

# Computer Vision

Software that can see

**James Boyden, Snap Disco**  
**<james@snapdisco.com>**

**National Computer Science School**  
**Masterclass, 2016**

# Introduction

# The Terminator



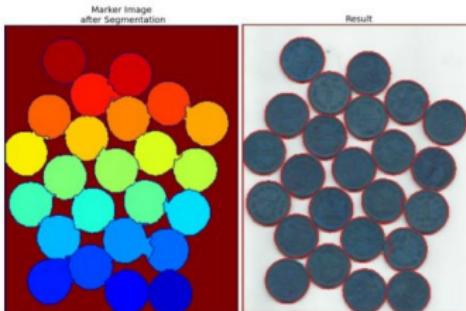
# Terminator Vision



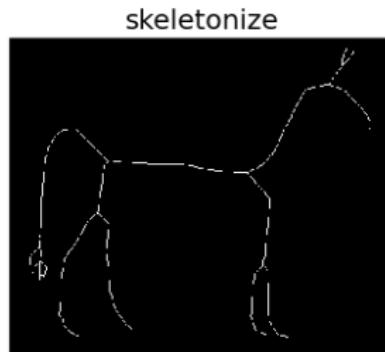
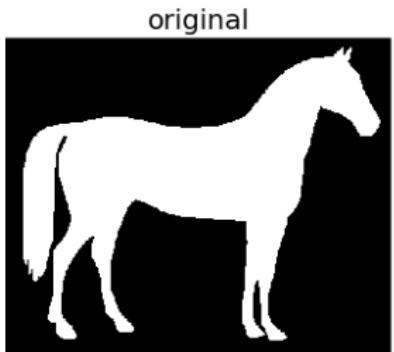
- Reddish colour
  - Foreground object segmented
  - White boundary traced around object

## What is Image Processing? Some examples...

- Image enhancement (sharpening, smoothing, brightening, ...)
  - Image analysis
  - Image segmentation
  - Image structure detection (skeletonization)
  - ⇒ Improve the image for **humans**
  - (or sometimes for later stages of Computer Vision)

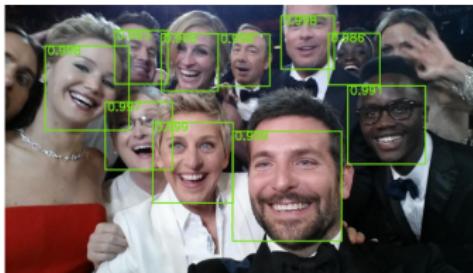


## An example of skeletonization



## What is Computer Vision? Some examples...

- Object component detection (edge & region detection)
- Object classification
- Object recognition
- Face detection
- Face recognition
- ⇒ Analyse the image for **software**



## Python packages we will use

- **Numpy**: efficient multi-dimensional arrays
- **Scipy**: scientific computing (like Matlab toolboxes)  
(image processing, statistics, spatial algorithms, signal processing, Fourier transforms, linear algebra, equation solvers, ...)
- **Skimage (a.k.a. scikit-image)**: image-processing
- **Matplotlib**: graph plotting (like Matlab plotting)

Also maybe of interest:

- **OpenCV**: A comprehensive collection of CV algorithms

# Maths that's useful for image processing & CV

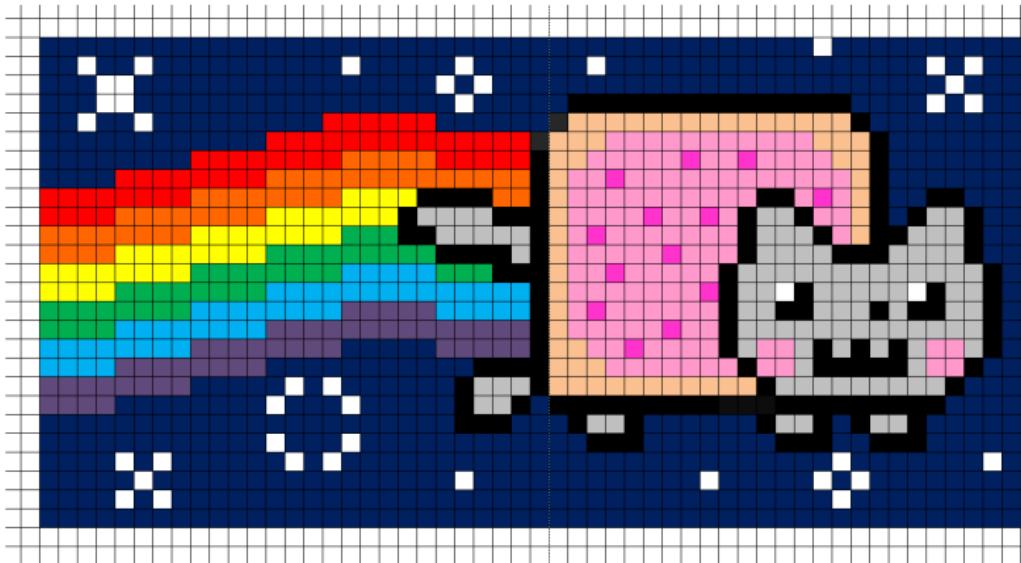
- Calculus (*esp.* differentiation) (in highschool)
- Trigonometry (in highschool)
- Linear Algebra (*esp.* matrices)
- Vector maths (*esp.* in 2-D & 3-D)
- 2nd-year Linear Algebra (*esp.* “eigenvalues & eigenvectors”)
- Fourier Analysis
- Graph Theory algorithms

# Today: 3x 40-min mini-tutorials: lecture then practical

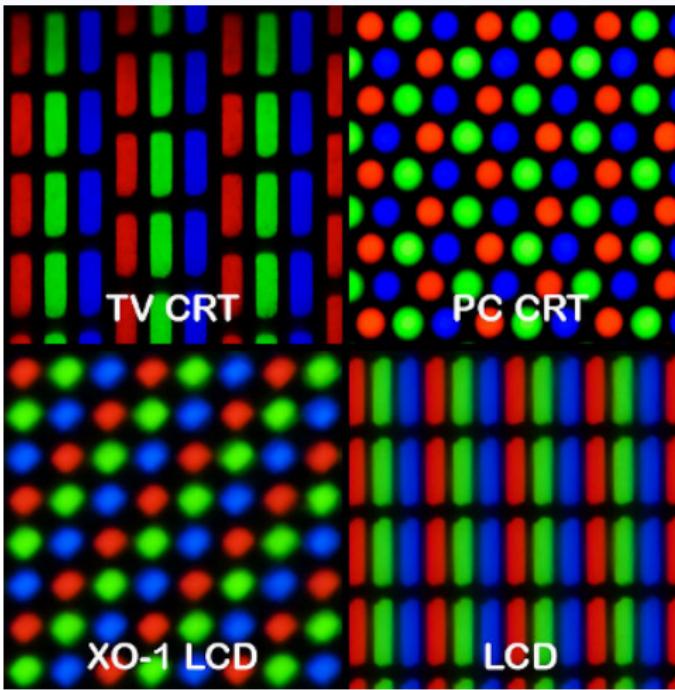
- 1** Images & colour
- 2** Numpy arrays
- 3** Filters, edges, histograms, regions

