Digital Image Analysis with Python

This notebook is the course materials for a crash course in digital image analysis using Python. The course is developed by Simula (https://www.simula.no/) and hosted in collaboration with Tekna Helse and Teknologi (https://www.tekna.no/fag-og-nettverk/IKT/helse-og-teknologi/).

The goal of this course is to give a quick, but efficient, introduction to how you can work with digital images in Python to perform analysis or automate tasks. The course is planned to be roughly 3 hours long, and so it is obviously limited how in-depth we are able to go into any single topic. Rather, we focus on giving you an overview of different packages, tools and approaches you can use in Python - as well as resources you can use to learn more on your own.

Note that we do assume you have some experience with Python's basic syntax, but not that you are an expert programmer by any means. As programming is a skill you need to apply to properly learn, we have also set aside some time for you to try your hand at applying examples as we go.

Course Plan

Part 0 - An example to start with

- A sneak peak of what is to come

Part 1 - Information on working in Python

- A quick introduction to `jupyter`
- An overview of Python packages for working with images
- How to install packages using `pip`

Part 2 - Digital images in Python

- How to read and save images in Python
- How to plot images (with `matplotlib` or `plotly.express`)
- How images are represented as `numpy` arrays
- How to work on images using `numpy` operations

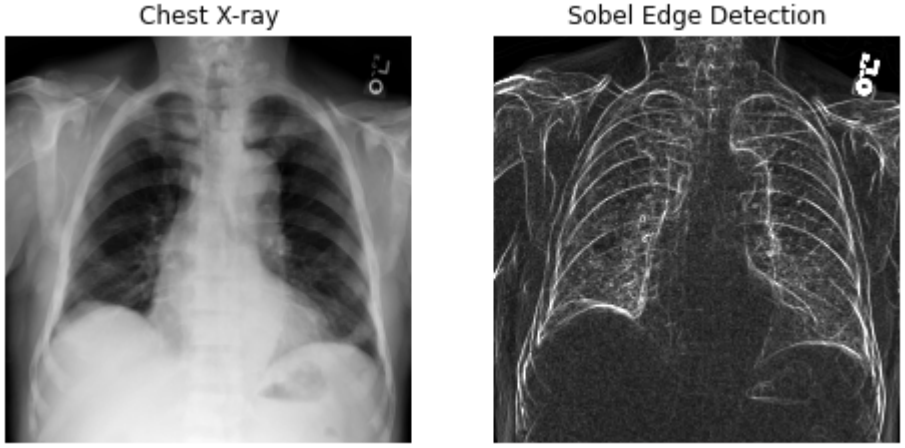
Part 3 - Digital image analysis

- Histograms and contrast enhancement
- Thresholding
- Denoising
- Edge detection
- Deconvolution

Part 0 - An example to start of with

Before we get into any details, let us start of with a quick example. We will return to this exact example a bit later on, so the goal isn't that you understand all the details of what is going on - it is more to show you an example of some Python code working with an image.

In [1]:
```
# Import packages needed to work with images
from skimage import io, filters
import matplotlib.pyplot as plt

# Read in an x-ray image
img = io.imread("img/chest_000.png")

# Plot the original image
plt.figure(figsize=(8, 6))
plt.subplot(1, 2, 1)
plt.imshow(img, cmap='gray')
plt.title('Chest X-ray')
plt.axis('off')

# Apply a Sobel filter to detect edges
edges = filters.sobel(img)

# Plot edges as another image
plt.subplot(1, 2, 2)
plt.imshow(edges, cmap='gray', vmax=0.05)
plt.title('Sobel Edge Detection')
plt.axis('off')
plt.show()
```



This code reads in a Chest x-ray stored as a `.png` image and displays it. Then we use an algorithm known as the Sobel filter. The Sobel filter is an example of *edge detection*, a common digital image analysis technique.

We have split our code into five different segments, with a Python-comment explaining what each does. Again, at this point it is *not* the point that you understand what each code line does, or how it works. Try instead to focus on the code as a whole, and the step-by-step approach we use, which can be summarized as:

- Import necessary packages and functions
- Read in the data or images we want to work with
- Perform some analysis or computation

- Display and/or save the results

As mentioned we will get back to this exact example later in the course. For now, let us take a step back and talk a bit more about how it is to work with Python from a practical point of view.

Part 1 - Information on working in Python

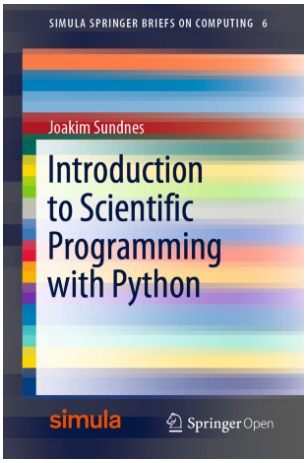
Python is a so-called *high-level* scripting language, meaning it has a high level of abstraction and we can typically perform a lot in a few lines of code without having to worry to much about technical details that happen behind the scences. It is also a programming language that is considered quite readable and easy to use, making it perfect for beginners and novice programmers.

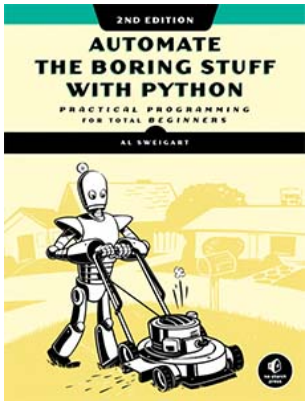
For these reasons, Python has quickly grown to become one of the most popular programming languages for scientific computing, data analysis, machine learning and digital image analysis.

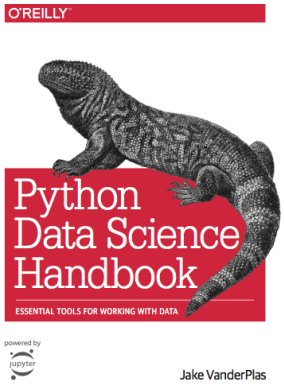
We won't talk too much about Python basics in this course, and assume you have some knowledge from before. If you don't, or you want to learn more Python basics, here are a few good open source resources online.

(Free) digital books

- Introduction to Scientific Programming with Python (https://doi.org/10.1007/978-3-030-50356-7) by Joakim Sundnes. This is a so-called *Springer Brief*, which aims to give a concise introduction to scientific programming with Python, aimed at readers with little, to no, background in programming.
- Automate the Boring Stuff (https://automatetheboringstuff.com/) by Al Sweigart. This is a free-to-read ebook about using Python for automating small tasks.
- The Python Data Science Handbook (https://jakevdp.github.io/PythonDataScienceHandbook/) by Jake VanderPlas. This free-to-read ebook assumes you know Python basics, but is a great resource if you want to learn more about working with data, data analysis and machine learning with Python.







A note on text-editors for Python and using Jupyter

Python is a programming language, and like most langugages, we can use many different programs and text editors when working with Python code. Typically one uses a text editor to write or change code, and then run the program either directly from the text editor, or through a console. An alternative is to run a so-called *integrated developer enviroment* (IDE), which are more jazzed up and feature rich text editors. Some popular text editors and IDE's for working with Python are

- Visual Studio Code (https://code.visualstudio.com/)
- PyCharm (https://www.jetbrains.com/pycharm/)
- SublimeText (https://www.sublimetext.com/)
- Spyder (https://www.spyder-ide.org/)
- Atom (https://atom.io/)
- ...and many, many more

If you already have a preferred text-editor to use, please feel free to stick to that one in this course. If you do not, we usually recommend using Spyder in our courses. This isn't because we consider Spyder to be the best, but we consider it to be the easiest to install and use for beginners, as you can get it a long with a complete Python-installation if you install the Anaconda (https://www.anaconda.com/) package.

Another alternative altogether is to work in a tool called *Jupyter*. This is not a traditional text editor, but a tool that allows for combining traditional text with code that is executable. It can be especially nice to use when creating tutorials and the like, and is what we have used to create these lecture notes. Jupyter has become a popular way to work with Python scripting, especially for scientific computing and data science, as data can be explored and illustrated interactively while working with it.

We present the material using Jupyter notebooks, and you can easily use Jupyter for yourself for working with digital image analysis if you so prefer. Let us very briefly show how Jupyer works before we move on to working with actual digital images.

Jupyter Notebooks and Jupyterlab

When we work with Python, we typically make files containing Python-code that can be run at a later time, and these files are given a *.py* file ending. When working in Jupyter, we do not make normal Python scripts, but rather *notebook* files, which are given a *.ipynb* file ending. These are not traditional scripts or programs, but rather a combination of rich text, math, figures and code all contained in the same file. You can then interact with this notebook using your browser, which enables Jupyter to use HTML and other useful features.

To use Jupyter you first need to install it on your machine, which is most easily done through installing the Anaconda (https://www.anaconda.com/) package. If you have done so, you should have access to a program called *Jupyer Notebook* from your start menu (on Windows). Running this program will start a Python-kernal and webserver in the background and open a tab in your default web browser to interact with it.

The Dashboard

When you first open Jupyter Notebook you are on a dashboard. You can then navigate to an existing notebook file on your computer, or start a new notebook to work on. If you want to open up these resources in Jupyter, you first need to download them from the github pages (the easiest

option, unless you know how to use git, is to click the green `Code` button and then select `Download Zip`, which you then unpack. Then you can navigate to wherever you unpacked this folder and open the notebook. The benefit of opening the lecture notes in Jupyter is that you can then run all the examples, play around with them. You can for example copy examples, change the code or extend them.

Working with code in notebooks

The basic idea of a Jupyter notebook is that the document is divided into a series of *cells*. A cell is a collection of content, and are split into two main types. *Markdown cells* contain content we associate with a typical text, such as written text, math, figures, tables and so on. *Code cells* contain code which can be executed, in which the output is written directly into the notebook, right below the code itself.

When navigating through a notebook, you will always be in one of two modes. Either you are actively making edits to an existing cell, this is known as *edit mode*. When you are in edit mode, the cell you are currently editing is marked in a green box, and you have a blinking cursor inside the cell. When we are not actively editing a cell, we are in *command mode*. We see we are in command mode because the active cell is marked in a blue box, not a green one, and there is no cursor. When in command mode, pressing the keyboard will invoke commands through keyboard shortcuts, such as generating new cells. When in command mode, you can enter edit mode on a given cell by double clicking it, or by pressing enter. When in edit mode, you can exit by clicking outside the cell with your cursor, or by pressing escape.

Example: Creating and running a code cell

Let us generate a code cell, write some code, and then run it. First we need to generate a new cell. Here you can either press the `+` button in the toolbar on the top, or simply use the keyboard shortcut `b`. This will generate a new cell. We see that the cell is a code cell, because it has the stylized `In []` to its left.

