image analysis

- we've already dealt a bit with images using numpy and matplotlib
- 3 main packages specifically for image manipulation and analysis in python:
    i. scikit image (skimage)
        - skimage image gallery: http://scikit-image.org/docs/dev/auto_examples
    ii. scipy.ndimage
    iii. OpenCV - more advanced algorithms (machine learning), multiple languages and versions, tricky to install
        - OpenCV Python tutorials
- recent local "SuperPython" talk by Joe Donovan for lots of nice examples:
    - https://github.com/superpythontalks/image_analysis/blob/master/image%20processing.ipynb

review: display numpy arrays as images

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io # functions for image file input/output
faceg = io.imread('face_gray.png') # 2D array, dtype is uint8
faceg.shape # 782 x 782, fairly big
f, ax = plt.subplots()
im = ax.imshow(faceg) # this uses default colormap 'viridis'
im.get_cmap().name
```

- why is matplotlib displaying colour, when we know the image is grayscale?
- 2D arrays represent only "single channel" images, each pixel can only represent one thing: luminance
- our screens are colour, so MPL puts single channel images through a color map, default is `'viridis'`, but we can choose `'gray'` instead:

```
f.colorbar(im) # add color bar to figure
plt.colormaps() # list all available colormaps
im.set_cmap('gray') # use grayscale instead
```

- 3 general types of colour maps:
    - sequential - data going ranging from low to high value,
    - diverging - data that deviates around a center value like 0
    - categorical - discrete colours for particular ranges of data
- see colormap plots and explanations at: http://matplotlib.org/users/colormaps.html
- can also directly specify desired colormap in imshow():

```
f, ax = plt.subplots()
im = ax.imshow(faceg, cmap='gray')
f.colorbar(im)
```

- how to turn off the axes x and y ticks? set them to empty lists:
  - `ax.set_xticks([]), ax.set_yticks([])`
- how to rotate arrays, and therefore images?

```
f, ax = plt.subplots()
face90 = np.rot90(faceg) # 90 deg counterclockwise
ax.imshow(face90, cmap='gray')
io.imsave('face90.png', face90) # save it back to a new file
```

- can use `np.flipud()` and `np.fliplr()` to flip arrays vertically or horizontally
  - since the image is horizontally symmetric, make a big black mark on one side of it before trying to flip it horizontally:
  - `faceg[:100, :100] = 0`
  - what's another way to flip vertically and horizontally, using array slicing?
- for specific (non 90 deg) angles of rotation:

```
from scipy import ndimage
face45 = ndimage.rotate(faceg, 45) # 45 deg counterclockwise
face45.shape # shape is bigger now: (1106, 1106)
ax.imshow(face45, cmap='gray')
```

subsampling & resizing

- subsample an image using array slicing:

```
f, ax = plt.subplots()
lowres = faceg[::10, ::10] # every 10th pixel in both dimensions
lowres.shape # 79x79, 1/10 the size in both dimensions
ax.imshow(lowres, cmap='gray')
```

- `imshow` can *display* images using different kinds of interpolation, default is no interpolation

```
f, ax = plt.subplots()
ax.imshow(lowres, cmap='gray', interpolation='gaussian')
```

- `'bilinear', 'bicubic', 'gaussian', 'spline16'` are common, see list of all interpolation methods
- this is just for display, doesn't modify the array. Let's create a new array by interpolating the existing one, with more control over the smoothing:

```
from skimage import filters
lowresgauss = filters.gaussian(lowres, sigma=2) # sigma in number of pixels
lowresgauss.shape # it's still a small 79x79 array...
```

```
f, ax = plt.subplots()
ax.imshow(lowresgauss, cmap='gray') # ... but its pixels have been smoothed
```

- resizing an image, either up or down:

```
biglowresgauss = ndimage.zoom(lowresgauss, 10) # resize up to original size
biglowresgauss.shape # 790 x 790
f, ax = plt.subplots()
ax.imshow(biglowresgauss, cmap='gray') # ... but its pixels have been smoothed
```

- to plot histogram of pixel values, have to ravel (flatten) to 1D array firs

