# Python Tutorial

**CV201:** Practical session

Oren Freifeld and Meitar Ronen
Computer Science, Ben-Gurion University


*These slides are based on a previous version by Ron Shapira-Weber.*

# Python: NumPy

## NumPy

Numpy is a core Python package which supports multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays.

- For more information, visit the quick start tutorial. If you are a veteran MATLAB user, Numpy for MATLAB user is also available and is highly recommended, even for non-matlab users.

# Python: NumPy

## np.allclose

np.allclose() compares two arrays **element wise**, and returns True if they are equal within a tolerance. $absolute(a - b) \leq (atol + rtol * absolute(b))$

```
1 a= np.arange(5)
2 b = a+1
3 np.allclose(a,b) # Returns False
4 c = a*1.001
5 np.allclose(a,c) # Returns False
6 d = a
7 d[0] += 0.001
8 np.allclose(a,d) # Returns True
```

# Python: NumPy

## Broadcasting

Arrays with different sizes cannot be added, subtracted, or generally be used in arithmetic.
One of the key advantages of NumPy is that it allows **broadcasting**.
Broadcasting is how NumPy solves the problem of arithmetic between arrays of differing shapes by in effect replicating the smaller array along the last mismatched dimension, so that they have compatible shapes.
It does this without making needless copies of data and usually leads to efficient algorithm implementations.

However, there are cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

# Python: NumPy

**Broadcasting**: scalar and array

```
1 a = np.arange(5) # [0,1,2,3,4]
2 b = 2
3 a*b # array([0, 2, 4, 6, 8])
```

# Python: NumPy

**Broadcasting**: arrays in different dimentions

```
a= np.eye(5) # 5\times 5 identity matrix
b = np.arange(5) # [0,1,2,3,4]
a+b
# array([[1., 1., 2., 3., 4.],
#        [0., 2., 2., 3., 4.],
#        [0., 1., 3., 3., 4.],
#        [0., 1., 2., 4., 4.],
#        [0., 1., 2., 3., 5.]])
```

# Python: NumPy

**Broadcasting**: rules of thumb

- Broadcasting can only be performed when dimensions are equal or one dimension has the size of 1.

```
    a= np.eye(5) # 5\times 5 identity matrix
    b = np.arange(5) # [0,1,2,3,4]
    a.shape # (5,5)
    b.shape # (5)
    # In effect, this becomes a comparison between:
    a.shape # (5,5)
    b.shape # (1,5)
    a+b  # broadcasting works
    # array([[1., 1., 2., 3., 4.],
    #        [0., 2., 2., 3., 4.],
    #        [0., 1., 3., 3., 4.],
    #        [0., 1., 2., 4., 4.],
    #        [0., 1., 2., 3., 5.]])

```

# Python: NumPy

**Broadcasting**: rules of thumb

- Broadcasting can only be performed when dimensions are equal or one dimension has the size of 1.

```
    # Will this work?
    A = array([[1, 2, 3], [1, 2, 3]])
    A.shape # (2,3)
    b = array([1, 2])
    b.shape # (1,2)
    C = A + b

```

# Python: NumPy

**Broadcasting**: rules of thumb

- Broadcasting can only be performed when dimensions are equal or one dimension has the size of 1.

```
    # Will this work?
    A = np.array([[1, 2, 3], [1, 2, 3]])
    A.shape # (2,3)
    b = np.array([1, 2])
    b.shape # (1,2)
    C = A + b

    # No....
    ValueError: operands could not be broadcast together
    with shapes (2,3) (2,)

```

# Python: NumPy

**Broadcasting**: rules of thumb

- Broadcasting can only be performed when dimensions are equal or one dimension has the size of 1.

```
    # Will this work?
    A = np.random.rand(1,2,3)
    # array([[[0.95756752, 0.49543907, 0.3638771 ],
    #         [0.60439745, 0.22096005, 0.37968279]]])
    A.shape # (1,2,3)
    b = np.random.rand(2,3)
    # array([[1.60595512e-01, 5.81226204e-01, 8.67794641e-01],
    #        [7.22824820e-04, 8.93400656e-01, 6.29571437e-01]])
    C = A+b

```

# Python: NumPy

**Broadcasting**: rules of thumb

- Broadcasting can only be performed when dimensions are equal or one dimension has the size of 1.

```
    # Will this work?
    A = np.random.rand(1,2,3)
    # array([[[0.95756752, 0.49543907, 0.3638771 ],
    #         [0.60439745, 0.22096005, 0.37968279]]])
    A.shape # (1,2,3)
    b = np.random.rand(2,3)
    # array([[1.60595512e-01, 5.81226204e-01, 8.67794641e-01],
    #        [7.22824820e-04, 8.93400656e-01, 6.29571437e-01]])
    C = A+b
    # Yes... The comparison is between (1,2,3) and (2,3) -> (1,2,3).

```

# Python: NumPy

**Broadcasting**: rules of thumb

- The arrays dimensions' size are equal or one dimension has the size of 1.
- if all input arrays do not have the same number of dimensions, a "1" will be repeatedly prepended to the shapes of the smaller arrays until all the arrays have the same number of dimensions.

