Image Processing With Python

Stephen Granade
@Sargent
stephen@granades.com

WTF is Image Processing?

- Modify images
  - Change colors
  - Adjust brightness and contrast
  - Remove parts of an image
- Extract information (signal) from images
  - Identify what's in an image (like faces)
  - Track objects as they move

Five Things

- Data structure to hold the image data
- Way to read in image data
- Ways to manipulate the image data

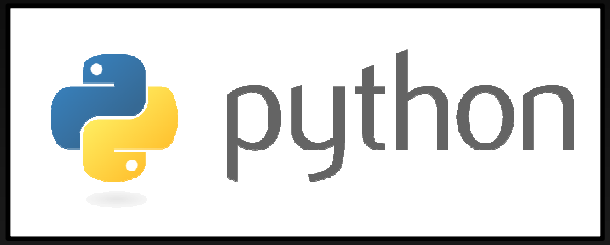
Three ~~Five~~ Things

- Data structure to hold the image data
- Way to read in image data
- Ways to manipulate the image data

Three ~~Five~~ Things

- Data structure to hold the image data
  - NumPy arrays
- Way to read in image data
  - SciPy library
- Ways to manipulate the image data
  - SciPy and scikit-image libraries

Numpy, SciPy, Scikits, Matplotlib



+

Get and Use the Libraries

- NumPy, SciPy and Matplotlib: http://www.scipy.org/
  - `import numpy as np`
  - `from scipy import ndimage`
  - `from scipy import misc`
  - `import matplotlib.pyplot as plt`
- Scikit-image: http://scikit-image.org/
  - `import skimage`
  - Lots of sub-libraries

NumPy Arrays: Ordered Collections of Homogeneous Data

- *Ordered*: You get ahold of items by their index number
- *Collection*: Holds other types of data, like ints or strings or lists or even other arrays
- *Homogeneous*: only holds data of the same type

Arrays are Sequences, But Slices Aren't Copies!

Lists

```
l1 = range(10)
l2 = l1[1:-1]
print('l1: %s\nl2: %s' % (l1, l2))


l1[4] = -1
print('l1: %s\nl2: %s' % (l1, l2))
```

Arrays

```
a1 = np.arange(10)
a2 = a1[1:-1]
print('a1: %s\na2: %s' % (a1, a2))


a1[4] = -1
print('a1: %s\na2: %s' % (a1, a2))
```

Arrays are Sequences, But Slices Aren't Copies!

Lists

```
l1 = range(10)
l2 = l1[1:-1]
print('l1: %s\nl2: %s' % (l1, l2))
  l1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  l2: [1, 2, 3, 4, 5, 6, 7, 8]
l1[4] = -1
print('l1: %s\nl2: %s' % (l1, l2))
  l1: [0, 1, 2, 3, -1, 5, 6, 7, 8, 9]
  l2: [1, 2, 3, 4, 5, 6, 7, 8]
```

Arrays

```
a1 = np.arange(10)
a2 = a1[1:-1]
print('a1: %s\na2: %s' % (a1, a2))
  a1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  a2: [1, 2, 3, 4, 5, 6, 7, 8]
a1[4] = -1
print('a1: %s\na2: %s' % (a1, a2))
  a1: [0, 1, 2, 3, -1, 5, 6, 7, 8, 9]
  a2: [1, 2, 3, -1, 5, 6, 7, 8]
```

Need a copy? Use `a1.copy()`

List Slices are New Objects But Array Slices Aren't

```
l1 = range(10)
l2 = l1[1:-1]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```
l1

```
[1, 2, 3, 4, 5, 6, 7, 8]
```
l2

```
a1 = np.arange(10)
a2 = a1[1:-1]
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```
a1

a2

Useful Array Functions

```
A = np.ones([3, 2])
```

Function	Result	Notes
`np.shape(A)` OR `A.shape`	`(3, 2)`	Tuple of array dimensions
`np.ndim(A)` OR `A.ndim`	`2`	Number of array dimensions
`np.size(A)` OR `A.size`	`6`	Flat length of the array (number of elements)
`A.T` OR `A.transpose()`	`2x3 array`	Transpose
`np.append(A, 2)`	`7x1 vector`	Flatten and append values
`np.append(A,[2,2])`	`8x1 vector`	Flatten and append values
`np.append(A,[[2,2]], axis=0)`	`4x2 array`	Append values along *axis* dimension
`np.concatenate( (A, A) )`	`6x2 array`	Given a tuple of arrays, concatenates them

Images Are (Usually) a 2D or 3D Array

Images Are (Usually) a 2D or 3D Array

- 2D array
  - Effectively grayscale image
  - Usually integers, but the same image functions that work on a uint8 grayscale image often work on 2D floating-point or Boolean arrays
- 3D array
  - 2D picture with a third dimension for things like color
  - But how is color defined?

Color Spaces Map Colors to Numbers

- RGB
  - Probably the one you've seen before
  - Red, green, and blue channels (values along the 3rd dimension)
  - Additive color mixing
- CMYK
  - Cyan, magenta, yellow, black
  - Subtractive color mixing
  - Mainly print – doesn't show up much in computer images

Color Spaces Map Colors to Numbers

- YCbCr
  - Luminance, Chroma blue, Chroma red
  - Shows up in JPEG & MPEG
- HSV
  - Hue, Saturation, Value
  - Hue: what color. Saturation: how colorful. Value: kind of how light it is, but not really.
  - Cylindrical coordinate representation of RGB

```
#####
# SIMPLE IMAGE LOADING AND DISPLAYING

import numpy as np
from scipy import ndimage
from scipy import misc
import matplotlib.pyplot as plt

img = misc.imread('Schroedinger.jpg')

plt.imshow(img)

# Why's it all rainbow like that? Because SCIENCE. We can fix that, though
plt.close('all')
plt.imshow(img, cmap='gray')
```

Read In Images Using SciPy

- A lot of the `scipy.misc` and `scipy.ndimage` functions are using the Python Imaging Library behind the scenes, and the SciPy docs about them are sketchy
- `scipy.misc.imread()` figures out an image file's type by the file contents
- `scipy.misc.imsave()` uses the filename extension to determine the file type
  - Supported types: bmp, gif, jpg (or jpeg), png, tiff, xbm
  - For a full list, see http://effbot.org/imagingbook/formats.htm

Convert an Image to Python Imaging Library Format for All PIL Options

- `scipy.misc.imsave()` won't let you set options like JPEG quality, but PIL will
- Solution: convert your ndarray image to a PIL image and then use its save function
  - ```
    pil_img = misc.toimage(img)
    pil_img.save('filename.jpg',
          quality=30)
    ```
- http://effbot.org/imagingbook/formats.htm has information on all save options
- Note: there's both `misc.toimage()` and `misc.fromimage()` for moving between NumPy and PIL
    ```

