Image Handling

Last updated on 2024-07-02 | Edit this page ✏️

[Download Chapter notebook (ipynb)](#)

Mandatory Lesson Feedback Survey

OVERVIEW

Questions

- How to read and process images in Python?

- How is an image mask created?

- What are colour channels in images?

- How to deal with big images?

Objectives

- Understanding 2-dimensional greyscale images.

- Learning image masking.

- 2-dimensional colour images, colour channels

- Decreasing memory load

PREREQUISITES

- Numpy arrays

- Plots and subplots with matplotlib

EXERCISES

This lesson has no explicit exercises. At each step, use images of your own choice to practice. There are many image file formats, different colour schemes etc for which you can try to find similar or analogous solutions.

Challenge

Reading and Processing Images

In biology, we often deal with images, for example from microscopy and different medical imaging modalities. In many cases, we wish to extract some quantitative information from these images. The focus of this lesson is to read and process images in Python. This includes:

- Working with 2-dimensional greyscale images
- Creating and applying binary image masks
- Working with 2-dimensional colour images, and interpreting colour channels
- Decreasing the memory for further processing by reducing resolution or patching
- Working with 3-dimensional images

Image Example

The example in Figure 1 is an image from the cell image library with the following description:

> "Midsaggital section of rat cerebellum, captured using confocal imaging. Section shows inositol trisphosphate receptor (IP3R) labelled in green, DNA in blue, and synaptophysin in magenta. Honorable Mention, 2010 Olympus BioScapes Digital Imaging Competition®."

We might want to, for example, determine the relative amounts of IP3R, DNA and synaptophysin in this image. This tutorial will guide you through some of the steps to get you started with processing images of all sorts using Python. At the end, you will have the opportunity to come back to this image example and perform some analysis of your own.

Figure 1: Example image, rat cerebellum

Figure 1: Example image, rat cerebellum

Work Through Example

Reading and Plotting a 2-dimensional Image

First, we want to read in an image. For this part of the lesson, we use a histological slice through an axon bundle as an example. We use Matplotlib's image module, from which we import `imread` to store the image in a variable called img. The function `imread` can interpret many different image formats, including jpg, png and tif images.

```python
from matplotlib.image import imread

img = imread('fig/axon_slice.jpg')
```

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/axon_slice.jpg'
```

We can check what type of variable this is:

```python
print(type(img))
```

```
NameError: name 'img' is not defined
```

This tells us that the image is stored in a Numpy array. We can check some other properties of this array, for example, what the image dimensions are.

```python
print(img.shape)
```

```
NameError: name 'img' is not defined
```

This tells us that our image is composed of 2300 by 3040 data units, or *pixels* as we are dealing with an image. It is equivalent to the image resolution. The array has two dimensions, and so we can expect our image to be two-dimensional as well. Let us now use matplotlib.pyplot's `imshow` function to plot the image to see what it looks like. We set the colour map to `gray` to overwrite the default colour map.
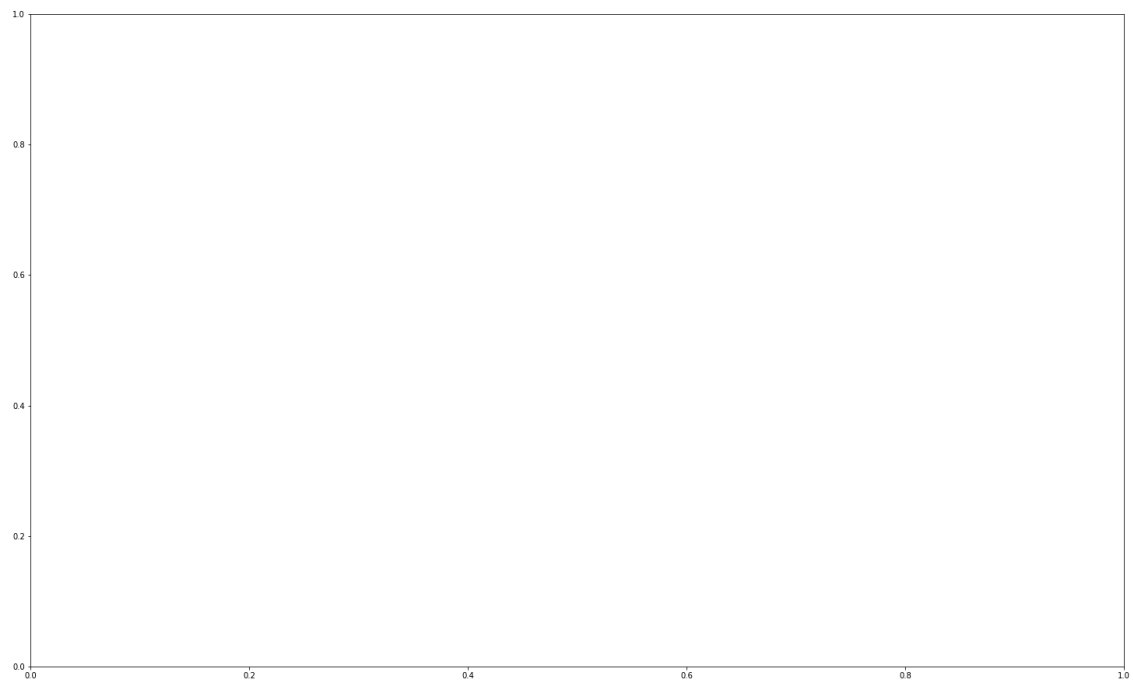
```python
from matplotlib.pyplot import subplots, show

fig, ax = subplots(figsize=(25, 15))

ax.imshow(img, cmap='gray');
```

```python
show()
```

`imshow` has allowed us to plot the Numpy array of our image data as a picture. The figure is divided up into a number of pixels, and each of those pixels is assigned an intensity value stored in the Numpy array. Let's have a closer look by selecting a smaller region of our image and plotting that.

```python
from matplotlib.pyplot import subplots, show

fig, ax = subplots(figsize=(25, 15))

ax.imshow(img[:50, :70], cmap='gray');
```
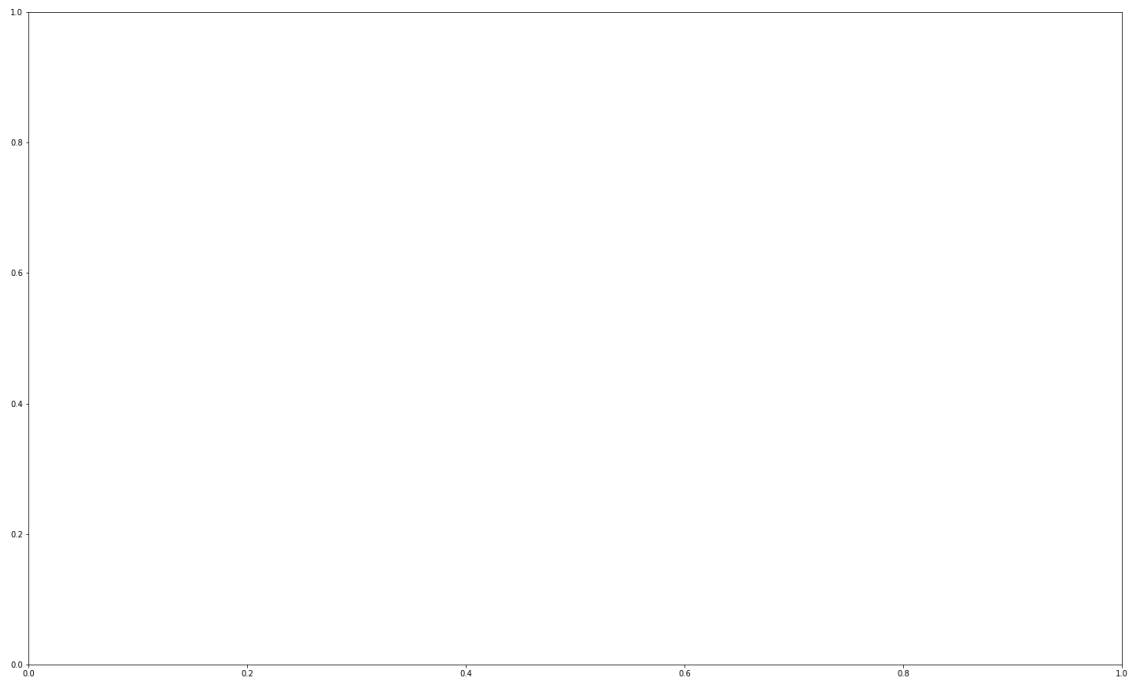
```
NameError: name 'img' is not defined
```