# Python: NumPy

## **Broadcasting** examples

```
1     a = np.array([0.0, 10.0, 20.0, 30.0])
2     b = np.array([1.0, 2.0, 3.0])
3     a*b # Will not work...
4
```

# Python: NumPy

**Broadcasting** examples

outerproduct

```
    a = np.array([0.0, 10.0, 20.0, 30.0])
    b = np.array([1.0, 2.0, 3.0])
    a[:, np.newaxis]*b # np.newaxis adds a new axis (dim=1)
    #array([[ 1., 2., 3.],
    # [ 11., 12., 13.],
    # [ 21., 22., 23.],
    # [ 31., 32., 33.]])
    # equally, we can do:
    c = a.reshape(4,1) # 1 is the trailing axis
    c * b
    #array([[ 1., 2., 3.],
    # [ 11., 12., 13.],
    # [ 21., 22., 23.],
    # [ 31., 32., 33.]])

```

# Python: NumPy

**mgrid**
A very useful NumPy function, that returns an instance of a meshgrid which could be used for creating coordinate arrays over some function. Please also review the mgrid.ipynb jupyter notebook which is provided in

the tutorial files, especially the speed-up when comparing to for-loops.

# Python: NumPy

## mgrid

```
np.mgrid[0:5,0:5] # from 0 to 5 (exclusive) steps of 1
# array([[[0, 0, 0, 0, 0],
#         [1, 1, 1, 1, 1],
#         [2, 2, 2, 2, 2],
#         [3, 3, 3, 3, 3],
#         [4, 4, 4, 4, 4]],
#        [[0, 1, 2, 3, 4],
#         [0, 1, 2, 3, 4],
#         [0, 1, 2, 3, 4],
#         [0, 1, 2, 3, 4],
#         [0, 1, 2, 3, 4]]])
```

# Python: NumPy

## mgrid

```
y, x = np.mgrid[-2:3:1, -2:3:1] # from -2 to 3 (exclusive)
    steps of 1
print(y)
#array([[-2, -2, -2, -2, -2],
# [-1, -1, -1, -1, -1],
# [ 0, 0, 0, 0, 0],
# [ 1, 1, 1, 1, 1],
# [ 2, 2, 2, 2, 2]])
print(x)
#array([[-2, -1, 0, 1, 2],
# [-2, -1, 0, 1, 2],
# [-2, -1, 0, 1, 2],
# [-2, -1, 0, 1, 2]])
```

# Python: NumPy

## mgrid

Useful (instead of using for loops)

```
1 def f(x,y):
2 return np.cos(x)*np.sin(y)
3 fxy = f(x,y)
4 #array([[ 0.37840125, 0.35017549, -0. , -0.35017549,
      -0.37840125],
5 # [-0.4912955 , -0.45464871, 0. , 0.45464871, 0.4912955 ],
6 # [-0.90929743, -0.84147098, 0. , 0.84147098, 0.90929743],
7 # [-0.4912955 , -0.45464871, 0. , 0.45464871, 0.4912955 ],
8 # [ 0.37840125, 0.35017549, -0. , -0.35017549,
      -0.37840125]])
```

# Python: Random

**random.seed()**

Sometimes, we would like to test code that samples random numbers.
However, debugging is hard when the number are truely random...

# Python: Random

**random.seed()**
Sometimes, we would like to test code that samples random numbers. However, debugging is hard when the number are truely random...

Fortunately, numbers generated by Python's random module are not truly random, it is pseudo-random. The numbers are produced from some value which is called a seed value.
Generally, the seed value is the previous number generated by the generator. However, When the first time you use the random generator, there is no previous value. So by-default current system time is used as a seed value. Using the seed, we can generate over and over again the same "random" numbers.

# Python: Random

## random.seed()

```
random.randint(25,50) # 44
random.randint(25,50) # 28
# Each time a different number

random.seed(30)
random.randint(25,50) # 42
random.seed(30)
random.randint(25,50) # 42
```

This could also work with sequences of random numbers...

# Python: Matplotlib

## Matplotlib

# Python: Matplotlib

Matplotlib is a plotting library which enables several 2D/3D plotting capabilities. We will mostly focus on the matplotlib.pyplot which is a collection of command style functions that make matplotlib work like MATLAB.