```
#####
# CROPPING, SCALING, ROTATING, FLIPPING

# Crop using standard array slicing notation
imgy, imgx = img.shape
cropped_img = img[0:imgy //2, :]

# To scale, use the imresize() function from scipy.misc
resized_img = misc.imresize(img, 0.30)
plt.close('all')
compare_images([img, resized_img], title='Float-Resized Image',
               subtitles=['Original', 'Resized'])
# imresize takes the size as a float (fraction), integer (percent), or
# tuple (final size)...or so it says. As far as I can tell, integer
# scaling is broken
resized_img = misc.imresize(img, 10)
# Tuples work, though
resized_img = misc.imresize(img, (img.shape[0] // 2, img.shape[1]))
plt.close('all')
compare_images([img, resized_img], title='Tuple-Resized Image')

# To rotate, use the rotate() function from scipy.ndimage
rotated_img = ndimage.rotate(img, 30)
plt.close('all')
compare_images([img, rotated_img], title='Rotated Image')
# By default rotate will make the array big enough that nothing gets cut
# off. You can change that, of course
cropped_rotated_img = ndimage.rotate(img, 30, reshape=False)

# To flip, use the standard NumPy functions flipud() and fliprl()
flipped_img = np.flipud(img)
```

1

2

3

4

1

2

3

4

Local Filtering Determines a Pixel's Values By Looking At Its Neighbors

- The basic idea: replace a pixel's value by a function involving neighbors' values
- Top right: a 3x3 Gaussian blurring filter. Each pixel's value is partially replaced by the value of its neighbors
- Bottom right: Sobel horizontal filter for finding horizontal lines
- Local filters let you blur, sharpen, detect edges, and more

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

1/8	1/4	1/8
0	0	0
-1/8	-1/4	-1/8

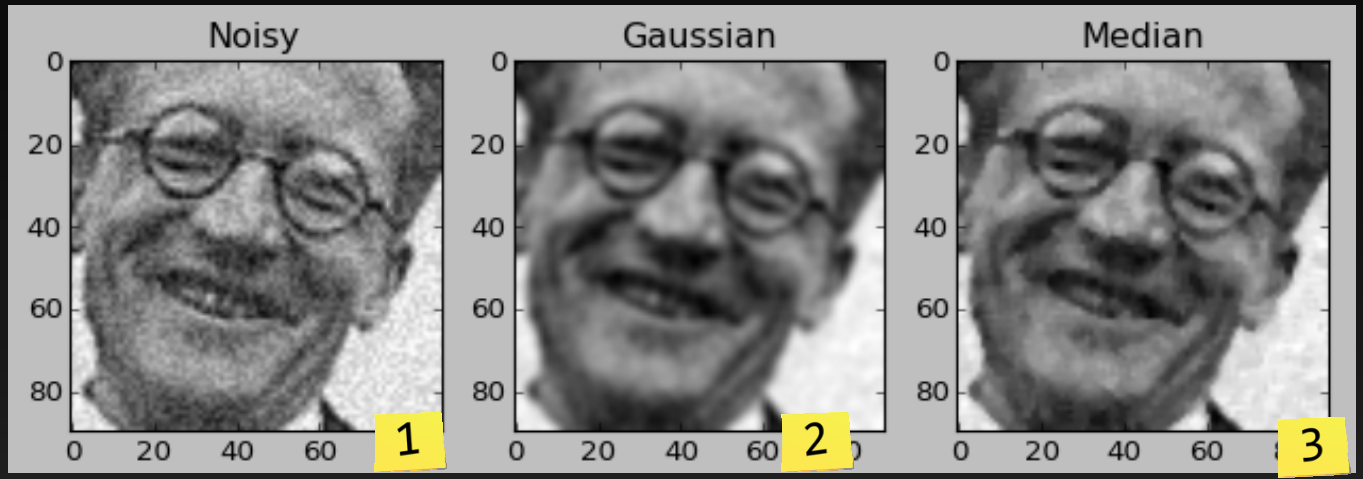
1

2

```python
# scipy.ndimage includes several common filters. For example, Gaussian
# filters, which soften and blur images, are in scipy.ndimage
blurred_img = ndimage.gaussian_filter(img, sigma=1)

# The larger the Gaussian's sigma, the more it blurs
more_blurred_img = ndimage.gaussian_filter(img, sigma=3)
```

1

2




```
# What if you have a noisy image?
cropped_img = img[50:140, 90:180]
noisy_img = cropped_img + (cropped_img.std()*np.random.random(cropped_img.shape) -
        (cropped_img.std()/2)*np.ones(cropped_img.shape))

# You can use a Gaussian filter to de-noise the image
denoised_img = ndimage.gaussian_filter(noisy_img, sigma=1)

# Or you can use a median filter to better preserve edges
median_denoised_img = ndimage.median_filter(noisy_img, 3)
plt.close('all')
compare_images([noisy_img, denoised_img, median_denoised_img],
               title="Gaussian vs Median Denoising",
               subtitles=['Noisy', 'Gaussian', 'Median'])
```

1

2

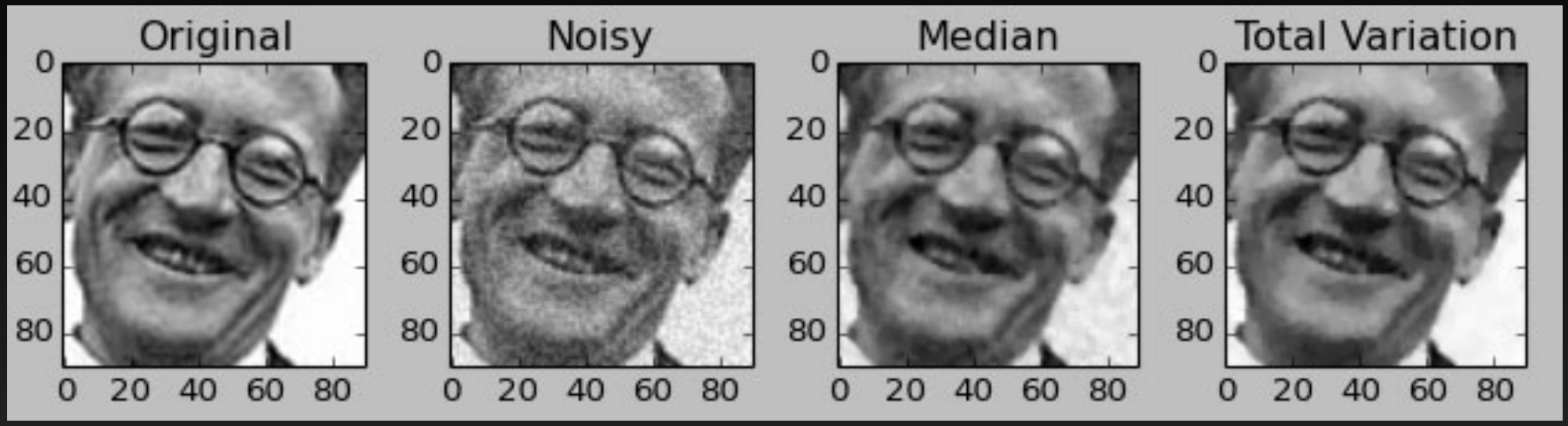
3

SciKit-Image Adds to SciPy

