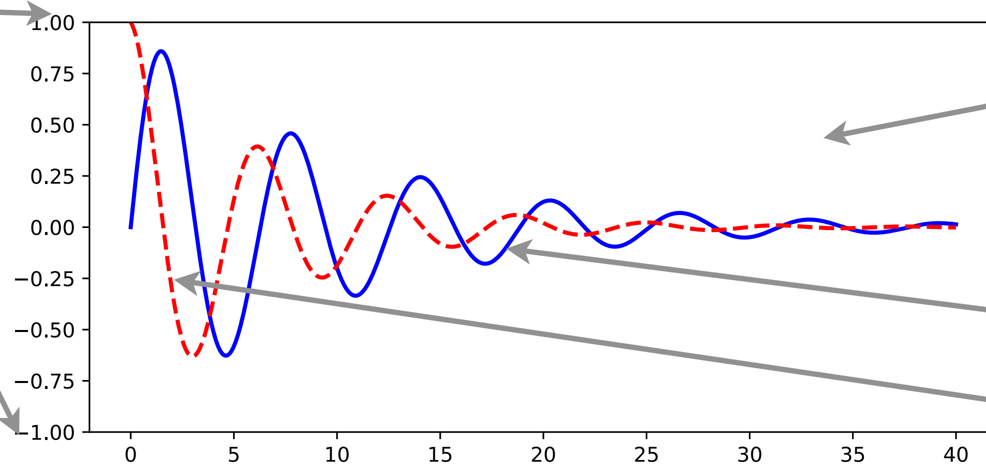
l203-example-02.py

A LITTLE BIT ON THE PLOTTING STYLE (II)

- This is what you should see as the result:

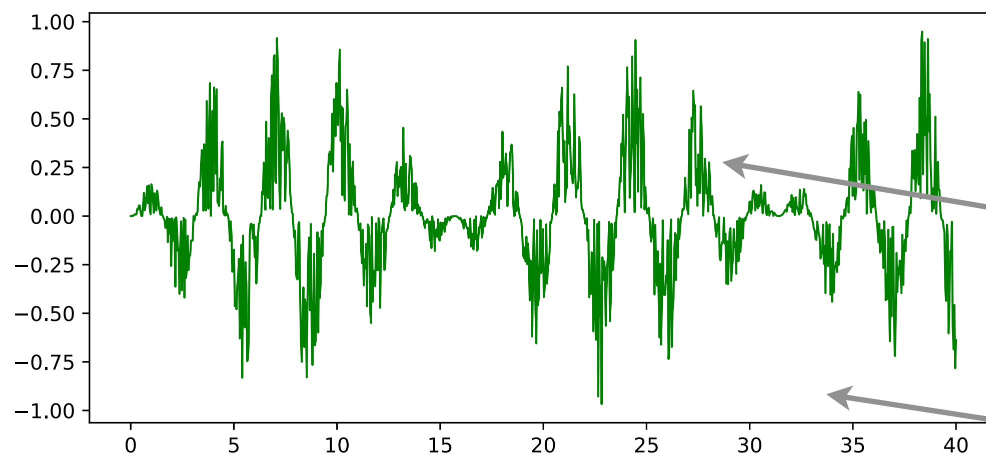
plt.ylim(-1.,+1.)

plt.subplot(2, 1, 1)



plt.plot(x, y1,...'-')

plt.plot(x, y2,...'--')



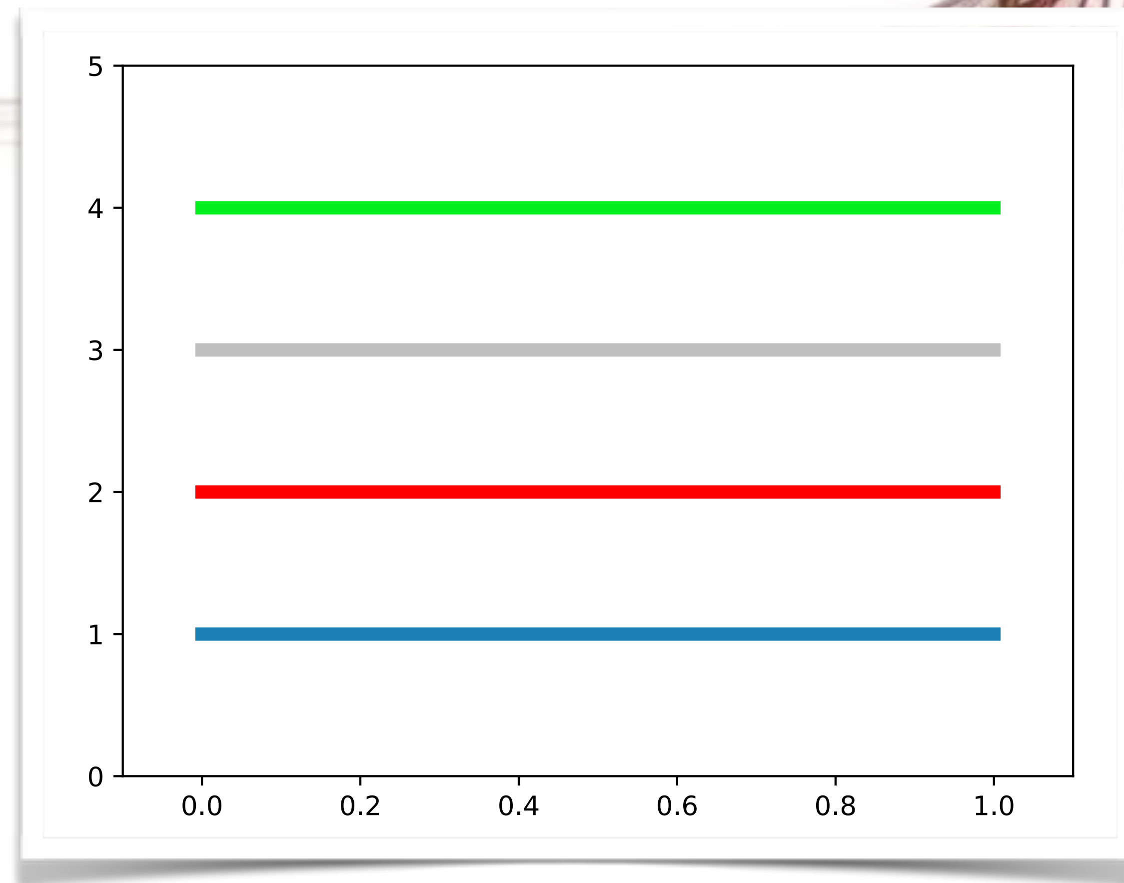
plt.plot(x, y3,...)

plt.subplot(2, 1, 2)

plt.figure(...)