# Images & colour

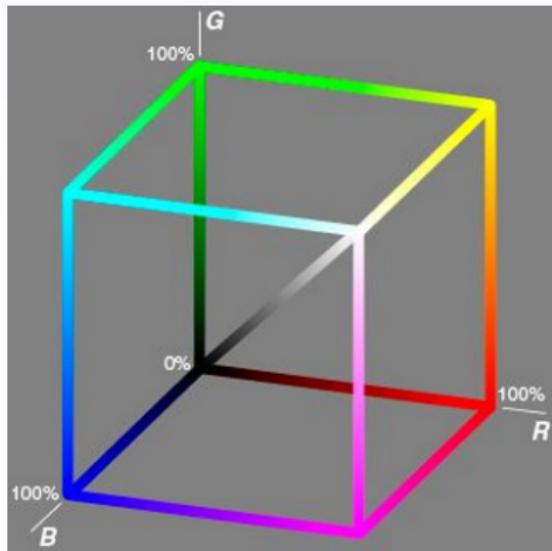
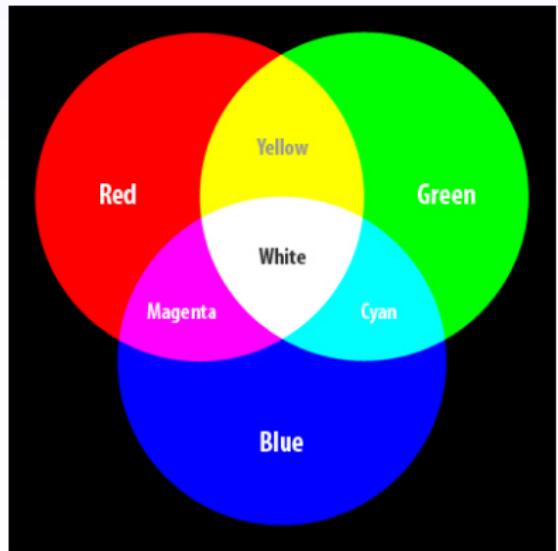
## A “raster image” is a grid of coloured pixels



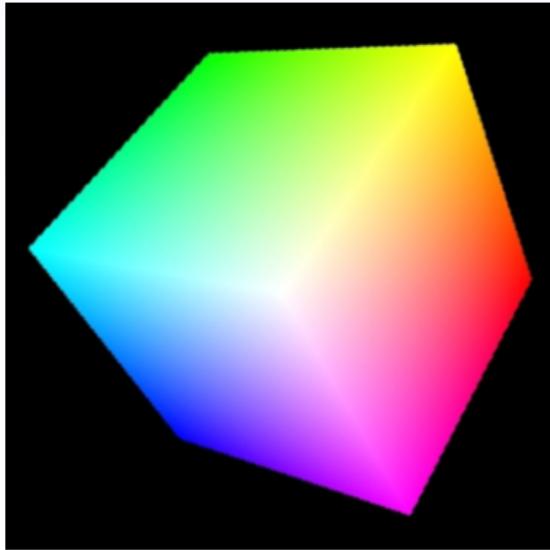
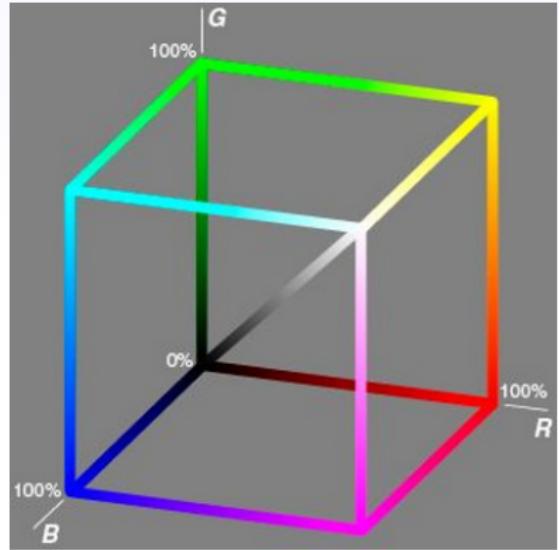
## Pixels consist of Red, Green, Blue (RGB) components



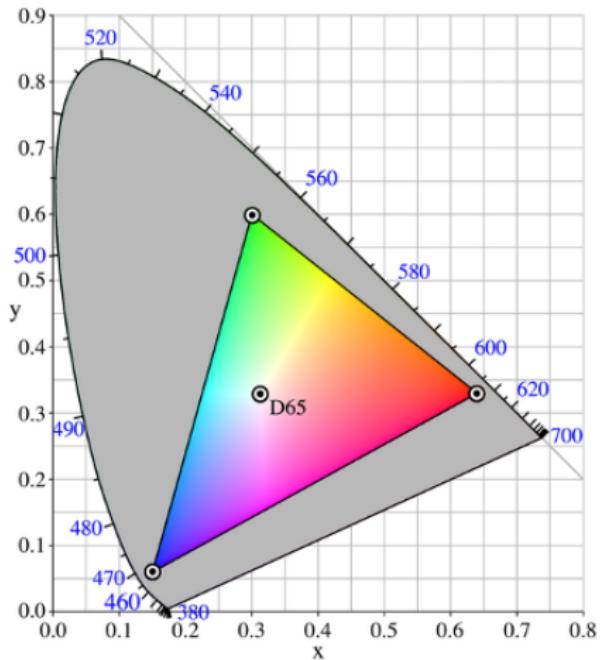
## Combine RGB colour components in different amounts ...



