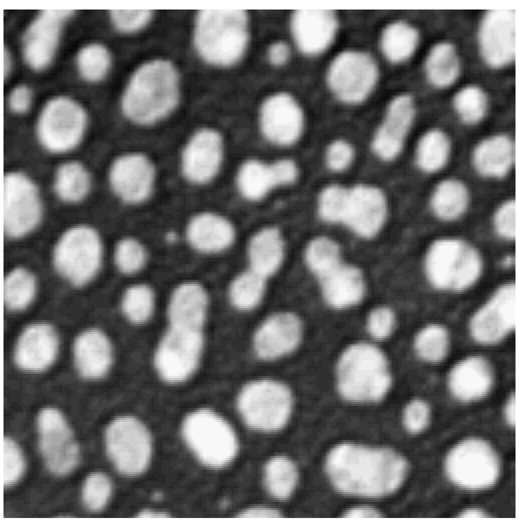
Introduction to Python Image Processing

There are many tools available for image processing in python. We will do some very simple excercises that will prepare you to think about images as data and some simple tools for manipulating that data

In [1]:
```
#PIL is pillow--the standard python imaging library
from PIL import Image
#tifffile reads tif's into a multidimensional array
import tifffile
#matplotlib let's us plot data
import matplotlib.pyplot as plt
#the matplotlib colors library let's us display those as colormaps
from matplotlib import colors
#pandas handles tables
import pandas as pd
#numpy handles multidimensional data sets
import numpy as np
```

In [2]:
```
#let's open a blobs image in the matplotlib image viewer
img=Image.open('blobs.tif')
#if we output an image, jupyter knows to show it as an image
img
```

Out[2]:



In [3]:
```
#that was cool but makes it difficult to see the image data
#if we load it with tifffile, we get a better look at things
img2=tifffile.imread('blobs.tif')
img2
```

Out[3]:
```
array([[ 40,  32,  24, ..., 216, 200, 200],
       [ 56,  40,  24, ..., 232, 216, 216],
       [ 64,  48,  24, ..., 240, 232, 232],
       ...,
       [ 72,  80,  80, ...,  48,  48,  48],
       [ 80,  80,  80, ...,  48,  48,  48],
       [ 96,  88,  80, ...,  48,  48,  48]], dtype=uint8)
```

Here we see that the image is an array that contains lots of other arrays (rows of the image)

In [4]:
```
#let's put it in a dataframe so we can look at it easier
imgdf=pd.DataFrame(img2)
print(imgdf.shape)
imgdf.head(20)
```

```
(254, 256)
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	...	246	247	248	249	250	251	252	253	2
0	40	32	24	24	16	24	24	32	32	32	...	232	240	240	240	240	240	232	216	2
1	56	40	24	24	24	32	32	32	32	32	...	240	248	248	248	248	240	240	232	2
2	64	48	24	32	32	32	32	32	32	32	...	248	248	248	248	248	248	240	240	2
3	40	40	32	40	40	40	40	32	32	24	...	248	248	248	248	248	248	248	240	2
4	16	24	32	40	48	48	40	32	24	24	...	248	248	240	248	248	248	248	240	2
5	24	32	40	48	56	48	40	32	24	24	...	240	240	240	240	240	240	240	240	2
6	32	40	48	56	56	48	32	24	16	16	...	232	232	232	232	232	232	232	232	2
7	40	48	48	56	56	40	32	24	16	24	...	232	232	232	232	232	232	232	232	2
8	48	48	48	48	48	40	24	24	16	24	...	232	232	232	232	232	232	232	232	2