COLORS

- There are several methods to allocate the **colors** in matplotlib. For example:




```
plt.plot([0,1], [1,1], color = (0.1,0.5,0.7), linewidth = 5)
                               ⇑ a tuple of (R,G,B) scale factors [0–1]
plt.plot([0,1], [2,2], color = 'red', linewidth = 5)
                               ⇑ a readable name following HTML standard
plt.plot([0,1], [3,3], color = '0.75', linewidth = 5)
                               ⇑ a gray scale factor [0–1]
plt.plot([0,1], [4,4], color = '#02ef20', linewidth = 5)
                               ⇑ hex representation
```

l203-example-03.py (partial)

LINE & MARKER STYLE

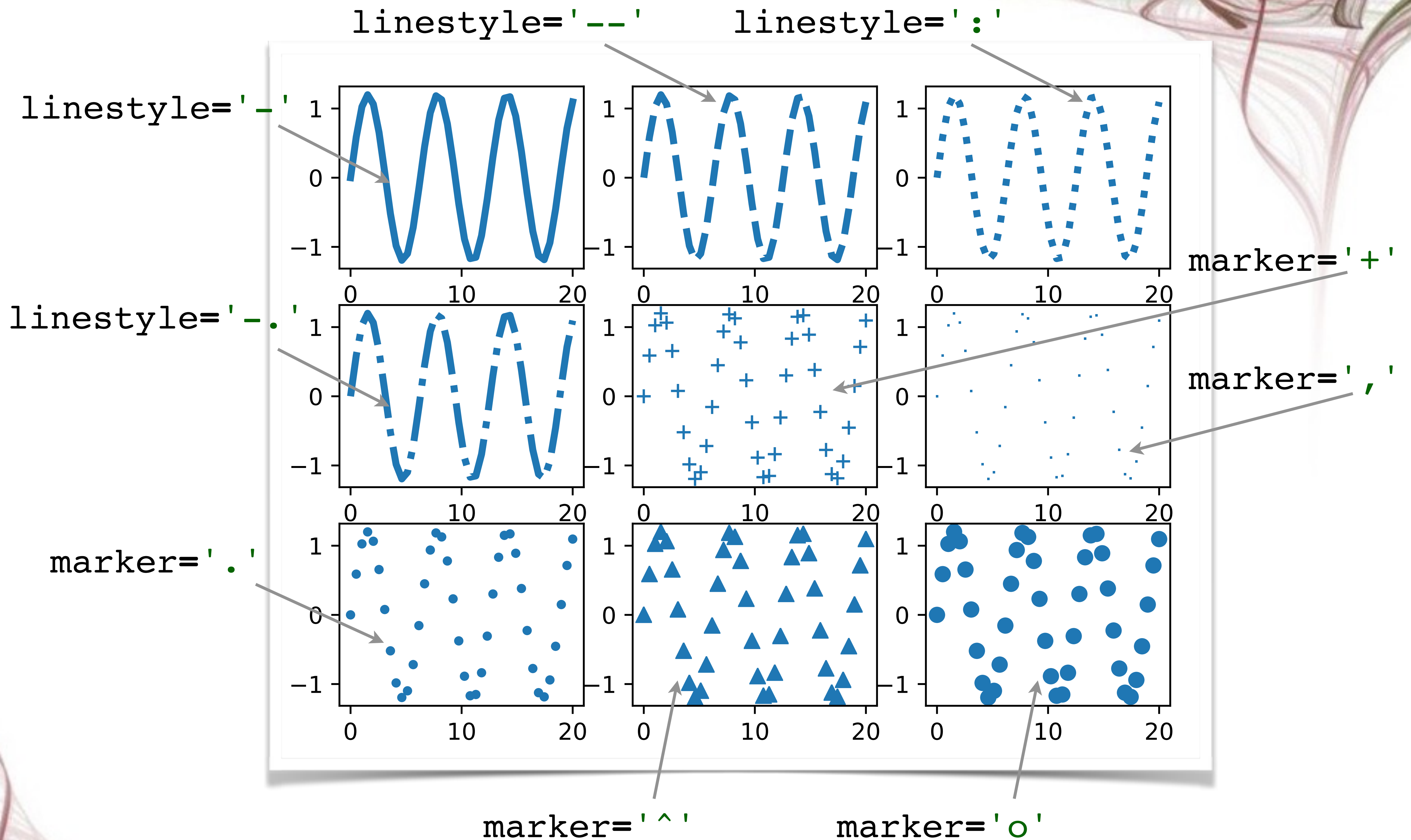
```
plt.subplot(3,3,1)
plt.plot(x, y, linewidth = 3, linestyle='-')
plt.subplot(3,3,2)
plt.plot(x, y, linewidth = 3, linestyle='--')
plt.subplot(3,3,3)
plt.plot(x, y, linewidth = 3, linestyle=':')
plt.subplot(3,3,4)
plt.plot(x, y, linewidth = 3, linestyle='-.')
plt.subplot(3,3,5)
plt.plot(x, y, linestyle='None', marker='+')
plt.subplot(3,3,6)
plt.plot(x, y, linestyle='None', marker=',')
plt.subplot(3,3,7)
plt.plot(x, y, linestyle='None', marker='.')
plt.subplot(3,3,8)
plt.plot(x, y, linestyle='None', marker='^')
plt.subplot(3,3,9)
plt.plot(x, y, linestyle='None', marker='o')
```

l203-example-04.py (partial)

Several typical line style and markers.