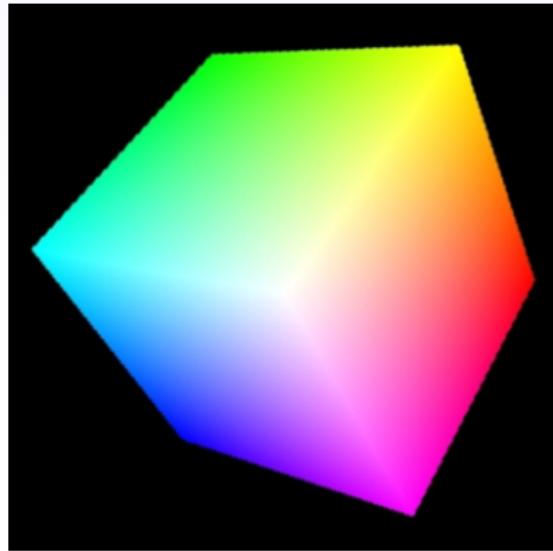
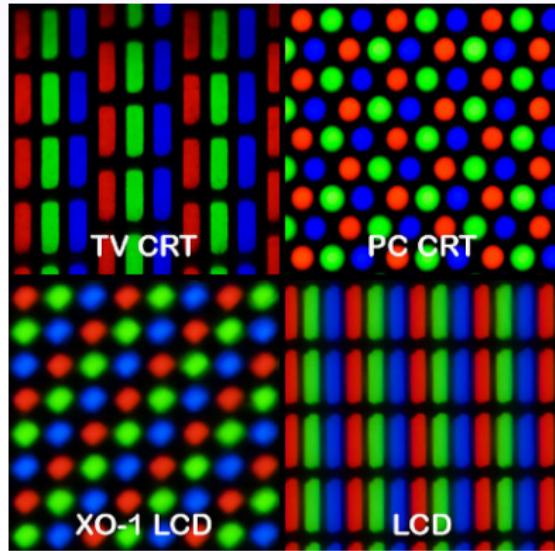
... to create (almost) any colour



## RGB can represent most of the colours visible to humans



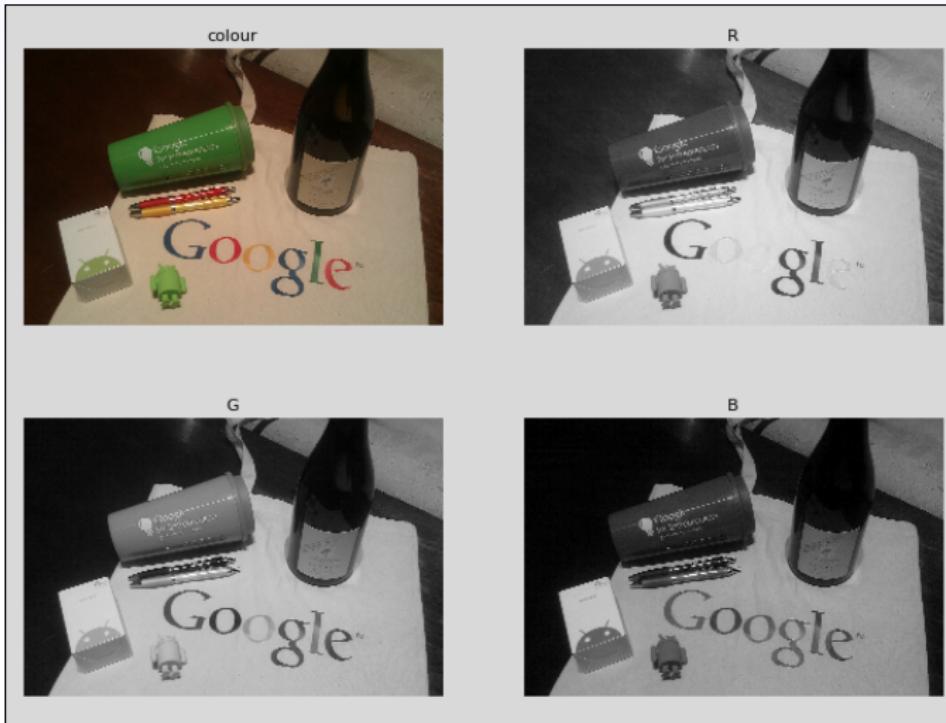
So, each pixel displays some combination of R, G, B



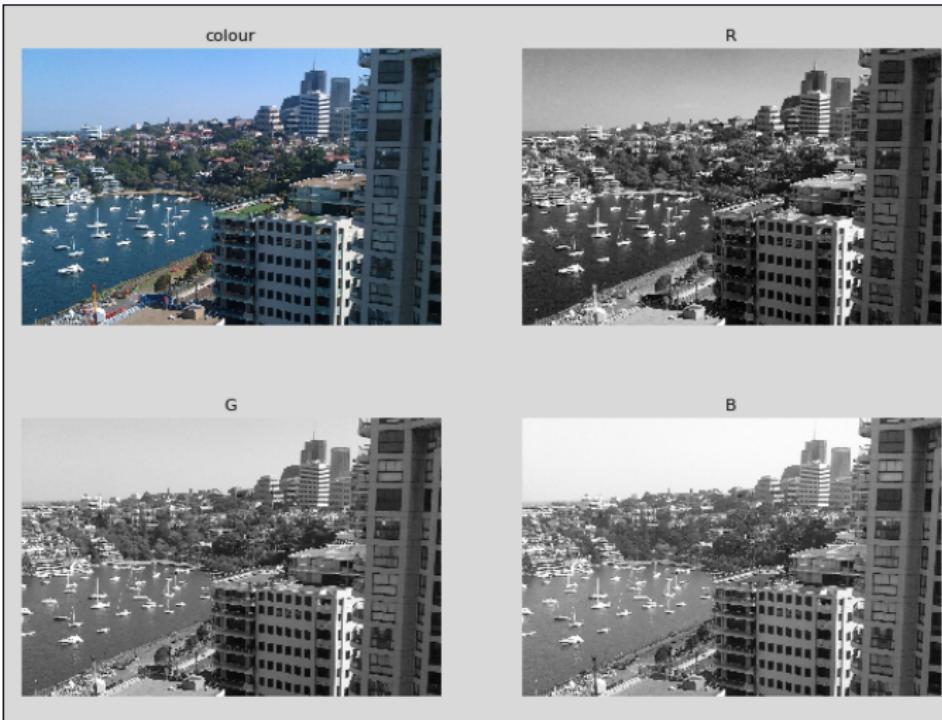
## A single-colour slice of R, G or B is called a “channel”