9	48	48	48	48	40	40	32	32	24	40	...	232	232	232	232	232	232	232	232	2
10	48	48	40	40	32	32	32	32	32	48	...	224	232	232	232	232	232	232	224	2
11	40	40	32	32	32	32	32	40	48	64	...	224	232	232	232	232	232	232	224	2
12	32	32	24	24	24	32	32	48	56	80	...	224	232	232	232	232	232	232	216	2
13	32	32	24	24	24	32	40	56	64	96	...	224	232	232	232	232	232	232	224	2
14	32	32	24	24	24	32	40	56	72	104	...	216	224	232	232	232	232	232	224	2
15	48	40	24	24	24	32	40	56	72	96	...	216	224	232	232	232	232	232	224	2
16	56	40	24	24	24	32	40	56	64	88	...	216	224	224	232	232	232	232	224	2
17	80	56	40	32	24	32	32	48	64	88	...	216	224	224	232	232	232	232	232	2
18	96	72	48	40	24	24	24	40	56	88	...	216	216	216	224	232	232	232	232	2
19	120	96	64	48	32	32	24	48	64	96	...	216	216	216	224	232	232	232	232	2

20 rows × 256 columns

This table has a number for each pixel in the image. We can think of that number as an "intensity" or perhaps as a "color"

In [5]:
```
#let's use describe to get some stats on each image "column" of pixels
imgdf.describe()
```

Out[5]:

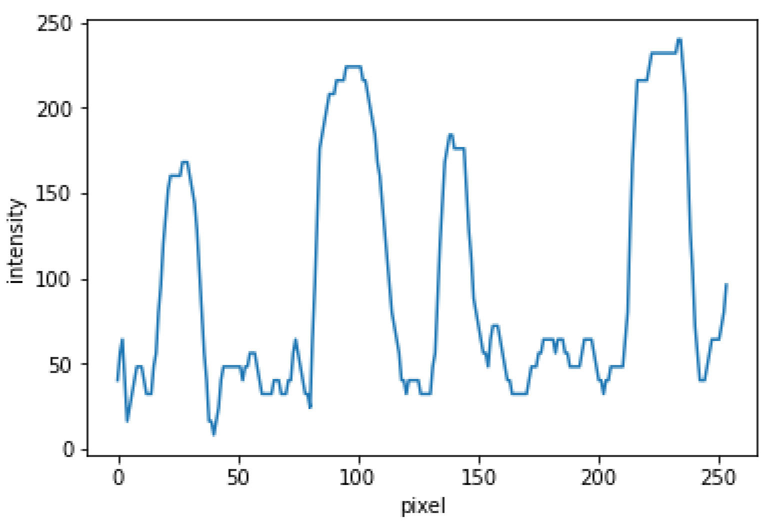
	0	1	2	3	4	5	6	
count	254.000000	254.000000	254.000000	254.000000	254.000000	254.000000	254.000000	2

	0	1	2	3	4	5	6	
mean	96.251969	98.425197	97.637795	100.125984	99.401575	102.992126	103.590551	
std	69.919431	71.234681	74.511072	73.557914	74.844266	73.317707	73.844335	
min	8.000000	16.000000	16.000000	24.000000	8.000000	16.000000	8.000000	
25%	40.000000	42.000000	40.000000	40.000000	40.000000	48.000000	40.000000	
50%	64.000000	56.000000	56.000000	56.000000	64.000000	64.000000	64.000000	
75%	160.000000	158.000000	160.000000	174.000000	182.000000	182.000000	184.000000	1
max	240.000000	240.000000	248.000000	248.000000	248.000000	240.000000	232.000000	2

8 rows × 256 columns

A few things about these data: the maximum values for each column are around 250. The minimum values are around 10.

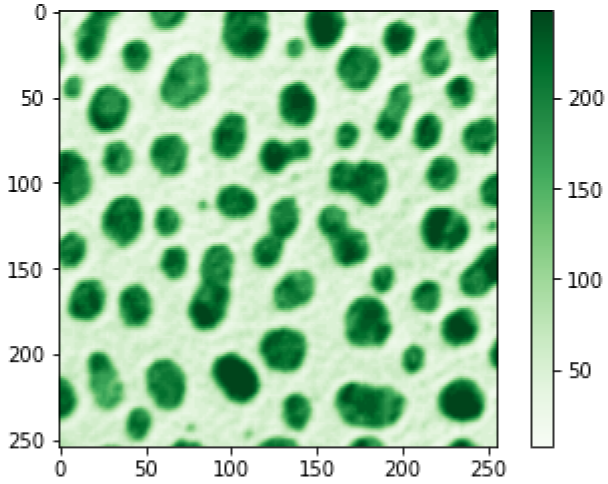
In [6]:
```
#let's plot the intensities in the first column
#we will use the matplotlib pyplot plot function which simply plots a list of da
plt.plot(imgdf[0])
plt.xlabel('pixel')
plt.ylabel('intensity')
plt.show()
```



Looking at the image above on the left side, we see 4 white blobs. Those correspond to the high intensity values in our plot. We see those values as "white" in the image above but they could be a different color too.