```
from scikit import…
```

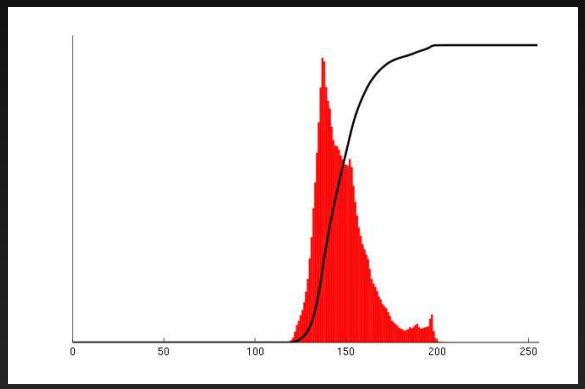
Subpackage	Description
color	Color space conversion (RGB, YCbCr, etc.)
data	Test images and example data
draw	Drawing primitives – lines, text, etc. – for NumPy arrays
exposure	Image intensity adjustment, like histogram equalization
feature	Feature detection and extraction
filter	Sharpening, edge-finding, rank filters, thresholding
graph	Graphi-theoretic operations like finding the shortest path
io	Reading, saving, and displaying images
measure	Measurement of image properties like similarity and contours
morphology	Morphological operations like opening or skeletonization
restoration	Restoration algorithms like denoising or deconvolution algorithms
segmentation	Partitioning an image into multiple regions
transform	Geometric and other transforms like rotation, Radon transform

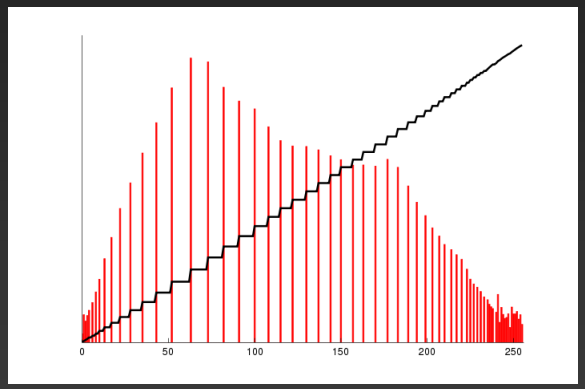
```
#####
# LOCAL FILTERING

# skimage has a lot of local filters available, like Sobel
import skimage.filter

img = misc.imread('Schroedinger.jpg')
schro_vsobel = skimage.filter.vsobel(img)
```




```
# Remember my noise reduction example earlier? skimage has better routines
cropped_img = img[50:140, 90:180]
noisy_img = cropped_img + (
        (cropped_img.std()*np.random.random(cropped_img.shape) -
        (cropped_img.std()/2)*np.ones(cropped_img.shape)))
median_denoised_img = ndimage.median_filter(noisy_img, 3)
total_var_denoised_img = skimage.filter.denoise_tv_chambolle(noisy_img,
                                                             weight=30)
```

Histogram Equalization Improves Contrast



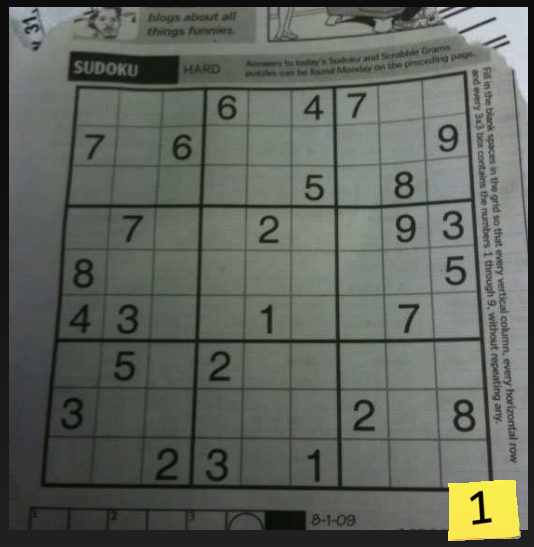


Hawkes Bay image courtesy Phillip Capper (PhillipC)

```
#####
# ADJUSTING EXPOSURE

import skimage.exposure

# Using skimage, we can perform contrast enhancement automatically by
# equalizing the picture's histogram. The presentation has more information
# on histogram equalization

sudoku = ndimage.imread('sudoku.jpg', flatten=True)
# "flatten=True" reads the picture in as grayscale and also converts the
# image to floating-point numbers.