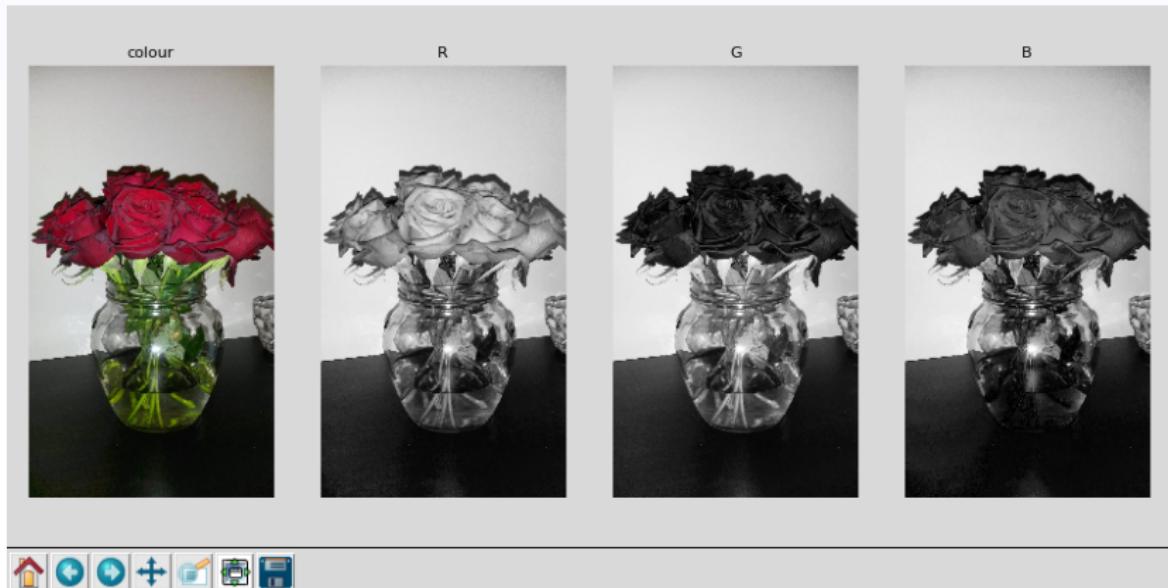
## Each channel represents the “intensity” of R, G, B ...



... where the maximum intensity appears white ...



... and the minimum intensity appears black



## There are also single-channel “greyscale” images



## And “binary” images of just black and white



## Different image types have different pixel types

- **binary image**: just a “boolean” value (True or False)
- **greyscale**: 1 numeric value (a greyscale intensity)
- **colour** (RGB): 3 numeric values (the R, G, B intensities)

A numeric value is generally either:

- an integer in the range 0–255 (**uint8**)
- a floating-point number in the range 0.0–1.0 (**float64**)

## A bit of code to get you started...

```
>>> # To load an image in Numpy & Scipy:  
>>> from scipy.misc import imread  
>>> img = imread("some-image.jpg")  
  
>>> # To display an image using Matplotlib:  
>>> from matplotlib import pyplot as plt  
>>> plt.imshow(img)  
>>> plt.show()  
  
>>> # To show the pixels as "pixelated" rather than smooth:  
>>> plt.imshow(img, interpolation="none")  
>>> plt.show()
```

## A bit of code to get you started...

```
>>> # To load an image in Numpy & Scipy:  
>>> from scipy.misc import imread  
>>> img = imread("some-image.jpg")  
  
>>> # To display an image using Matplotlib:  
>>> from matplotlib import pyplot as plt  
>>> plt.imshow(img)  
>>> plt.show()  
  
>>> # To show the pixels as "pixelated" rather than smooth:  
>>> plt.imshow(img, interpolation="none")  
>>> plt.show()  
  
# Test images & more code at https://github.com/jboyd/ncss2016/images  
# and https://github.com/jboyd/ncss2016/part1
```

# Numpy arrays

## Python list: Pros

- A very convenient general-purpose collection
- Simple, convenient syntax:  
[5, 4, 0, 1, 9, 5, 2]
- Able to contain multiple different data types at once:  
[5, "hello", 2.2]
- Just a few easy-to-remember methods:  
append, count, extend, index, insert, pop, remove, reverse, sort
- Indexing & slicing are intuitive

## Python list examples: Access an element at an index

```
>>> my_list = [5, 4, 0, 1, 9, 5, 2]
>>> len(my_list)
7

# Index an element:
# Indices run from 0 to (len-1),
# inclusive.
>>> my_list[0]
5
>>> my_list[1]
4
>>> my_list[6]
2

# Negative indices count backwards
# from the end of the list.
# -1 is the index of the last
# element in the list.
>>> my_list[-1]
2
>>> my_list[-2]
5
```

## Python list examples: “Slice” to obtain a sub-list

```
>>> my_list = [5, 4, 0, 1, 9, 5, 2]  
  
# Slice a range of elements:  [start:stop]  
>>> my_list[1:5]  
[4, 0, 1, 9]
```

Index from rear:	-6	-5	-4	-3	-2	-1	
Index from front:	0	1	2	3	4	5	
	-----+-----+-----+-----+-----+						
	a   b   c   d   e   f						
	-----+-----+-----+-----+-----+						
Slice from front:	:	1	2	3	4	5	:
Slice from rear:	:	-5	-4	-3	-2	-1	:

## Python list examples: “Slice” to obtain a sub-list

```
# Slice from the start:  
>>> my_list[:5]  
[5, 4, 0, 1, 9]  
  
>>> my_list = [5, 4, 0, 1, 9, 5, 2]  
  
# Slice a range of elements:  
>>> my_list[1:5]  
[4, 0, 1, 9]  
  
# Slice to the end:  
>>> my_list[1:]  
[4, 0, 1, 9, 5, 2]  
  
# Slice "the whole list":  
>>> my_list[:]  
[5, 4, 0, 1, 9, 5, 2]
```