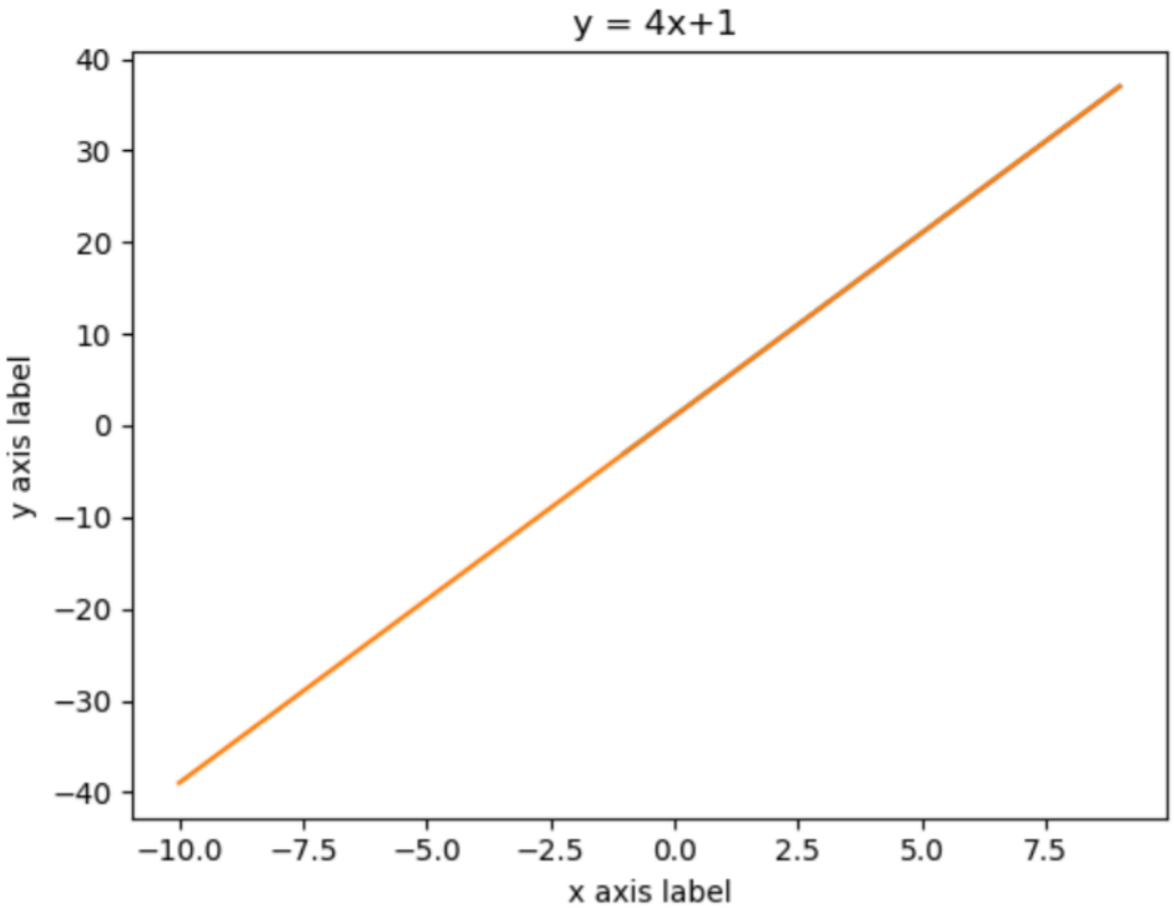
# Python: Matplotlib

## matplotlib

```
1  import numpy as np
2  import matplotlib.pyplot as plt # plt is the convention
3  x = np.arange(-10,10) # from -10 until 9, steps of 1
4  y = [4*a+1 for a in x] # y = 4x+1
5  y = np.array(y) # casting y to np.array, so we could use
     min()
6  #We'll set our axis to the min/max values of x,y:
7  plt.axis([x.min(), x.max(), y.min(), y.max()])
8  # [-10, 9, -39, 37]
9  # set axes names and title
10 plt.plot(x,y)
11 plt.xlabel('x axis label')
12 plt.ylabel('y axis label')
13 plt.title(' y = 4x+1')
14 plt.show()
```

# Python: Matplotlib

## Matplotlib



# Python: Matplotlib

## matplotlib

The general formatting of the 'plot' function is the following:

```
plot([x], y, [fmt], [x2], y2, [fmt2],...)
```