# In a lot of cases, skimage wants image data to be floating point numbers
# scaled to the range [-1, 1].
# Because of the flattening, sudoku is a floating-point number, but it's
# scaled to the range [0, 255]. Adjust it to a [-1, 1] scale.
sudoku_scaled = (sudoku - 127.5)/256
sudoku_equalized = skimage.exposure.equalize_hist(sudoku_scaled)
```

1

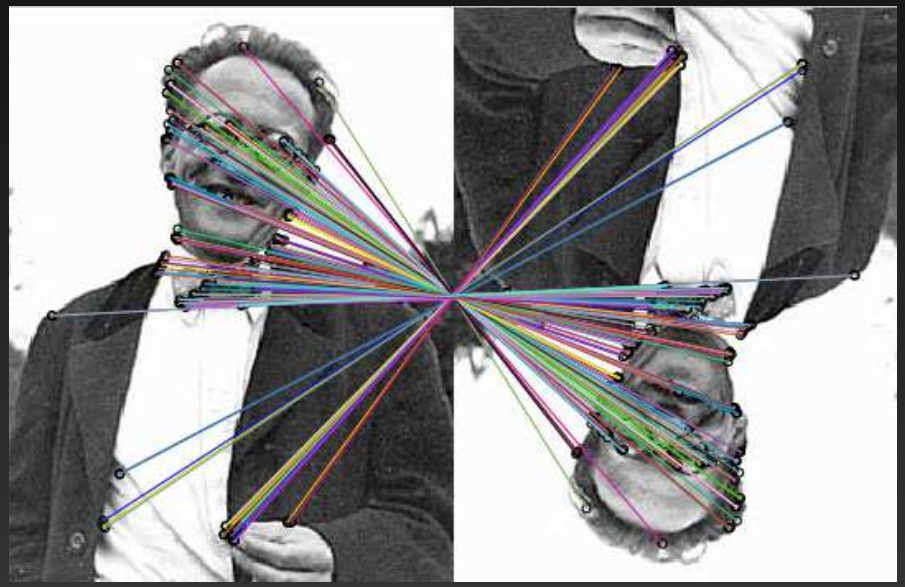
2

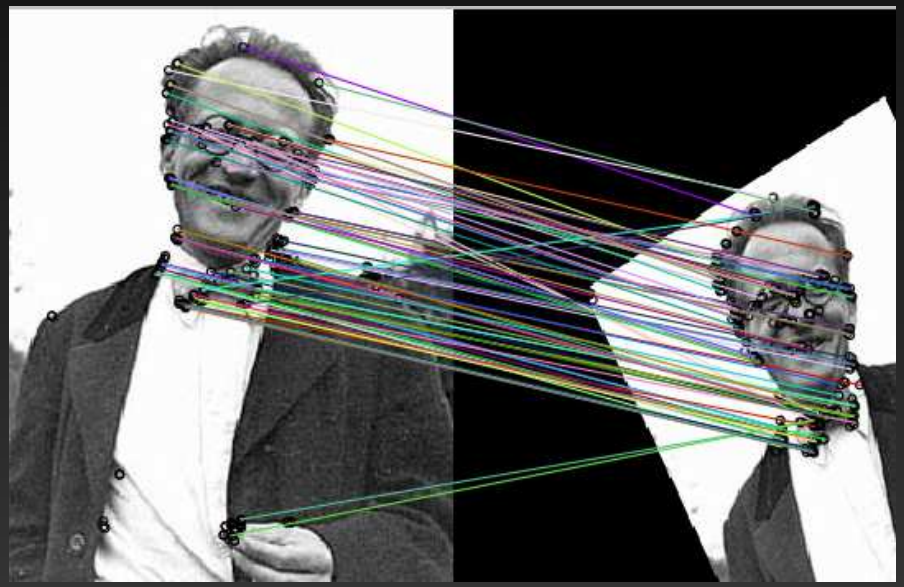


1

2

Matching Images





Example: Extract the Square Glyphs

How to draw an Owl.

"A fun and creative guide for beginners"

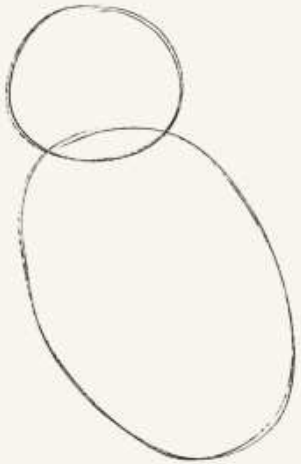
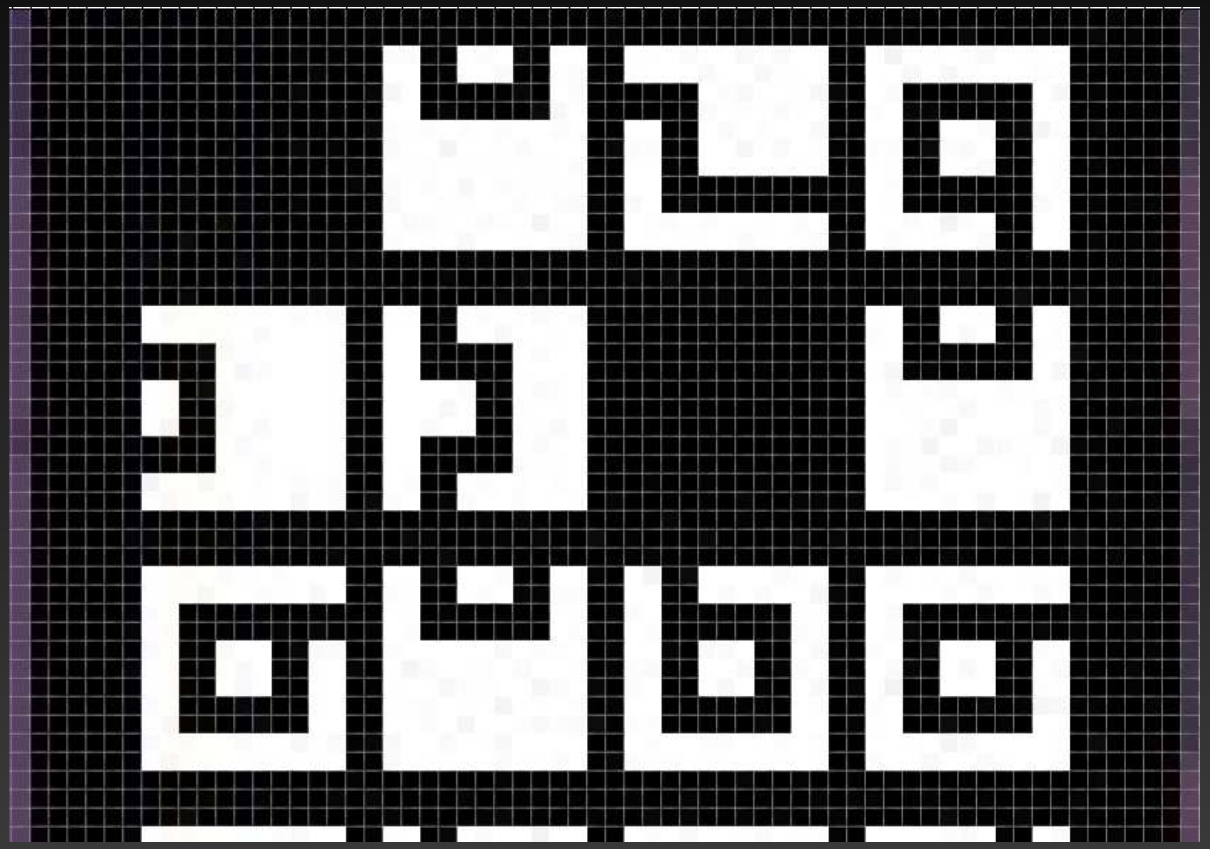


Fig 1. Draw two circles

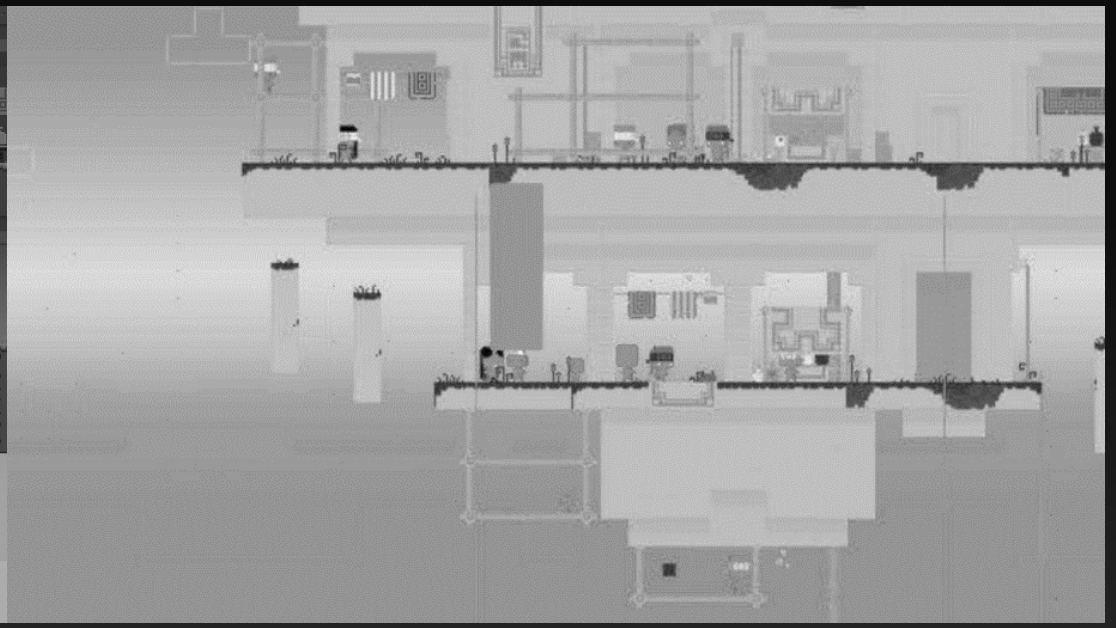
Fig 2. Draw the rest of the damn Owl

Example: Extract the Square Glyphs

The Glyphs are 11x11 B&W Squares

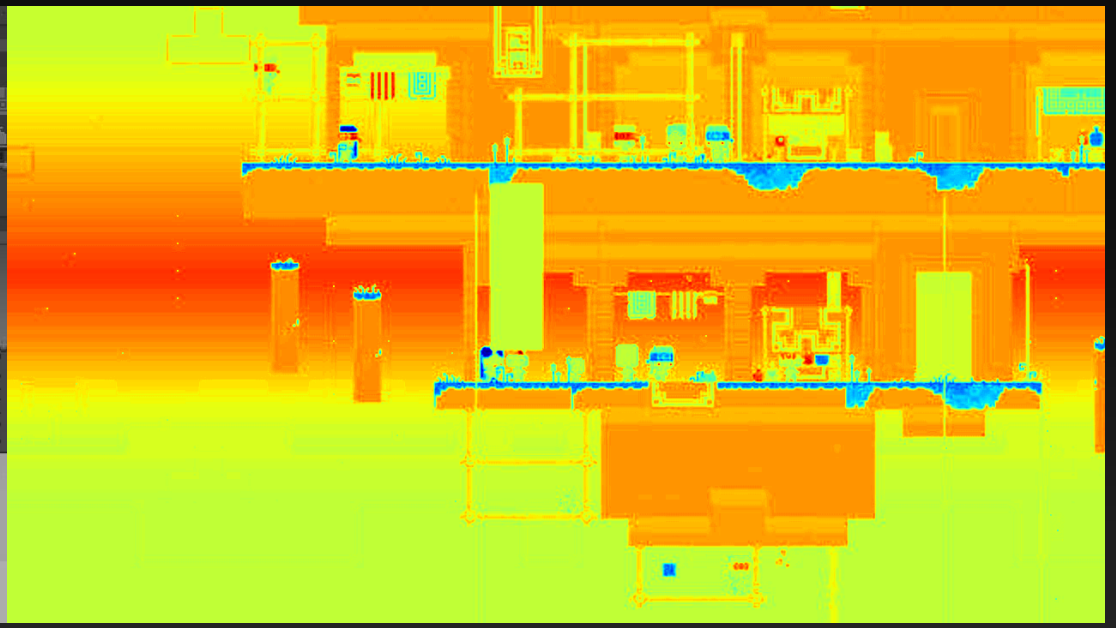


Luminance
(Brightness)



Chrominance
(Color)