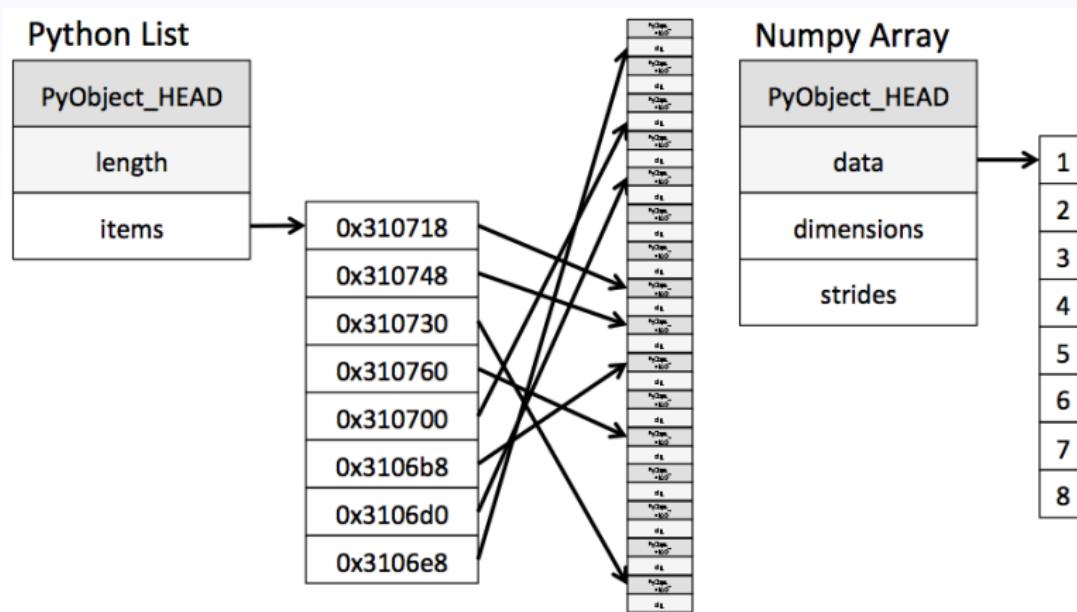
## Python list: Cons (for image processing)

- Python lists use “indirect” storage of Python objects
- This takes up more memory (a problem for large images)
- This also makes loops slower
  
- No numerical methods or functions beyond `max`, `min`, `sum`
- So you have to write your own basic functions...
  
- Not easy to create a list of an arbitrary length
- Especially not easy to create a desired 2-D shape
  
- In fact, there's **no way** to tell a list to “be 2-D”

## Numpy arrays

- A single Numpy array holds only **one specified data type**
- Benefit: The data can be held compactly in memory
- Benefit: It takes up less memory, and makes loops faster
- Numpy arrays have many numerical methods and functions:  
`max, mean, min, sum, argmax, argmin, clip, round`
- And many more!
- It's very easy to tell the Numpy array what shape you want

# Python lists vs. Numpy arrays in memory



## Creating a Numpy array

```
>>> import numpy as np # ("np" is short for "numeric python")  
  
>>> my_arr = np.array([5, 4, 0, 1, 9, 5, 2])  
>>> my_arr  
array([5, 4, 0, 1, 9, 5, 2])  
  
>>> my_arr.dtype  
dtype('int64')  
# ^ data type is 64-bit integer  
  
>>> my_arr.shape  
(7,)  
# ^ 1-D, of length 7
```

## Creating a specific length: lists vs. Numpy arrays

```
# Python list
```

```
# Numpy array
```

```
>>> import numpy as np
```

```
# A range
```

```
# A range
```

```
>>> range(6)
```

```
>>> np.arange(6)
```

```
[0, 1, 2, 3, 4, 5]
```

```
array([0, 1, 2, 3, 4, 5])
```

```
# All zeros
```

```
# All zeros (floating-point by default)
```

```
>>> [0] * 6
```

```
>>> np.zeros(6)
```

```
[0, 0, 0, 0, 0, 0]
```

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

```
# All ones
```

```
# All ones (floating-point by default)
```

```
>>> [1] * 6
```

```
>>> np.ones(6)
```

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

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

## Specify a desired data type using the `dtype` parameter

```
>>> import numpy as np
```

```
# A range
```

```
>>> range(6)
```

```
[0, 1, 2, 3, 4, 5]
```

```
# A range
```

```
>>> np.arange(6)
```

```
array([0, 1, 2, 3, 4, 5])
```

```
# All zeros
```

```
>>> [0] * 6
```

```
[0, 0, 0, 0, 0, 0]
```

```
# All zeros (now int32 type)
```

```
>>> np.zeros(6, dtype=np.int32)
```

```
array([0, 0, 0, 0, 0, 0], dtype=int32)
```

```
# All ones
```

```
>>> [1] * 6
```

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

```
# All ones (now int32 type)
```

```
>>> np.ones(6, dtype=np.int32)
```

```
array([1, 1, 1, 1, 1, 1], dtype=int32)
```

## Specify a desired data type using the `dtype` parameter

```
>>> import numpy as np
```

```
# All False (for binary images)
>>> [False] * 4
[False, False, False, False]
```

```
# All False (for binary images)
>>> np.zeros(4, dtype=np.bool)
array([False, False, False, False],
      dtype=bool)
```

```
# All True (for binary images)
>>> [True] * 4
[True, True, True, True]
```

```
# All True (for binary images)
>>> np.ones(4, dtype=np.bool)
array([ True,  True,  True,  True],
      dtype=bool)
```

## Specify a desired 2-D shape using a shape tuple

```
# We want a shape of 4 rows x 7 columns.  
>>> my_arr = np.zeros((4, 7), dtype=np.int32)  
>>> my_arr  
array([[0, 0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0, 0]], dtype=int32)  
>>> my_arr.shape  
(4, 7)
```

## Or reshape an existing array

```
>>> my_long_array = np.arange(28) # 'arange' creates a 1-D array.  
>>> my_long_array  
array([ 0,  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])  
>>> my_long_array.shape  
(28,)
```

```
# Note that 4*7 == 28, so total number of elements is unchanged.  
>>> my_arr2 = my_long_array.reshape((4, 7)) # Reshape to 2-D.  
>>> my_arr2  
array([[ 0,  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]])  
>>> my_arr2.shape  
(4, 7)
```

## We can index the array in 2-D to access an element

```
>>> my_long_array = np.arange(28)                                >>> my_arr2[0,0]
>>> my_arr2 = my_long_array.reshape((4, 7))                      0
>>> my_arr2
>>> my_arr2[1,1]                                                 8
array([[ 0,  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]])
>>> my_arr2.shape                                              13
(4, 7)                                                       22
>>> my_arr2[3,1]                                                 27
>>> my_arr2[3,6]                                                 27
>>> my_arr2[-1,-1]                                              27
```