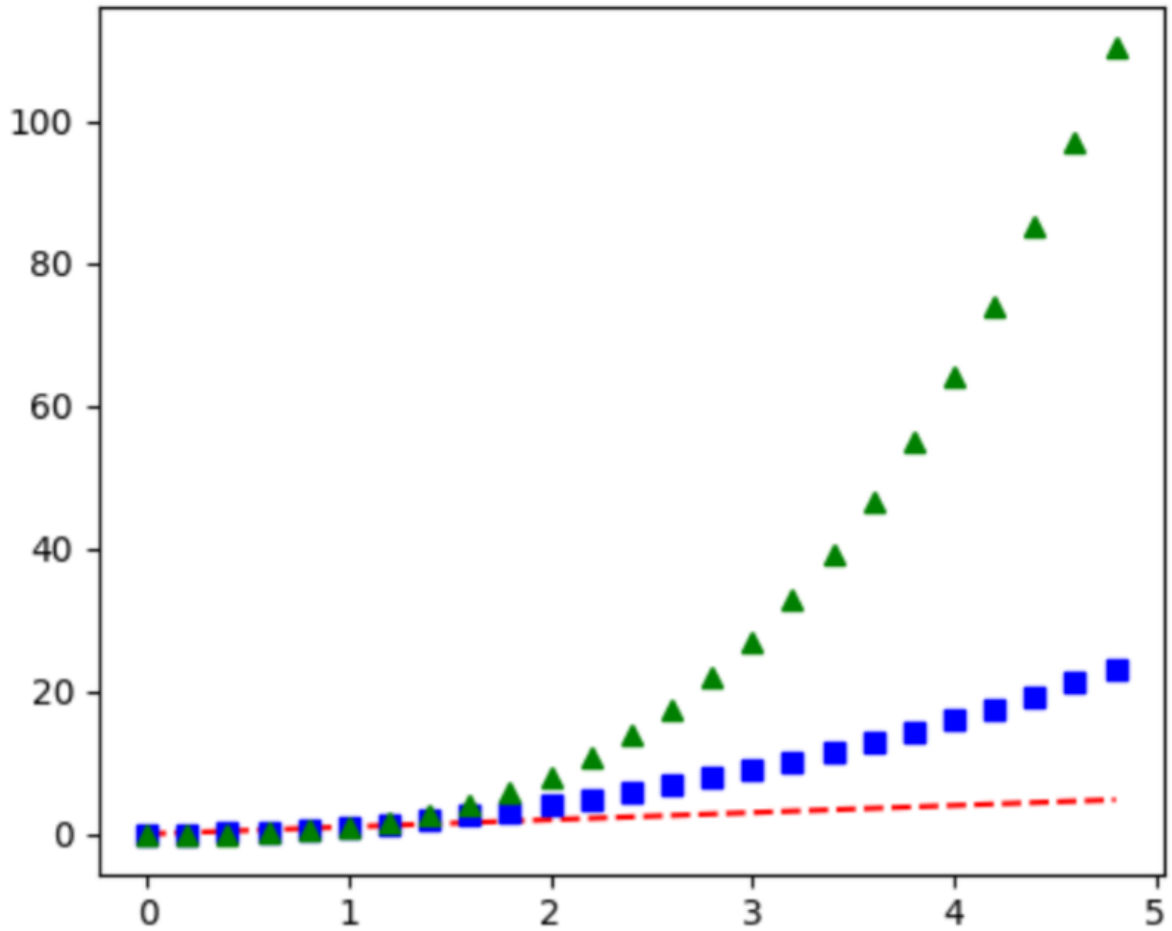
Where [fmt] stands for formatting:
fmt = '[color][marker][line]' For instance:

```
    # evenly sampled time at 200ms intervals
    t = np.arange(0., 5., 0.2)
    # red dashes: ('r--') r = red, -- = dashed marker
    # blue squares: ('bs') b = blue, s = square marker
    # green triangles ('g^') g = green, ^ = truangle
    plt.plot(t, t, 'r--', t, t**2,'bs', t, t**3,'g^')
    plt.show()

```

# Python: Matplotlib

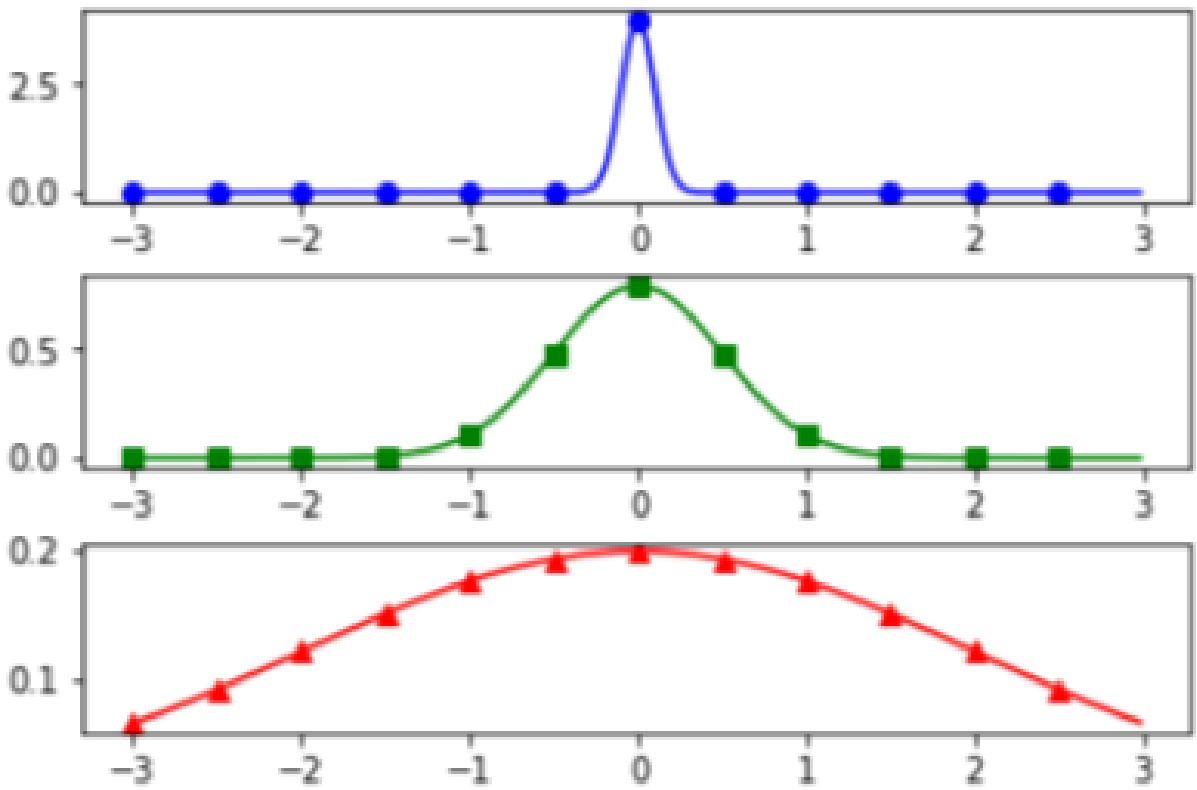
## Matplotlib



# Python: Matplotlib

## Subplots

The subplot command allows for plotting several figures in the same window. Let's say we want to explore how changing the variance affects the gaussian distribution:



# Python: Matplotlib

# Python: Matplotlib

## Subplots

As you might recall, the gaussian distribution function is defined by two parameters:

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

```
# define our gaussian function
def g(mu, sigma, x):
  return np.exp((x-mu)**2.) / (2.*(sigma)**2.)) /
         (np.sqrt(2.*np.pi*(sigma**2.))))
# Define different values of sigma
sigma1, sigma2, sigma3 = 0.1, 0.5, 2
mu = 0
```

# Python: Matplotlib

## Subplots

```
# Define X axis:
# we define two arrays of values for the following reasons
# 1) highlights the value of our function (every 0.5
  points)
# 2) plots our line
x1 = np.arange(-3.0, 3.0, 0.5)
x2 = np.arange(-3.0, 3.0, 0.02)
# First subplot
plt.figure(1) # declare the figure
plt.subplot(311) # 311 -> 3 rows, 1 columns, 1st subplot
plt.plot(x1, g(0, sigma1, x1), 'bo', x2, g(0, sigma1, x2),
   'b') # 'bo': blue circle markers, 'b': black line
```

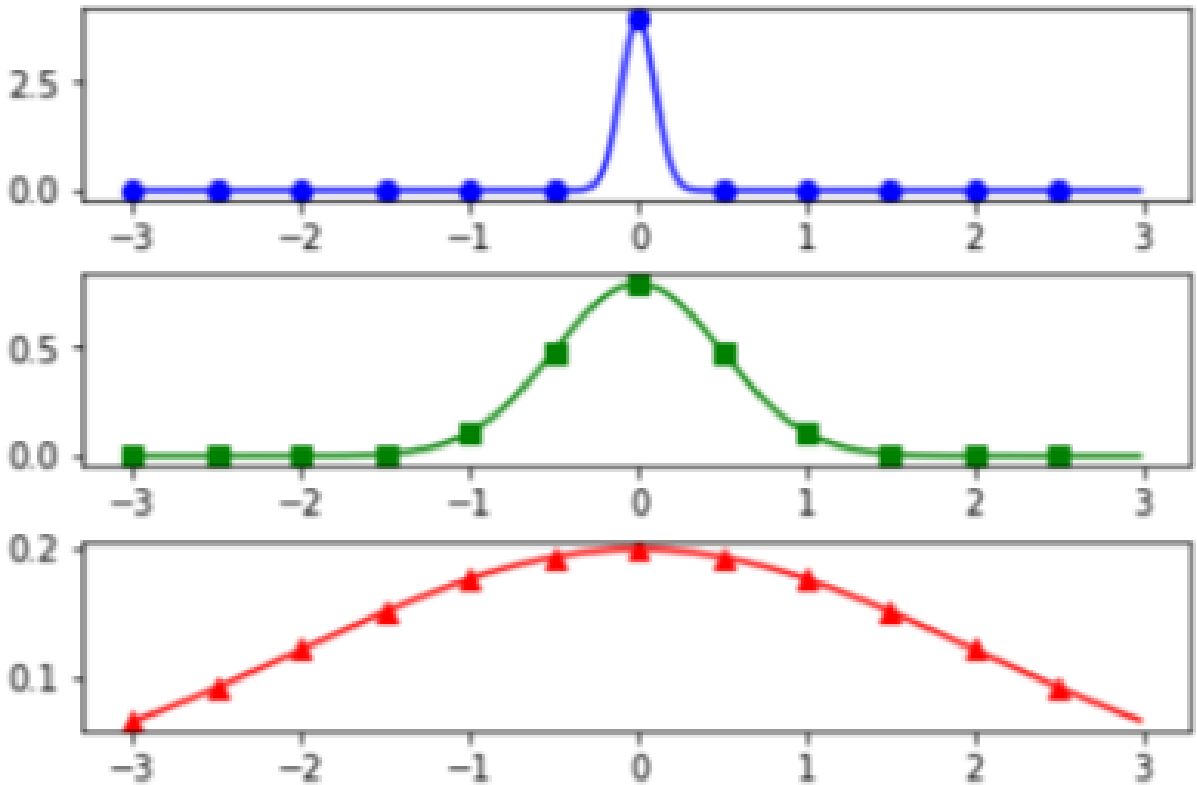
# Python: Matplotlib

## Subplots

```
# Second subplot, notice we're not actually plotting anything yet
# Also, we're not declaring another figure, since we're subplotting into the same figure.
plt.subplot(312) # 311 -> 3 rows, 1 columns, 2nd subplot
plt.plot(x1, g(0, sigma2, x1), 'gs', x2, g(0, sigma2, x2), 'g')
# Third subplot
plt.subplot(313) # 311 -> 3 rows, 1 columns, 3nd subplot
plt.plot(x1, g(0, sigma3, x1), 'r^', x2, g(0, sigma3, x2), 'r')
# Plotting
plt.subplots_adjust(hspace=0.4) # define space between subplots
plt.show() # only now we are actually plotting
```