```
FILENAME = 'fez-image.jpg'

glyphimg = misc.imread(FILENAME)

## Read in the image in YCbCr (luminance/chroma blue/chroma red) color space
glyphimg_ycbcr = ndimage.imread(FILENAME, mode='YCbCr')
lum = glyphimg_ycbcr[:,:,0]
chrom = glyphimg_ycbcr[:,:,1] / 2 + glyphimg_ycbcr[:,:,2] / 2
```

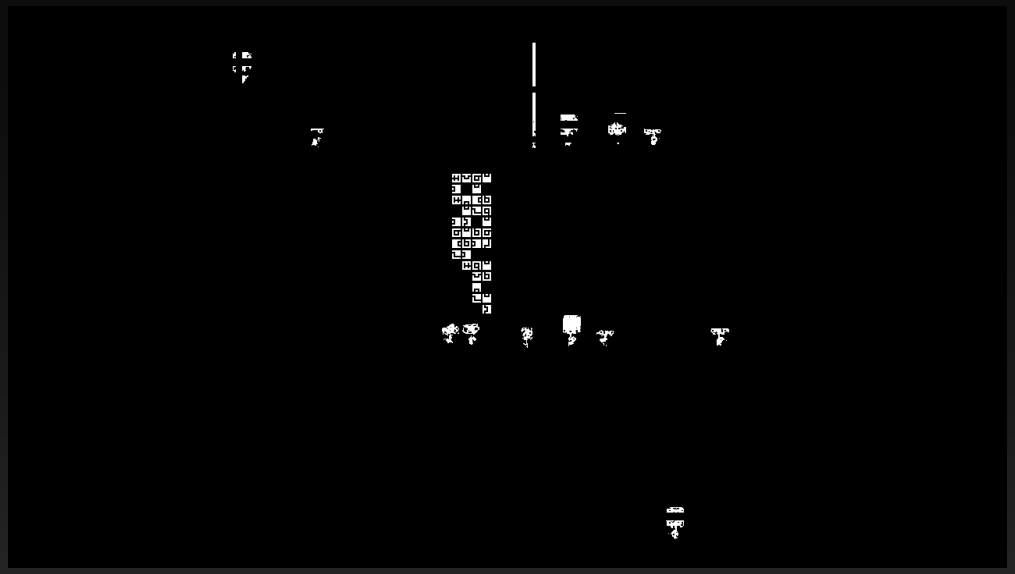


Luminance
(Brightness)

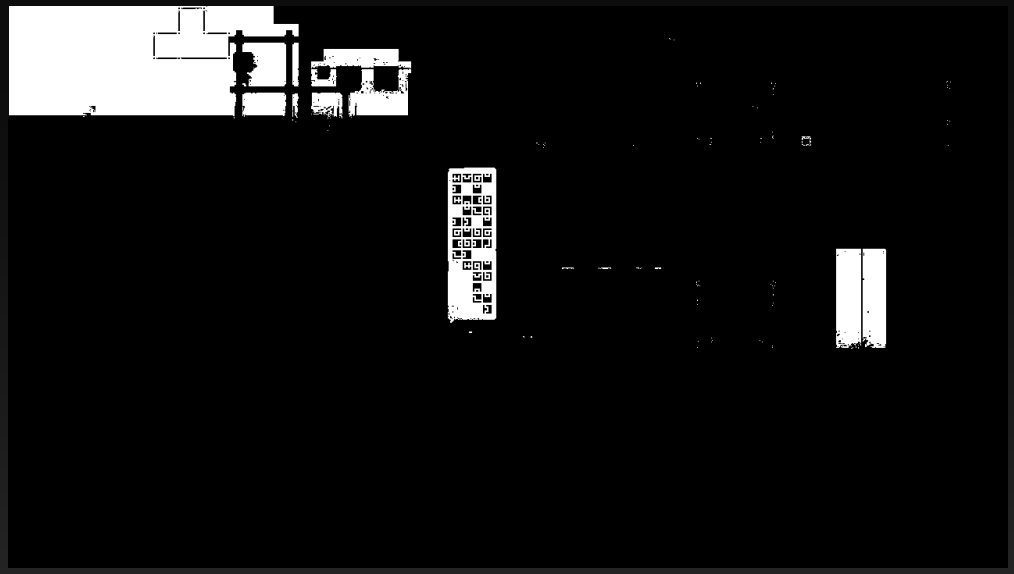
Chrominance
(Color)

