3. Image import/export

For the moment, we have only used images that were provided internally by skimage. We are however normally going to use data located in the file system. The module skimage.io deals with all in/out operations and supports a variety of different import mechanisms.

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
import skimage.io as io
```

3.1 Simple case

Most of the time the simples command imread() will do the job. One has just to specifiy the path of the file or a url. In general your path is going to look something like:

```
image = io.imread('/This/is/a/path/MyData/Klee.jpg')
```

In [5]:
```
file_path = 'Data/Klee.jpg'
print(file_path)
```

```
Data/Klee.jpg
```

Here we only use a relative path, knowing that the `Data` folder is in the same folder as the notebook. However you can also give a complete path. We can also check what's the complete path of the current file:

In [6]:
```
import os
print(os.path.realpath(file_path))
```

```
/home/marie/Documents/CAS_data_science/CAS_21.01.2020_Python_Image_Proces
sing/PyImageCourse-master/Data/Klee.jpg
```

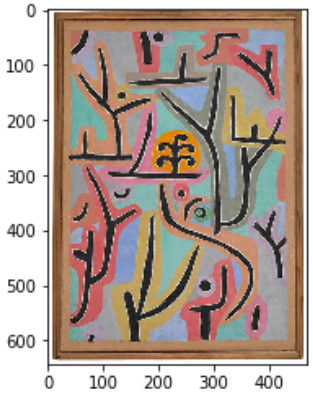
Now we can import the image:

In [7]:
```
image = io.imread(file_path)
```

In [8]:
```
image.shape
```

Out[8]: (643, 471, 3)

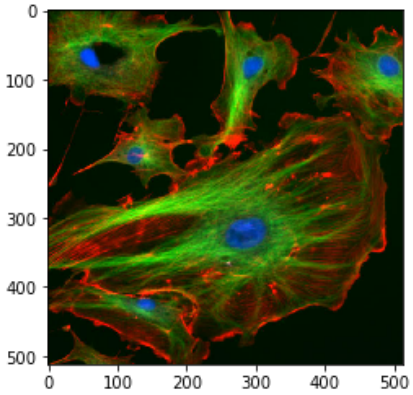
In [9]:
```
plt.imshow(image);
```



Now with a url:

In [18]:
```
image = io.imread('https://upload.wikimedia.org/wikipedia/commons/0/09/FluorescentCells.jpg')
```

In [19]:
```
plt.imshow(image)
plt.show()
```



3.2 Series of images (.tif)

Popular compressed formats such as jpg are usually used for natural images e.g. in facial recognition. The reason for that is that for those applications, in most situations one does not care about quantitative information and effects of information compression occurring in jpg are irrelevant. Also, those kind of data are rarely multi-dimensional (except for RGB).

In most other cases, the actual pixel intensity gives important information and one needs a format that preserves that information. Usually this is the .tif format or one of its many derivatives. One advantage is that the .tif format allows to save multiple images within a single file, a very useful feature for multi-dimensional acquisitions.

You might encounter different situations.

3.2.1 Series of separate images

In the first case, you would have multiple single .tif files within one folder. In that case, the file name usually contains indications about the content of the image, e.g a time point or a channel. The general way of dealing with this kind of situation is to use regular expressions, a powerful tool to parse information in text. This can be done in Python using the `re` module.

Here we will use an approach that identifies much simpler patterns.

Let's first see what files are contained within a folder of a microscopy experiment containing images acquired at two wavelengths using the `os` module:

In [10]:
```
import glob
import os
```

In [11]:
```
folder = 'Data/BBBC007_v1_images/A9'
```

Let's list all the files contained in the folder

In [13]:
```
files = os.listdir(folder) # MZ: list all files that are within the dire
ctory
print(files)
```