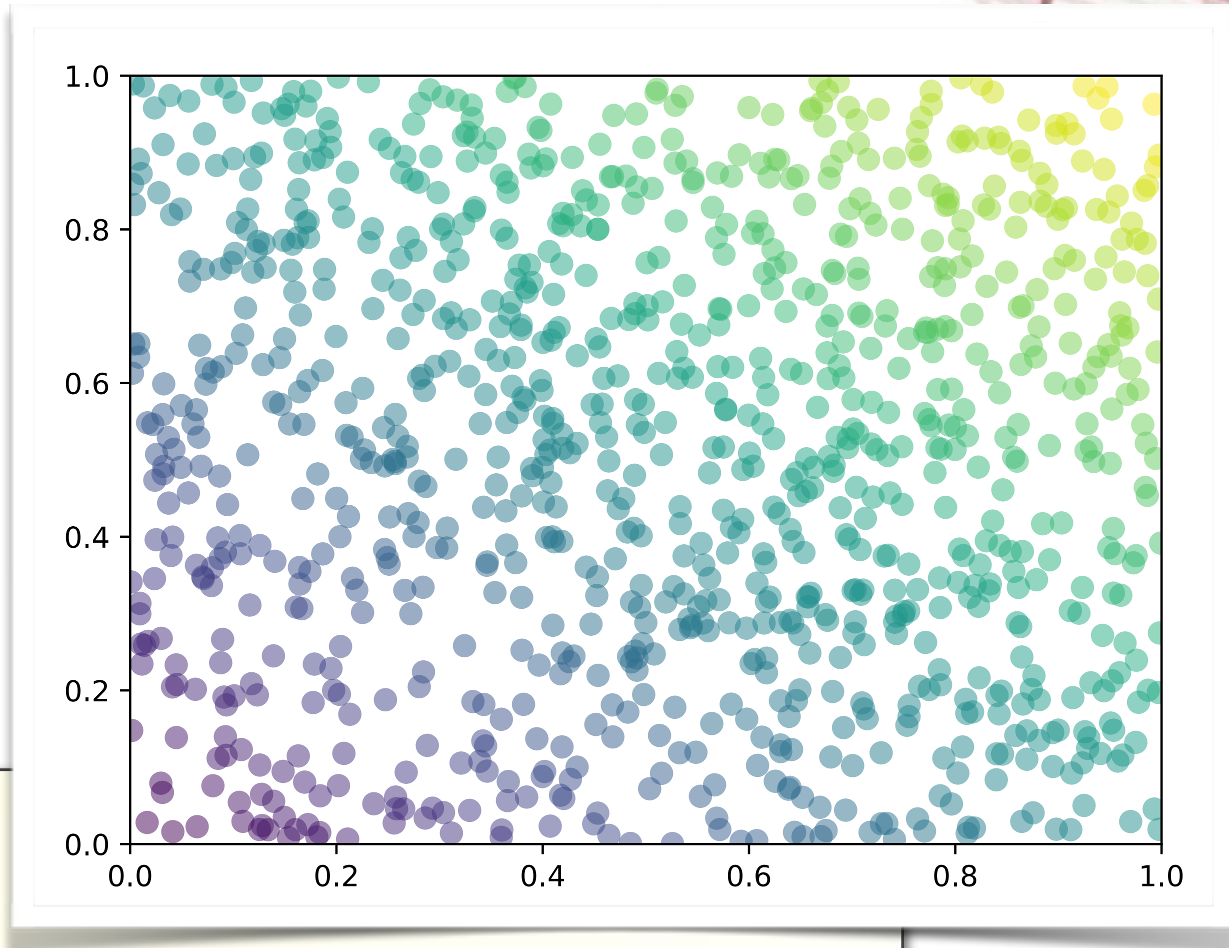
Check the matplotlib manual for more options!

LINE & MARKER STYLE (II)



MORE PLOTTING TYPES: SCATTER PLOT

- Instead of simple X-Y plots, there are few more options. For example, the **scatter plot**.




```
x = np.random.rand(1000)
y = np.random.rand(1000)

plt.scatter(x, y, c = (x+y)*0.5, s=50, alpha=0.5)

plt.xlim(0.,1.)
plt.ylim(0.,1.)
plt.show()
```

⇑ can be a fixed color, or a series of colors!

l203-example-05.py (partial)