## Shapes and indexing in 2-D arrays

- [0, 0] (the origin) is always in the top-left.
- [-1, -1] is always in the bottom-right.
- **Tip: Don't** think of the coordinates as  $(x, y)$ .
- Instead, think of the coordinates as  $(row, column)$ .
- To specify a shape: `(num_rows, num_cols)`
- To index an array: `arr[row_idx, col_idx]`

`num_cols == 4`

`num_rows == 3`

	Column 0	Column 1	Column 2	Column 3
Row 0	<code>arr[0, 0]</code>	<code>arr[0, 1]</code>	<code>arr[0, 2]</code>	<code>arr[0, 3]</code>
Row 1	<code>arr[1, 0]</code>	<code>arr[1, 1]</code>	<code>arr[1, 2]</code>	<code>arr[1, 3]</code>
Row 2	<code>arr[2, 0]</code>	<code>arr[2, 1]</code>	<code>arr[2, 2]</code>	<code>arr[2, 3]</code>

## We can create a single-channel (greyscale) image

```
>>> arr_256 = np.arange(256, dtype=np.uint8)
>>> arr_256.shape
(256,)
>>> arr_1616 = arr_256.reshape((16, 16))
>>> arr_1616.shape
(16, 16)

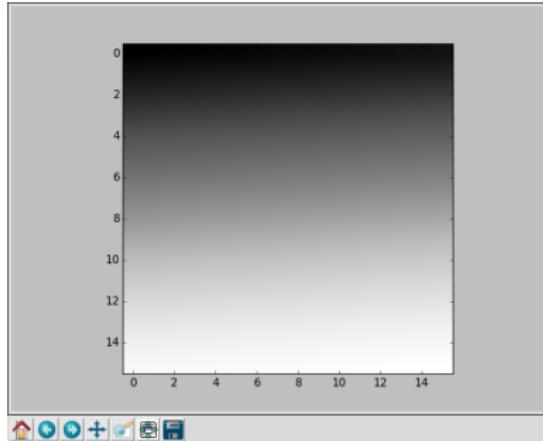
# This is a valid greyscale image:
# dtype == uint8, values are in the range [0, 255]
>>> arr_1616.dtype
dtype('uint8')
>>> arr_1616.min()
0
>>> arr_1616.max()
255
```

## We can create a single-channel (greyscale) image

```
array([[ 0,  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],
       [ 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47],
       [ 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63],
       [ 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [ 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95],
       [ 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111],
       [112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127],
       [128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143],
       [144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159],
       [160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175],
       [176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191],
       [192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207],
       [208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223],
       [224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239],
       [240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255]],
```

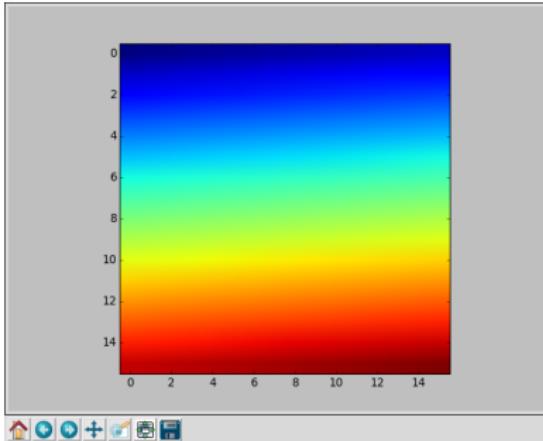
## Display the image using Matplotlib

```
>>> import matplotlib.pyplot as plt  
# Note that we specify a greyscale colourmap.  
>>> plt.imshow(arr_1616, cmap=plt.cm.Greys_r)  
>>> plt.show()
```



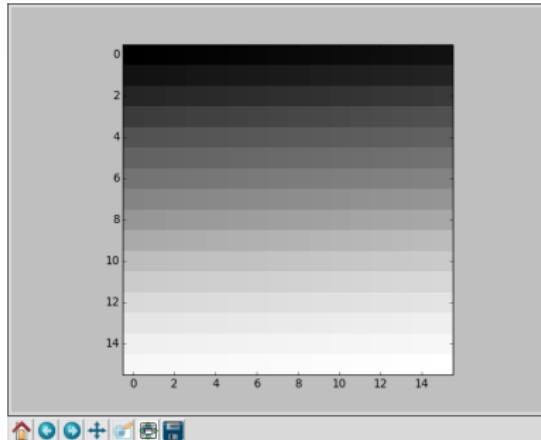
## By default, Matplotlib displays greyscale images in colour

```
# If we don't specify a colourmap, Matplotlib uses false colour:  
# black -> blue, white -> red, with a rainbow in-between.  
>>> plt.imshow(arr_1616)  
>>> plt.show()
```



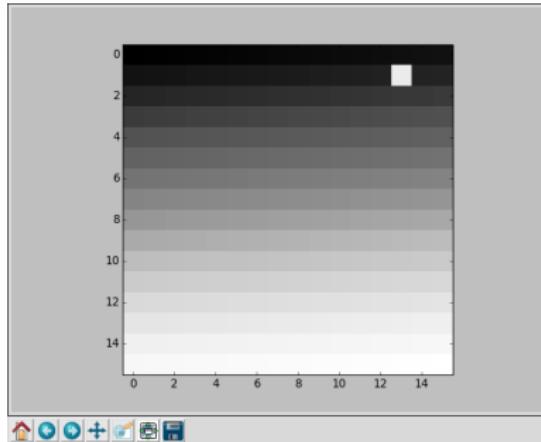
## Switch off smoothing to see individual image pixels

```
# Use interpolation="none" to switch off the colour smoothing.  
>>> plt.imshow(arr_1616, cmap=plt.cm.Greys_r,  
              interpolation="none")  
>>> plt.show()
```