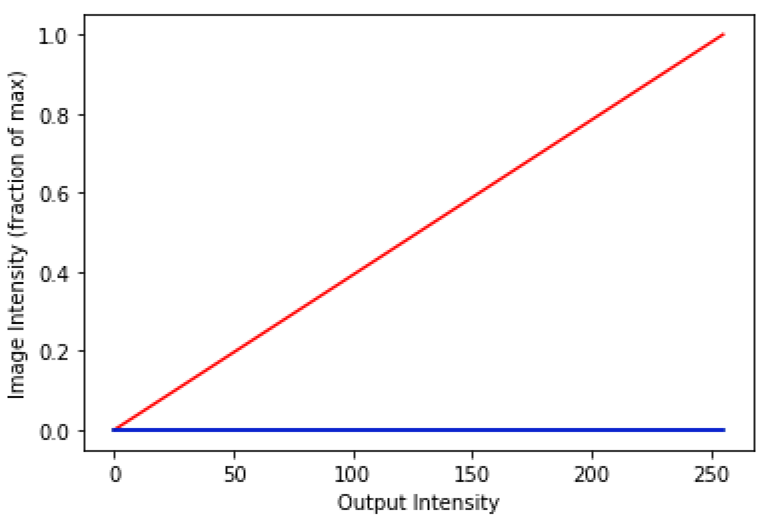
In [7]:
```
#matplotlib's imshow let's us show data as images in other ways
plt.imshow(img2,cmap='Greens')
plt.colorbar()
```

Out[7]: `<matplotlib.colorbar.Colorbar at 0x7fe823328340>`



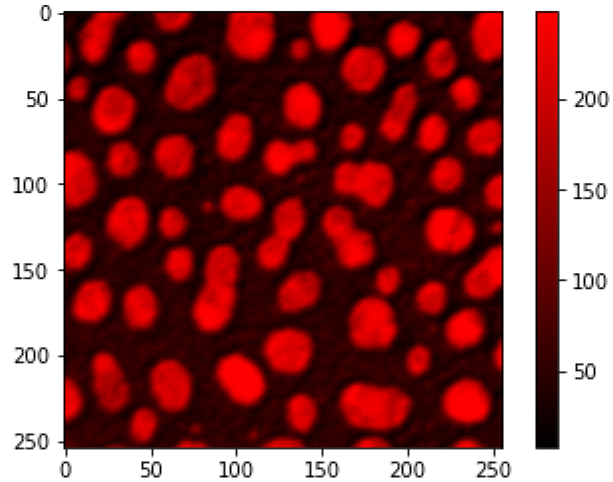
In that version of things high intensities are dark green while low intensities are light green. The colormap (cmap argument in imshow) maps how different colors are displayed. The colorbar at the right shows how this mapping is done. There are a large variety of ways that colormapping can be done. Let's explore this a bit more. Computer colors are usually a mixture of red, green, and blue colors. If only the red color get's brighter at higher image intensities, we have a red colormap:

In [8]:
```
redmap=np.arange(256)/255.0
greenmap=np.zeros(256)
bluemap=np.zeros(256)
plt.plot(redmap,'r')
plt.plot(greenmap,'g')
plt.plot(bluemap,'b')
plt.xlabel('Output Intensity')
plt.ylabel('Image Intensity (fraction of max)')
plt.show()
```



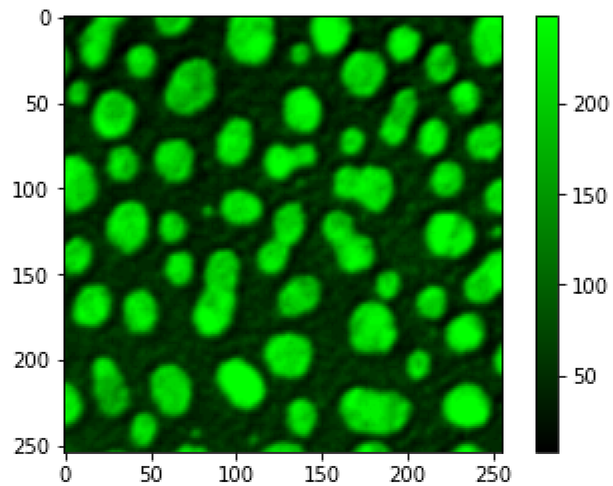
Now let's show the corresponding image for that colormap.

In [9]:
```
redcmap=colors.ListedColormap(np.array([redmap,greenmap,bluemap]).T)
plt.imshow(img2,cmap=redcmap)
plt.colorbar()
plt.show()
```



If we swap the red and green here we have a green colormap.