In [ ]:
```
1
```

You can now click the empty cell to enter edit mode, and start writing some Python code. Now, we haven't started looking at Python code yet, so let us start with the simplest possible example, a `print` command:

In [2]:
```
1 print("Hello, World!")
```

```
Hello, World!
```

We have now entered some code into our cell, but if we simply exit edit-mode, the code will not run automatically. To run code inside the notebook, we must *execute* a given code cell. You can do this either by clicking the `Run` command in the toolbar at the top, or use the more efficient keyboard shortcuts `Ctrl+Enter` or `Shift+Enter`.

When you execute a cell like this, all the code inside the cell is executed and the output is printed immediately below the cell. In addition, the `In []:` will update to note which execution this was in the notebook, as this is the first cell we execute, it will become `In [1]:`.

Execution order of cells

One element of notebooks that can be a bit confusing to beginners and experts alike is that a notebook is one big interactive session. This means that variables and results are remembered after running a cell, and can be used further down in a notebook, in different code cells.

However, when shutting down the notebook altogether and reloading it, this interactive data is forgotten. When opening a notebook, one can often *not* run a single cell in the middle or towards the end of the notebook, rather, one should run all the cells in the notebook from the start. A helpful tool here is to use the `Cell > Run all` option from the toolbar, which will run all cells in order, and load everything into memory. Alternatively you can use the `Kernel > Restart and Run All` if you want to completely reset and then run all cells in order.

A more thorough introduction to Jupyter Notebooks

We have now covered how you can create code cells, write code and execute it in Jupyter. We won't cover how to write other types of text in Jupyter here, as it is now too important for the topic at hand. If you want a more detailed introduction to Jupyter, you can take a look at this guide to Jupyter basics (https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Notebook%20Basics.html) by the original creators.

Useful Python-packages for working with Digital Image Analysis

Python is a *general* programming language, and is not specialzed for working with scientific applications or digital image analysis for that matter. One consequence of this is that it has been designed to be quite simple and minimialist in its basic features. This means that by default you have access to quite few built-in keywords and functions in Python. The major idea here is then that when programming in Python you are supposed to rely on additional libraries and packages to provide additional features as needed. To use these we simply need to *import* them, before we use them. In a traditional Python-script, we simply do this at the top of our *.py*-file, while in Jupyter we typically either place it early in the notebook in its own cell, or we place the import directly in the cell where the code is used.

Installing packages with pip

Python ships with quite a few *standard libraries*, which are useful to know about, for instance *math* for mathematical functions and *random* for generating random numbers. But there is a much bigger work of open source developed packages developed for Python we can use. Typically these are shared through a site called the Python Packaging Index, or *pypi* for short:

- https://pypi.org/ (https://pypi.org/)

When we want to install packages shared through *pypi*, we typically use a tool called *pip*, which makes installing packages borderline trivial (under most cases at least, occasionally installing things gets annoyingly tricky in IT). Say for instance you want to install the package `scikit-image` (which we will introduce shortly), then we simply run pip. We can either invoke `pip` through a command-line by writing

- `python -m pip install scikit-image`

or if you are working in Jupyter or Spyder you can invoke `pip` directly from the program by writing

- `%pip install scipy` in a code cell (for Jupyter) or directly into the iPython console (in Spyder). Here the percent-sign indicates what we call "cell magic", which are some nice extra features we can invoke in Jupyter.

In [3]:
```
1 %pip install scikit-image
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: scikit-image in /home/jonas/.local/lib/python3.
8/site-packages (0.19.2)
Requirement already satisfied: scipy>=1.4.1 in /home/jonas/.local/lib/python3.
8/site-packages (from scikit-image) (1.8.0)
Requirement already satisfied: networkx>=2.2 in /home/jonas/.local/lib/python3.
8/site-packages (from scikit-image) (2.7.1)
Requirement already satisfied: PyWavelets>=1.1.1 in /home/jonas/.local/lib/pyth
on3.8/site-packages (from scikit-image) (1.2.0)
Requirement already satisfied: packaging>=20.0 in /home/jonas/.local/lib/python
3.8/site-packages (from scikit-image) (21.3)
Requirement already satisfied: pillow!=7.1.0,!=7.1.1,!=8.3.0,>=6.1.0 in /home/j
onas/.local/lib/python3.8/site-packages (from scikit-image) (9.0.1)
Requirement already satisfied: numpy>=1.17.0 in /home/jonas/.local/lib/python3.
8/site-packages (from scikit-image) (1.22.2)
Requirement already satisfied: imageio>=2.4.1 in /home/jonas/.local/lib/python
3.8/site-packages (from scikit-image) (2.16.1)
Requirement already satisfied: tifffile>=2019.7.26 in /home/jonas/.local/lib/py
thon3.8/site-packages (from scikit-image) (2022.2.9)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /home/jonas/.local/l
ib/python3.8/site-packages (from packaging>=20.0->scikit-image) (3.0.7)
Note: you may need to restart the kernel to use updated packages.
```

Running the pip cell above will install `scikit-image`, and its dependencies, on your system if you don't already have it. If you *do* have it, it will let you know so by saying "requirement already satisfied". Note that if you installed through *Anaconda* you get a bunch of packages for scientific computing pre-installed, meaning you might very well already have many packages installed even if you have never touched `pip`.

Perhaps the most important package to know about: `numpy` (*Numerical Python*)

The first, and perhaps most important, package we want to mention is `numpy`. This is one of the most important packages to use in Python when working with *numerical data*. An all digital images are numerical data. When we use `numpy` we get access to a very important data type called

`arrays`, which are used to represent numerical data in the form of matrises and tensors.

Python is a high level programming language, and put simply, it is actually quite a slow language. However, `numpy` mostly relised on underlying code written in C, and manipulating larger sets of numerical data through numpy arrays is typically much faster than doing so directly with Python code. If you for example are working with linear algebra operations such as matrix-matrix multiplication, `numpy` is a crucial tool in doing such operations in a timely fashion.

Most digital image analysis libraries for Python are built on top of `numpy`, and use `numpy` for its mathematical operations. It is therefore very useful to have a good understanding of how `numpy` and its `arrays` work. We will therefore shortly look closer at how digital images are represented as arrays, and how we can work with these.

Let us now show how a Python-package is imported

In [4]:
```
1 import numpy as np
```

Here we import the package `numpy` by writing `import numpy`. However, we choose to do so under an *alias* by also writing `as np`. The second part isn't needed, but it has become a convention to use the shorthand `np`, to save ourselves some writing later on. If this code executes without a warning, it means you have a package called `numpy` installed on your system.

Once the packcage is installed, we can use functions it contains by writing the name of the package or module, then a dot, then the name of the function. for example:

In [5]:
```
1 empty = np.zeros((400, 400))
```

In this example we use the function `np.zeros`, which is contained in the `numpy` package. This will produce a 400x400 grid of zeroes, which corresponds to an empty picture.

If you try to import a package you have not installed, you will get an error message. Let us try:

In [6]:
```
1 import simula
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-6-d6b50bea82e5> in <module>
----> 1 import simula

ModuleNotFoundError: No module named 'simula'
```

The code fails with a `ModuleNotFoundError`. A *module*, in Python-lingo, is more or less equivalent to a file.

If you get such an error with any of our examples later in this course, then try to install the package using `pip` first, and then rerun the example.

The Scikit-image package for working with digital images

The next package we cover is the `scikit-image` package. Which is an open source digital image analysis package developed especially for Python by Stéfan van der Walt et al., who also developed the much used `scipy` (scientific python) package. When importing this package, we actually need to import it as `skimage`.

In [7]:
```
1 import skimage
```

Scikit-image is by no means the only package for digital image analysis. But it is the one we will focus on the most in this course. The main reason we focus on scikit-image is that it is developed especially for Python, and thus it is quite often the simplest to actually use. It also has very nice webpages with plenty of examples you can use to learn more about the package and how to use it on your own after the course.

- https://scikit-image.org/ (https://scikit-image.org/)

OpenCV

Another very popular package for image analysis in Python is called *OpenCV* (CV stands for *computer vision*). Unlike scikit-image, OpenCV is a package made for multiple programming languages, and so the Python-version is simply one version. You can read more about the package here

- https://opencv.org/ (https://opencv.org/)

To install OpenCV through `pip`, it is actually the package called `opencv-python` you should install (You can read more here: https://pypi.org/project/opencv-python/ (https://pypi.org/project/opencv-python/)). Then once you have installed it, you import it under the name `cv2`

In [8]:
```
1 %pip install opencv-python
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: opencv-python in /home/jonas/.local/lib/python3.
8/site-packages (4.5.5.64)
Requirement already satisfied: numpy>=1.14.5 in /home/jonas/.local/lib/python3.
8/site-packages (from opencv-python) (1.22.2)
Note: you may need to restart the kernel to use updated packages.
```

In [9]:
```
1 import cv2
```

OpenCV is a powerful package with plenty of features. However, we recommend trying to use scikit-image first, unless you have specific reasons for going the OpenCV route. This is because we find scikit-image to be more lightweight and easy to install and use than OpenCV, which can be a bit tricker to find documentation for and to use in practice.

Other packages for digital image analysis

We consider Scikit-image and OpenCV to be the most important packages for digital image analysis in Python. However, many other packages exist, which are either more specialized to a given topic, or they might be older packages that have fallen a bit out of popular use. One example is an older package that is no longer maintained called PIL (Python Image Library), which is no longer maintained, but which you still find in some older examples you might find online. Using PIL isn't a cardinal sin or anything, but might not be the preferred approach any longer.

For specific applications or specific operations/algorithms there might definitely be very useful packages out there. Giving general advice about *specific* applications is of course always tricky. You will therefore generally need to to more research on your own to find the packages that works best for you depending on your own needs and desires. Two examples that might be useful for you could for instance be

- `nipy` https://nipy.org/ (https://nipy.org/) A package for working with neuroimaging in Python
- `dipy` https://dipy.org/ (https://dipy.org/) A package for working with Diffusion MRI

How to plot images (`Matplotlib`)

Depending on what package you use for working with digital images themselves, they might come with built in functions for displaying images to the screen (typically referred to as "plotting"). For example, if we use `OpenCV`, this comes with its own function for displaying an image. However, most packages do not contain such functionality. Therefore its also useful to know how to plot images, and to do this we typically use the most popular package for plotting in Python, which is the Python mathematical plottling library, or `Matplotlib` for short.

- https://matplotlib.org/ (https://matplotlib.org/)

This package is used for much, much more than simply displaying digital images, but you can look more into that by going to their webpages and looking at their examples pages.