```python
show()
```

With `img[:50, :70]` we select the first 50 values from the first dimension, and the first 70 values from the second dimension. Thus, the image above shows a very small part of the upper left corner of our original image. As we are now zoomed in quite close to that corner, we can easily see the individual pixels here. Let's take a quick look at an even smaller section.
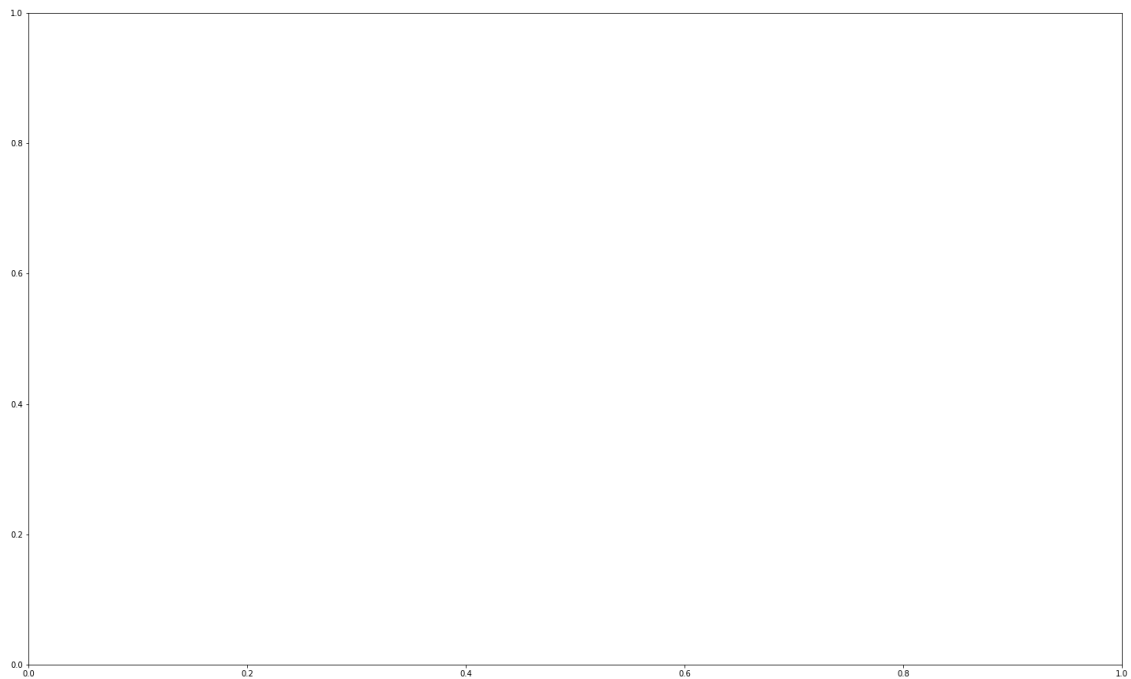
```python
fig, ax = subplots(figsize=(25, 15))

ax.imshow(img[:20, :15], cmap='gray');
```

```
NameError: name 'img' is not defined
```

```python
show()
```

This is a small section from that same upper left corner. Each square is a pixel and it has one grey value. So how exactly are the pixel values assigned? By the numbers stored in the Numpy array, `img`. Let us have a look at those values by picking a slice from the array.

```python
print(img[:20, :15])
```

```
NameError: name 'img' is not defined
```

Each of these numbers corresponds to an intensity in the specified colourmap. These numbers range from 0 to 255, implying 256 shades of grey.

We chose `cmap = gray`, which assigns darker grey colours to smaller numbers, and lighter grey colours to higher numbers. However, we can also pick a colourmap to plot our image, and we can even show a colourbar to keep track of the intensity values. Matplotlib has a large number of very nice colourmaps that you can look through here. We show an example of the colourmaps called `viridis` and `magma`:

```python
fig, ax = subplots(nrows=1, ncols=2, figsize=(25, 15))

p1 = ax[0].imshow(img[:20, :15], cmap='viridis')
```

```python
p2 = ax[1].imshow(img[:20, :15], cmap='magma')
```

```python
fig.colorbar(p1, ax=ax[0], shrink = 0.8)
```
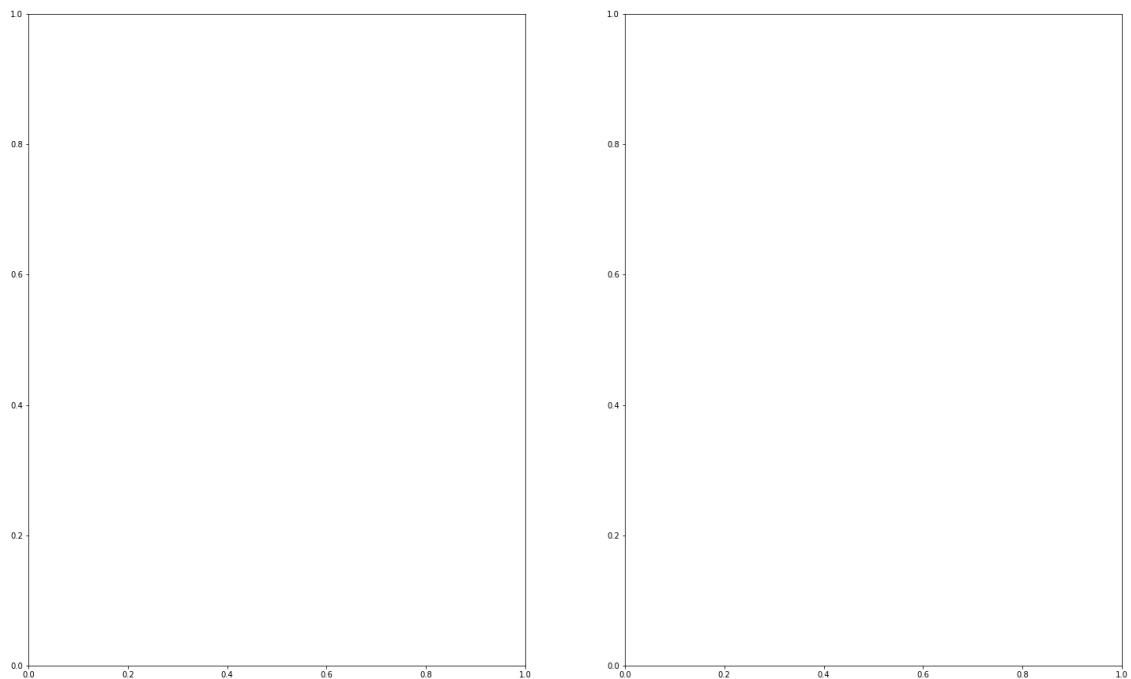
```python
fig.colorbar(p2, ax=ax[1], shrink = 0.8);
```