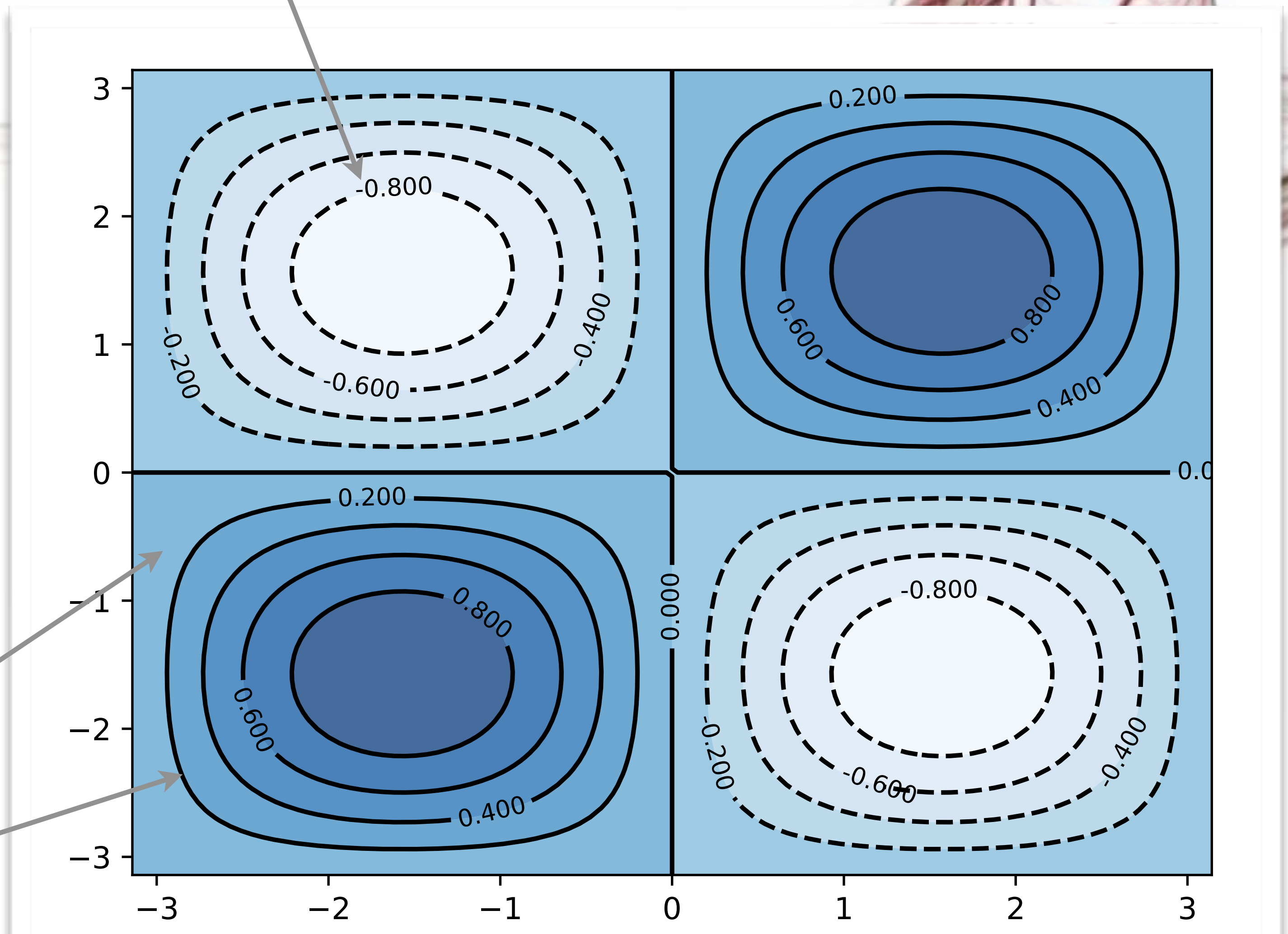
CONTOUR

- **Contour** plots are also very useful for many cases!

plt.clabel()

plt.contourf()

plt.contour()




```
x = np.linspace(-np.pi, np.pi, 100)
y = np.linspace(-np.pi, np.pi, 100)
xv, yv = np.meshgrid(x, y)
zv = np.sin(xv)*np.sin(yv)

plt.contourf(xv, yv, zv, 12, alpha=.75, cmap='Blues')
ctr = plt.contour(xv, yv, zv, 12, colors='black')
plt.clabel(ctr, fontsize=8)
```