In our course, we will mainly focus on using the combination of `skimage + matplotlib` to analyse and plot digital images.

In [10]:
```
1 import matplotlib.pyplot as plt
```

Reading Medical Image Formats (`pydicom`)

If we are working with medical images, these often come in image formats that are not supported by more general image analysis packages. The most important example being DICOM (dicomstandard.org)-images. To work with such formats, we will need to install an additional package that can read and write such a format. In the case of DICOM, this can be done with a package called `pydicom` (https://pypi.org/project/pydicom/).

Another often used format, especially in neuroimaging, is the NIfTI format. To read such a format we can use the `nibabel` package (The `nipy` / `dipy` packages use `nibabel` behind the scenes).

If you have images or data in other formats that are not supported by the standard Python packages, a quick google search will typically point you in the right direction. For example, I collaborated with an OUS imaging lab using Zeiss microscopes, which produce image files in a propreitary format called CZI, so I googled "python read czi file" and quickly found my way to the package `czifile` for doing this.

Summary on Packages (Don't Panic!)

Ok. We have now just given you a *lot* of information on different packages. But please don't panic! You don't have to go around remembering all the different packages that exist and what they are used for. Rather, you should search up features as you need them, find an appropriate package that has that functionality, and then apply as needed.

For now, the most important packages are

- `numpy` for its `array` type
- `skimage` for digital image analysis
- `matplotlib` for plotting images

And `pydicom` if you are working with DICOM images.

Part 2 - Digital images in Python

Enough information and tlak! Let us get to working with actual images! Let us start of by looking at how we can read images, and how these are actually represented in Python.

Reading and plotting images

If we are reading in an image that is stored in a traditional, general image format like `.png`, `.jpg` or `.tif`, we can simply use `skimage` directly. In this case we load in the `io` submodule (`io` stands for input/output), and use the function `imread` (short for *image read*).

When we use the `imread` function we must apply a *path*, which tells Python where the image we want to load in is located on disk (or it can be the URL to an image located on the net). If you are not used to writing paths, it is simply a string saying where the file is located. You can specify a path in two ways:

- Relative path
- Absolute path

A relative path is specified in relation to the script or notebook you are writing the code and running it. They are usually short to write and easy to work with. They are also nice when sharing code, as they can work just as well for others, as long as they place their images in the same manner as you. So for a given project you could store the image and the code in the same folder, and it will work.

An absolute path is the full and unique path to the files specific location path on your machine. On a Windows path, it will usually start with "C:" if it is located on the C-disk. You can easily find the absolute path of any file by using your file explorer and shift right-clicking any file and selecting "Copy as path", in which case you can paste in the full path into your code. Absolute paths are usually long, and they are applicable only to your specific machine - so sharing code with absolute paths means it won't work for others.

In this course we assume you download the Github folder and run the notebook and make examples in the same folder. Meaning relative paths are easy. All our example images are located in a subfolder we call `img`.

In [11]:
```
1 from skimage import io
2
3 img = io.imread("img/brain_tumors/glioma3.jpg")
```

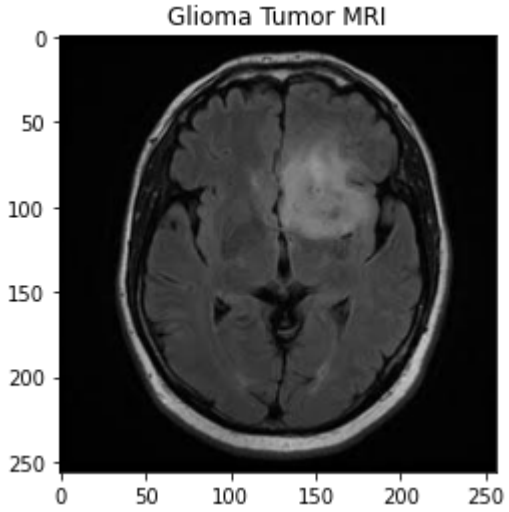
Tip: In Jupyter (and some other editors) you can get help to fill in the path as you are writing it by clicking the Tab button as you write.

Here we specify that from the folder our notebook is stored in, we first enter the `img` folder, then the `brain_tumors` subfolder, then find the file called `glioma3.jpg`. This is an example of a MRI scan of a glioma tumor, which I took from an open dataset on MRI scans of brain tumors (https://www.kaggle.com/datasets/sartajbhuvaji/brain-tumor-classification-mri?resource=download) found on the website Kaggle.com - a site for sharing larger open source datasets and notebooks analysing them.

When we use the `io.imread` function, it gets the correct file on the disk and reads the image information. We then specify that the information found are to be stored in a Python object, which we have chosen to call `img` here. Thus we need the `img =` part of the code line to declare a variable to keep the data around.

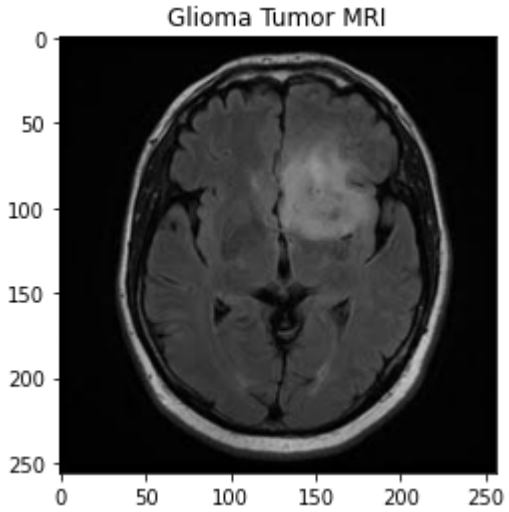
To plot the actual image we use `matplotlib.pyplot` (typically imported under the shorthand `plt`. It has a function `imshow` which shows an image object. After using `imshow` we also need to call `plt.show()` (without arguments) to show the final image.

In [12]:
```
1 plt.imshow(img)
2 plt.title('Glioma Tumor MRI')
3 plt.show()
```



Note that the image automatically shows in *false color*, this is because the default color scheme in matplotlib is not in greyscale. If you prefer a greyscale image, you can simply add the colormap as a so-called keyword optional argument as follows

In [13]:
```
1 plt.imshow(img, cmap='gray')
2 plt.title('Glioma Tumor MRI')
3 plt.show()
```



Let us explore this `img` variable a bit. It is actually a `numpy` array

In [14]:
```
1 print(type(img))
```

```
<class 'numpy.ndarray'>
```

As mentioned earlier, `numpy`, is a Python package for working with numerical data. `ndarray` is the specific data type, which stands for n-dimensional array. It represents a grid of numerical

numbers, which can represent a vector, a matrix, tensor or in this case, a digital image.

We can acces properties such as `ndim`, `shape` and `dtype` to get more information about the array

In [15]:
```
print(img.ndim)
```

```
3
```

Our array is two dimensional. This is because our image is 2D.

In [16]:
```
print(img.shape)
```

```
(256, 256, 3)
```

Our image consists of 1024 elements in each dimension, meaning we have a 1024×1024 pixel image.

In [17]:
```
print(img.dtype)
```

```
uint8
```

The `dtype` is a bit more technical. It is the *data type* of each individual element in the array, i.e., each pixel. In this case the image is an `uint8` image. This stands for `unsigned integer, 8 bits`. Simply put this means that the array consists of numbers that are between 0 (for black pixels) to 255 (for white pixels).

Some images use floating point numbers (i.e., decimals) rather than integers. It is then typical to use values in the interval 0 to 1.

You can read more about data types of pixels here:

- https://scikit-image.org/docs/stable/user_guide/data_types.html (https://scikit-image.org/docs/stable/user_guide/data_types.html)

If we try to print out the array we don't get the full thing, but we can see that it is indeed an array of values

In [18]:
```
1 print(img)
```

```
[[[0 0 0]
  [0 0 0]
  [0 0 0]
  ...
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  ...
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[0 0 0]
  [0 0 0]
  [0 0 0]
  ...
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 ...

 [[1 1 1]
  [1 1 1]
  [1 1 1]
  ...
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[1 1 1]
  [1 1 1]
  [1 1 1]
  ...
  [0 0 0]
  [0 0 0]
  [0 0 0]]

 [[1 1 1]
  [1 1 1]
  [1 1 1]
  ...
  [0 0 0]
  [0 0 0]
  [0 0 0]]]
```

As the image is over a million pixels in total, writing them all out is of course completely uninteresting. But numpy arrays has plenty of useful properties to read out average information and such

In [19]:
```
1 print("Minimum pixel intensity:", img.min())
2 print("Maximum pixel intensity:", img.max())
3 print("Average pixel intensity:", img.mean())
4 print("St. dev pixel intensity:", img.std())
```

```
Minimum pixel intensity: 0
Maximum pixel intensity: 214
Average pixel intensity: 37.751220703125
St. dev pixel intensity: 40.628137255354964
```

Here we check the lowest and highest pixel intensity. Note that 251 is a bit lower than the theoretically highest possible value in the uint8, but it is close. In some cases however, the minimum and maximum might be far away, in which case we might want to rescale or renormalize the image. We talk a bit more about histogram equalization later.

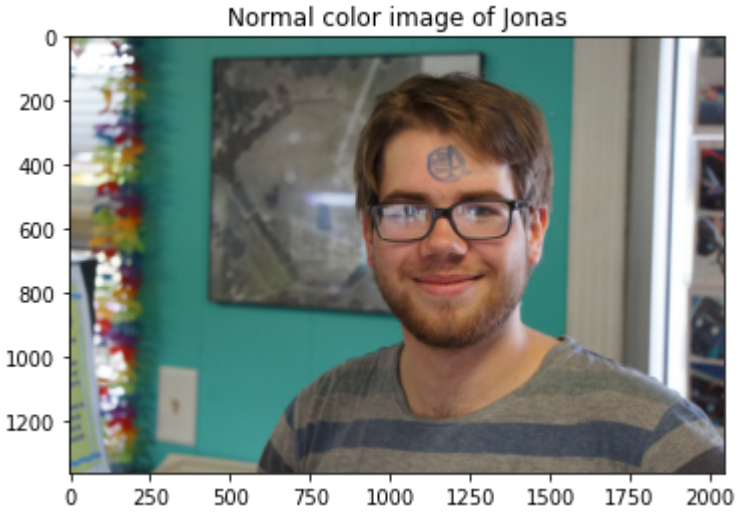
We then compute the numerical average of the pixels and the standard deviation of all pixels.

If you want to do any more fancier computation on the pixel intesities, you can easily compute on the numpy array as if it was a normal mathematical matrix. Let us get back to this later.