```
from skimage import data # example data
moon = data.moon() # lunar surface
f, ax = plt.subplots(1, 2)
ax[0].imshow(moon, cmap='gray', vmin=0, vmax=255) # vmin/vmax turn off auto normalize
ax[1].hist(moon.ravel(), bins=np.arange(256+1)) # +1 adds right bin edge
```

- how to change image brightness?
  - increase the average pixel value by adding a constant
  - watch out for excessive saturation, or integer overflow!
- how to change image contrast?
  - scale pixel values to span the full possible range of 0 to 255
  - do this by shifting left via subtraction, then multiplying by a constant to get full range:
  - general equation: `output = (input - min) / (max - min) * 255`
  - `moon = (np.float64(data.moon()) - 75) / (150-75) * 255`
  - watch out for excessive saturation, or integer overflow!
  - more sophisticated methods: `skimage.exposure` module
- colour:
  - if you have image data in a 2D array, the values for each pixel can only represent luminance, there's no colour information
  - can map the luminance to some set of colours using a colour map, but you can't independently represent luminance and colour with only a single value per pixel
  - how can colour be represented in an array? as red, green and blue channels (RGB)
  - you could use 3 separate 2D arrays to represent RGB for an image, but normally these are stacked into a single 3D array: `nrows x ncols x 3`

```
facec = io.imread('face.png')
facec.shape # 3D array, last dimension represents colour
f, ax = plt.subplots()
im = ax.imshow(facec) # no sense using a colormap when image already has colour
```

Exercise 1

1. Load the `ohki2005.png` image into a numpy array named `ohki` and display it in a matplotlib figure. What shape does the data have? Is it colour?

2. Rotate the image data 90 degrees *clockwise* and display it in a figure. Add a colorbar. Try plotting with different kinds of colormaps (`im.set_cmap()` is the most convenient way):

   i. sequential: e.g. `'gray', 'viridis'`
   ii. Diverging: e.g. `'Spectral', 'coolwarm'`
   iii. Categorical: e.g. `'Dark2', 'Pastel1'`

   See https://matplotlib.org/users/colormaps.html

3. Plot a histogram of the pixel values.

4. The image is a bit dark, and the contrast could be better.

   i. Do something to the pixel data to increase the brightness of the image. Name this new array `ohki2`. Try not to saturate the image too much. Plot the new image and its histogram, and compare to the original.
   ii. Increase the contrast of the image, and name this new array `ohki3`. To make this easier, you can use:

```
from skimage import exposure
ohki3 = exposure.rescale_intensity(ohki, (newminval, newmaxval))
```

5. Let's say we want to focus on the middle section of the image. Slice out the middle third of the image (vertically and horizontally), so you end up with an array with about 1/9th as many pixels as the original. Name this subset array `subset`. Display it in a figure.

6. Plot a histogram of this subset in another figure. Is it fairly different from the histogram of the original full image?

7. Scale up the subset by a factor of 2 using `scipy.ndimage.zoom` and display that as well. Should this scaled up version have a different histogram from the non-scaled version? Check it.

modifying colours

- scikit-image has several demo images built in:

```
from skimage import data
immun = data.immunohistochemistry()
immun.shape # notice it's a 3D array, last dimension represents colour
f, ax = plt.subplots()
ax.imshow(immun)
```

- we can modify the R, G and B channels separately. Let's remove all the red from the image. How can we do this by manipulating the array data?
  - set the red channel to 0, and then the green channel:

```
temp = immun.copy()
temp[:, :, 0] = 0 # no more red
ax.imshow(temp) # teal remains
temp[:, :, 1] = 0 # no more green
ax.imshow(temp) # only blue remains
```

- color conversion functions in `skimage.color`
  - convert RGB image to grayscale:

```
from skimage import color
immung = color.rgb2gray(immun)
immung.shape # now it's only 2D
ax.imshow(immun, cmap='gray')
```

- images can also have a 4th channel: alpha, aka transparency. The alpha channel is a transparency mask for the image. 0 is fully transparent, 255 is fully opaque. So pixels you don't want painted are set to 0:

```
facea = io.imread('face_alpha.png')
facea.shape # gives (782, 782, 4), i.e. RGBA
alpha = facea[:, :, 3] # prints last (4th) hypercolumn
f, ax = plt.subplots()
ax.imshow(alpha, cmap='gray')
bg = np.zeros_like(facea) # init a background of the same shape
bg[:, :] = 255, 0, 0, 255 # set to red, and fully opaque
```

- now display background, and then image with transparency to see the effect:

```
f, ax = plt.subplots()
ax.imshow(bg) # first show background
ax.imshow(facea) # see red where foreground image alpha is low
```

- can also blend array with a red background with `color.rgba2rgb(facea, [1, 0, 0])`

edge detection & segmentation

- edge detection:

```
ohki = io.imread('ohki2005.png')
f, ax = plt.subplots()
ax.imshow(ohki, cmap='gray')
f, ax = plt.subplots()
ax.hist(ohki.ravel(), bins=np.arange(256))
edges = filters.sobel(ohki)
f, ax = plt.subplots()
ax.imshow(edges, cmap='gray')
```

- edge-based segmentation:

```
from skimage.feature import canny
edges = canny(ohki, sigma=1.5)
from scipy import ndimage
mask = ndimage.binary_fill_holes(edges)
```

```
f, ax = plt.subplots()
ax.imshow(mask, cmap='gray')
```

- watershed-based segmentation:
  - see exercise 2

movies

- load movies using pyav:
- to install pyav in anacoda: `conda install av -c conda-forge` at the command line
- in linux/mac, if you don't have anaconda: `pip install av` might work

```
import av
v = av.open('movie.avi')
mv = [] # init empty list to fill with frames
for frame in v.decode(video=0): # get the first video stream in file
    mv.append(np.asarray(frame.to_image()))
mv = np.asarray(mv)
mv.shape # 4D array: nframes x nrows x ncols x 3 colour channels
f, ax = plt.subplots()
ax.imshow(mv[0]) # plot the first frame
```

- how might you (naively) animate the movie in a matplotlib figure?
  - doing this properly is a bit complicated: https://matplotlib.org/api/animation_api.html
- movies can also be loaded using opencv, but again, that might be harder to install (and use)

Exercise 2

1. Load the immunohistochemistry example data from skimage (`immun = data.immunohistochemistry()`). Check its shape. Does it have colour? Display the image in a figure
2. Convert it to grayscale using `skimage.color.rgb2gray` . Check its shape and display it.
3. Remove the blue channel from the image and display it.
4. Apply the "Sobel" edge enhancement filter (`skimage.filters.sobel`) to the `ohki2005.png` image and display the result. Compare to the original image. Inspect both images to see why the algorithm worked better for some cell bodies in the image than for others.
5. Repeat the above for the contrast enhanced version of the image you created earlier in Exercise 1. Does the edge enhancement work better with enhanced contrast?
6. Bonus: try doing image segmentation by following the tutorial at http://scikit-image.org/docs/dev/user_guide/tutorial_segmentation.html

Edge-based segmentation:

```
from skimage import data
from skimage.feature import canny
```

```
coins = data.coins()
f1, ax1 = plt.subplots()
im = ax1.imshow(coins, cmap='gray') # image looks quite binary to our eyes
f2, ax2 = plt.subplots()
ax2.hist(coins.ravel(), bins=np.arange(256)) # histogram isn't very binary

edges = canny(coins) # edge detection
f3, ax3 = plt.subplots()
ax3.imshow(edges, cmap='gray')

fill_coins = ndimage.binary_fill_holes(edges) # fill in those edges
f4, ax4 = plt.subplots()
ax4.imshow(fill_coins, cmap='gray') # mostly works
```

Region-based (watershed) segmentation:

```
from skimage.filters import sobel
from skimage.morphology import watershed
elevation = sobel(coins) # edge detection

markers = np.zeros_like(coins)
# these markers designate foreground and background, see above histogram of coins
markers[coins < 30] = 1
markers[coins > 150] = 2

segmentation = watershed(elevation, markers)
segmentation = ndimage.binary_fill_holes(segmentation - 1) # remove some holes
labels, _ = ndimage.label(segmentation) # label the coins
f, ax = plt.subplots()
ax.imshow(labels, cmap='Dark2')
```