```
FILENAME = 'fez-image.jpg'

glyphimg = misc.imread(FILENAME)

## Read in the image in YCbCr (luminance/chroma blue/chroma red) color space
glyphimg_ycbcr = ndimage.imread(FILENAME, mode='YCbCr')
lum = glyphimg_ycbcr[:,:,0]
chrom = glyphimg_ycbcr[:,:,1] / 2 + glyphimg_ycbcr[:,:,2] / 2
```

```
# The glyph graphics are an 11x11 square of just black and white surrounded
# by a black square. That means we're looking for areas with both high luminance
# (brightness) and low chromaticity (color).

lum_mask = lum > .94*np.max(lum)
# Low color corresponds to a chrominance of around 128 (halfway between 0 & 255)
chrom_mask = np.logical_and(chrom >= 126, chrom <= 136)

# Let's identify the part of the image we care about
image_mask = np.logical_and(lum_mask, chrom_mask)
```

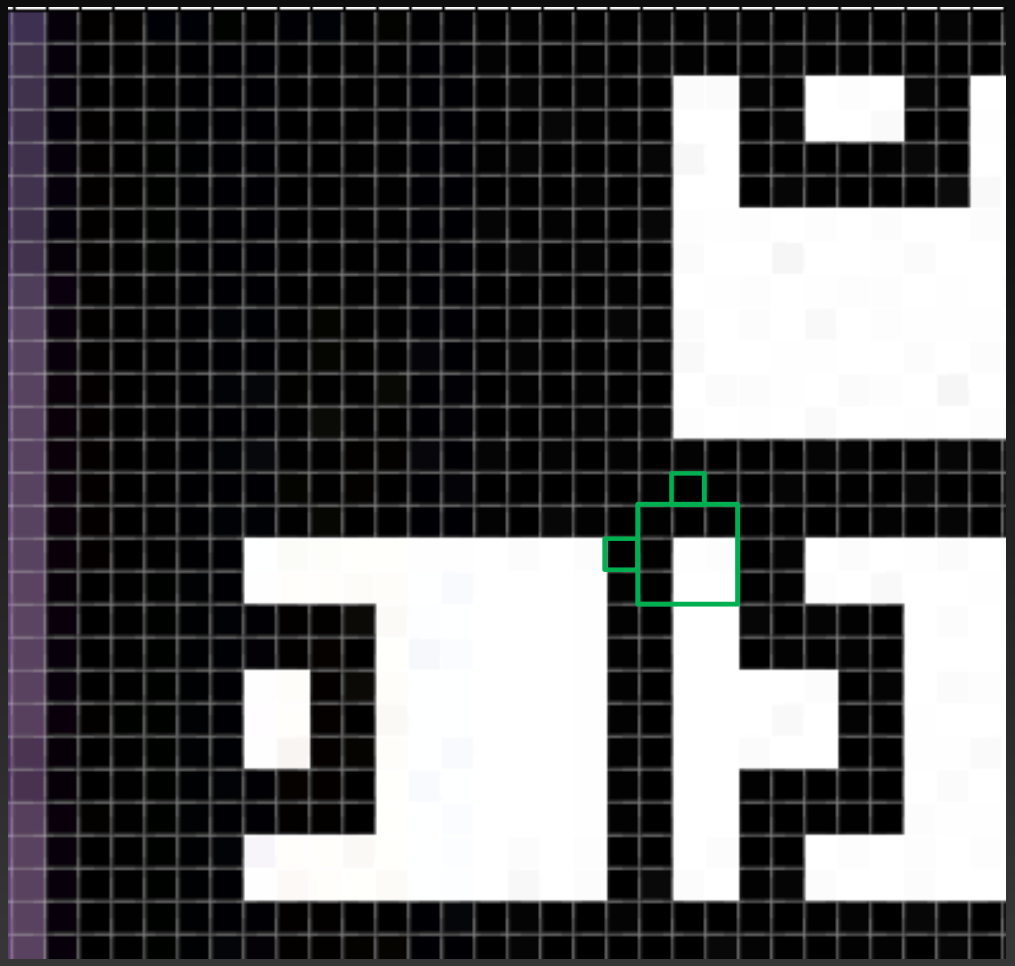


Possible Glyph Pixels
(Bright, No Color)

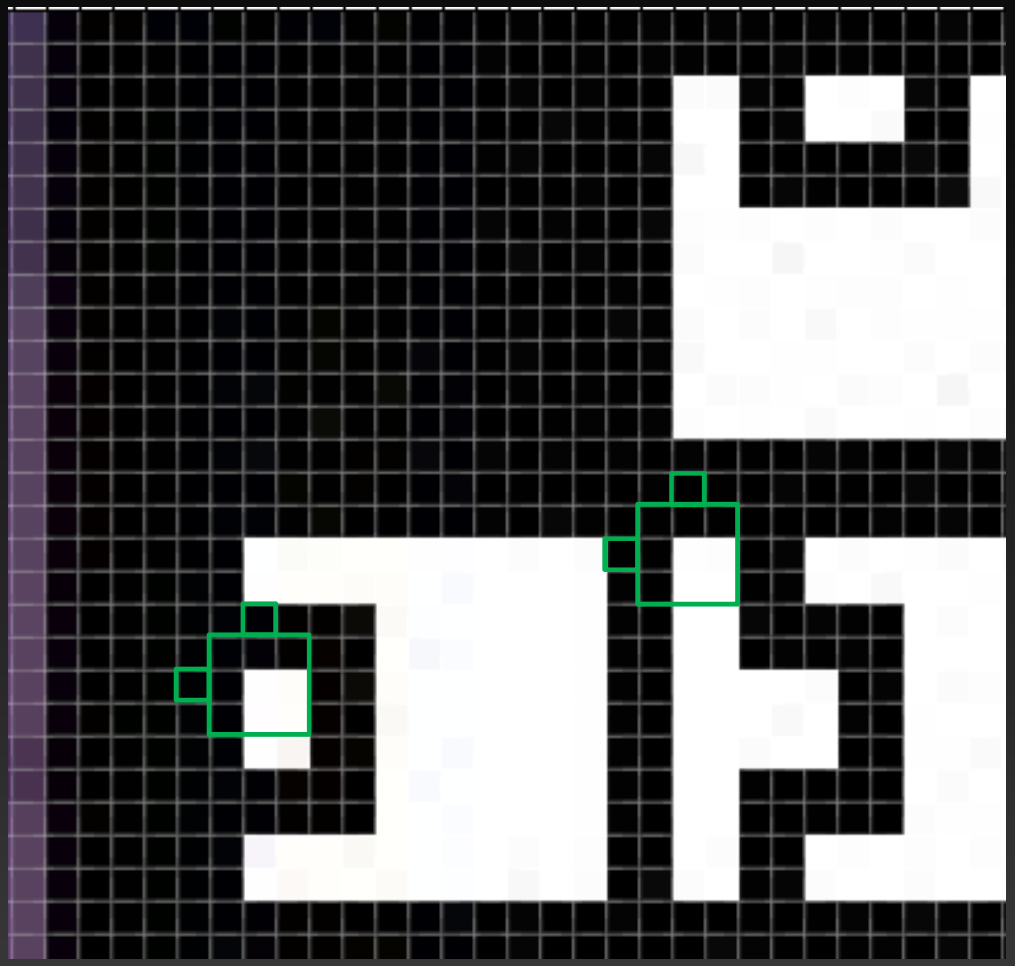


Possible Border Pixels
(Dark, No Color)