Higher dimensional images

Our first example was a two-dimensional image. But images are actually often higher dimensional than this. The simplest reason is that most normal (non-medical) images are *color* images, in which case we need to have multiple *channels* that convery the color information. As an example, I will use an image of myself

In [20]:
```
1 img = io.imread("img/jonas.jpg")
2
3 plt.imshow(img)
4 plt.title("Normal color image of Jonas")
5 plt.show()
6
7 print("Image dimension:", img.ndim)
8 print("Image array shape:", img.shape)
9 print("Pixel dtype:", img.dtype)
```




```
Image dimension: 3
Image array shape: (1362, 2048, 3)
Pixel dtype: uint8
```

The image itself is of course completely uninteresting. The point I am trying to show here is that while the image is displayed as a 2D image, the actual data of the image is three-dimensional. As we can see from the `ndim` and `shape` properties.

This image consists of $1362 \times 2048 \times 3$ pixels. The last *3* is because this is an RGB image, meaning each pixel in the final image is a combination of a *red*, a *green* and a *blue* pixel intensity. Because the third dimension consists of exactly three pixels, `matplotlib` interprets it as a RGB image and therefore shows it as a color image.

Example: Going from color to grayscale

While most medical images are greyscale, and working with RGB might not be that interesting to you, let us do a few simple manipulations of this image to get a better feel for the underlying numpy arrays and how they work.

Because the array is three dimensional, it can be seen as three different 2D arrays stacked on top of each other. And if we know a bit of `numpy` we can easily seperate out these.

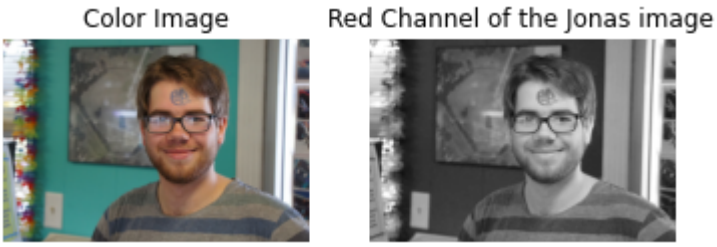
In [21]:
```
red = img[:, :, 0]

print(red.shape)
```

```
(1362, 2048)
```

Here I pull out the *red channel* by *indexing*. Indexing means we are selecting only parts of the array. We write what parts of the array we want in square brackets. Here I write `:` to mean all pixels along this dimension. So `:, :, 0` means *all pixels* along the first two dimension. For the third dimension I write 0, which means I only want a single value. Python starts counting at 0 (this is simply a convention you must get used to), so 0 is the first channel, i.e., the red one. Printing the shape of the red array reveals it to be a 2-dimensional array.

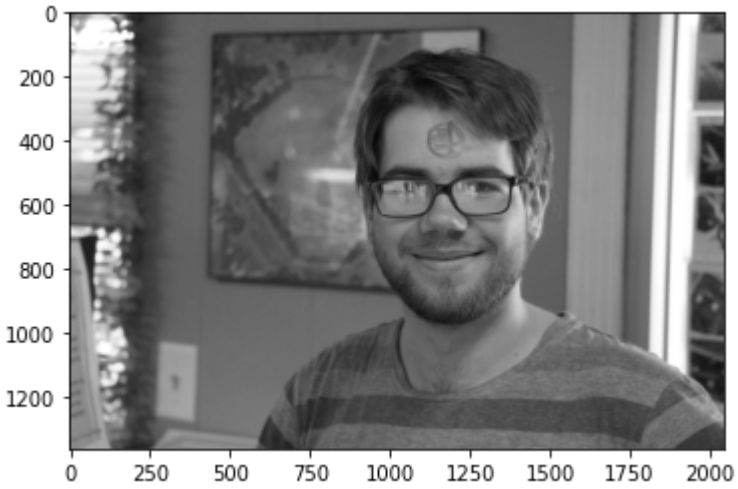
We can now plot the red channel as a greyscale image

In [22]:
```
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title('Color Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(red, cmap='gray')
plt.title('Red Channel of the Jonas image')
plt.axis('off')
plt.show()
```



What we now can do is pull out the blue and the green channels as well. And from these three we can actually compute the *average* of the three channels, which will turn out original color image into a greyscale image

In [23]:
```
red = img[:, :, 0]
green = img[:, :, 1]
blue = img[:, :, 2]

greyscale_img = red/3 + green/3 + blue/3

plt.imshow(greyscale_img, cmap='gray')
plt.show()
```



Here we have effectively used `numpy` to implement our own color to greyscale filter (as we might find in any smartphone picture app).

As a small aside, we actually shouldn't weight each channel equally, by thirds, as the human eye has a preference for green. Mathworks (https://www.mathworks.com/help/matlab/ref/rgb2gray.html) suggest the following weighting to be the one most corresponding to human vision

$$0.299R + 0.587G + 0.114B$$

So in our code we could simply write

In [24]:
```
greyscale_img = 0.299*red + 0.587*green + 0.114*blue
```

Now that I have created a greyscale version of my image, I might want to save the results for later analysis or use. In this case I must use the `skimage.io.imsave` functions. This works similar to `imread` in that I must specify a path to say *where* I want my image to be saved. In addition I of course need to attach my actual image data

In [25]:
```
io.imsave("results/greyscale_jonas.png", greyscale_img)
```

```
Lossy conversion from float64 to uint8. Range [4.153, 255.0]. Convert image to
uint8 prior to saving to suppress this warning.
```

Your turn: A deuteranopi filter

A fun exercise you can now do is to implement a *deuteranopi* filter. Deuteranopi, or "red-green color blindness" as it is often called, is a condition where it is hard to separate red and green colors.

- a) Read in the example image `img/color_wheel.png` and plot it
- b) Use indexing the pick out the red, green and blue color channels
- c) Now define a new array `mixed`, by adding `0.5*red + 0.5*green`.
- d) Now go back and replace both the red and the green channels with this mix, by writing
  - `colorwheel[:, :, 0] = mixed`
- e) Plot the modified colorwheel
- f) Run the same process through the `deuteranopia_test.jpg` image. You should be able to read the number in the image before you apply the filter, but not after.

Actual 3D images

In our previous example, we should that a 2D color image uses a 3D numpy array. However, as mentioned, most medical images are greyscale. However, they are often *volume* images. These are thus closer to what we might think of when we say "3D" images. Let us look at an example.

In this case we want to use an example stored in `stanford_t1.nii.gz` (we have taken this example dataset from the `dipy.org` data examples: https://dipy.org/documentation/1.5.0/data/ (https://dipy.org/documentation/1.5.0/data/))

The `.nii.gz` file format is a NIfTI image, and we cannot load it directly with `skimage`. Instead we use the `nibabel` package (you might need to install it with `pip` first)

In [26]:
```
1 from nibabel import load
2
3 data = load("img/stanford_t1.nii.gz")
```

When we load a NIfTI image like this, we do not get a `numpy` array imediately like we do with `imread`. Rather we get a specific `nibabel` object. This is not really important, as we can read out the actual image data directly with a `get_fdata` function. It also has some metadata stored in a `.header` property

In [27]:
```
1 print(data.header)
```

```
<class 'nibabel.nifti1.Nifti1Header'> object, endian='<'
sizeof_hdr      : 348
data_type       : b''
db_name         : b''
extents         : 0
session_error   : 0
regular         : b''
dim_info        : 0
dim             : [  3  81 106  76   1   1   1   1]
intent_p1       : 0.0
intent_p2       : 0.0
intent_p3       : 0.0
intent_code     : none
datatype        : int16
bitpix          : 16
slice_start     : 0
pixdim          : [1. 2. 2. 2. 1. 1. 1. 1.]
vox_offset      : 0.0
scl_slope       : nan
scl_inter       : nan
slice_end       : 0
slice_code      : unknown
xyzt_units      : 0
cal_max         : 0.0
cal_min         : 0.0
slice_duration  : 0.0
toffset         : 0.0
glmax           : 0
glmin           : 0
descrip         : b''
aux_file        : b''
qform_code      : unknown
sform_code      : aligned
quatern_b       : 0.0
quatern_c       : 0.0
quatern_d       : 0.0
qoffset_x       : -80.0
qoffset_y       : -120.0
qoffset_z       : -60.0
srow_x          : [  2.   0.   0. -80.]
srow_y          : [   0.    2.    0. -120.]
srow_z          : [  0.   0.   2. -60.]
intent_name     : b''
magic           : b'n+1'
```

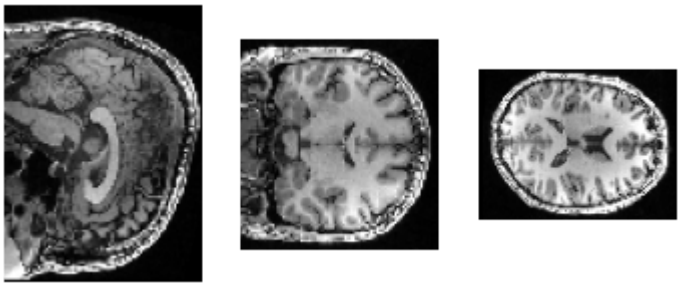
In [28]:
```
img = data.get_fdata()
print(type(img))
print(img.shape)
print(img.dtype)
```

```
<class 'numpy.ndarray'>
(81, 106, 76)
float64
```

Here we have a 3D numpy array that consists of *floating points* (decimals). This data thus corresponds to a 3D volume block of data. We cannot easily plot such a 3D space, but we can again use numpy to easily pull out specific planes to plot, at least along the three principle directions

We can pick a specific plane by using indexing. Let us for instance pick out the three middle planes along each axis

In [29]:
```
nz, ny, nx = img.shape

plt.subplot(1, 3, 1)
plt.imshow(img[nz//2, :, :], cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(img[:, ny//2, :], cmap='gray')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(img[:, :, nx//2], cmap='gray')
plt.axis('off')
plt.show()
```



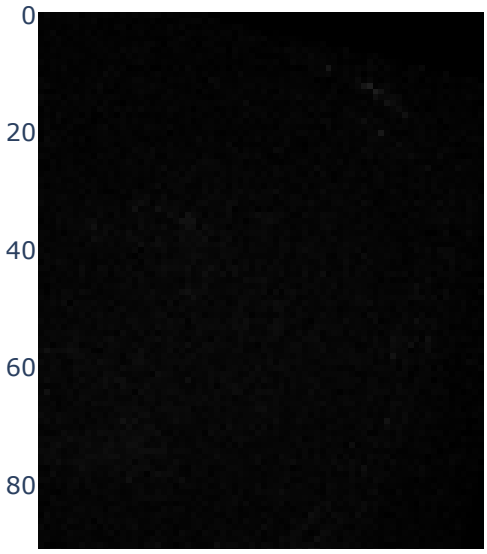
You might be a bit confused that I order the name of the axes in the order z, y, x. This is simply a common convention that the z axis (axial direction) is placed as the first dimension, then y, then x. In the code they are simply names of variables, so it is just convention.