```python
show()
```

Note, that even though we can plot our greyscale image with colourful colourschemes, it still does not qualify as a colour image. This is because colour images will require **three sets** of intensities for each pixel, not just one as in this example. In the case above, the number in the array represented a grey value and the colour was assigned to that grey value by Matplotlib. These represent 'false' colours.

Creating an Image Mask

Now that we know that the images are composed of a set of intensities that are just numbers in a Numpy array, we can start using these numbers to process our image.

As a first approach, we can plot a histogram of the original image intensities. We use the `.flatten()` method to turn the original 2300 x 3040 array into a one-dimensional array with 6,992,000 values. This rearrangement allows the inclusion of an image as a single column in a matrix or dataframe!
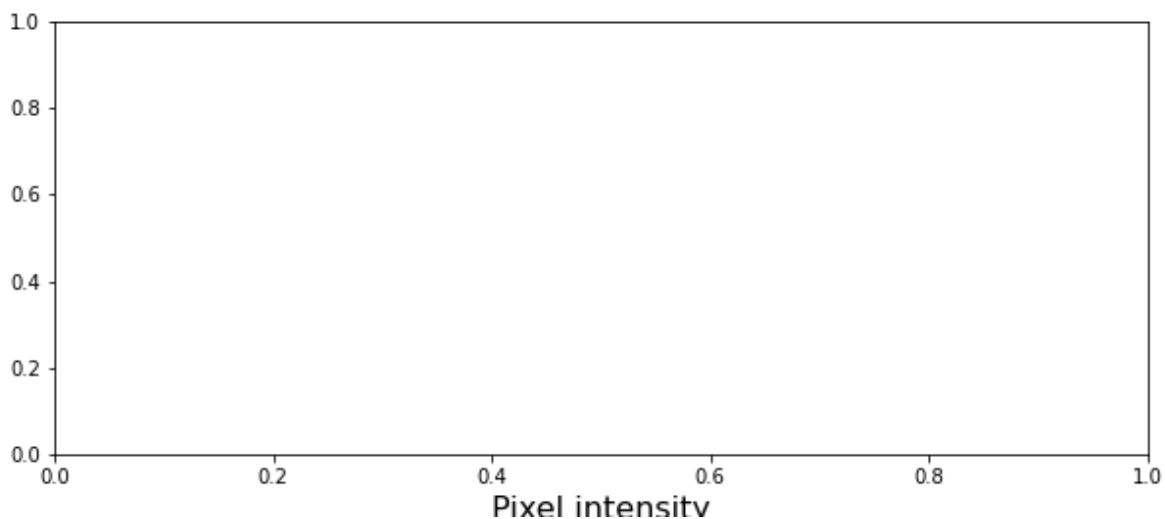
The histogram plot shows how many of each of the intensities are found in this image:

```PYTHON
fig, ax = subplots(figsize=(10, 4))

ax.hist(img.flatten(), bins = 50)
```

```PYTHON
ax.set_xlabel("Pixel intensity", fontsize=16);

show()
```


The histogram is a distribution with intensity values mostly between about 50 and 250.

The image shows a cut through an axon bundle. Say we are now interested in the myelin sheath surrounding the axons (the dark rings). We can create a **mask** that isolates pixels whose intensity value is below a certain threshold (because darker pixels have lower intensity values). Everything below this threshold can be assigned to e.g. 1 (representing True), and everything above will be assigned to 0 (representing False). This is called a binary or Boolean mask.

Based on the histogram above, we might try to adjust that threshold somewhere between 100 and 200. Let's see what we get with a threshold set to 125. We first use a conditional statement to create the mask. Then we apply the mask to the image. As a result we plot both the mask and the masked image.

```python
threshold = 125

mask = img < threshold
```

```python
img_masked = img*mask
```

```python
fig, ax = subplots(nrows=1, ncols=2, figsize=(20, 10))

ax[0].imshow(mask, cmap='gray')
```
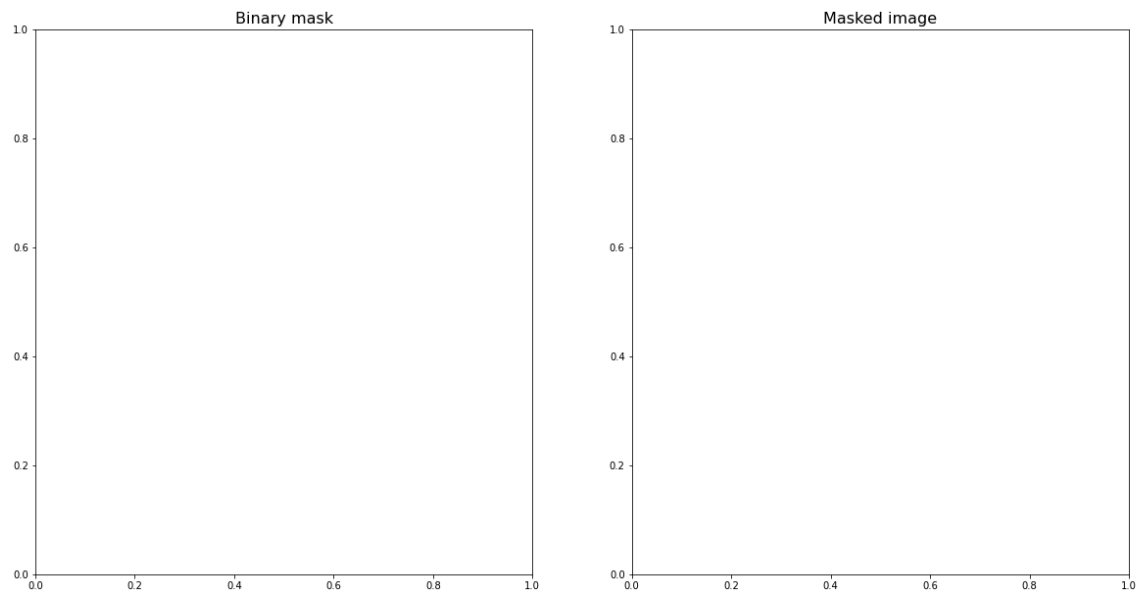
```python
ax[0].set_title('Binary mask', fontsize=16)
ax[1].imshow(img_masked, cmap='gray')
```

```python
ax[1].set_title('Masked image', fontsize=16)

show()
```

The left subplot shows the binary mask itself. White represents values where our condition is true, and black where our condition is false. The right image shows the original image after we have applied the binary mask, i.e. the original pixel intensities in regions where the mask value is true.

Note that "applying the mask" means that the intensities where the condition is true are left unchanged and the intensities where the condition is false are multiplied with zero ans therefore set to zero.

Let's have a look at the resulting image histograms.

```python
fig, ax = subplots(nrows=1, ncols=2, figsize=(20, 5))

ax[0].hist(img_masked.flatten(), bins=50)
```

```python
ax[0].set_title('Histogram of masked image', fontsize=16)
ax[0].set_xlabel("Pixel intensity", fontsize=16)

ax[1].hist(img_masked[img_masked != 0].flatten(), bins=25)
```
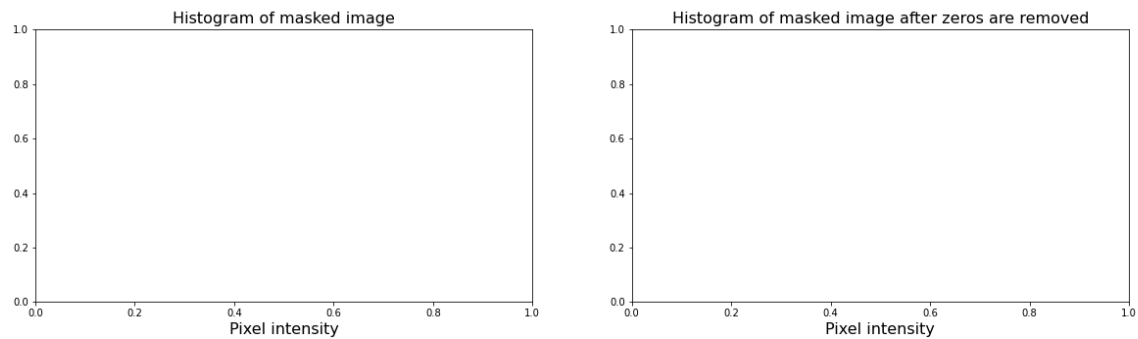
```python
ax[1].set_title('Histogram of masked image after zeros are removed', fontsize=16)
ax[1].set_xlabel("Pixel intensity", fontsize=16)

show()
```

On the left we show all the values for the masked image. There is a large peak at zero, as a large part of the image is masked. On the right, we show only the non-zero pixel intensities. We can see that our mask worked as expected, only values up to 125 are found. This is because our threshold causes a sharp cut-off at a pixel intensity of 125.

Colour Images

Often we want to work with colour images. So far, our image had a single intensity value for each pixel. In colour images, we will have three so-called channels corresponding to red, green and blue intensities. Any colour will be a composite of the intensity value for each of these colours. We now show an example with a colour image of the rat cerebellar cortex. Let us import it and check its shape.