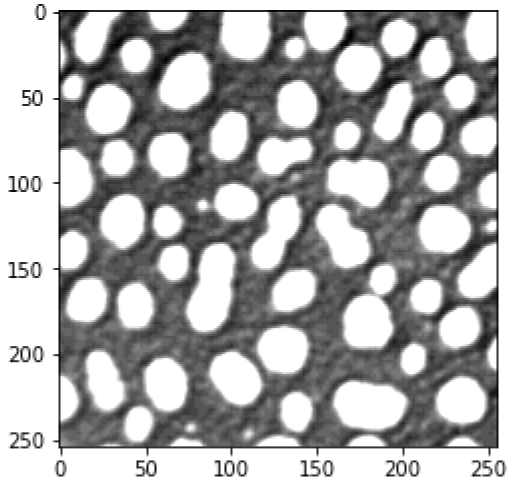
In [10]:
```
greencmap=colors.ListedColormap(np.array([greenmap,redmap,bluemap]).T)
plt.imshow(img2,cmap=greencmap)
plt.colorbar()
plt.show()
```



Scientific images can have a large range of intensities. We have to decide how bright everything should be in the display. The software usually set's the maximum intensity in the image to be the maximum displayed value. We can change that value to brighten or dim the image.

In [11]:
```
#let's brighten our image up by setting the display maximum to 128
plt.imshow(img2,cmap='gray',vmax=128)
```

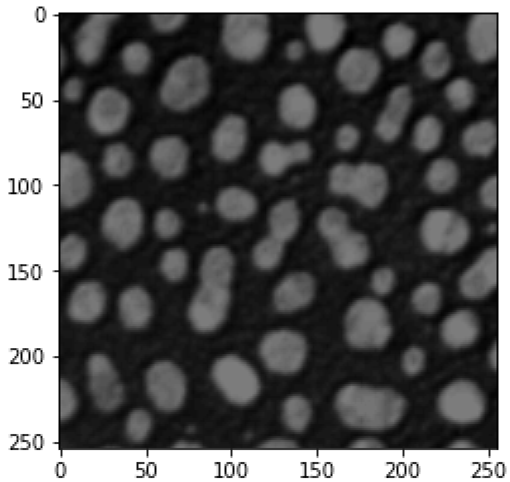
Out[11]: <matplotlib.image.AxesImage at 0x7fe823474550>



Now "white" in the image is at the intensity value of 128 rather than around 250 as above. Anything above 128 shows up as white as well.

In [12]:
```
#let's try going the other way and setting the maximum to 500
plt.imshow(img2,cmap='gray',vmax=500)
```

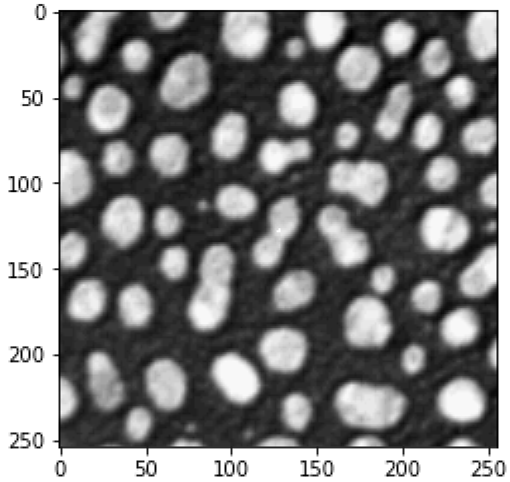
Out[12]: <matplotlib.image.AxesImage at 0x7fe7f0570550>



Now "white" is 500 and none of our pixels are bright enough to reach that level. We can also set the minimum display value.