Looking at 3D data

If you want to work with 3D data *interactively*, it might be better to use a dedicated program for things like this. As Python is not really the best tool for things like that. However, there are definitely ways to do things like animate a stack of images. However, we consider this slightly

outside the scope of this course. Here is an example that creates an animation of the stack in question using `plotly.express`:

In [30]:
```
import plotly.express as px

px.imshow(img, animation_frame=0, binary_string=True)
```



Here the `animation_frame` argument is saying *which* axis we want to animate. So you can write 0, 1, or 2, to get the animation to go along either of the three dimensions. This was meant as a simple example, and I won't cover any more on plotly.express, but they have plenty of examples and documentation on their website if you are curious.

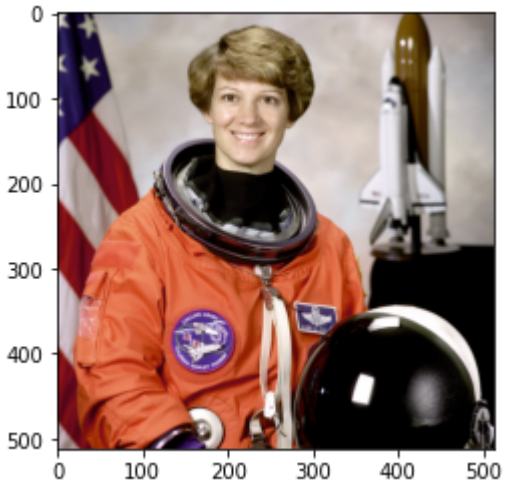
Cropping Images

We have seen how we can use indexing with `numpy` arrays to pick out individual channels, or slices. But let us now see how we can pick out ranges to keep. The most straight-forward use for this is to simply *crop* images. Cropping images is often useful, as we might not be interested in a full dataset which we have loaded in, and want instead to focus only on parts of it (we will also get back to this a bit later with *masking*).

As an example, we will use an example image contained within `skimage` (packages often contain example images, to make it easier to make tutorials and examples and so forth).

In [31]:
```
1 img = skimage.data.astronaut()
2
3 print(img.shape)
4
5 plt.imshow(img)
6 plt.show()
```

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



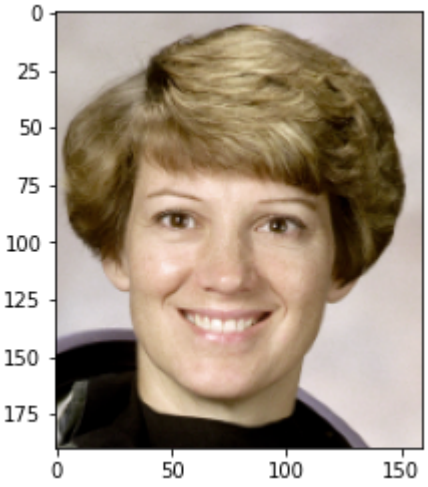
This image is 512 by 512 pixels. As it's a color image, we see that it has a third dimension, but we don't really need to think about that in our example. Let us say we want to focus only on the astronauts head now, and we want to crop out this part.

To select only the pixels we are interested in, we need to specify what indexes we want to keep along either axis. Note that they both start at 0, as you can see from the image. To specify a range, simply put a number before and/or after the colon, for instance `100:200` would mean the pixels between 100 and 200. If you omit a number, for instance `100:` that means all pixels from 100 to the end of the axis.

To find specific numbers, you can either plot the image in an interactive viewer and check the coordinates by hovering, or you can use trial-and-error. Some quick testing gives some useful numbers

In [32]:

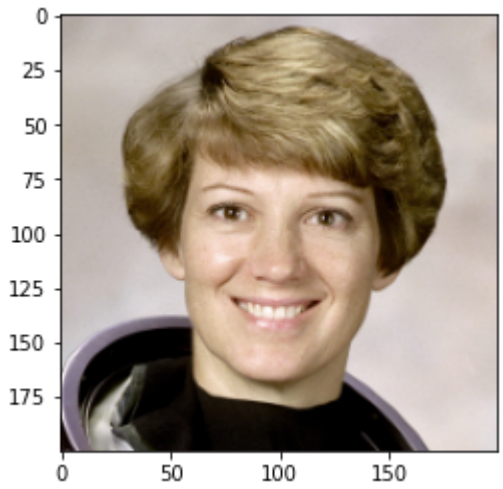
```
face = img[10:200, 150:310]

plt.imshow(face)
plt.show()
```



If we know the image size we want ahead of time, say we want a square 200x200 pixel crop-out. It might be easier to do it like this:

In [33]:

```
x0 = 125
y0 = 10
width = 200

face = img[y0:y0+width, x0:x0+width]

plt.imshow(face)
plt.show()
```



Here we simply adjust the `x0` and `y0` values with some trial and error untill we are happy.

The point of this example was mainly to show you *slicing* of `numpy` -arrays, though cropping is a very useful step in digital image analysis. However, as one of our goals with programming is usually to automate image processing tasks - selecting crops or windows by hand, manually. Especially in a trial-and-error way, is rarely what we want to do. Ideally, we find a way to automatically pick out interesting features. This can for instance be done through using masks (as we will explain later).

Your Turn: Cropping using slicing

- Read in the example image `skimage.data.chelsea` .
- Plot the image, what does it show?
- Use slicing to pick out a window of 64x64 pixels that captures an eye

Hint: You can copy my code from above and change it if that makes it easier.

Graphical tool for selecting ROI

At this point you might be asking if there exists a graphical interface tool for simply *selecting* a region of interest or cropping. The short answer is that *yes*, there does. The OpenCV-package contains some GUI tools, and the function `cv2.selectROI` opens up an interactive dialog box that lets you simply pick a window. However, this interactive feature doesn't work that well with Jupyter. Instead we have made an example script and put it in the github `examples` folder under the name `select_roi_and_crop.py` .

However, the longer answer is that again, a GUI tool like that will make batch processing and automating a bit tricker. And so it is often preferable to simply avoid using manual selection at all, if possible. Though whether that is possible is of course highly problem specific.

Example: More than three dimensions

Scientific images can quickly get more complex than simple grey-level images, or 2D/3D datasets. Once I worked with a microscope that output a 9D image dataset! It's important to remember that `numpy` itself allows its data to be placed in *any* shape, and what order to put data in is typically just conventions. A given imaging modality or machine might output data in a type that is different from what we cover here, and you should take some time to double check the dimensions and shape of your data to make sure everything makes sense. Then you can start pulling out the data you are interested in and work on it the way you want to.

As an example, let us load in a dataset and explore it

In [34]:
```
from skimage import data

kidney_img = data.kidney()

print(kidney_img.ndim)
print(kidney_img.shape)
print(kidney_img.dtype)
```

```
4
(16, 512, 512, 3)
uint16
```

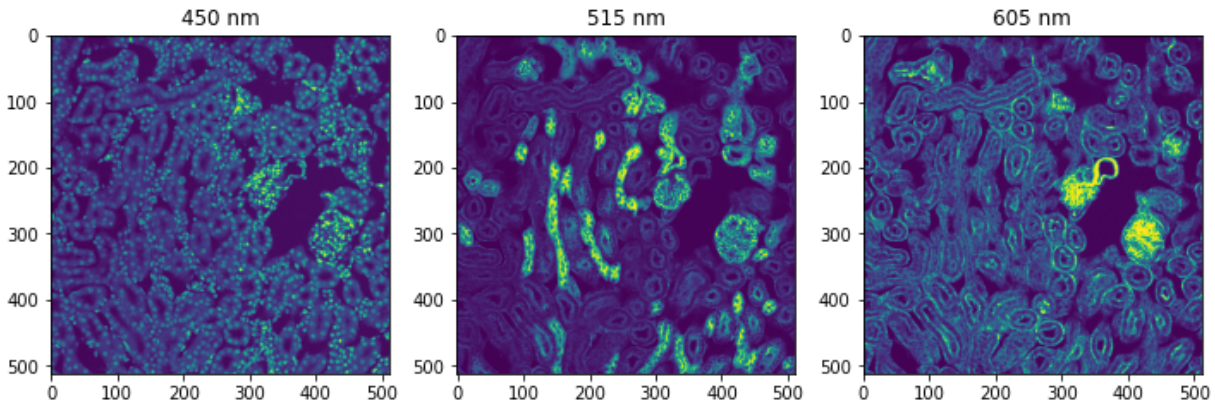
This "image" is four-dimensional, and looking up the official reference (https://scikit-image.org/docs/stable/api/skimage.data.html#skimage.data.kidney), we will learn that these are

- The first dimension are 16 slices in the axial direction (i.e., z)
- The second and third dimension are the x- and y-dimensions
- The fourth represents three individual microscopy color channels (450 nm, 515 nm, 605 nm)

So this "image" is actually a stack of 16 images, where each image is the combination of three individual color channels. They are thus not normal RGB-images. Let us select the middle of the stack, and plot the three channels one-by-one

In [35]:

```
middle = kidney_img[8]

plt.figure(figsize=(12, 10))
plt.subplot(1, 3, 1)
plt.imshow(middle[:, :, 0])
plt.title('450 nm')

plt.subplot(1, 3, 2)
plt.imshow(middle[:, :, 1])
plt.title('515 nm')

plt.subplot(1, 3, 3)
plt.imshow(middle[:, :, 2])
plt.title('605 nm')

plt.show()
```

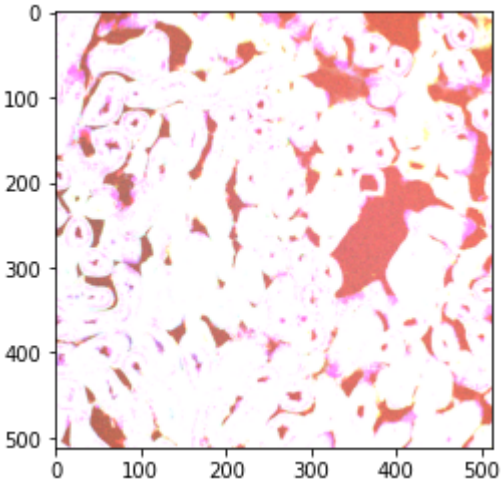


In this case the microscope has picked up three individual fluorophores in a fluroessence image. And the three image channels therefore contain comepletely seperate data. This can be the case in a lot of dataset, where it is possible that different imagaing modalities or information sources are simply stacked. In other cases, we might have different datasets that need to be *registered* (i.e., overlaid and matched in coordinates).