```python
img_col = imread('fig/rat_brain_low_res.jpg')
```

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_brain_low_res.jpg'
```

```python
img_col.shape
```

```
NameError: name 'img_col' is not defined
```

Our image array now contains three dimensions. The first two are the spatial dimensions corresponding to the pixel positions. The last one contains the three colour channels. So we have three layers of intensity values on top of each other.
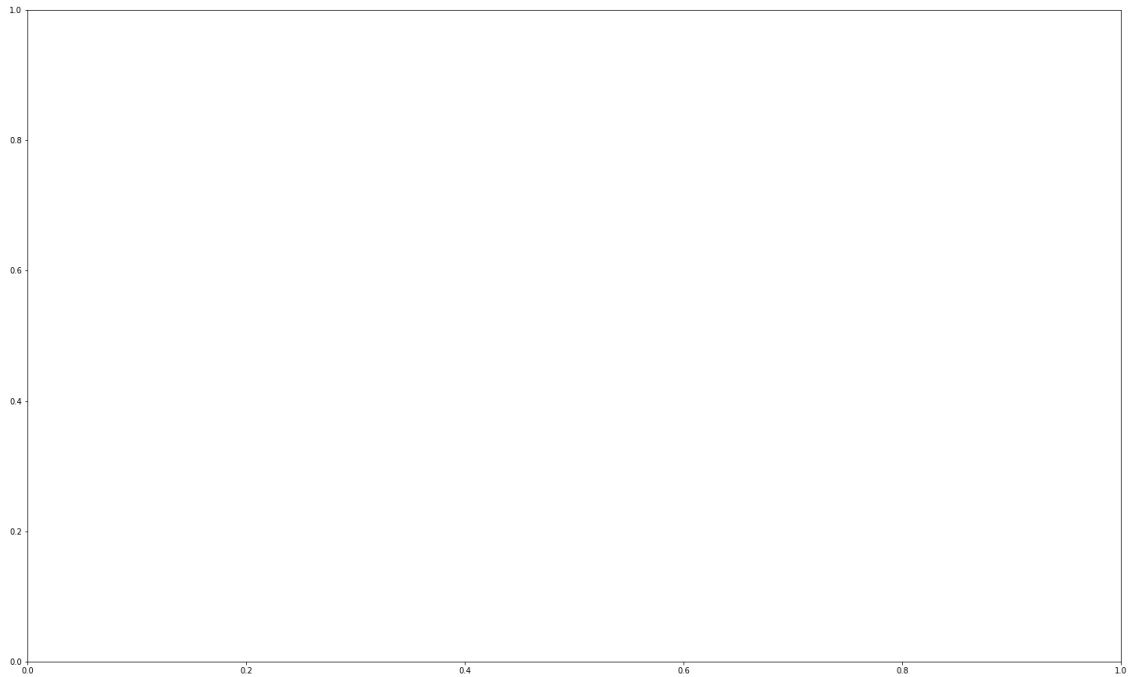
First, let us plot the whole image.

```python
fig, ax = subplots(figsize=(25, 15))

ax.imshow(img_col);
```

```
NameError: name 'img_col' is not defined
```

```
show()
```


The sample is labeled for Hoechst stain (blue), the Inositol trisphosphate (IP3) receptor (green) and Glial fibrillary acidic protein (GFAP) (red).

Now we can visualise the three colour channels individually by slicing the Numpy array. The stack with index 0 corresponds to 'red', index 1 corresponds to 'green' and index 2 corresponds to 'blue':

```
red_channel   = img_col[:, :, 0]
```

```
NameError: name 'img_col' is not defined
```

```python
green_channel = img_col[:, :, 1]
```

```
NameError: name 'img_col' is not defined
```

```python
blue_channel  = img_col[:, :, 2]
```

```
NameError: name 'img_col' is not defined
```

```python
fig, ax = subplots(nrows=1, ncols=3, figsize=(20, 10))

imgplot_red = ax[0].imshow(red_channel, cmap="Reds")
```

```python
imgplot_green = ax[1].imshow(green_channel, cmap="Greens")
```

```python
imgplot_blue = ax[2].imshow(blue_channel, cmap="Blues")
```

```python
fig.colorbar(imgplot_red, ax=ax[0], shrink=0.4)
```

```python
fig.colorbar(imgplot_green, ax=ax[1], shrink=0.4)
```

```python
fig.colorbar(imgplot_blue, ax=ax[2], shrink=0.4);
```

```python
show()
```
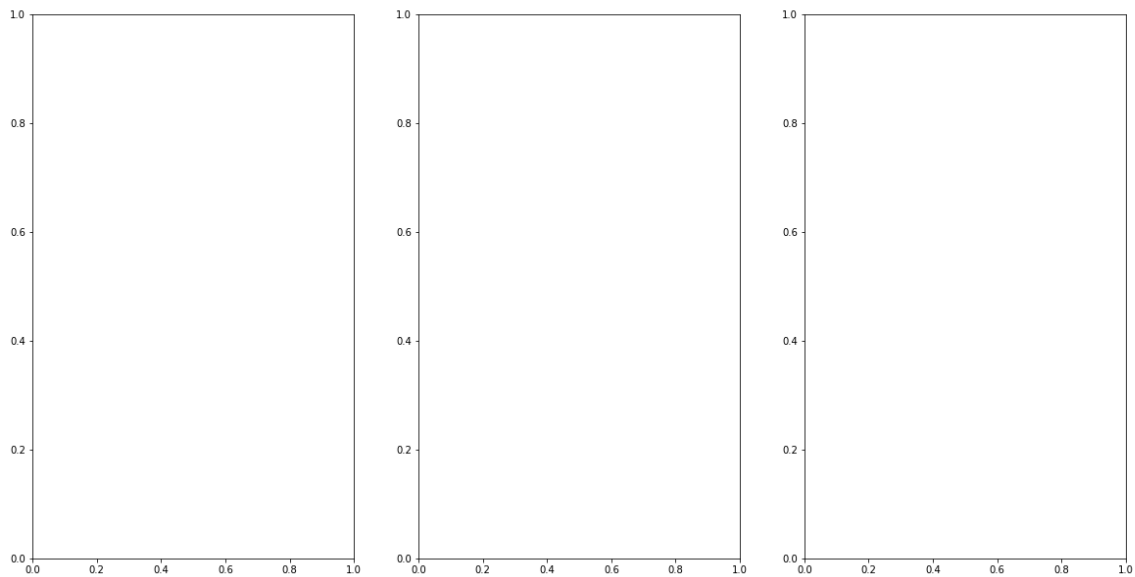

This shows what colour combinations each of the pixels is made up of. Notice that the intensities go up to 255. This is because RGB (red, green and blue) colours are defined within the range 0-255. This gives a total of 16,777,216 possible colour combinations!

We can plot histograms of each of the colour channels.

```python
fig, ax = subplots(nrows=1, ncols=3, figsize=(20, 5))

ax[0].hist(red_channel.flatten(), bins=50)
```

```python
ax[0].set_xlabel("Pixel intensity", fontsize=16)
ax[0].set_xlabel("Red channel")
ax[1].hist(green_channel.flatten(), bins=50)
```
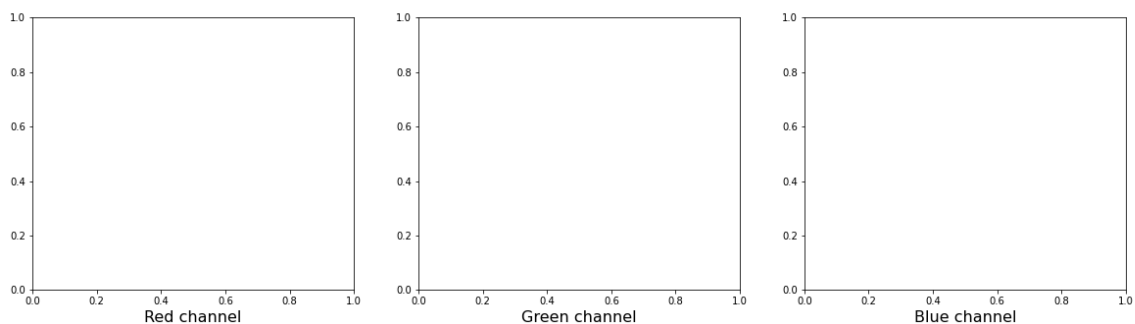
```python
ax[1].set_xlabel("Pixel intensity", fontsize=16)
ax[1].set_xlabel("Green channel")
ax[2].hist(blue_channel.flatten(), bins=50)
```

```python
ax[2].set_xlabel("Pixel intensity", fontsize=16)
ax[2].set_xlabel("Blue channel")

show()
```


Dealing with Large Images

Sometimes (or quite often, depending on the field of research), we have to deal with very large images that are composed of many pixels. It can be quite difficult to process these images, as they can require a lot of computer memory when they are processed. We will look at two different strategies for dealing with this problem: decreasing resolution and using patches from the original image. We will use the full-resolution version of the rat brain in the above example.

```python
img_hr = imread('fig/rat_brain.jpg')
img_hr.shape
```

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_brain.jpg'
NameError: name 'img_hr' is not defined
```

In fact, we can even get an warning from python that say something like "Image size (324649360 pixels) exceeds limit of 244158474 pixels, could be decompression bomb DOS attack." This refers to malicious files which are designed to crash or cause disruption by using up a lot of memory.

We can get around this by changing the maximum pixel limit as follows.

To do this, we import Image from the Python Image Library PIL:

```python
from PIL import Image

Image.MAX_IMAGE_PIXELS = 1000000000
```

Let's try again. Be patient, it might take a moment.