```
['A9 p10f.tif', 'A9 p5f.tif', 'A9 p9d.tif', 'A9 p7f.tif', 'A9 p7d.tif', '
A9 p10d.tif', 'A9 p9f.tif', 'A9 p5d.tif']
```

The two channels are defined by the last character before .tif. Using the wild-card sign *we can define a pattern to select only the 'd' channel:* d.tif. We complete that name with the correct path. Now we use the native Python module `glob` to parse the folder content using this pattern:

In [14]:
```
d_channel = glob.glob(folder+'/*d.tif')
d_channel
```

Out[14]:
```
['Data/BBBC007_v1_images/A9/A9 p9d.tif',
 'Data/BBBC007_v1_images/A9/A9 p7d.tif',
 'Data/BBBC007_v1_images/A9/A9 p10d.tif',
 'Data/BBBC007_v1_images/A9/A9 p5d.tif']
```

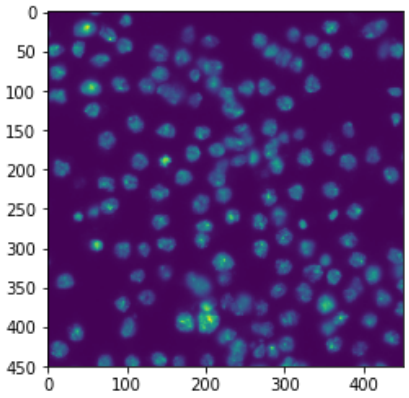
Then we use again the `imread()` function to import a specific file:

In [15]:
```
image1 = io.imread(d_channel[0])
```

In [16]:
```
image1.shape
```

Out[16]:
```
(450, 450)
```

In [17]:
```
plt.imshow(image1);
```



These two steps can in principle be done in one step using the `imread_collection()` function of skimage.

We can also import all images and put them in list if we have sufficient memory:

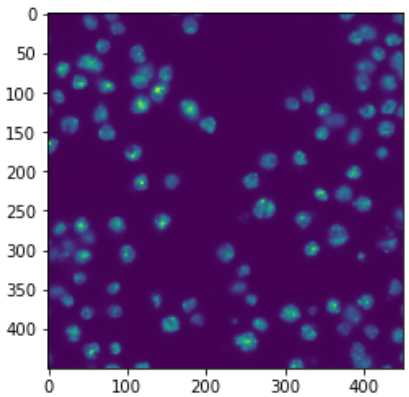
In [18]:
```
channel1_list = []
for x in d_channel:
    temp_im = io.imread(x)
    channel1_list.append(temp_im)
```

Let's see what we have in that list of images by plotting them:

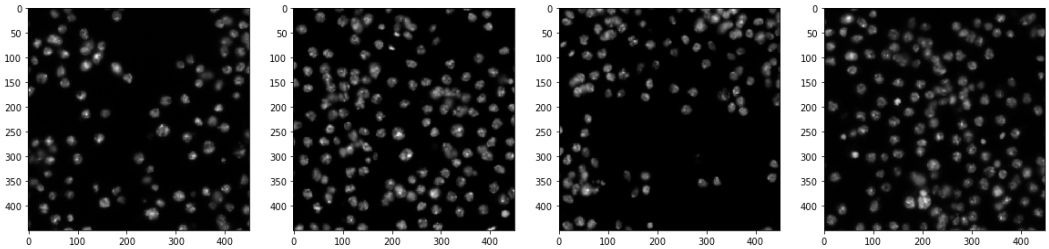
In [94]:
```
channel1_list[0].shape
```

Out[94]: (450, 450)

In [106]:
```
plt.imshow(channel1_list[0]);
```



In [105]:
```
num_plots = len(channel1_list)
plt.figure(figsize=(20,30))
for i in range(num_plots):
    plt.subplot(1,num_plots,i+1)
    plt.imshow(channel1_list[i],cmap = 'gray')
```



3.2.2 Multi-dimensional stacks

We now look at a more complex multi-dimensional case taken from a public dataset (J Cell Biol. 2010 Jan 11;188(1):49-68) that can be found here (http://flagella.crbs.ucsd.edu/images/30567).

We already provide it in the datafolder:

In [19]:
```
file = 'Data/30567/30567.tif'
# MZ: tif can contain many data and also can contain metadata -> very us
eful
```

In [21]:
```
image = io.imread(file)
```

The dataset is a time-lapse 3D confocal microscopy acquired in two channels, one showing the location of tubulin, the other of lamin (cell nuclei).

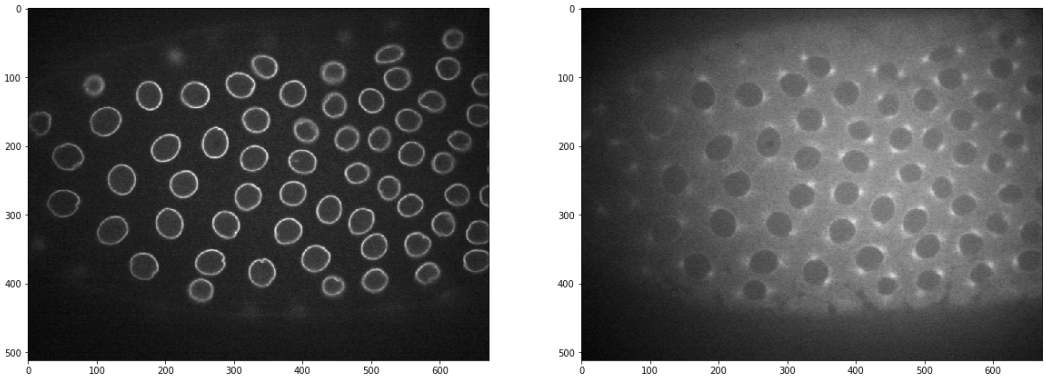