In this case we happen to have three data channels. If we *want* to we can produce a *false color* image of this dataset simply by sending the image to `plt.imshow`, which will interpret it as RGB data. However, note that the three channels do not correspond to the colors `matplotlib` will use. It is just a coincidence that we have three color channels in this case.

In [36]:
```
1 plt.imshow(middle)
2 plt.show()
```

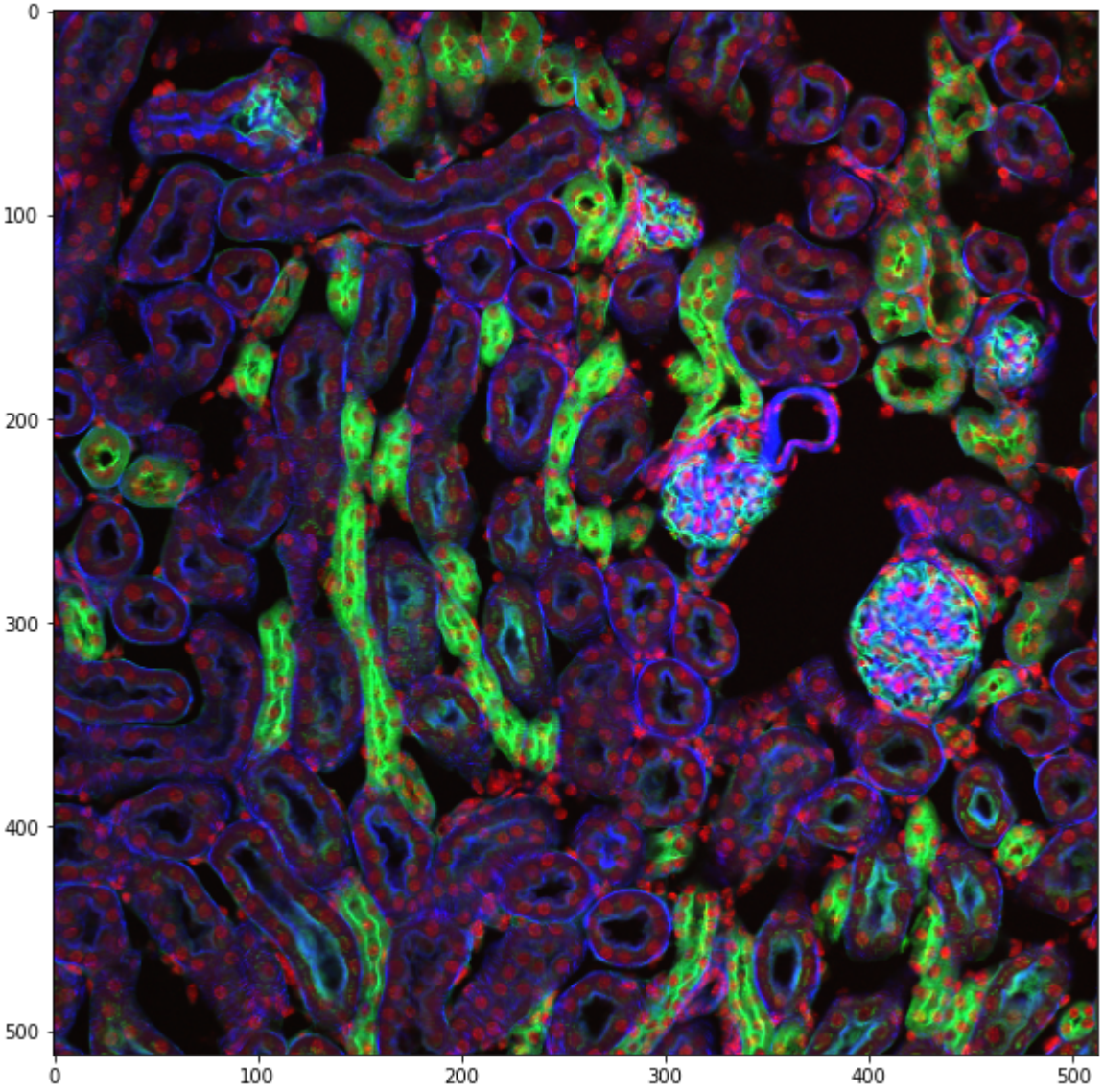
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for flo
ats or [0..255] for integers).
```



Here we get a warning that the data range of the pixels is not was `matplotlib` wants it, so let us normalize the image data first, then replot

In [37]:

```
1 plt.figure(figsize=(12, 10))
2 plt.imshow(middle/middle.max())
3 plt.show()
```



Working with DICOM-images

Let us finally look closer at how we can read and work with DICOM images. Like with the NIfTI images, we here use a package especially for reading the images, and we again get metainformation along with the image data itself. In this case, however, let us use a bit of time to look at how we can work with DICOM-tags to read or change this metainformation.

We wil now work on the example image of `img/CT_small.dcm`. This is a file I took directly from the `pydicom` documentation page:

- https://pydicom.github.io/pydicom/stable/auto_examples/index.html (https://pydicom.github.io/pydicom/stable/auto_examples/index.html)

Specifically, you can look at the "Read a Dataset and plot Pixel Data" (https://pydicom.github.io/pydicom/dev/auto_examples/input_output/plot_read_dicom.html)

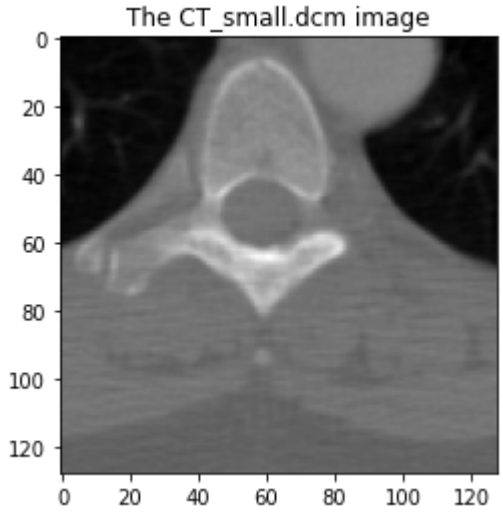
In [38]:
```
from pydicom import dcmread

data = dcmread("img/CT_small.dcm")

print(type(data))
```

```
<class 'pydicom.dataset.FileDataset'>
```

As before we import a specific function for reading an image file, in this case it is `pydicom.dcmread`, which then reads the `CT_small.dcm` file. The object we have created in memory is now a specical `pydicom` object.

To read out the actual image data, we must access the `pixel_array` property, which then gives us the actual image data as a `numpy` array (just like for all other image formats)

In [39]:
```
img = data.pixel_array

plt.imshow(img, cmap='gray')
plt.title("The CT_small.dcm image")
plt.show()

print(type(img))
print(img.shape)
print(img.dtype)
print(img.min())
print(img.max())
```




```
<class 'numpy.ndarray'>
(128, 128)
int16
128
2191
```

Working with DICOM-tags

If all you want is to read out the image data from a DICOM image, we are already done. However, the metadata of the DICOM-image contains a lot of information, which you might be interested in working with. This metainformation is sorted under different *tags*, which are specificed in the DICOM-standard.

For any of the standard tags you can access them as properties of the `pydicom` -object. See for example:

In [40]:
```
print(f"Patient Name: {data.PatientName}")
print(f"Patient ID: {data.PatientID}")
print(f"Modality: {data.Modality}")
print(f"Study Date: {data.StudyDate}")
print(f"Pixel Spacing: {data.PixelSpacing}")
```

```
Patient Name: CompressedSamples^CT1
Patient ID: 1CT1
Modality: CT
Study Date: 20040119
Pixel Spacing: [0.661468, 0.661468]
```

These are just some of the many possible DICOM tags that exist. For an extensive list of possible tags you can see here (https://dicom.nema.org/medical/dicom/current/output/chtml/part06/chapter_6.html).

Note that not all tags necessarily *need* to be used. And any given image might use, nor not use, any of the standard tags. If you want to check *whether or not* an image has any data stored on a given tag, you can use the Python `in` keyword to write a quick conditional:

In [41]:
```
print('WindowWidth' in data)
print('PatientName' in data)
```

```
False
True
```

From the results we see that our example image has a `PatientName` tag, but no `WindowWidth` tag.

Set information on a tag

We have so far see that you can *read* information on a tag, or check whether information is present at all. But what if you want to add a new tag? Or change information on a given tag? Then you can simply change it as if it were a normal Python variable:

In [42]:
```
data.WindowWidth = 500
```

Here we define the `WindowWidth` to be 500 (we can imagine this is needed by a general script we have already made for instance). After we define the variable, the tag is now present:

In [43]:
```
print('WindowWidth' in data)
```

```
True
```

Here we defined a tag that wasn't present, but you can also change an existing tag.

Deleting tags

Instead of changing the value of a given tag, we can delete the information altogether using the Python keyword `del` (short for delete), For instance

In [44]:
```
del data.PatientName
del data.PatientID
```

Let us double check that these tags are indeed missing:

In [45]:
```
print('PatientName' in data)
print('PatientID' in data)
```

```
False
False
```

Deleting tags might be a necessary step in certain cases, for instance if we want to anonymize data. (Disclaimer: I am not claiming that deleting the tags `PatientName` / `PatientID` is sufficient to anonymize a given dataset, I am simply giving an example of what might be a step in a larger workflow).

Saving dicom-images

Note that as we are adding, changing or deleting tags, we are not working on the actual `.dcm` file as it is stored on the hard drive. Instead we are working on a Python-object stored in memory. If we want to keep our modified data, we will need to save it as a .dcm file before we end our program. We can save it as the same name we loaded in, but note that this will *overwrite* our existing image. As a best practice this is rarely what we want to do, because a small bug in our code and we can break and destroy data we want to keep. Therefore, let us save our file under a new path, which will then result in a new image on disk. Let us store this in the `results` folder, instead of the `img` folder

In [46]:
```
data.save_as("results/modified_CT_small.dcm")
```

Your turn: Reading a dicom image and changing tags

- a) Read in the file `img/vannfantom.dcm`
- b) Read the `PatientName` tag, what does it say?
- c) Does the file have a `PatientID` tag? If so, what does it say?
- d) Delete both tags
- e) Plot the image data contained within the file (Hint: `.pixel_array`)
- f) Find the size of the image, the data type of the pixels and the average value of the entire image.
- g) Save the image with the saved tags under `results/vannfantom.dcm`

In [47]:
```
import pydicom as dcm

data = dcm.dcmread("img/vannfantom.dcm")

print(data.PatientName)
print(data.PatientID)
```

```
Fysiker Test
23042100001
```

Example: Batch-processing dicom images

As an example, let us write a quick code that walks through all images in a given folder, reads them in, deletes the patient name and patient ID and saves the results in a new folder

In [48]:
```
import os
import glob

input_folder = 'img'
output_folder = 'results'

input_files = glob.glob(os.path.join(input_folder, "*.dcm"))

print(f"Finding all *.dcm files in {input_folder}")

for infile in input_files:
    print(f"Processing {infile}.")