```python
img_hr = imread('fig/rat_brain.jpg')
```

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_brain.jpg'
```

```python
img_hr.shape
```

```
NameError: name 'img_hr' is not defined
```

Now we can plot the full high-resolution image:

```python
fig, ax = subplots(figsize=(25, 15))

ax.imshow(img_hr, cmap='gray');
```

```
NameError: name 'img_hr' is not defined
```

```python
show()
```

Although now we can plot this image, it consists of over 300 million pixels, and we could run into memory problems when trying to process it. One approach is simply to reduce the resolution. One way to do this is to import the image using Image from the PIL library that we imported above. This library gives us more tools to process images, including decreasing the resolution. It is a rich library with lots of useful tools. As always, having a look at the [documentation](#) and playing around is recommended!

We use `resize` to downsample the image:

PYTHON ‹ ›

```python
img_pil = Image.open('fig/rat_brain.jpg')
img_small = img_pil.resize((174, 187))

print(type(img_small))
```

OUTPUT ‹ ›

```
PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_brain.jpg'
NameError: name 'img_pil' is not defined
NameError: name 'img_small' is not defined
```
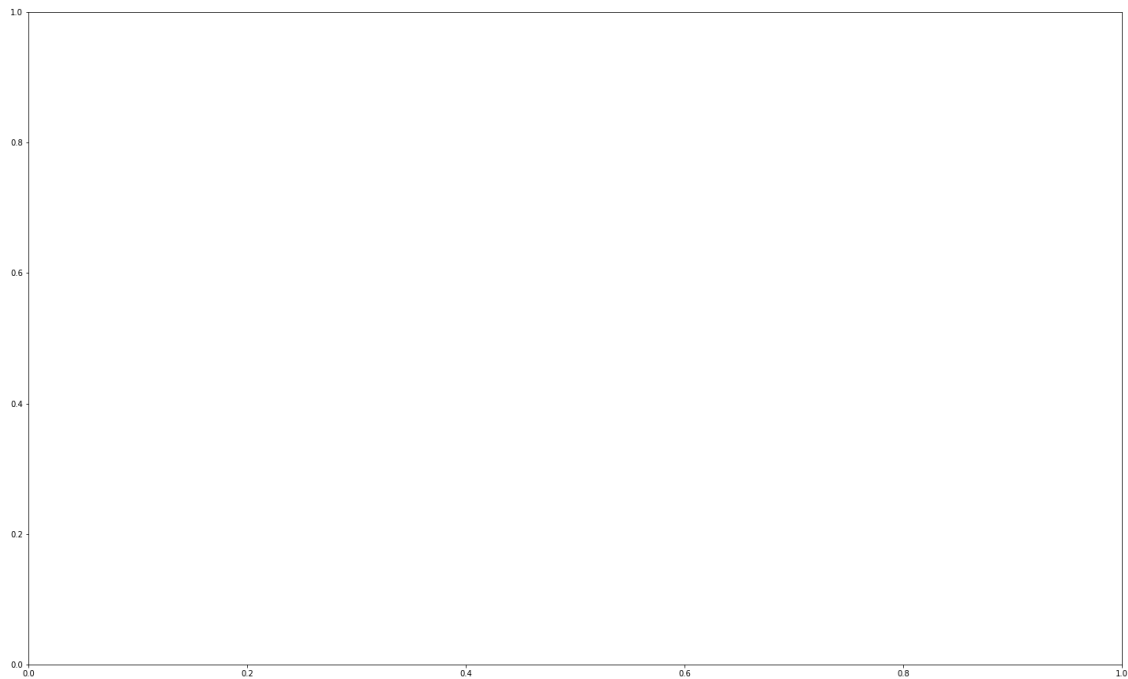
Plotting should now be quicker.

```python
fig, ax = subplots(figsize=(25, 15))

ax.imshow(img_small, cmap='gray');
```

```
NameError: name 'img_small' is not defined
```

```python
show()
```


With this code, we have resized the image to 174 by 187 pixels. We should be aware though, that our image is no longer in a Numpy array form but rather it now has type 'PIL.Image.Image'. We can, however, easily convert it back into a Numpy array using `array`, if we wish.

```python
from numpy import array

img_numpy = array(img_small)
```

```
NameError: name 'img_small' is not defined
```

```python
print(type(img_numpy))
```

```
NameError: name 'img_numpy' is not defined
```

Often, we like to have full resolution images, as resizing causes a loss of information. An alternative approach to downsampling that is commonly used is to *patch* the images, i.e. divide the picture up into smaller chunks, or patches.

For this, we can use functionality from the Scikit-Learn library.

```python
from sklearn.feature_extraction.image import extract_patches_2d
```

'extract_patches_2d' is used to extract parts of the image. The shape of each patch as well as maxiaml number of patches can be specified.

```python
patches = extract_patches_2d(img_hr, (174, 187), max_patches=100)
```

```
NameError: name 'img_hr' is not defined
```

```python
patches.shape
```

```
NameError: name 'patches' is not defined
```

Note that patching itself can be a memory-intensive task. Extracting lots and lots of patches might take a long time. To look at the patches we can use a for loop:

```python
fig, ax = subplots(nrows=10, ncols=10, figsize=(25, 25))

ax = ax.flatten()

for index in range(patches.shape[0]):
    ax[index].imshow(patches[index, :, :, :])
```
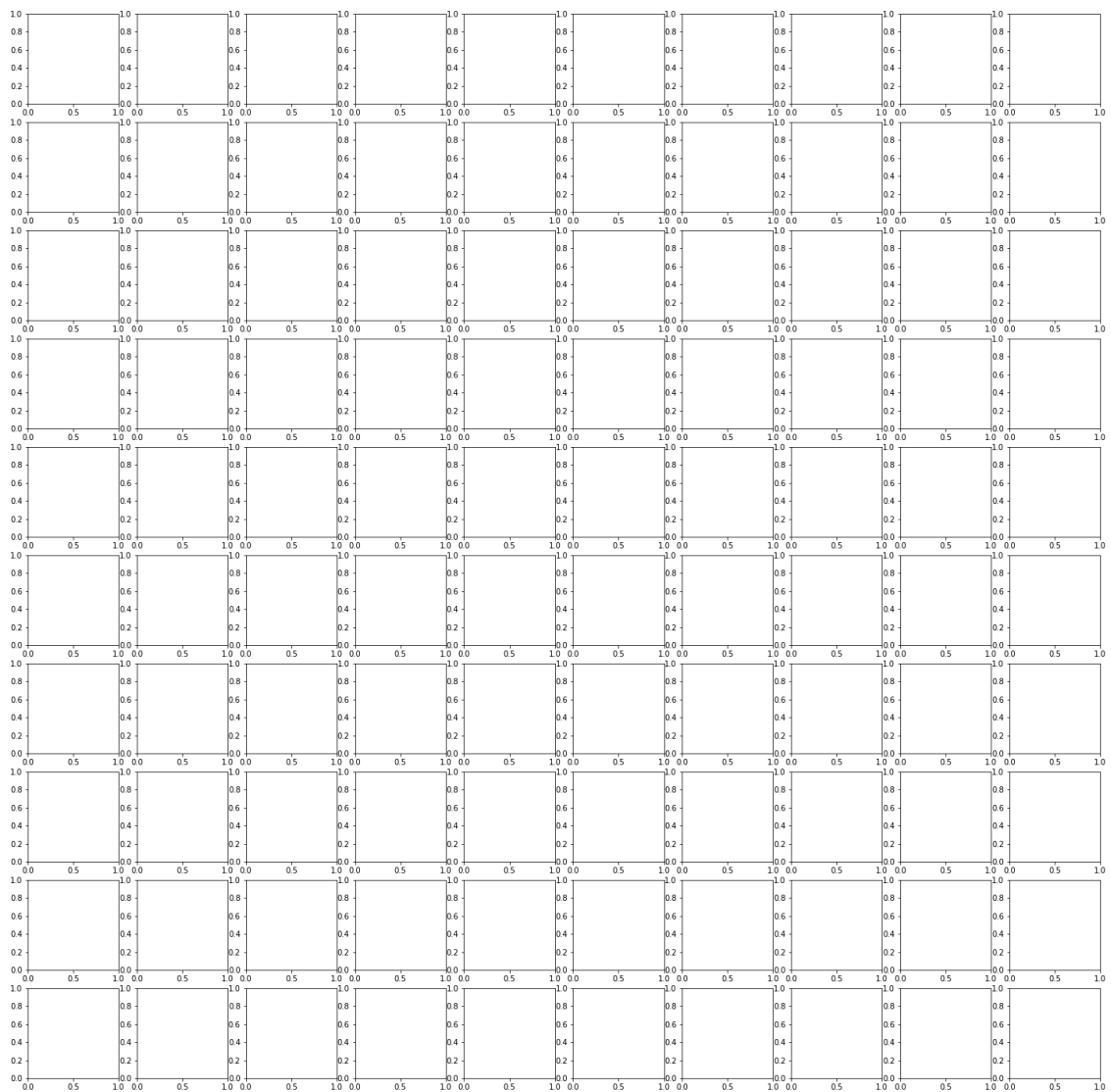
```
NameError: name 'patches' is not defined
```

```python
show()
```