# Python: Matplotlib

## Subplots



# Python: Matplotlib

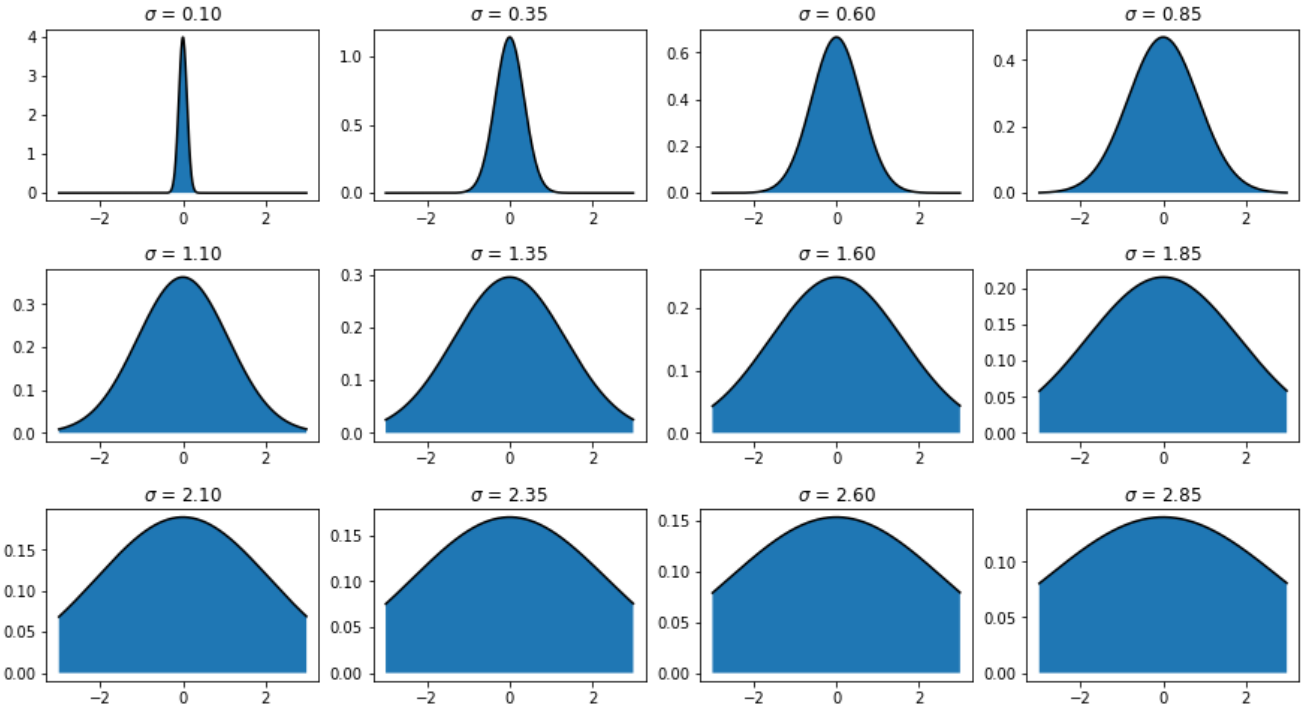
## Subplots

We can iterate over the axes we've created as well by creating declaring the size of our subplots explicitly.

```
# Let's plot a Nx3 grid with different values of sigma
x1 = np.arange(-3.0, 3.0, 0.01)
sigmas = np.arange(0.1,3,0.25)
(rows, cols) = sigmas.reshape(3,-1).shape # declare the size and
number of subplots
fig, axes = plt.subplots(nrows=rows, ncols=cols, figsize=(15, 8))
# we can iterate of axes:
for idx, ax in enumerate(axes.flat):
    sig = sigmas[idx]
    ax.plot(x1, g(0, sig, x1), 'k')
    # we can add LateX to our plot title by using r" " syntax, for instance:
    ax.set_title(r'$\sigma$ = {0:.2f}'.format(sig)) # .format(sig)
    # color the area under the graph between y = 0 and y = g
```

# Python: Matplotlib

## Subplots



# Python: Matplotlib

## Images

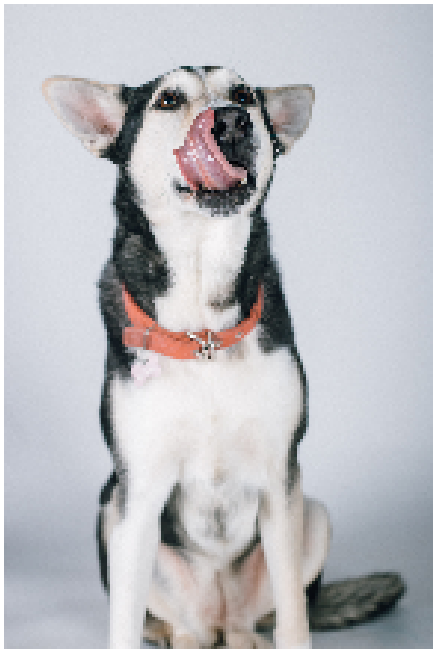
Matplotlib support image operations. While we'll mostly work with OpenCV, it is worth knowing the basics. You can find more on Matplotlib official image tutorial.