```
# Find the border around the glyphs (low luminance, low chromaticity)
border_mask = np.logical_and(lum < 20, chrom_mask)
```

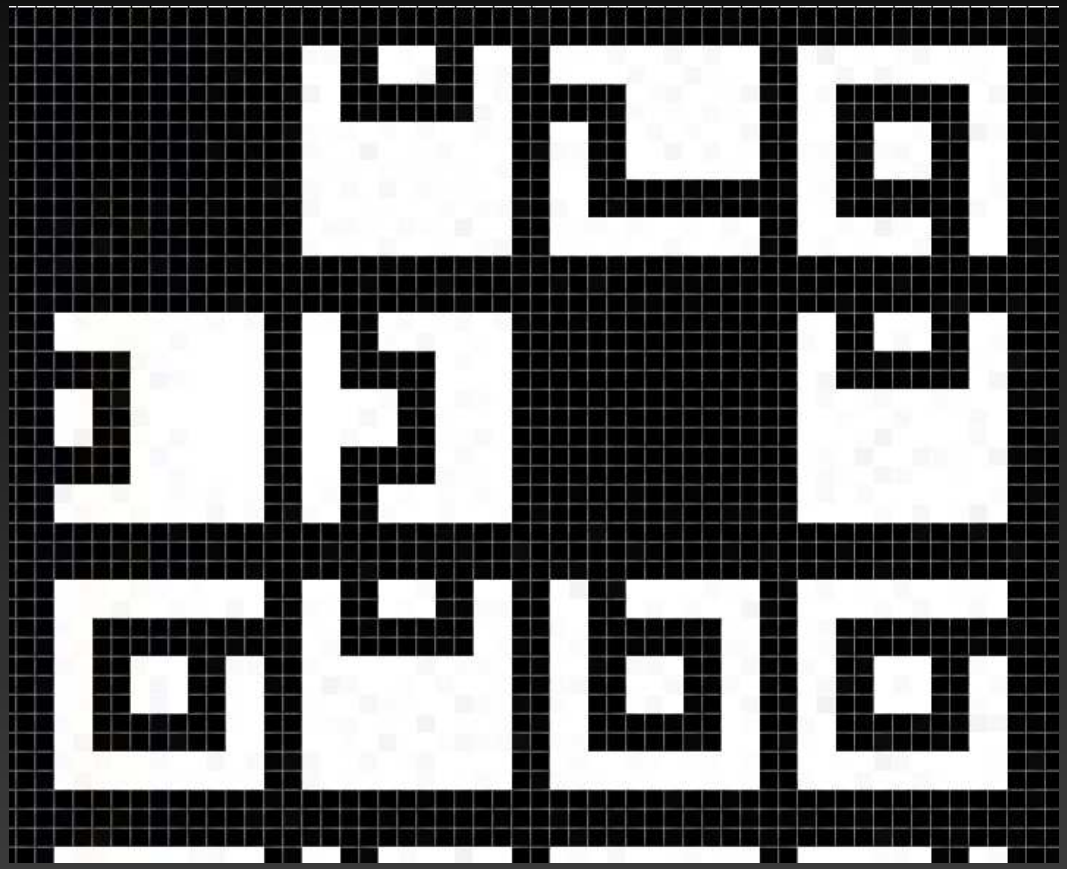


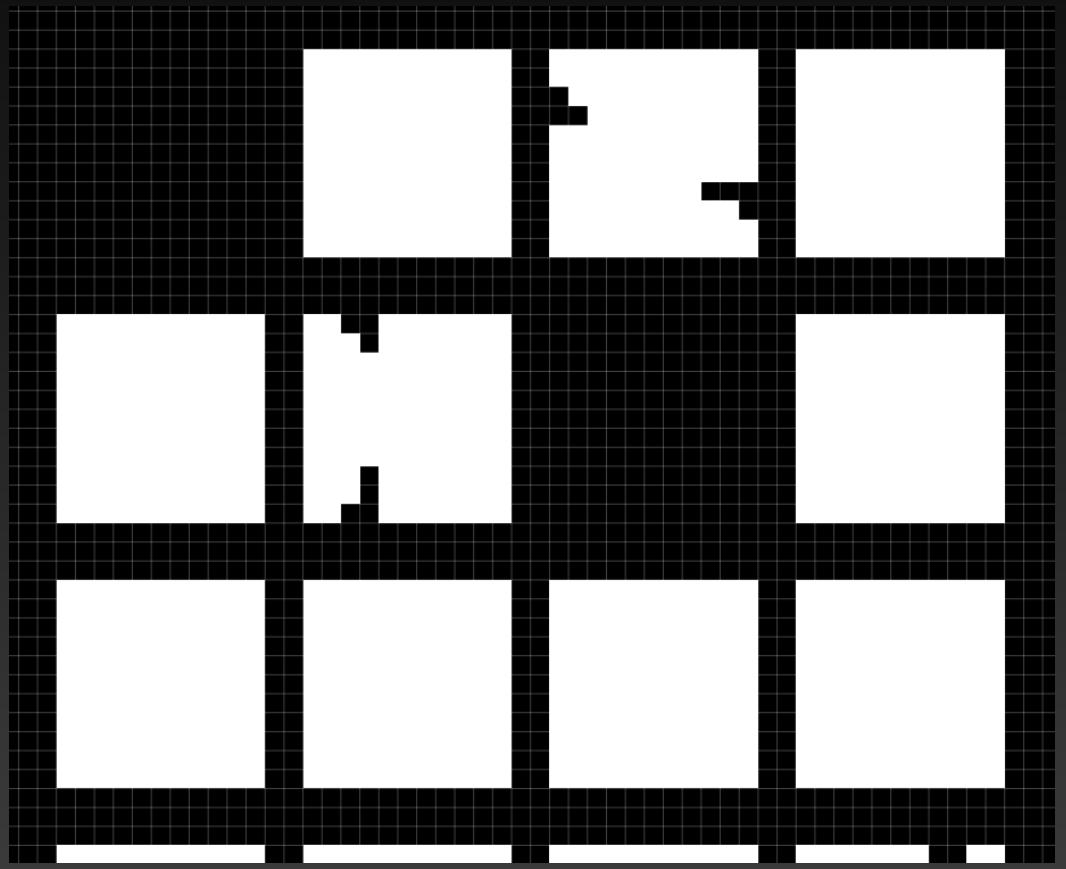
```python
# Routines to find all of the top left corners in an image
def mask_is(mask, i, j):
    """Return the true/false value of a Boolean mask image at index [i, j].