⇐ make 2D arrays of y versus x

⇓ color map

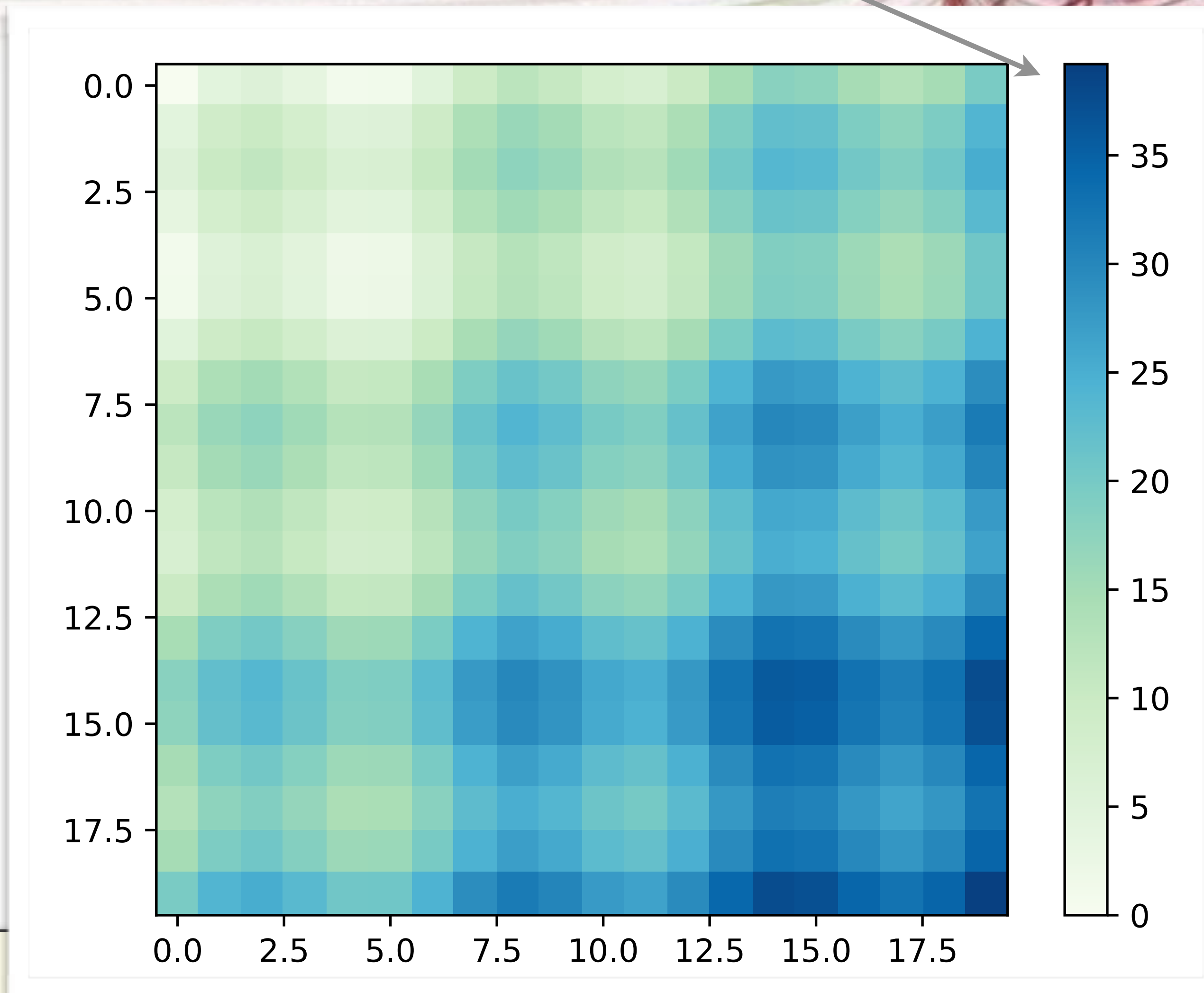
⇑ 12 levels

l203-example-06.py (partial)

SHOW AS AN IMAGE

- Sometimes you may just want to show the 2D array as an **image**.

plt.colorbar()

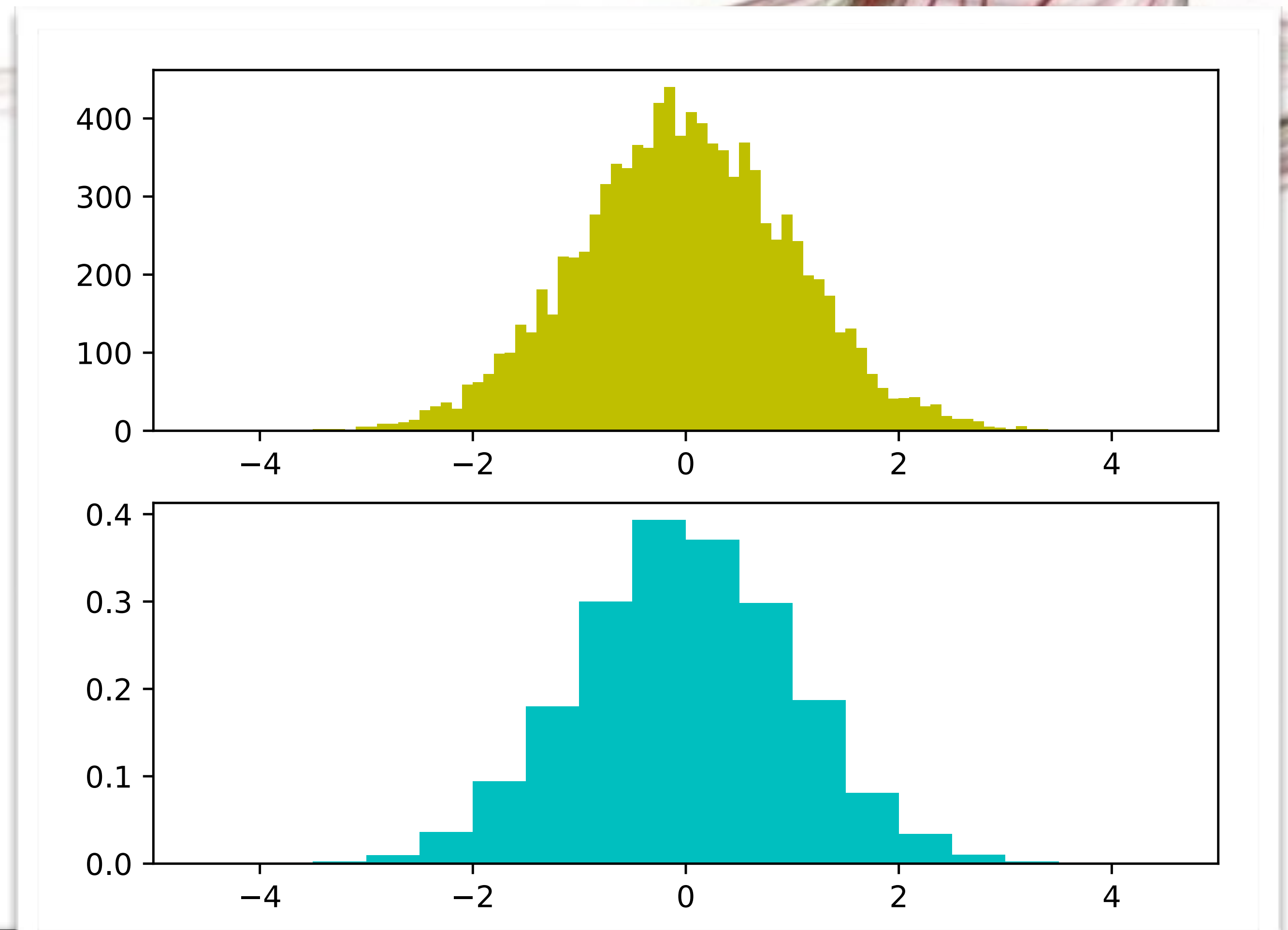



```
def f(i,j):
        return i+j+np.sin(i)*4.+np.sin(j)*4.

img = np.fromfunction(f,(20,20))

plt.imshow(img, interpolation='nearest', cmap='GnBu')
plt.colorbar()
```

l203-example-07.py (partial)