Now, working with these smaller, individual patches will be much more manageable!

3D Images

Sometimes we might want to work with 3D images. A good example for this are MRI scans. These don't come as 'csv' format but in specialised image formats. One example is `nii`, the *Neuroimaging Informatics Technology Initiative (NIfTI)* open file format. For these types of images we will need special software. In particular, we will be using the open source library called **nibabel**. Documentation for this package is available at https://nipy.org/nibabel/.

As it is not contained in your Python installation by default, it needs to be installed first.

To install it, please run:

```
conda install -c conda-forge nibabel
```

in your command line or terminal if you have an **Anaconda distribution** of Python.

Alternatively, you can install it using:

```
pip install nibabel
```

in your command line or terminal.

```python
import nibabel as nib
```

The package is now available for use. If a function comes from that package, we call it by referring to the package using `nib`, followed by a period and the name of the function:

```python
img_3d = nib.load('fig/brain.nii')

img_data = img_3d.get_fdata()

print(type(img_data))
```

```
<class 'numpy.memmap'>
```

```python
print(img_data.shape)
```

```
(256, 256, 124)
```

We can see that this image has three dimensions, and a total of 256 x 256 x 124 volume pixels (or voxels). To visualise our image, we can plot one slice at a time. Below, we show three different slices, in the transverse direction (from chin to the top of the head. To access an image from the transverse direction, you pick a single value from the third dimension of the image:

```python
fig, ax = subplots(ncols=3, figsize=(25, 15))

p1 = ax[0].imshow(img_data[:, :, 60], cmap='gray')
p2 = ax[1].imshow(img_data[:, :, 75], cmap='gray')
p3 = ax[2].imshow(img_data[:, :, 90], cmap='gray')

fig.colorbar(p1, ax=ax[0], shrink=0.4)
```
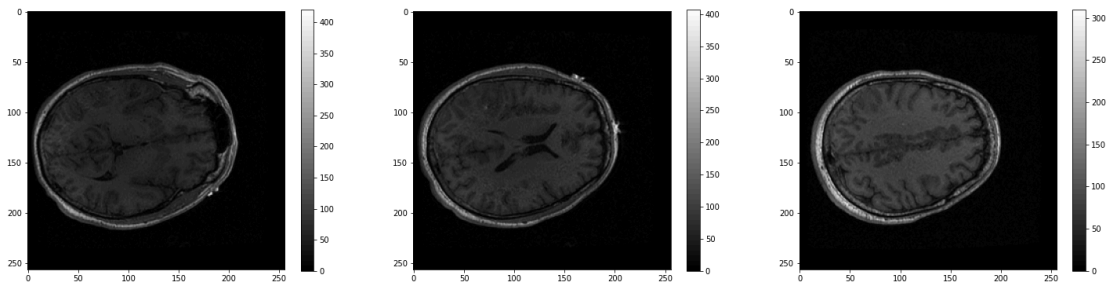
```python
fig.colorbar(p2, ax=ax[1], shrink=0.4)
```

```python
fig.colorbar(p3, ax=ax[2], shrink=0.4);

show()
```


These look fairly dark. We can improve the contrast, by adjusting the intensity range. This requires setting of the keyword arguments `vmin` and `vmax`.

`vmin` and `vmax` define the data range that the colormap (in our case the 'grey' map) covers. By default, the colormap covers the complete value range of the supplied data. For an image that will be somewhere between 0 and 255. If we

want to brighten up the darker shades of grey, we can reduce the value of `vmax`

Expanding the above code:

```python
fig, ax = subplots(ncols=3, figsize=(25, 15))

p1 = ax[0].imshow(img_data[:, :, 60], cmap='gray', vmin=0, vmax=150)
p2 = ax[1].imshow(img_data[:, :, 75], cmap='gray', vmin=0, vmax=150)
p3 = ax[2].imshow(img_data[:, :, 90], cmap='gray', vmin=0, vmax=150)

fig.colorbar(p1, ax=ax[0], shrink=0.4)
```

```python
fig.colorbar(p2, ax=ax[1], shrink=0.4)
```

```python
fig.colorbar(p3, ax=ax[2], shrink=0.4);

show()
```
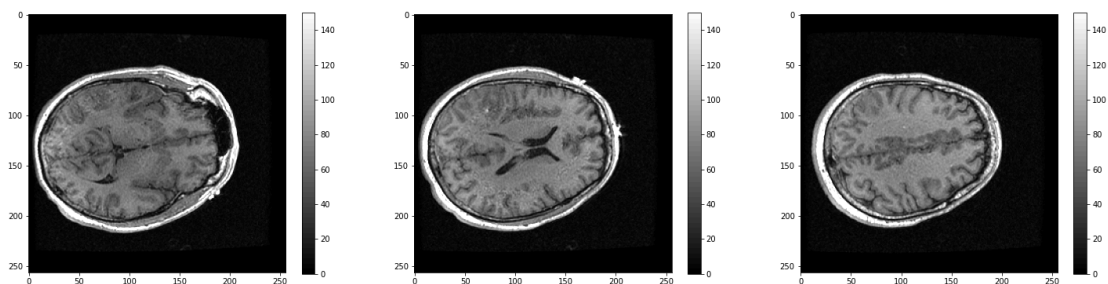