- The imread() command allows us to load images as NumPy arrays. Matplotlib rescaled the 8 bit data from each channel to floating point data between 0.0 and 1.0.
- Matplotlib plotting can handle float32 and uint8, but image reading/writing for any format other than PNG is limited to uint8 data.
- As you'll see when we'll reach OpenCV, using cv2.imread() has its downsides. Therefore, when possible, we recommend using plt.imread().

# Python: Matplotlib

## Images

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  nooka = plt.imread(r'<your_path>\ nooka.jpg')
4  plt.imshow(nooka)
5  plt.axis('off')
```



# Python: Matplotlib

## Images

Matplotlib imshow() is quite useful for visuallzing functions:

```
1  #recall this from the mgrid section
2  y, x = np.mgrid[-3:4:0.5, -3:4:0.5]
3  def f(x,y):
4    return np.cos(x)*np.sin(y)
5  fxy = f(x,y)
6  # We can explicitly declare the figure and axes
7  fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(8, 5))
8  fig.suptitle("f(x,y) = cos(x)*sin(y)") # main title
```

# Python: Matplotlib

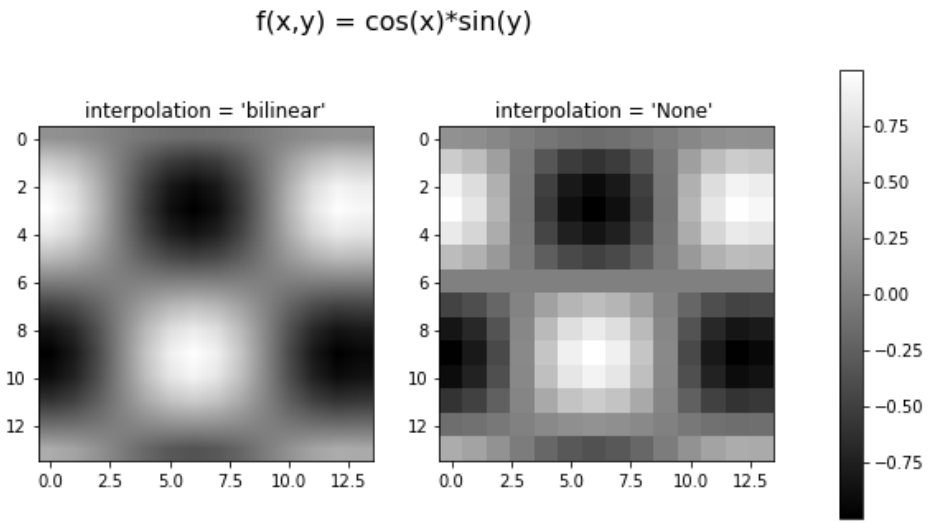
## Images

Matplotlib imshow() is quite useful for visuallzing functions:

```
1  # You can choose the type of interpolation between
     adjacent cells in the (discrete) mgrid as well as the
     color map (cmap).
2  ax = plt.subplot(121)
3  plt.imshow(fxy, interpolation = 'bilinear', cmap = 'gray')
4  ax.set_title("interpolation = 'bilinear'") #set title to
     this 'ax'
5  ax = plt.subplot(122)
6  ax.set_title("interpolation = 'None' ")
7  plt.imshow(fxy, interpolation="None", cmap = 'gray')
8  # Note that it is the string "None", which is different
     from interpolation=None which uses Python's built-in
     constant, None.
9  # we can play with the anchor's location to relocate our
     colorbar
10 cbar = plt.colorbar(ax=axes, anchor = (2,0))
11 plt.show()
```

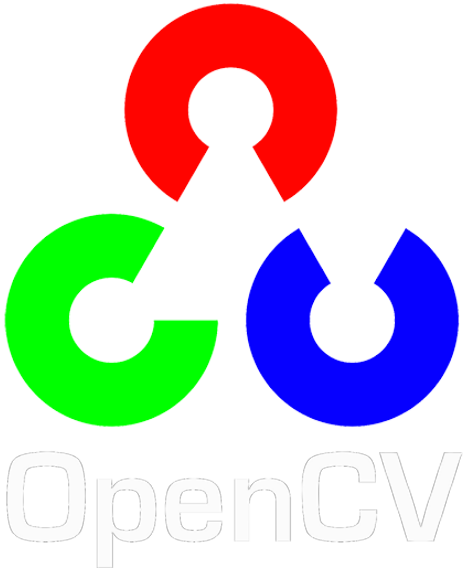
# Python: Matplotlib

## Images




It is sometimes useful, when debugging CV code to view the images without any interpolation (i.e., look directly at the pixel level).

# Python: openCV



# Python: openCV

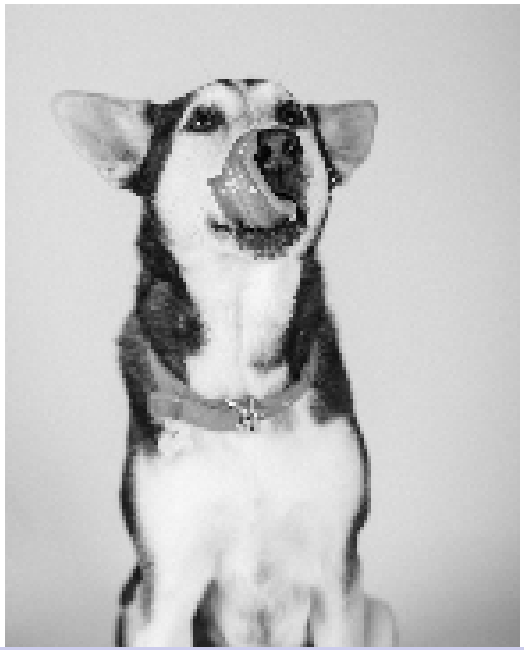
OpenCV (Open Source Computer Vision Library) is a library with Python, C++ and Java interfaces. We shall use OpenCV with Python bindings. Please see getting started section in the course website for the correct version and installation guide.

# Python: openCV