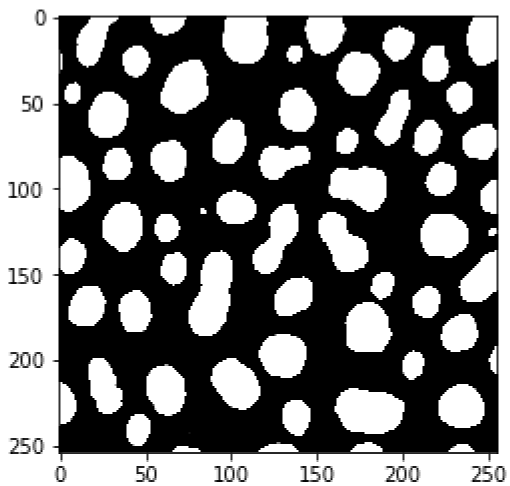
In [13]:
```
img3=img2.astype(float)
img3[128,128]=3000
plt.imshow(img3,cmap='gray',vmax=255)
```

Out[13]: <matplotlib.image.AxesImage at 0x7fe7e0069100>



In [14]:
```
#what if we set the display range very narrowly: between 127 and 128
plt.imshow(img2,cmap='gray',vmin=127,vmax=128)
```

Out[14]: <matplotlib.image.AxesImage at 0x7fe823397dc0>



Now everything below 127 is black and everything above 128 is white, effectively giving us a black and white image. The range of values that are shown in an image is called "contrast". The overall intensity of the image is called "brightness". We can utilize a black and white image like the one above to measure things about our image.

In [15]:
```
#lets' count the number of pixels above 128
bwdf=imgdf>128
bwdf.head()
```

Out[15]:

	0	1	2	3	4	5	6	7	8	9	...	246	247	248	249	250
0	False	False	False	False	False	False	False	False	False	False	...	True	True	True	True	True
1	False	False	False	False	False	False	False	False	False	False	...	True	True	True	True	True
2	False	False	False	False	False	False	False	False	False	False	...	True	True	True	True	True
3	False	False	False	False	False	False	False	False	False	False	...	True	True	True	True	True
4	False	False	False	False	False	False	False	False	False	False	...	True	True	True	True	True

5 rows × 256 columns

In [16]:
```
#the pandas sum function counts the number of true values, call it twice for the
countwhite=bwdf.sum().sum()
countwhite
```

Out[16]: 21413

In [17]:
```
#so 21413 pixels are white, how many pixels are in our image?
total_pixels=bwdf.shape[0]*bwdf.shape[1]
total_pixels
```

Out[17]: 65024

In [18]:
```
countwhite/total_pixels
```

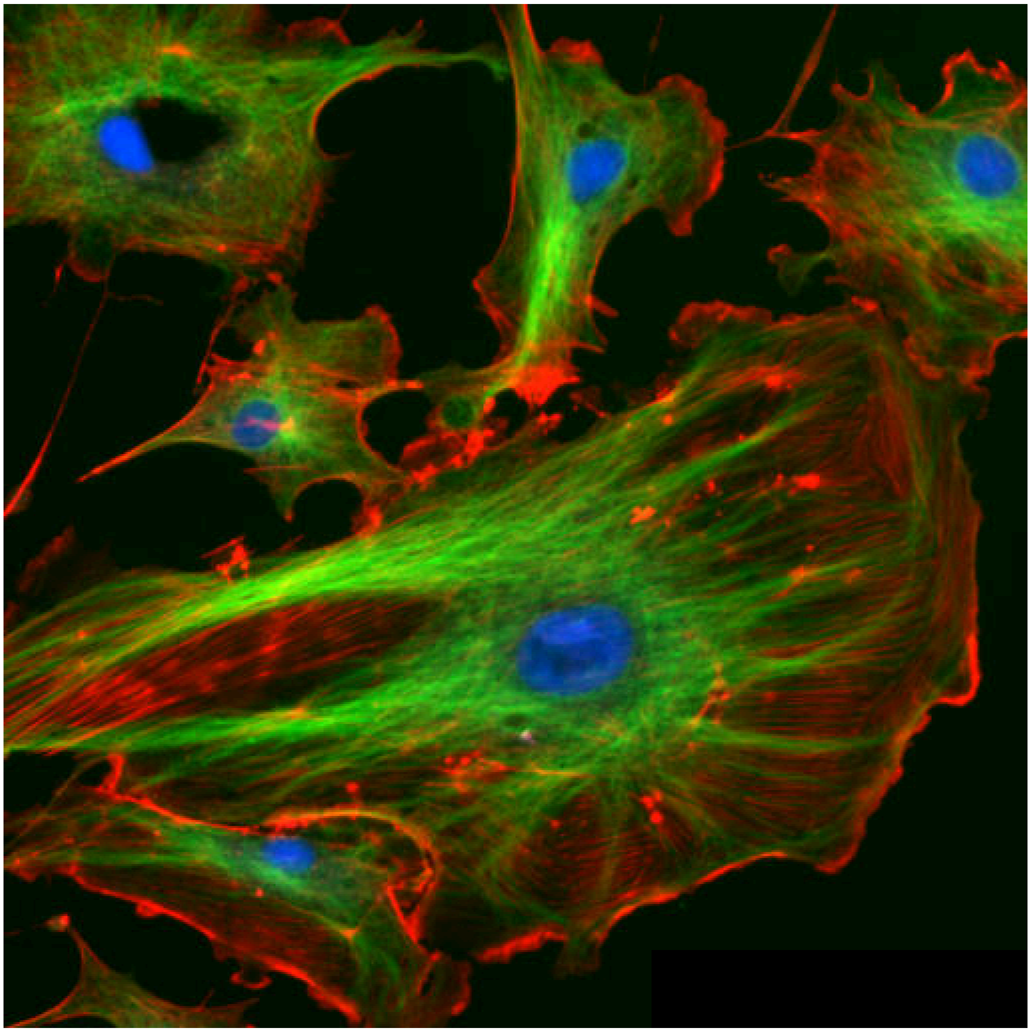
Out[18]: 0.3293091781496063

Now we know that around 1/3 of our pixels are above 128 intensity.