All .tif variants have the same basic structure: single image planes are stored in individual "sub-directories" within the file. Some meta information is stored with each plane, some is stored for the entire file. However, how the different dimensions are ordered within the file (e.g. all time-points of a given channel first, or alternatively all channels of a given time-point) can vary wildly. The simplest solution is therefore usually to just import the file, look at the size of various dimensions and plot a few images to figure out how the data are organized.

In [22]:
```
image.shape
```

Out[22]: (72, 2, 5, 512, 672)

We know we have two channels (dimension 2), and five planes (dimension 3). Usually the large numbers are the image dimension, and therefore 72 is probably the number of time-points. Using slicing, we look at the first time point, of both channels, of the first plane, and we indeed get an appropriate result:

In [23]:
```
plt.figure(figsize=(20,10))
plt.subplot(1,2,1)
plt.imshow(image[0,0,0,:,:],cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(image[0,1,0,:,:],cmap = 'gray');
```



We can check that our indexing works by checking the dimensions of the sliced image:

In [24]:
```
# where are the metadata and how to access them -> data-specific
image[0,0,0,:,:].shape
```

Out[24]: (512, 672)

As we have seen in the Numpy chapter, we can do various operations on arrays. In particular we saw that we can do projections. Let's extract all planes of a given time point and channel:

In [109]:
```
stack = image[0,0,:,:,:]
stack.shape
```

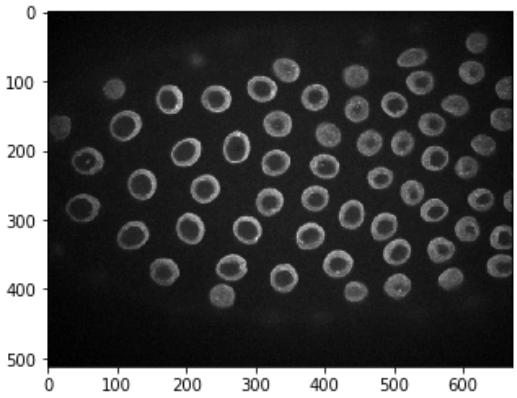
Out[109]: (5, 512, 672)

Here, to do a max projection, we now have to project all the planes along the **first** dimension, hence:

In [25]:
```
maxproj = np.max(image[0,0,:,:,:],axis = 0)
#MZ:  1st time point, 1st channel, but all the planes; take all max alon
g 1st dimension -> projection
# project on the 1st dimension (axis=0)
maxproj.shape
```

Out[25]: (512, 672)

In [26]:
```
plt.imshow(maxproj,cmap = 'gray')
plt.show()
```



skimage allows one to use specific import plug-ins for various applications (e.g. gdal for geographic data, FITS for astronomy etc.).

In particular it offers a lower-lever access to tif files through the tifffile module. This allows one for example to import only a subset of planes from the dataset if the latter is large.

In [27]:
```
# load only what you want (e.g. the 1st time point)
# so you don't need to load all the timepoints in memory