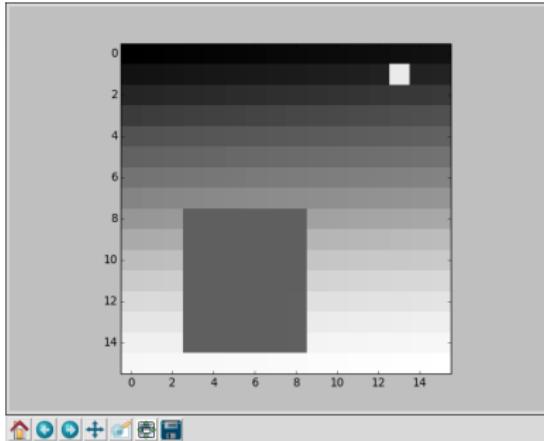
## We can modify individual pixels using indexing...

```
>>> arr_1616[1, 13] = 220 # 220 == light grey, almost 255 (white)
>>> plt.imshow(arr_1616, cmap=plt.cm.Greys_r,
              interpolation="none")
>>> plt.show()
```



## ... or modify larger rectangles using slicing

```
>>> arr_1616[8:15, 3:9] = 77 # rows == [8:15], columns == [3:9]
>>> plt.imshow(arr_1616, cmap=plt.cm.Greys_r,
              interpolation="none")
>>> plt.show()
```



## Load colour images using `scipy.misc.imread`

```
>>> from scipy.misc import imread
>>> img = imread("20141226_222803.jpg")
# Test images at https://github.com/jboy/ncss2016/images

>>> img.shape
(1836, 3264, 3)
# ^ Note: 1836 rows x 3264 columns x 3 colour channels (RGB)

>>> img.dtype
dtype('uint8')
>>> img.min(), img.max()
(0, 255)

# No need to specify a colourmap for a colour image.
>>> plt.imshow(img)
>>> plt.show()
```

## Some image-processing functions only operate on greyscale

```
# You can access individual channels by indexing in the 3rd dimension
# (the colour-channel dimension).

# An individual channel is like a greyscale image.

# The 1st ':' is for "all rows"; the 2nd ':' is for "all columns".
>>> red_chan = rgb_img[:, :, 0]
>>> red_chan.shape
(1836, 3264)

# The ellipsis ("...") means "fill in as many ':' as necessary",
# so rgb_img[..., 0] is equivalent to rgb_img[:, :, 0]

green_chan = rgb_img[..., 1]
blue_chan = rgb_img[..., 2]
```

## Some image-processing functions only operate on greyscale

```
# You can also convert a colour image entirely to greyscale.  
# See http://scikit-image.org/docs/dev/api/skimage.color.html#rgb2grey  
>>> from skimage.color import rgb2grey  
>>> rgb_img.shape  
(1836, 3264, 3)  
>>> greyscale_img = rgb2grey(rgb_img)  
>>> greyscale_img.shape  
(1836, 3264)
```

## You can change the overall image tint by editing a channel

```
>>> import matplotlib.pyplot as plt  
>>> from scipy.misc import imread  
>>> img = imread("IMAG0537.jpg")  
>>> img[...,0] = 255 # set red channel to max  
>>> plt.imshow(img) ; plt.axis("off") ; plt.show()
```



## Here are some short scripts to save you some typing...

- Scripts are at <https://github.com/jboy/ncss2016/part1>

```
useful.py  
display_rgb_colour.py  
display_rgb_grey.py
```

- Slides are at <https://github.com/jboy/ncss2016/part2>

# Filters, edges, histograms, regions

# Filters

- A **filter** applies the same operation to every pixel in an image.
- Filters can be used for image enhancement (smoothing, sharpening)
- or for edge detection (part of image segmentation)

## Gaussian filter blurs the image

```
>>> import matplotlib.pyplot as plt
>>> from scipy.misc import imread
>>> img = imread("IMAG0537.jpg")

>>> from skimage.filters import gaussian
>>> blurred15 = gaussian(img, 15)

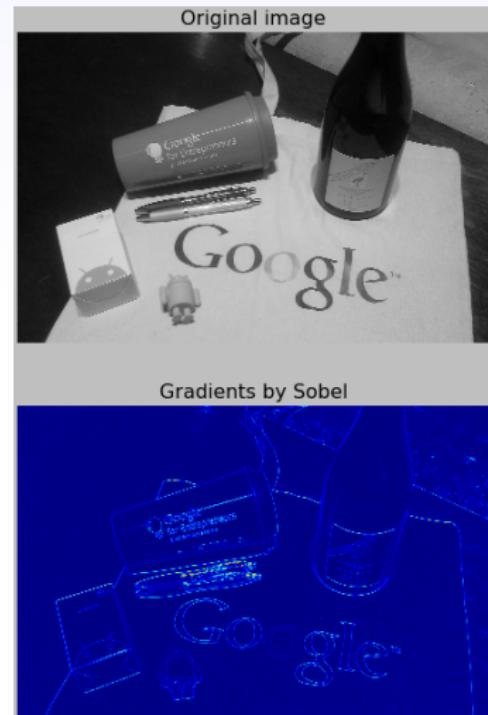
>>> fig, (ax0, ax1) = plt.subplots(nrows=2)
>>> ax0.imshow(img)
>>> ax0.set_title("Original image")
>>> ax0.axis("off")
>>> ax1.imshow(blurred15)
>>> ax1.set_title("Gaussian(15) blur")
>>> ax1.axis("off")
>>> plt.show()
```



## Sobel filter calculates greyscale gradient magnitude

```
>>> from skimage.color import rgb2grey
>>> from skimage.filters import sobel
# Sobel only accepts greyscale images.
>>> img_g = rgb2grey(img)
>>> gradients = sobel(img_g)

>>> fig, (ax0, ax1) = plt.subplots(nrows=2)
>>> ax0.imshow(img_g, cmap=plt.cm.Greys_r)
>>> ax0.set_title("Original image")
>>> ax0.axis("off")
>>> ax1.imshow(gradients)
>>> ax1.set_title("Gradients by Sobel")
>>> ax1.axis("off")
>>> plt.show()
```