Let's open a color image and see how that's different than our black and white image.

In [19]:
```
#first view it with the standard library as before
Image.open('FluorescentCells.tif')
```

Out[19]:



Pretty! Let's see what this data looks like.

In [20]:

```
cimg=tifffile.imread('FluorescentCells.tif')
cimg.shape
```

Out[20]:

```
(512, 512, 3)
```

The .shape attribute is used in numpy and in pandas and tells us the sizes of all of the dimensions of a multidimensional array. Here we have 3 dimensions: y, x, and color. Digital color images are typically in RGB format, i.e. there is a separate image for red, green and blue that is rendered by red, green and blue pixels in your computer screen. We can use "slicing" to get different dimensions. The ":" symbol means we want all of that dimensions values. Let's use this to look at just one of the colors of our image.

In [21]:

```
c1df=pd.DataFrame(cimg[:,:,0])
c1df
```

Out[21]:

	0	1	2	3	4	5	6	7	8	9	...	502	503	504	505	506	507	508	5
0	16	15	18	21	19	14	13	17	22	17	...	0	0	0	0	0	0	0	
1	13	13	15	19	18	15	15	18	25	21	...	0	0	0	0	0	0	0	
2	12	12	13	17	18	15	18	24	28	23	...	0	0	0	0	0	0	0	

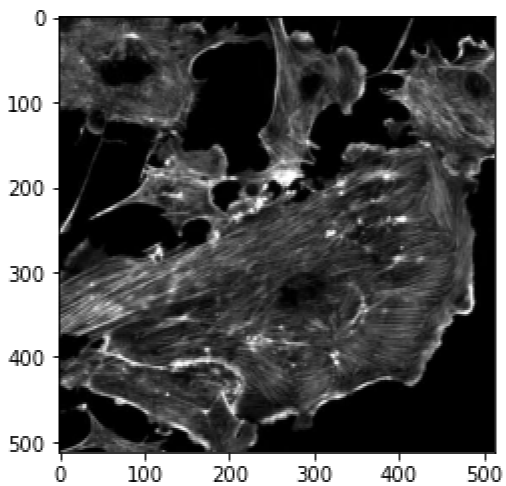
	0	1	2	3	4	5	6	7	8	9	...	502	503	504	505	506	507	508	5
3	12	12	12	16	18	17	21	27	26	20	...	0	0	0	0	0	0	0	
4	15	12	14	16	16	18	24	32	20	16	...	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
507	0	0	0	7	8	5	0	0	3	3	...	0	0	0	0	0	0	0	
508	16	10	3	0	0	0	0	0	42	72	...	0	0	0	0	0	0	0	
509	13	0	0	0	2	45	89	120	158	166	...	0	0	0	0	0	0	0	
510	0	0	13	44	85	130	171	194	163	140	...	0	0	0	0	0	0	0	
511	36	73	128	167	174	144	98	66	74	62	...	0	0	0	0	0	0	0	

512 rows × 512 columns

This is just a grayscale image like our blobs image above. Let's see what it looks like.