# tif -> most often used format for this kind of data
```

In [28]:
```
from skimage.external.tifffile import TiffFile

data = TiffFile(file)
```

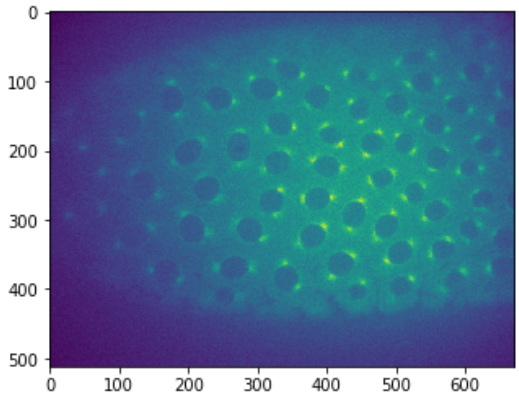
Now the file is open but not imported, and one can query information about it. For example some metadata:

In [29]:
```
data.info()
```

Out[29]:
```
'TIFF file: 30567.tif, 473 MiB, big endian, ome, 720 pages\n\nSeries 0: 7
2x2x5x512x672, uint16, TCZYX, 720 pages, not mem-mappable\n\nPage 0: 512x
672, uint16, 16 bit, minisblack, raw, ome|contiguous\n* 256 image_width
(1H) 672\n* 257 image_length (1H) 512\n* 258 bits_per_sample (1H) 16\n* 2
59 compression (1H) 1\n* 262 photometric (1H) 1\n* 270 image_description
(3320s) b\'<?xml version="1.0" encoding="UTF-8"?><!-- Wa\n* 273 strip_off
sets (86I) (182, 8246, 16310, 24374, 32438, 40502, 48566, 56630,\n* 277 s
amples_per_pixel (1H) 1\n* 278 rows_per_strip (1H) 6\n* 279 strip_byte_co
unts (86I) (8064, 8064, 8064, 8064, 8064, 8064, 8064, 8064, \n* 282 x_res
olution (2I) (1, 1)\n* 283 y_resolution (2I) (1, 1)\n* 296 resolution_uni
t (1H) 1\n* 305 software (17s) b\'LOCI Bio-Formats\''
```

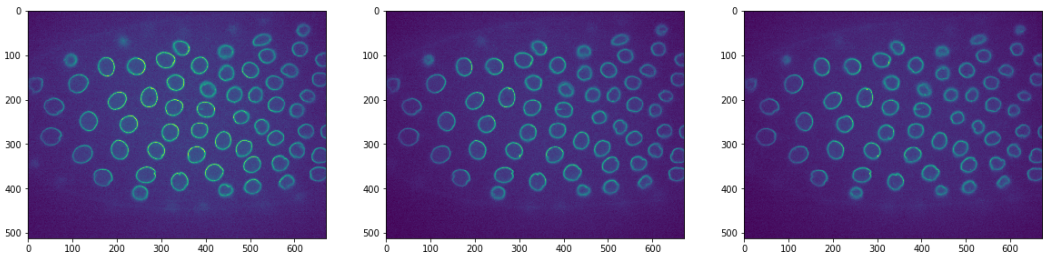
Some specific planes:

In [30]:
```
plt.imshow(data.pages[6].asarray())
plt.show()
```



In [31]:
```
image = [data.pages[x].asarray() for x in range(3)]
```

In [32]:
```
plt.figure(figsize=(20,10))
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(image[i])
plt.show()
```



3.2.3 Alternative formats

While a large majority of image formats is somehow based on tif, instrument providers often make their own tif version by creating a proprietary format. This is for example the case of the Zeiss microscopes which create the .czi format.

In almost all cases, you can find an dedicated library that allows you to open your particular file. For example for czi, there is a specific package (https://pypi.org/project/czifile/).

More generally your research field might use some particular format. For example Geospatial data use the format GDAL, and for that there is of course a dedicated package (https://pypi.org/project/GDAL/).

Note that a lot of biology formats are well handled by the tifffile package. io.imread() tries to use the best plugin to open a format, but sometimes if fails. If you get an error using the default io.imread() you can try to specific what plugin should open the image, *.e.g*

```
image = io.imread(file, plugin='tifffile')
```

3.3 Exporting images

There are two ways to save images. Either as plain matrices, which can be written and re-loaded very fast, or as actual images.

Just like for loading, saving single planes is easy. Let us save a small region of one of the images above:

In [119]:
```
image[0].shape
```

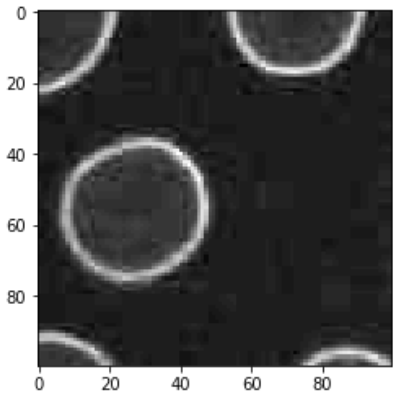
Out[119]:
```
(512, 672)
```

In [33]:
```
io.imsave('Data/region.tif',image[0][200:300,200:300])  # MZ: specify wh
at you want to save
io.imsave('Data/region.jpg',image[0][200:300,200:300])
```