    If the [i, j] index is out of bounds, returns True.
    """
    try:
        return mask[i, j]
    except IndexError:
        return True

def is_top_left_corner(image, border, i, j):
    """Determine if image[i, j] is a top left corner that we're looking for.

    Args
    ----
    image : array
        A Boolean array indicating where the image pixels are.
    border : array
        A Boolean array indicating where the border around the image is.

    Returns
    -------
    True if image[i, j] is a top left corner, False otherwise.
    """
    return (image[i, j] &
        mask_is(image,  i  , j+1) &
        mask_is(image,  i+1, j+1) &
        mask_is(image,  i+1, j  ) &
        mask_is(border, i+1, j-1) &
        mask_is(border, i  , j-1) &
        mask_is(border, i-1, j-1) &
        mask_is(border, i-1, j) &
        mask_is(border, i-1, j+1) &
        (not mask_is(image, i-2, j)) &
        (not mask_is(image, i, j-2)))
```

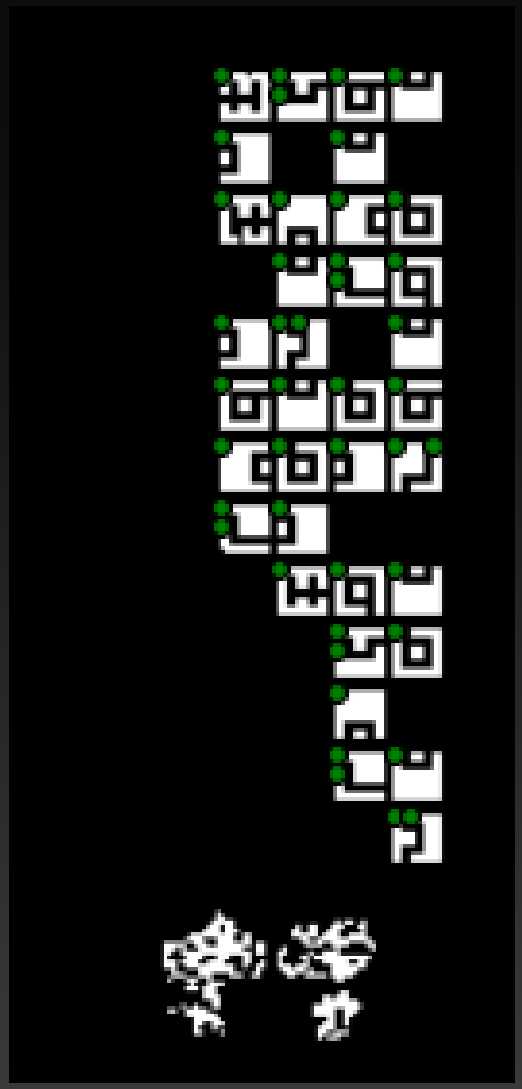


```python
# Routines to find all of the top left corners in an image
def mask_is(mask, i, j):
    """Return the true/false value of a Boolean mask image at index [i, j].

    If the [i, j] index is out of bounds, returns True.
    """
    try:
        return mask[i, j]
    except IndexError:
        return True

def is_top_left_corner(image, border, i, j):
    """Determine if image[i, j] is a top left corner that we're looking for.

    Args
    ----
    image : array
        A Boolean array indicating where the image pixels are.
    border : array
        A Boolean array indicating where the border around the image is.

    Returns
    -------
    True if image[i, j] is a top left corner, False otherwise.
    """
    return (image[i, j] &
        mask_is(image,  i  , j+1) &
        mask_is(image,  i+1, j+1) &
        mask_is(image,  i+1, j  ) &
        mask_is(border, i+1, j-1) &
        mask_is(border, i  , j-1) &
        mask_is(border, i-1, j-1) &
        mask_is(border, i-1, j) &
        mask_is(border, i-1, j+1) &
        (not mask_is(image, i-2, j)) &
        (not mask_is(image, i, j-2)))
```

Convex Hull Stretches a Rubber Band Over Objects







```
# Find the likely columns by counting occurrences and find the most-common
# spacing between adjacent columns
# (I'm using a Counter: a dict that stores elements as keys and the number of
# times that element occurs as the values)
columns = Counter(top_left_corners[1])
avg_column_count = np.mean(columns.values())
likely_columns = sorted([k for k in columns
        if columns[k] > avg_column_count])
col_delta, _ = Counter([i - j for i, j
        in zip(likely_columns[1:], likely_columns[:-1])]).most_common(1)[0]
# I happen to know that the spacing between rows is 1 more than that between
# columns, & there aren't enough columns to make the statistical analysis
# I did above work for rows.
# (I also know that the column spacing is the glyph size (11 pixels) + 2,
# but I wanted to show how I'd do this kind of analysis)
row_delta = col_delta + 1
```

Where to Go From Here

- Look at the sample code from this presentation
  - https://github.com/sgranade/python-image-processing-intro
- Try a more optimized image processing library
  - OpenCV: http://opencv.org/
- Try a simpler image processing library
  - SimpleCV: http://simplecv.org/
- Look at other tutorials
  - http://blog.yhathq.com/posts/image-processing-with-scikit-image.html (scikit-image)
  - http://www.pyimagesearch.com/ (OpenCV)
  - http://docs.opencv.org/trunk/doc/py_tutorials/py_tutorials.html (OpenCV)

Image Processing With Python

Stephen Granade
@Sargent
stephen@granades.com