    data = dcmread(infile)

    del data.PatientName
    del data.PatientID

    filename = os.path.basename(infile)
    outpath = os.path.join(output_folder, filename)

    data.save_as(outpath)
    print(f"Wrote results to {outpath}")

```

```
Finding all *.dcm files in img
Processing img/CT_small.dcm.
Wrote results to results/CT_small.dcm
Processing img/vannfantom.dcm.
Wrote results to results/vannfantom.dcm
```

Your turn: Batch-processing

The folder `img` contains a series of chest xrays from the same patient over time. These are named `chest_000.png`, `chest_001.png` and so on. Use `glob` to automatically loop over and plot out all of these.

Hint: `glob` finds all files matching a given pattern, but they don't come sorted. If you however do `sorted(glob(pattern))` (where `pattern` is a text-string), it finds all files and then sorts the list.

Part 3: Digital image analysis

Let us now turn to performing more actual analysis or manipulation of images. As mentioned earlier, this is simply a crash course, and we don't have too much time left - so here we focus on giving some examples. To get a better overview of possibilities with digital image analysis, I recommend you take a look at the scikit-image gallery, which has plenty of examples with corresponding code. You can find it here:

- https://scikit-image.org/docs/stable/auto_examples/ (https://scikit-image.org/docs/stable/auto_examples/)

Edge detection

One classic operation in image analysis is to use *edge detection*. Edge detection is often a useful step when trying to *segment* an image. Segmenting means trying to declare what different parts of an image are different objects or regions, for instance recognizing what part of a medical image corresponds to an organ, or say, a tumor.

Edge detection is process where an algorithm tries to look at how pixel intensities change to find the edges between different regions and objects. It can often be a bit tricky in noisy images, such as medical images.

In this example we will use a chest x-ray image. You can find a series of 9 such images in the `img` folder which I have taken from an open NIH dataset. Another interesting data set to explore on your own can be this open source data set of chest x-rays with pneumonia which you can find on Kaggle:

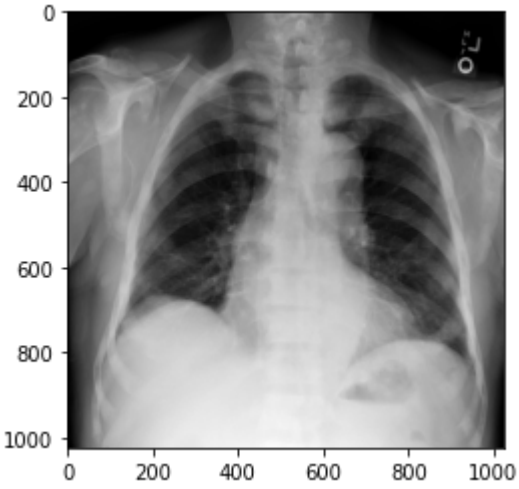
- The ChestX-ray8 dataset provided by the NIH: https://paperswithcode.com/dataset/chestx-ray8which (https://paperswithcode.com/dataset/chestx-ray8which) I took from an open source dataset
- An open Kaggle dataset: https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia (https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia)

We start of by loading in the image as usual

In [49]:

```
from skimage import io
import matplotlib.pyplot as plt

xray = skimage.io.imread("img/chest_000.png")

print(xray.shape)
print(xray.dtype)

plt.imshow(xray, cmap='gray')
plt.show()
```

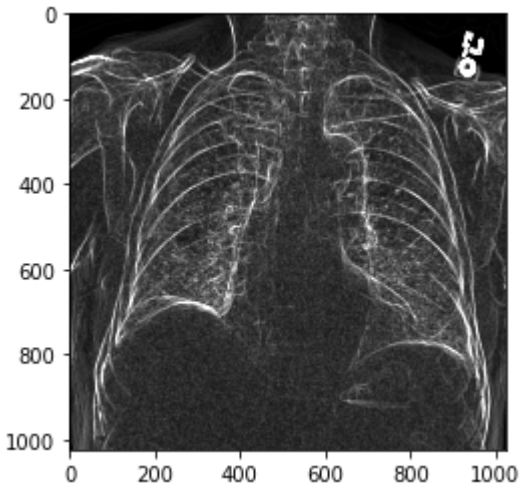
```
(1024, 1024)
uint8
```



We now want to do edge detection on this image. Luckily, `skimage` supports a few different algorithms for edge detection, and these are contained in the `filters` submodule. Let us start of with the *Sobel* edge detection algorithm

In [50]:

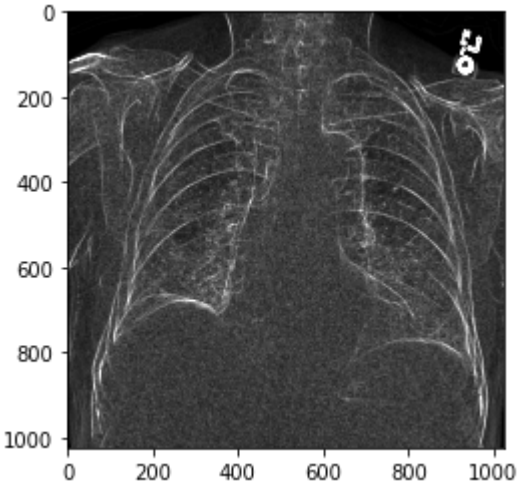
```
from skimage import filters

edges = filters.sobel(xray)

plt.imshow(edges, cmap='gray', vmax=0.05)
plt.show()
```



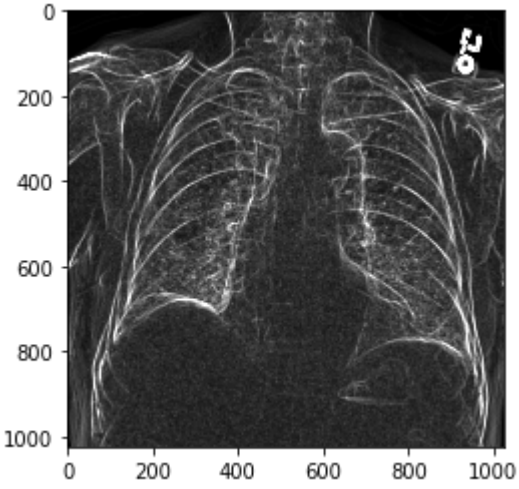
Note that we use the `vmax` to turn up the image intensities that are printed out. If you skip this, the image will be very dark. The main reason for this is that the algorithm gives an output that is "edge magnitude", which is very low numbers for our image, as there are not clear edges between the different regions. For other types of images, the output will typically be stronger.

As mentioned, different edge detection algorithms exist. You can read a bit more about them here (https://scikit-image.org/docs/stable/auto_examples/edges/plot_edge_filter.html). Without going into too much technical details, they work differently mathematical, and some might be better suited for different types of images, depending on what you are interested in.

In [51]:
```
from skimage import filters

edges = filters.roberts(xray)

plt.imshow(edges, cmap='gray', vmax=0.05)
plt.show()
```



In [52]:
```
from skimage import filters

edges = filters.prewitt(xray)

plt.imshow(edges, cmap='gray', vmax=0.05)
plt.show()
```

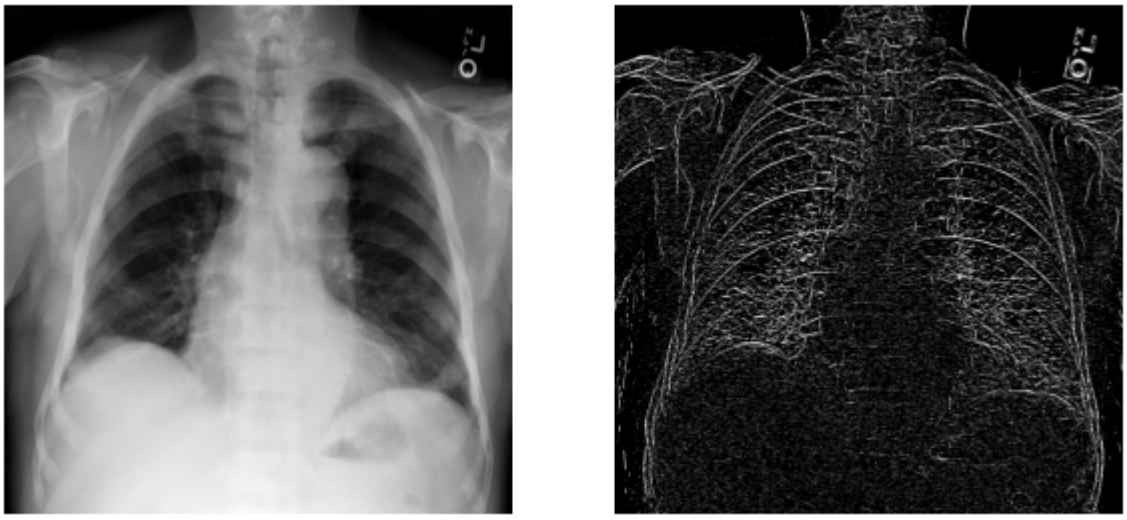


The Laplacian of Gaussian filter

A final edge detection algorithm I wanted to mention is the so-called Laplacian of Gaussian filter (LoG), which can also be used for edge detection. It is a slighlty more fancy algorithm that first takes the Gaussian of an image to smooth it, then takes the Laplacian of the image (which is a mathematical transform). It gets a bit more technical, so I don't want to get into the details here. Suffice to say, neither skimage or OpenCV has a built in LoG filter that works for our purposes. If you are interested in an LoG, I would instead advice you to go to the `scipy.ndimage` package, which has this function.

I mostly mention the LoG filter here as an example that not every algorithm, feature or filter will be available in one, single package. In this case we could instead turn to the `scipy` package. Or we could implement the LoG filter ourselves, as the `skimage` package does contain functions for computing both the Gaussian and Laplacian. This option, however, would require that we understand quite a bith about exactly how the LoG should be computed and analyzed.

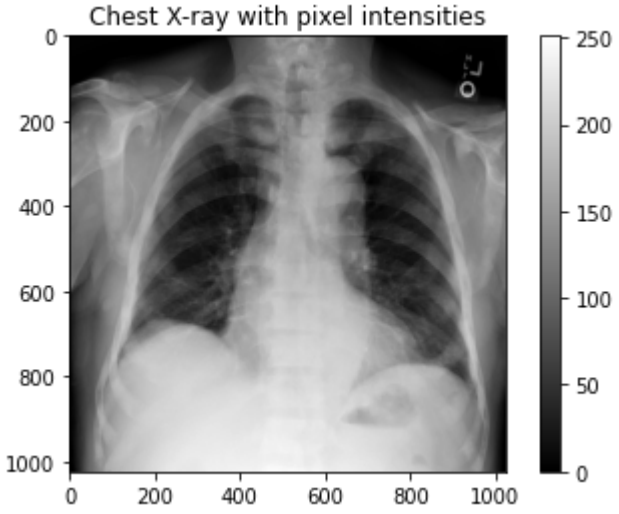
In [53]:
```
from scipy import ndimage