What about the other dimensions? We can also plot coronal and sagittal slices but note that the respective slices have different pixel resolution.

```python
fig, ax = subplots(nrows=3, ncols=5, figsize=(26, 18))

t1 = ax[0, 0].imshow(img_data[:, :, 45].T, cmap='gray', vmin=0, vmax=150, origin='lower')
t2 = ax[0, 1].imshow(img_data[:, :, 60].T, cmap='gray', vmin=0, vmax=150, origin='lower')
t3 = ax[0, 2].imshow(img_data[:, :, 75].T, cmap='gray', vmin=0, vmax=150, origin='lower')
t4 = ax[0, 3].imshow(img_data[:, :, 90].T, cmap='gray', vmin=0, vmax=150, origin='lower')
t5 = ax[0, 4].imshow(img_data[:, :, 105].T, cmap='gray', vmin=0, vmax=150, origin='lower')

c1 = ax[1, 0].imshow(img_data[:, 50, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
c2 = ax[1, 1].imshow(img_data[:, 75, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
c3 = ax[1, 2].imshow(img_data[:, 90, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
c4 = ax[1, 3].imshow(img_data[:, 105, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
c5 = ax[1, 4].imshow(img_data[:, 120, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')

s1 = ax[2, 0].imshow(img_data[75, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
s2 = ax[2, 1].imshow(img_data[90, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
s3 = ax[2, 2].imshow(img_data[105, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
s4 = ax[2, 3].imshow(img_data[120, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')
s5 = ax[2, 4].imshow(img_data[135, :, :].T, cmap='gray', vmin=0, vmax=150, origin='lower')

show()
```
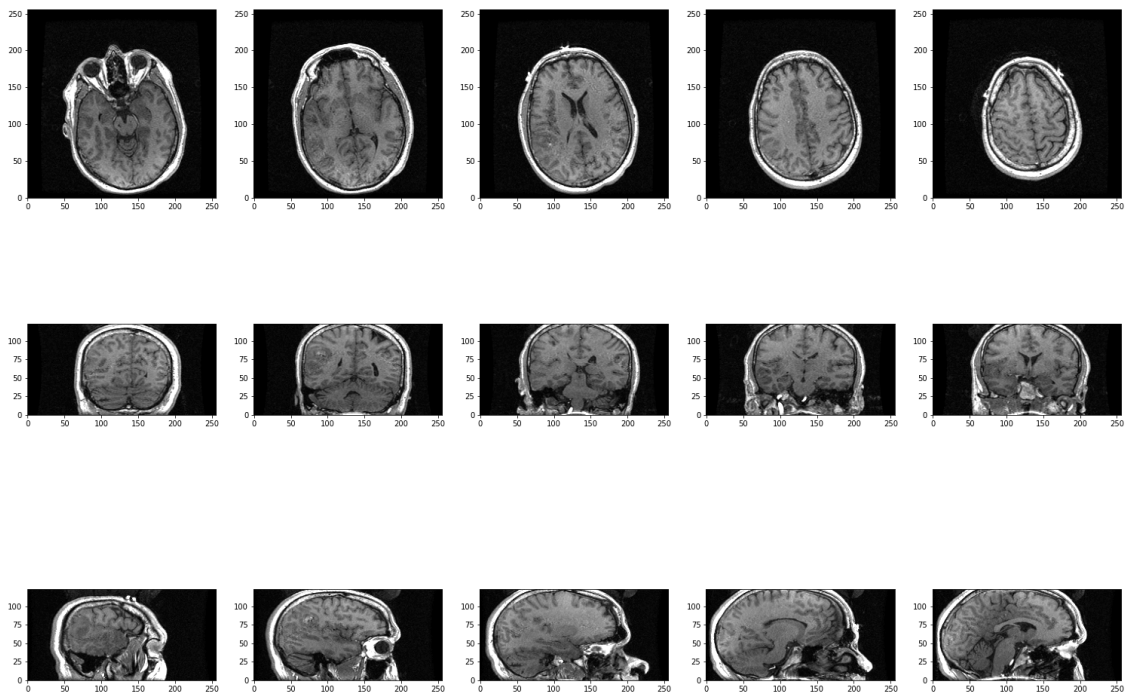
Now, we can see all three viewing planes for this 3-dimensional brain scan!

Exercises

Q1

```python
                                                    PYTHON < >

## Import the image
from matplotlib.image import imread

img_task = imread('fig/rat_cerebellum.jpg')
```

```
                                                    OUTPUT < >

PIL.UnidentifiedImageError: cannot identify image file 'fig/rat_cerebellum.jpg'
```
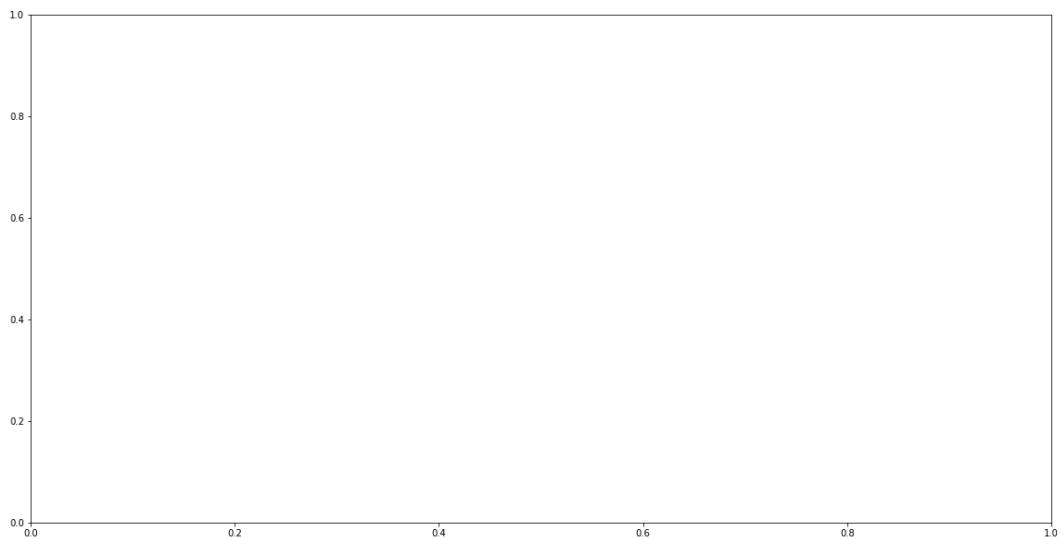
```python
                                                    PYTHON < >

## Display the image

from matplotlib.pyplot import subplots, show

fig, ax = subplots(figsize=(20, 10))

ax.imshow(img_task, cmap='gray');
```

```python
                                                    PYTHON < >

show()
```

Q2

```python
red_channel   = img_task[:, :, 0]
```

```python
green_channel = img_task[:, :, 1]
```

```python
blue_channel  = img_task[:, :, 2]
```

```python
fig, ax = subplots(ncols=3, figsize=(20, 5))

ax[0].hist(red_channel.flatten(), bins=50)
```
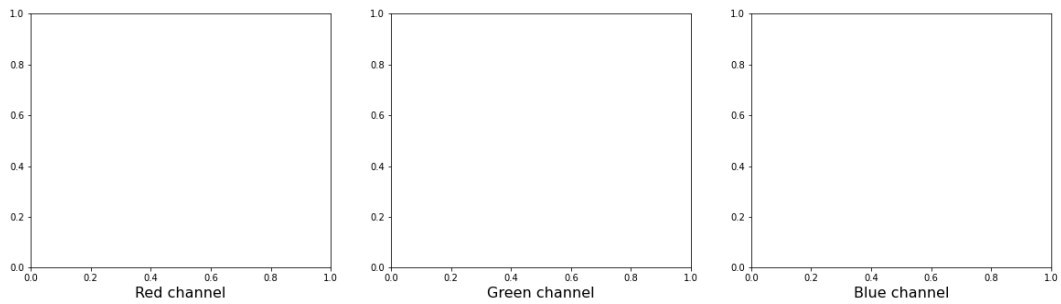
```python
ax[0].set_xlabel("Pixel intensity", fontsize=16)
ax[0].set_xlabel("Red channel")
ax[1].hist(green_channel.flatten(), bins=50)
```

```python
ax[1].set_xlabel("Pixel intensity", fontsize=16)
ax[1].set_xlabel("Green channel")
ax[2].hist(blue_channel.flatten(), bins=50)
```

```python
ax[2].set_xlabel("Pixel intensity", fontsize=16)
ax[2].set_xlabel("Blue channel");

show()
```

```python
fig, ax = subplots(ncols=3, figsize=(20, 10))

imgplot_red   = ax[0].imshow(red_channel, cmap="Reds")
```

```python
imgplot_green = ax[1].imshow(green_channel, cmap="Greens")
```

```python
imgplot_blue  = ax[2].imshow(blue_channel, cmap="Blues")
```

```python
fig.colorbar(imgplot_red,   ax=ax[0], shrink=0.5)
```
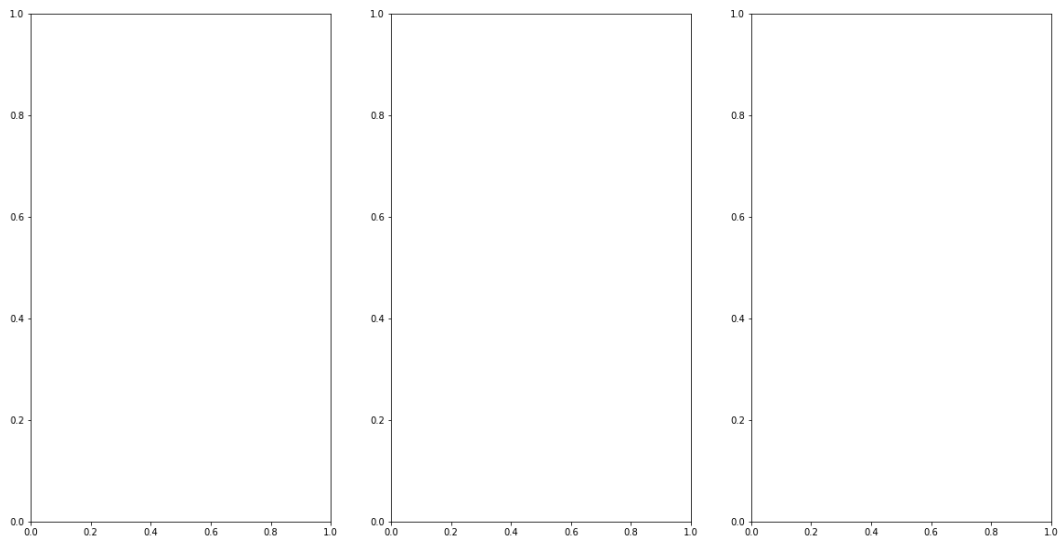
```python
fig.colorbar(imgplot_green, ax=ax[1], shrink=0.5)
```

```python
fig.colorbar(imgplot_blue,  ax=ax[2], shrink=0.5);
```

```python
show()
```