```
/usr/local/lib/python3.5/dist-packages/skimage/util/dtype.py:141: UserWar
ning: Possible precision loss when converting from uint16 to uint8
  .format(dtypeobj_in, dtypeobj_out))
```

In [34]:
```
reload_im = io.imread('Data/region.jpg') # MZ: jpg not good for scientif
ic purposes
```

In [35]:
```
plt.imshow(reload_im,cmap='gray')
plt.show()
```



Saving multi-dimensional .tif files is a bit more complicated as one has of course to be careful with the dimension order. Here again the tifffile module allows to achieve that task. We won't go through the details, but here's an example of how to save a dataset with two time poins, 5 stacks, 3 channels into a file that can then be opened as a hyper-stack in Fiji:

In [36]:
```
from skimage.external.tifffile import TiffWriter

data = np.random.rand(2, 5, 3, 301, 219)#generate random images
data = (data*100).astype(np.uint8)#transform data in a reasonable 8bit r
ange

with TiffWriter('Data/multiD_set.tif', bigtiff=False, imagej=True) as ti
f:
    for i in range(data.shape[0]):
        tif.save(data[i])
```

3.4 Interactive plotting

Jupyter offers a solution to interact with various types of plots: ipywidget

In [125]:
```
from ipywidgets import interact, IntSlider
```

The interact() function takes as input a function and a value for that function. That function should plot or print some information. interact() then creates a widget, typically a slider, executes the plotting function and adjusts the ouptut when the slider is moving. For example:

In [126]:
```
def square(num=1):
    print(str(num)+' squared is: '+str(num*num))
```

In [127]:
```
square(3)
```

```
3 squared is: 9
```

In [128]:
```
interact(square, num=(0,20,1));
```

Depending on the values passed as arugments, interact() will create different widgets. E.g. with text:

In [129]:
```
def f(x):
    return x
interact(f, x='Hi there!');
```

An important note for our imaging topic: when moving the slider, the function is continuously updated. If the function does some computationally intensitve work, this might just overload the system. To avoid that, one can explicitly specifiy the slider type and its specificities:

In [130]:
```
def square(num=1):
    print(str(num)+' squared is: '+str(num*num))
interact(square, num = IntSlider(min=-10,max=30,step=1,value=10,continuo
us_update = False));
```

If we want to scroll through our image stack we can do just that. Let's first define a function that will plot the first plane of the channel 1 at all time points:

In [131]:
```
image = io.imread(file)
```

In [132]:
```
def plot_plane(t):
    plt.imshow(image[t,0,0,:,:])
    plt.show()
```

In [133]:
```
interact(plot_plane, t = IntSlider(min=0,max=71,step=1,value=0,continuou
s_update = False));
```

Of course we can do that for multiple dimensions:

In [134]:
```
def plot_plane(t,c,z):
    plt.imshow(image[t,c,z,:,:])
    plt.show()

interact(plot_plane, t = IntSlider(min=0,max=71,step=1,value=0,continuou
s_update = True),
         c = IntSlider(min=0,max=1,step=1,value=0,continuous_update = Tr
ue),
         z = IntSlider(min=0,max=4,step=1,value=0,continuous_update = Tr
ue));
```

And we can make it as fancy as we want:

In [135]:
```
def plot_plane(t,c,z):
    if c == 0:
        plt.imshow(image[t,c,z,:,:], cmap = 'Reds')
    else:
        plt.imshow(image[t,c,z,:,:], cmap = 'Blues')
    plt.show()

interact(plot_plane, t = IntSlider(min=0,max=71,step=1,value=0,continuou
s_update = True),
         c = IntSlider(min=0,max=1,step=1,value=0,continuous_update = Tr
ue),
         z = IntSlider(min=0,max=4,step=1,value=0,continuous_update = Tr
ue));
```