HISTOGRAM

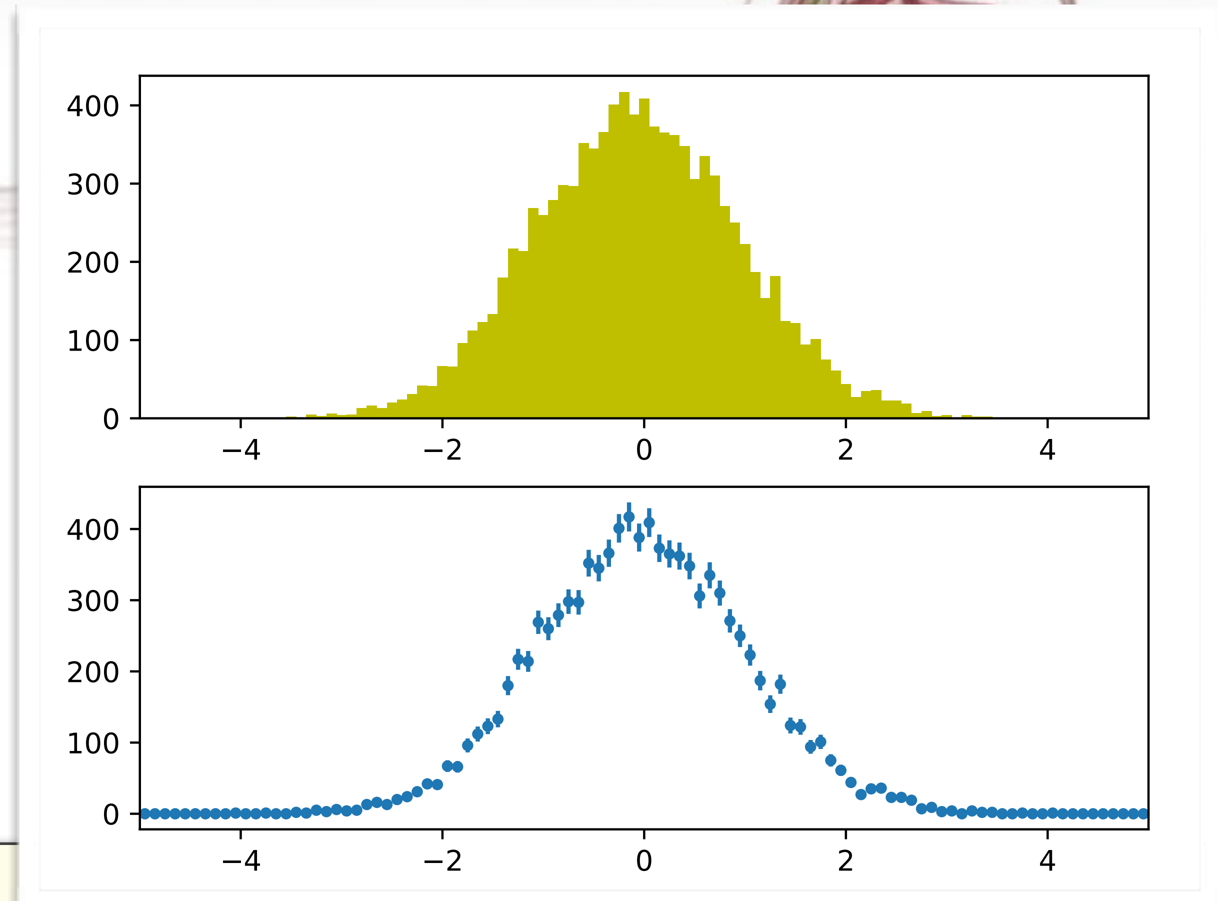
- **Histograms** are more commonly used in the statistical analysis!
- There is a straightforward command **`hist()`** in matplotlib that can draw a histogram directly.




```
data = np.random.randn(10000)  ⇐ generate random data with a Gaussian function

plt.subplot(2,1,1)
plt.hist(data, bins=100, range=(-5.,+5.), color='y')
plt.xlim(-5.,5.)          ⇑ Cut (-5.,+5.) with 100 slices,
                            accumulate the events for each bin
plt.subplot(2,1,2)
plt.hist(data, bins=20, range=(-5.,+5.), normed=True, color='c')
plt.xlim(-5.,5.)
```

l203-example-08.py (partial)

HISTOGRAM (II)



- Well, if you want to draw the error bar, you'll need some other way to do the same thing...

Make a histogram first ⇓

```
hist,bin_edges = np.histogram(data, bins=100, range=(-5.,+5.))
bin_centers = 0.5*(bin_edges[1:]+bin_edges[:-1])
bin_errors = np.sqrt(hist)
```

⇓ Draw a bar plot

```
plt.subplot(2,1,1)
plt.bar(bin_edges[:-1], hist, width=0.1, color='y')
plt.xlim(-5.,5.)
```

⇓ error-bar only

```
plt.subplot(2,1,2)
plt.errorbar(bin_centers, hist, yerr = bin_errors, fmt = '.')
plt.xlim(-5.,5.)
```

l203-example-08a.py (partial)

INTERMISSION

- Although we did not introduce this in the previous slides, adding a text to the matplotlib figure is also very simple. Please try the following little code and see what you get:

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

plt.text(0.5,0.6, 'Draw a LaTeX equation:',
fontsize = 30, ha='center', va='center')
plt.text(0.5,0.4, '$E = \sqrt{p^2 + m^2}$',
fontsize = 40, ha='center', va='center')

plt.show()
```

PUT ALL TOGETHER: BARNSLEY FERN

- The **Barnsley fern** is a fractal named after the British mathematician Michael Barnsley who first described it in his book **Fractals Everywhere**.
- It's a basic examples of self-similar sets: *the generated pattern that can be reproducible at any magnification or reduction*. The fern code is an example of an iterated function system (IFS) to create a fractal.
- "IFSs provide models for certain plants, leaves, and ferns, by virtue of the self-similarity which often occurs in branching structures in nature. But nature also exhibits randomness and variation from one level to the next; no two ferns are exactly alike, and the branching fronds become leaves at a smaller scale."
  — Michael Barnsley.

PUT ALL TOGETHER: BARNSLEY FERN (II)

PUT ALL TOGETHER: BARNSLEY FERN (III)

- Barnsley's fern uses four **affine transformations**:

$$f(x, y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

Randomly pick up one of the transformations according to the given probability, and iterate the position (x,y)

Prob	a	b	c	d	e	f
1%	0	0	0	0.16	0	0
85%	0.85	0.04	-0.04	0.85	0	1.60
7%	0.20	-0.26	0.23	0.22	0	1.60
7%	-0.15	0.28	0.26	0.24	0	0.44

PUT ALL TOGETHER: BARNSLEY FERN (IV)

- Naive code:

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

A = np.array([[[0.,0.],[0.,0.16]],
              [[0.85,0.04],[-0.04,0.85]],
              [[0.20,-0.26],[0.23,0.22]],
              [[-0.15,0.28],[0.26,0.24]]])

B = np.array([[0.,0.],
              [0.,1.60],
              [0.,1.60],
              [0.,0.44]])

ntrials = 40000
pos = np.zeros((ntrials,2))
for n in range(1,ntrials):
    type = np.random.choice(range(4),p=[0.01,0.85,0.07,0.07])
    pos[n] = A[type].dot(pos[n-1])+B[type]

plt.figure(figsize=(8, 12), dpi=80)
plt.scatter(pos[:,0],pos[:,1],c=pos[:,1]*0.5,alpha=0.5,s=10,marker='.')
plt.show()
```