Q3-4

```python
red_mask   = red_channel   > 120
```

```python
green_mask = green_channel > 100
```

```python
blue_mask  = blue_channel  > 100
```

```python
red_masked   = red_channel*red_mask
```

```python
green_masked = green_channel*green_mask
```

```python
blue_masked  = blue_channel*blue_mask
```

```python
fig, ax = subplots(nrows=3, ncols=2, figsize=(18, 20))

ax[0, 0].imshow(red_mask, cmap='gray')
```

```python
ax[0, 0].set_title('Red binary mask', fontsize=16)
ax[0, 1].imshow(red_masked, cmap='Reds')
```

```python
ax[0, 1].set_title('Masked image', fontsize=16)
ax[1, 0].imshow(green_mask, cmap='gray')
```

```python
ax[1, 0].set_title('Green binary mask', fontsize=16)
ax[1, 1].imshow(green_masked, cmap='Greens')
```

```python
ax[1, 1].set_title('Masked image', fontsize=16)
ax[2, 0].imshow(blue_mask, cmap='gray')
```
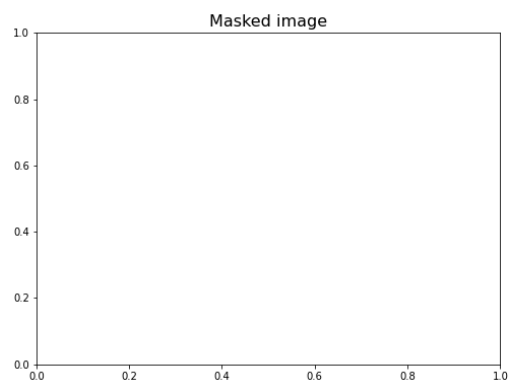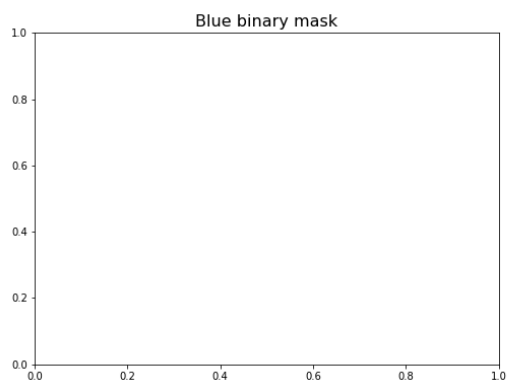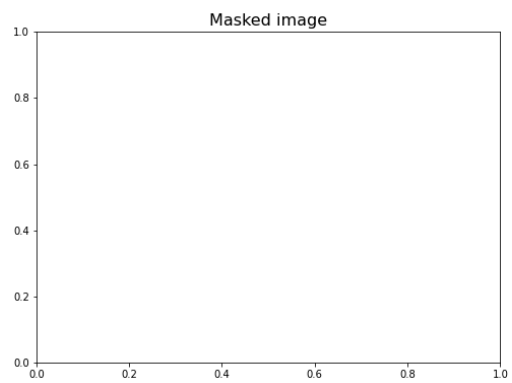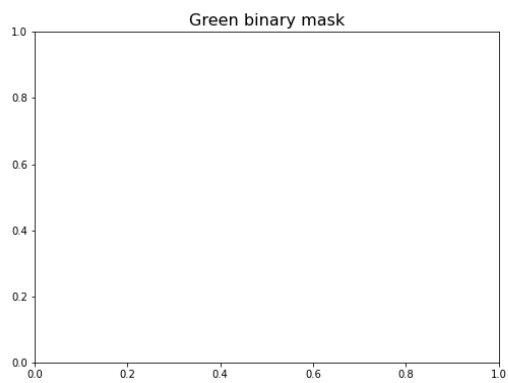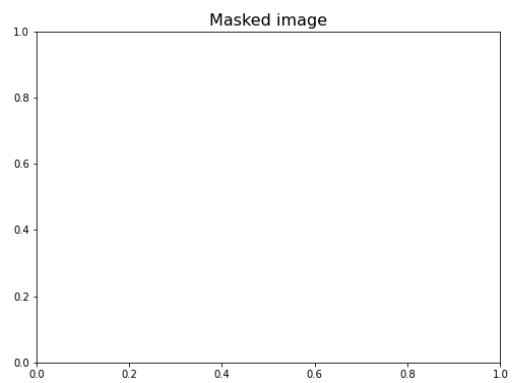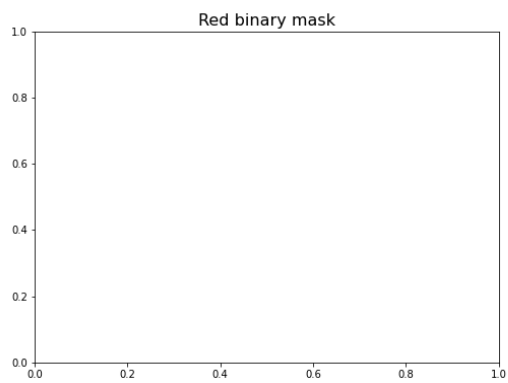
```python
ax[2, 0].set_title('Blue binary mask', fontsize=16)
ax[2, 1].imshow(blue_masked, cmap='Blues')
```

```python
ax[2, 1].set_title('Masked image', fontsize=16);

show()
```

Red binary mask

Masked image

Green binary mask

Masked image

Blue binary mask

Masked image

Q5

```python
from numpy import sum as numpy_sum
total_pixels = img_task.shape[0]*img_task.shape[1]

red_counts   = numpy_sum(red_mask)
green_counts = numpy_sum(green_mask)
blue_counts  = numpy_sum(blue_mask)
print("Approximately %d"%(red_counts/total_pixels*100)+"% of the image is synaptophysin")
print("Approximately %d"%(green_counts/total_pixels*100)+"% of the image is IP3R")
print("Approximately %d"%(blue_counts/total_pixels*100)+"% of the image is DNA")
```

```
NameError: name 'img_task' is not defined
NameError: name 'red_mask' is not defined
NameError: name 'green_mask' is not defined
NameError: name 'blue_mask' is not defined
NameError: name 'red_counts' is not defined
NameError: name 'green_counts' is not defined
NameError: name 'blue_counts' is not defined
```

Q6

```
[ad libitum]
```

KEY POINTS

- `imread` function can interpret many different image formats.

- Masking isolates pixels whose intensity value is below a certain threshold.

- The colour images are comprised of three channels (corresponding to red, green and blue intensities).

- Python Image Library (PIL) helps to set high pixel limit for larger images.