```
1 import cv2
2 # Load colored image
3 img = cv2.imread('nooka.jpg',1) # 1 color, 0 grayscale, -1
    unchanged
4 # Display image and destory window after any key is
5 pressed cv2.imshow('Doggie', img) # title, image
6 cv2.waitKey(0) # wait until any key is pressed
7 cv2.destroyAllWindows()
```

# Python: openCV

```
# Convert to grayscale
img_gray= cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# notice anything weird?
cv2.imshow('Gray doggie', img_gray) # title, image
cv2.waitKey(0) # wait until any key is pressed
cv2.destroyAllWindows()
# Saving
cv2.imwrite('gray_nooka.png', img_gray) # (new file name,
    image to save)
```



# Python: openCV

## Working with matplotlib

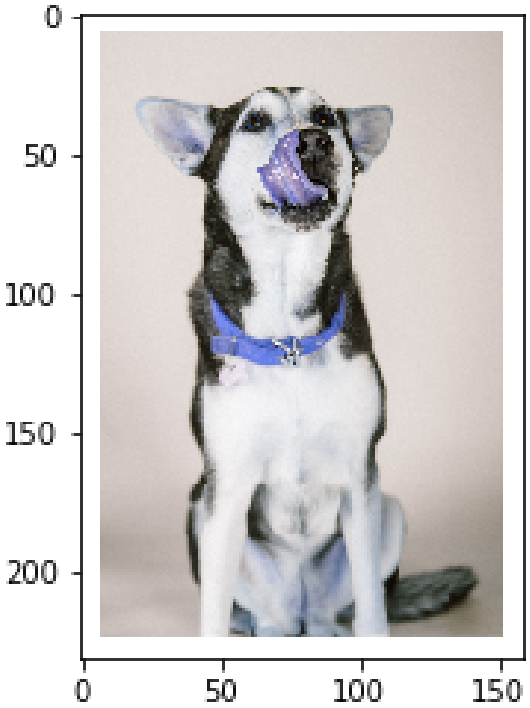
As you might have notice, when we converted the colored image to grayscale, we used the method:

```
1 img_gray= cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Converting BGR to GRAY (and not RBG to grayscale). This is because OpenCV uses BGR color channel instead of RGB.
When working with OpenCV it's ok, but if we wish to work with matplotlib as well, this will happen:

```
1 # Load colored image
2 img = cv2.imread('nooka.png',1)
3 # 1 color, 0 grayscale, -1 unchanged
4 plt.imshow(img)
5 # Notice we've used plt.imshow() and not cv2.imshow()
```

# Python: openCV

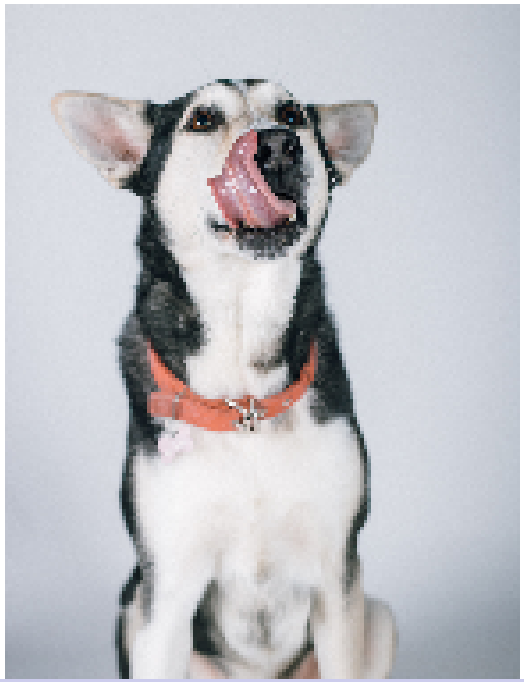


# Python: openCV

## Working with matplotlib

So, we'd usually want to convert BGR to RGB:

```
1 img = cv2.imread('nooka.jpg',1)
2 #default is BGR not RGB
3 img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
4 # We can also save images with matplotlib
5 plt.imsave('image rgb.png', img_rgb)
6 plt.imshow(img_rgb)
```



# Python: openCV

## Resizing images

```
# Loading image
img = cv2.imread('nooka.jpg',1)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# This image is a bit too big. So we will resize it.
nRows,nCols = img.shape[:2]
print('original shape:',nRows,nCols)
# original shape: 521 550
nRows /= 3
nCols /= 3
# Note that even if the returned number is integer, it
    automaticallt transfomed into float
nRows = int(nRows)
nCols = int(nCols)
print('new shape:',nRows,nCols)
# new shape: 130 137
img = cv2.resize(img,(nRows,nCols))
plt.imshow(img)
```

# Python: openCV