In [22]:
```
plt.imshow(c1df,cmap='gray')
```

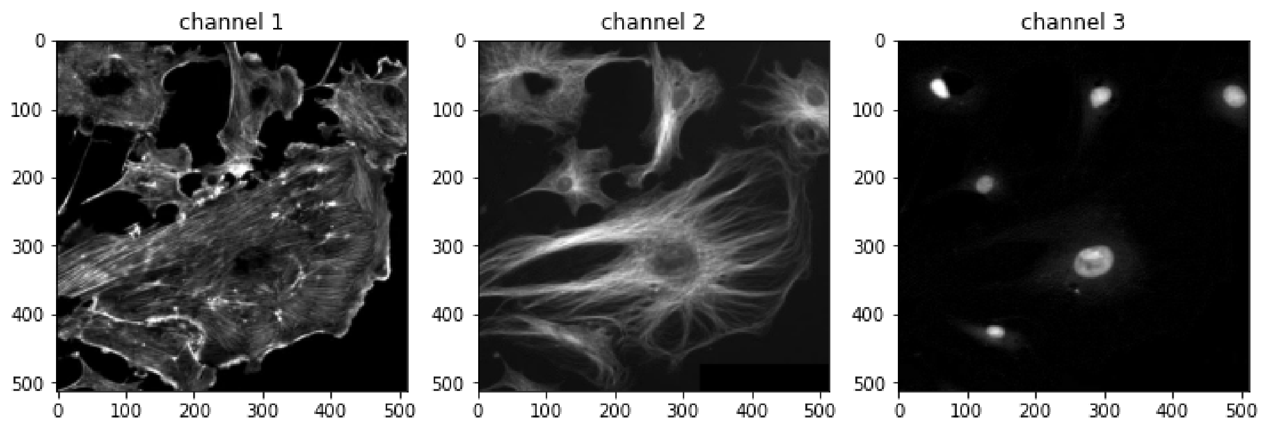
Out[22]: <matplotlib.image.AxesImage at 0x7fe7d00e2f40>



This looks just like the red color above! So the first color in a color image is red. Let's use the subplot feature of matplotlib to show all of the colors side by side.

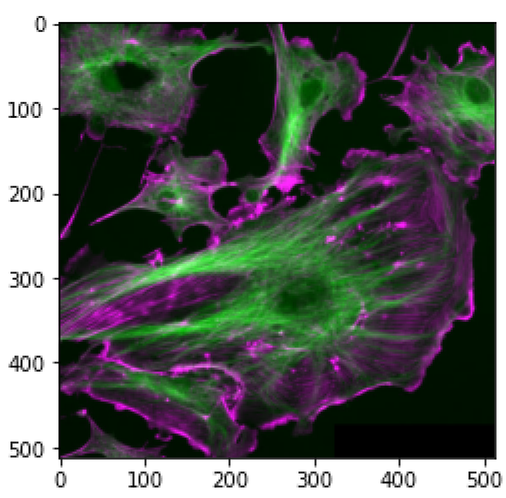
In [23]:
```
#start by making our figure a bit bigger so it's easier to see
plt.figure(figsize=(12,4))
#the subplot command let's us specify the dimensions of our plotting grid: here
ax=plt.subplot(1,3,1)
#ax is an axes object, we can use similar commands to the plt object
ax.imshow(cimg[:,:,0],cmap='gray')
ax.set_title('channel 1')
ax=plt.subplot(1,3,2)
ax.imshow(cimg[:,:,1],cmap='gray')
ax.set_title('channel 2')
ax=plt.subplot(1,3,3)
ax.imshow(cimg[:,:,2],cmap='gray')
```

```
ax.set_title('channel 3')
plt.show()
```



So now you know how to make RGB images: you just put (r,g,b) values in each pixel position. We can turn our red image into a magenta image by copying it into the 1st and 3rd channels of a blank image. Red and blue make magenta.

In [24]:
```
#this makes a blank image with the same shape as the one we imported
cimg2=np.zeros(cimg.shape,dtype=np.uint8)
#set the red channel
cimg2[:,:,0]=cimg[:,:,0]
#set the blue channel
cimg2[:,:,2]=cimg[:,:,0]
#keep the green channel as it was
cimg2[:,:,1]=cimg[:,:,1]
plt.imshow(cimg2)
plt.show()
```



Mixing colors can be tricky but fun! Note that computer screens can only display between 0 and 255 intensity levels, so if you add too many images in a channel everything will saturate. Anyway, play and have fun and see what you can create!

In [ ]:
```

```