Udemy OpenCV Notes  
numpy = basically does all the array stuff  
an image can be represented with a 2d array of RGB tuples.   
  
Using PIL library, matplotlib and numpy  
**from PIL import Image**   
This allows you to import an image from a file   
**import matplotlib.pyplot as plt  
%matplotlib inline  
  
plt.imshow**  
this allows you to show the image you translated into an array with grid lines. Super handy.  
  
# RED CHANNEL VALS

**plt.imshow(pic\_red[:,:,0], cmap='gray')**# GREEN CHANNEL VALS

**plt.imshow(pic\_red[:,:,1], cmap='gray')  
  
import cv2** 🡨 this allows you to import the open cv library  
**cv2.imread(** 🡨 this reads an image directly into an np array. Will not give errors if you import the wrong or non existent filepath

Open CV reads in colors as BGR, but matplotlib requires RGB, this function changes the RGB parameters.

**fix\_img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)**When resizing, note that the scaling matplotlib uses and opencv uses are opposite wrt coordinates

**rectangle(image, pt1, pt2, color, thickness)** 🡨 this creates a rectangle on any image you supply, probably very useful for bounding boxes later.

vertices = np.array([ [100,300], [200,200], [400,300], [200,400] ], dtype=np.int32) 🡨 Create all the points on a polygon  
**pts = vertices.reshape((-1,1,2))** 🡨 When creating a polygon, you must reshape for the matrices to work

**img = np.zeros((512,512,3))**Blending images  
cv2.addWeighted(src1, alpha, src2, beta, gamma) 🡨 produces a blended composition of src1 and src2 dependent on the values of alpha and beta.   
  
img = cv2.imread('DATA/rainbow.jpg', 0) 🡨 the 0 here allows you to read in as grey scale

image thresholding allows you to set all values below a certain grayscale or colour value to 0, and everything else above that goes to 255  
**ret, thresh1 = cv2.threshold(img, 150, 255, cv2.THRESH\_BINARY)**

**ADAPTIVE THRESHOLDING:**

Looks at a “neighbourhood” of pixels around a certain pixel to determine what should be included.  
**th2 = cv2.adaptiveThreshold(img,255,cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, 11, 8)**

Common practice is to use multiple threshold outputs weighted with addWeighted(). Experimenting with the various types of thresholding is necessary to find the best one.  
  
WINDOW CONTROL:

**while True:**

**cv2.imshow('my\_drawing', img)**

**if cv2.waitKey(20) & 0xFF == 27:** 🡨 this function waits for an escape key to be pressed after 20ms

**break**

**cv2.destroyAllWindows() 🡨** kills all opencv windows.  
  
Blurring and smoothing

Important for edge detection, high resolution images result in too many edges detected. Blurring the photo results in more accurate detection.  
  
Methods:   
1) Gamma correction

Gamma correction makes an image brighter or darker.

2) Kernel based filters

Image kernels are small matrix used to apply effects that can truthfully be anything. These kernels are multiplied to all the various pixels in the image. Kernels can be used to sharpen, blur, emboss, outline, and many other things. The matrix for blurring is ( 0.0625 0.125 0.0625, 0.125 0.25 0.125, 0.0625 0.125 0.0625)  
  
Blurring functions:  
  
**blurred = cv2.GaussianBlur(img, (7,7), 10)** 🡨 takes an average value and blurs it

Median blur keeps the detail within the image while removing some of the noise from the image.

**median\_result = cv2.medianBlur(withText(), 5)**

Morphological Operators

Morphological operators are sets of kernels that can achieve a variety of effects. They are often used to reduce noise. Some operators are very good at reducing black points on a white background.

They are also very good at achieving erosion and dilation effects from an existing image. This is really good at fixing up text in an image.  
  
Erosion code  
**result = cv2.erode(img, kernel, iterations=3)**  
Opening code  
**noiseless = cv2.morphologyEx(noise\_img, cv2.MORPH\_OPEN,kernel)** 🡨 This is really good for white noise or background noise. Opening is an erosion followed by a dilation.

Closing code

**closing = cv2.morphologyEx(black\_noise\_img, cv2.MORPH\_CLOSE, kernel)** 🡸 quite good for noise that corresponds to the background image. This will attempt to fill in the gaps where they are missing, as opposed to reducing down noise like with the open code

Gradient  
**gradient = cv2.morphologyEx(img, cv2.MORPH\_GRADIENT, kernel)** 🡨 decent for an initial edge detection kind of deal. This finds the difference between Closing and Opening  
  
Histograms  
  
Histograms are great for telling you the frequency of occurrences. This is useful with image recognition because you can perform histograms on color frequency as well as how frequent each channel is being represented.

Code for histogram of a single channel:  
**hist\_values = cv2.calcHist([dark\_horse], channels=[0], mask=None, histSize=[256], ranges=[0,256])**  
  
Code for histogram of all channels:

**img = dark\_horse  
color = ('b', 'g', 'r')  
for i, col in enumerate(color):  
 histr = cv2.calcHist([img],[i],None,[256],[0,256])  
 plt.plot(histr, color=col)  
 plt.xlim([0,50])  
 plt.ylim([0, 500000])  
plt.title("HISTOGRAM FOR HORSE")**