$$f(x,y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

⇓ randomly choose one of the transformation

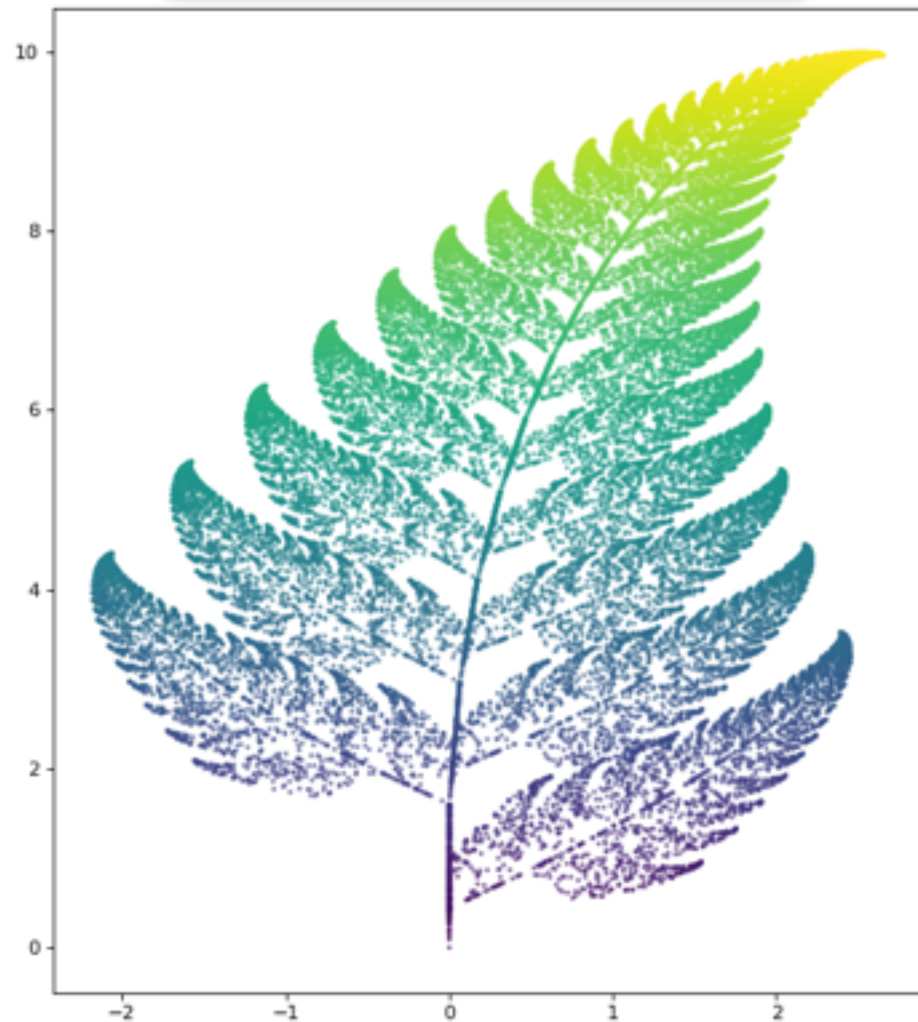
⇐ use dot() to calculate product

l203-example-09.py

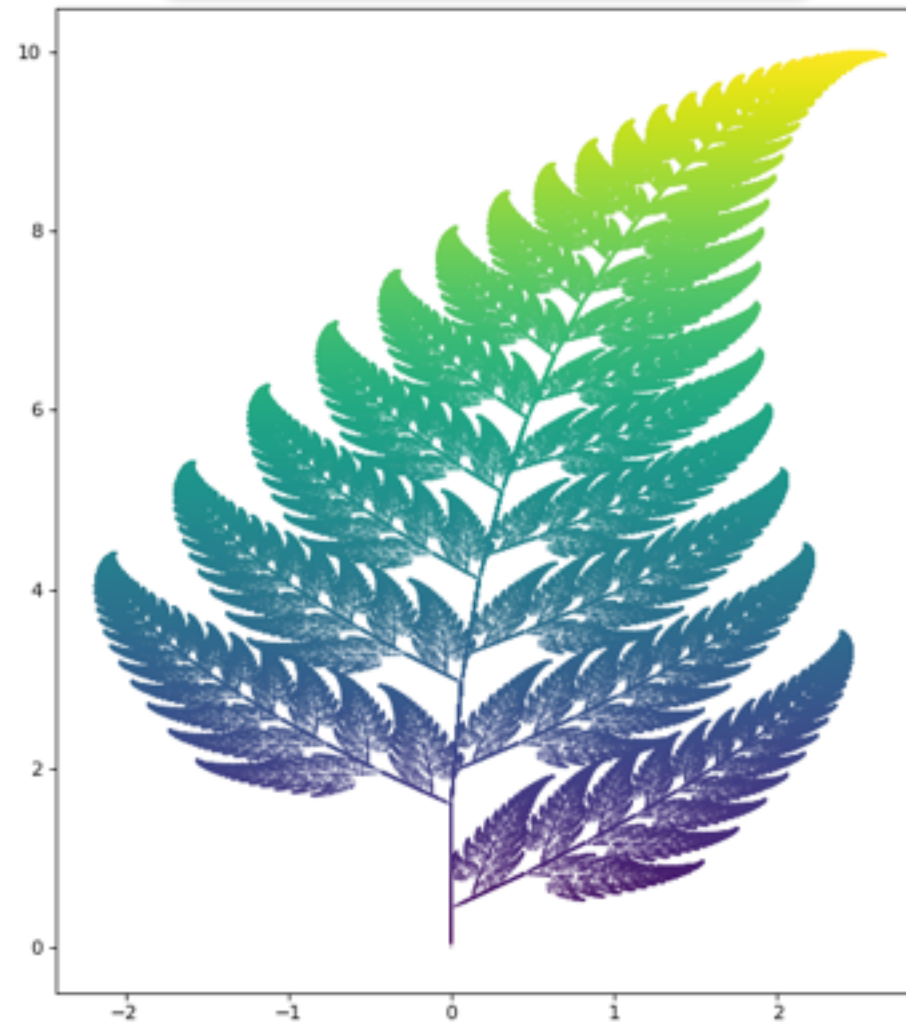
PUT ALL TOGETHER: BARNSLEY FERN (V)

- Output plots:

40K iterations



1M iterations



You may definitely play with much more different drawing options for fun!

- Here we quickly went though some of the typical plotting commands which will be used in the follow up lectures.
- There are far more options (plotting styles, etc) available for matplotlib. It might be useful if you can spend some time to go through some tutorials that are available on the web, e.g.:
  - https://matplotlib.org/users/pyplot_tutorial.html
  - http://www.loria.fr/~rougier/teaching/matplotlib/
- Surely you can find a lot of examples at the official matplotlib gallery:
  - https://matplotlib.org/gallery.html

HANDS-ON SESSION

- **Practice 01:**
  Create the following NumPy arrays:

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

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

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

HANDS-ON SESSION

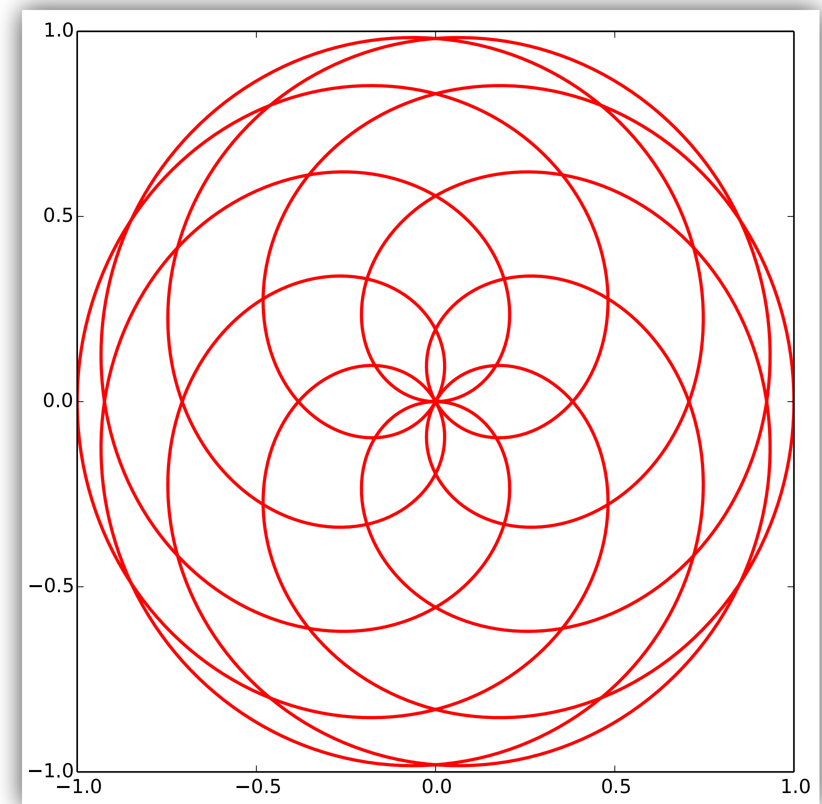
- **Practice 02:**
  Use matplotlib to draw the **Rose-rhodonea curve**:

$$x = \cos(kt)\cos(t)$$
$$y = \cos(kt)\sin(t)$$

where $k = \dfrac{n}{d}$, both n and d are integers.

Please produce the corresponding curve for the following combination of n and d: (4,1), (3,4), (3,7), (3,8),...etc.

for example, this is $(n,d) = (3,8)$



HANDS-ON SESSION

- **Practice 03:**
  Generate a different **Barnsley fern**, modify the `l203-example-09.py` and generate fern with the following table:

Prob	a	b	c	d	e	f
2%	0	0	0	0.25	0	-0.4
84%	0.95	0.005	-0.005	0.93	-0.002	0.5
7%	0.035	-0.2	0.16	0.04	-0.09	0.02
7%	-0.04	0.2	0.16	0.04	0.083	0.12