## Canny edge detector finds binary edges in greyscale

```
>>> from skimage.color import rgb2grey
>>> from skimage.filters import canny
# Canny only accepts greyscale images.
>>> img_g = rgb2grey(img)
>>> edges = canny(img_g)

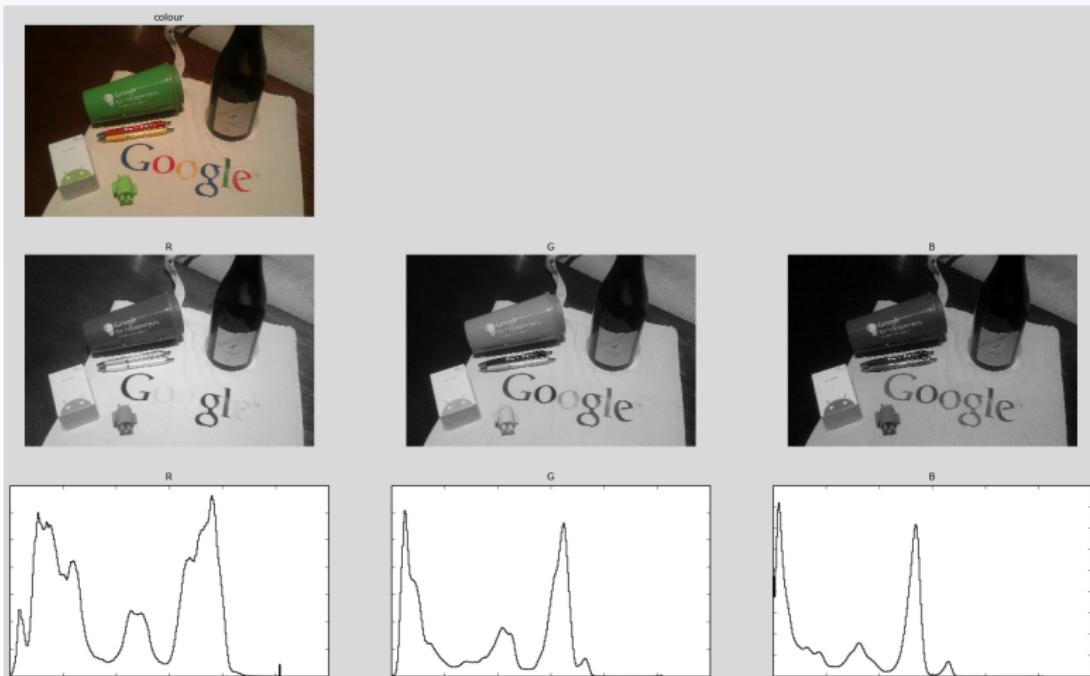
>>> fig, (ax0, ax1) = plt.subplots(nrows=2)
>>> ax0.imshow(img_g, cmap=plt.cm.Greys_r)
>>> ax0.set_title("Original image")
>>> ax0.axis("off")
>>> ax1.imshow(edges, cmap=plt.cm.Greys_r)
>>> ax1.set_title("Edges by Canny")
>>> ax1.axis("off")
>>> plt.show()
```



## Detected edges usually don't form closed boundaries

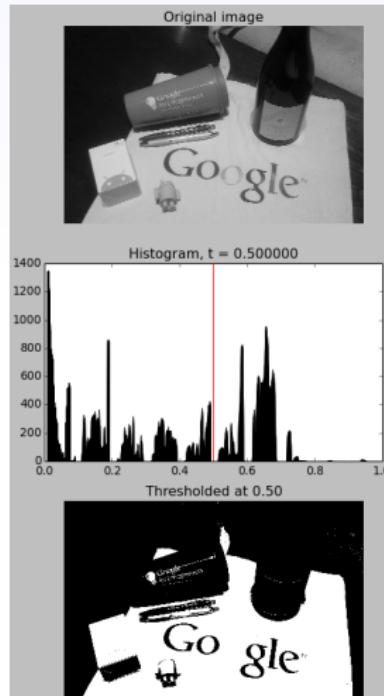
- **Problem:** If detected edges don't form into closed loops, they won't segment the image into regions.
- But we want to find image regions that correspond to objects.
- Hence, we consider **region-based image segmentation**.

## Histograms show the counts of light & dark grey pixels



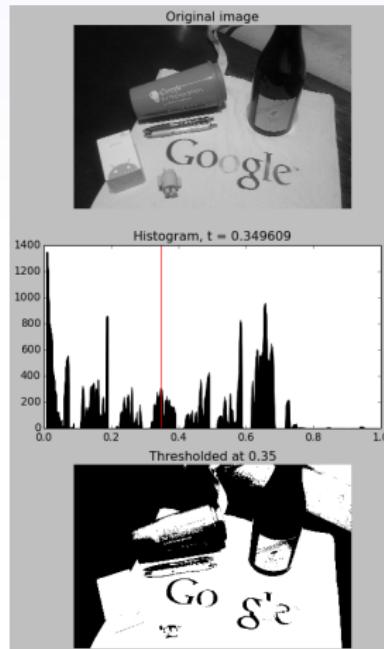
## You can “threshold” a greyscale image to binary

```
# We just take a guess: half the max.  
>>> t = img_g.max() / 2.0  
>>> thresh = img_g > t  
  
>>> fig, (ax0, ax1, ax2) =  
plt.subplots(nrows=3)  
>>> ax0.imshow(img_g, cmap=plt.cm.Greys_r)  
>>> ax0.set_title("Original image")  
>>> ax0.axis("off")  
>>> ax1.hist(img_g)  
>>> ax1.set_title("Histogram, t=%2f"%t)  
>>> ax1.axvline(t, color="r")  
>>> ax2.imshow(thresh, cmap=plt.cm.Greys_r)  
>>> ax2.set_title("Thresholded by  
guesstimate")  
>>> ax2.axis("off")  
>>> plt.show()
```



## Otsu's method chooses an “optimal” threshold automatically

```
>>> from skimage.filters import  
threshold_otsu  
  
>>> # Otsu determines "optimal" threshold.  
>>> t = threshold_otsu(img_g)  
>>> thresh = img_g > t  
  
>>> fig, (ax0, ax1, ax2) =  
plt.subplots(nrows=3)  
>>> ax0.imshow(img_g, cmap=plt.cm.Greys_r)  
>>> ax0.set_title("Original image")  
>>> ax0.axis("off")  
>>> ax1.hist(img_g)  
>>> ax1.set_title("Histogram, t=%2f"%t)  
>>> ax1.axvline(t, color="r")  
>>> ax2.imshow(thresh, cmap=plt.cm.Greys_r)  
>>> ax2.set_title("Thresholded by Otsu")  
>>> ax2.axis("off")
```



## Further reading (if you're interested)

- Watershed image segmentation

## Thanks for your time!

- If you have any more questions about image processing or computer vision, please ask!
- (whether in person today, or by email in the future)

## Some more short scripts to save you some typing...

- At <https://github.com/jboy/ncss2016/part3>