edges = ndimage.gaussian_laplace(xray, sigma=1.3)

plt.figure(figsize=(10, 8))
plt.subplot(1, 2, 1)
plt.imshow(xray, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(edges, cmap='gray')
plt.axis('off')
plt.show()
```



Histograms and contrast enhancement

Another topic that can be useful is to look closer at the histogram of an image. By that we mean a histogram of the pixel intensities. Let us again use the chest x-ray as an example. First let us plot the image and add a colorbar to get a better feel for the actual pixel values

In [54]:
```
1 plt.imshow(xray, cmap='gray')
2 plt.title("Chest X-ray with pixel intensities")
3 plt.colorbar()
4 plt.show()
```



From the color bar we see that pixel intensities range from 0 to around 250ish. Which makes a lot of sense, because this is a `uint8` image. However, we can make a histogram of all the pixel intensities to look at the actual distribution of pixel intensities. This can sometimes be useful, as some images with low contrast might only use a small portion of the intensity space

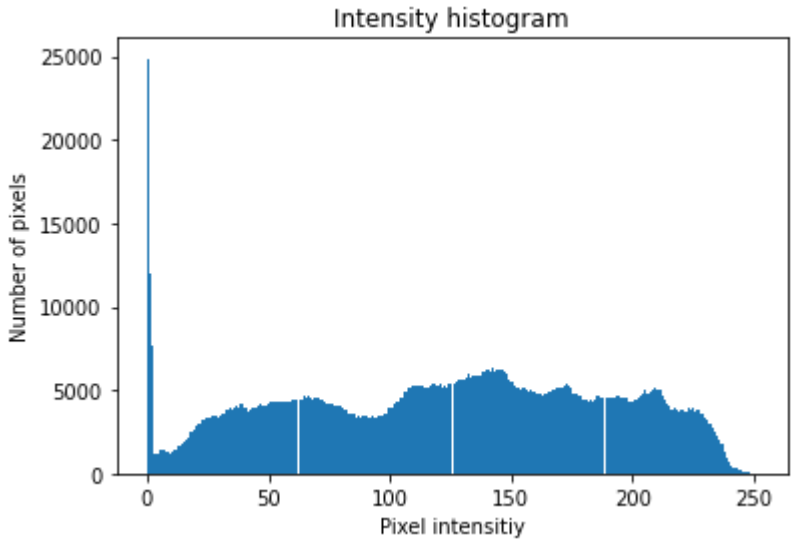
There are three different functions we can use to make the histogram of an image, and they work a bit different:

- `np.histogram`
- `skimage.exposure.histogram`
- `plt.hist`

You can read up on the three more in detail if you want, there are subtle differences, but they pretty much do the same thing. The major difference is that the two first ones just compute the histogram values, and you will need to plot them yourself. The last one (from `matplotlib`) plots it for us, so let us use that.

For `plt.hist` you just need to use `.ravel()` on the numpy array. This method turns your 2D (or higher dimensional) array into a 1D string of pixels. In essence it flattens your picture and removes any spatial information. This is OK for a histogram, as we only care about pixel intensities, not where they are located.

In [55]:
```
1 plt.hist(xray.ravel(), bins=255)
2 plt.xlabel('Pixel intensitiy')
3 plt.ylabel('Number of pixels')
4 plt.title('Intensity histogram')
5 plt.show()
```



We can see that this image uses the full spectrum of possible image intensities, which is a good thing.

The `skimage.exposure` submodule is used to directly manipulate an image to adjust its histogram. It can for example be used to improve the contrast of an image that has a skewed or bunched up histogram distribution.

- https://scikit-image.org/docs/stable/api/skimage.exposure.html (https://scikit-image.org/docs/stable/api/skimage.exposure.html)

Whether or not histogram equalization is something you have use for is of course highly problem specific. But it is useful to know about.

The code cell below is an example of histogram equalization using `skimage`, taken directly from their website:

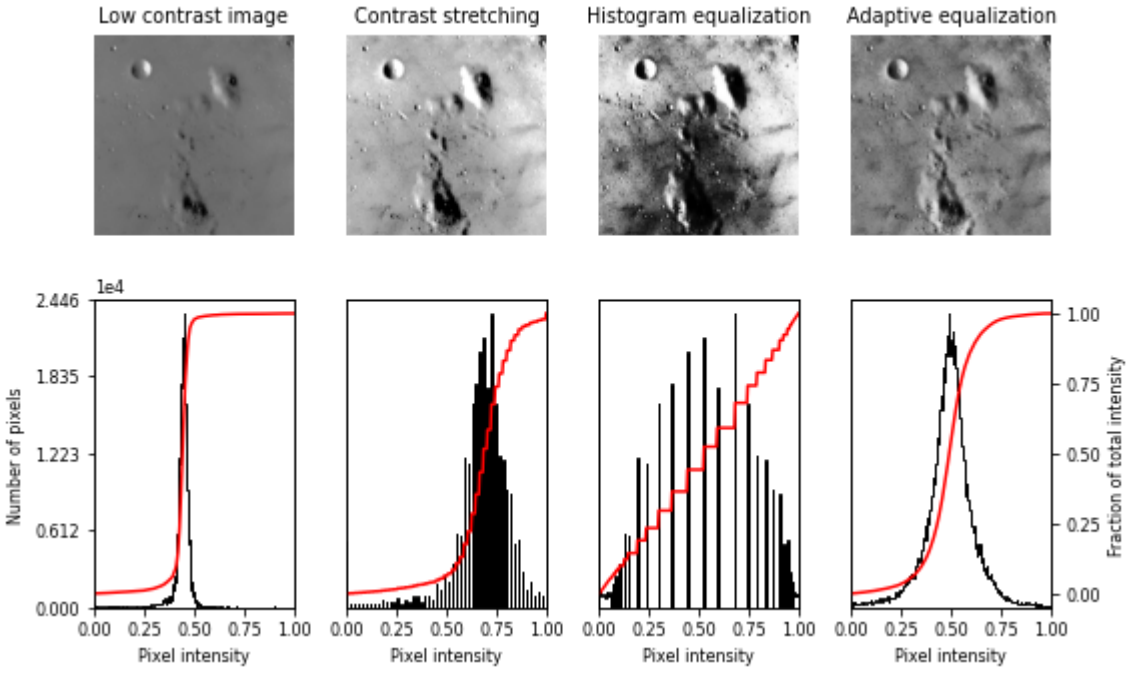
- https://scikit-image.org/docs/stable/auto_examples/color_exposure/plot_equalize.html#sphx-glr-auto-examples-color-exposure-plot-equalize-py (https://scikit-image.org/docs/stable/auto_examples/color_exposure/plot_equalize.html#sphx-glr-auto-examples-color-exposure-plot-equalize-py)

The code itself is a bit more technical than it needs to be, as it goes a bit fancy on the plotting and such. But the basic premise is that it uses and example image with a histogram that is really bunched up, giving the image low contrast. Using a few different approaches, this can be stretched out to improve contrast.

In [56]:

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

from skimage import data, img_as_float
from skimage import exposure


matplotlib.rcParams['font.size'] = 8


def plot_img_and_hist(image, axes, bins=256):
    """Plot an image along with its histogram and cumulative histogram.

    """
    image = img_as_float(image)
    ax_img, ax_hist = axes
    ax_cdf = ax_hist.twinx()

    # Display image
    ax_img.imshow(image, cmap=plt.cm.gray)
    ax_img.set_axis_off()

    # Display histogram
    ax_hist.hist(image.ravel(), bins=bins, histtype='step', color='black')
    ax_hist.ticklabel_format(axis='y', style='scientific', scilimits=(0, 0))
    ax_hist.set_xlabel('Pixel intensity')
    ax_hist.set_xlim(0, 1)
    ax_hist.set_yticks([])

    # Display cumulative distribution
    img_cdf, bins = exposure.cumulative_distribution(image, bins)
    ax_cdf.plot(bins, img_cdf, 'r')
    ax_cdf.set_yticks([])

    return ax_img, ax_hist, ax_cdf


# Load an example image
img = data.moon()

# Contrast stretching
p2, p98 = np.percentile(img, (2, 98))
img_rescale = exposure.rescale_intensity(img, in_range=(p2, p98))

# Equalization
img_eq = exposure.equalize_hist(img)

# Adaptive Equalization
img_adapteq = exposure.equalize_adapthist(img, clip_limit=0.03)

# Display results
fig = plt.figure(figsize=(8, 5))
axes = np.zeros((2, 4), dtype=object)
axes[0, 0] = fig.add_subplot(2, 4, 1)
for i in range(1, 4):
```

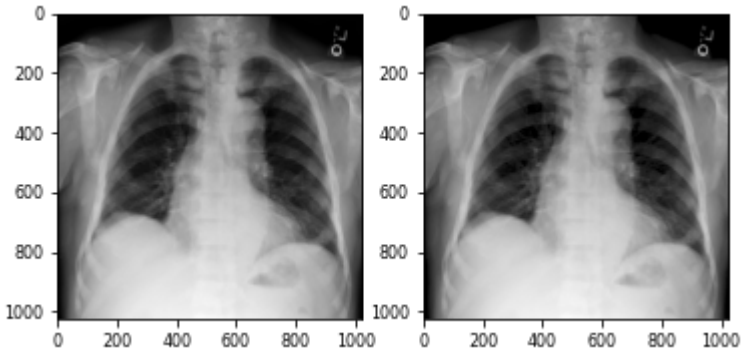
```
    axes[0, i] = fig.add_subplot(2, 4, 1+i, sharex=axes[0,0], sharey=axes[0,
for i in range(0, 4):
    axes[1, i] = fig.add_subplot(2, 4, 5+i)

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img, axes[:, 0])
ax_img.set_title('Low contrast image')

y_min, y_max = ax_hist.get_ylim()
ax_hist.set_ylabel('Number of pixels')
ax_hist.set_yticks(np.linspace(0, y_max, 5))

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_rescale, axes[:, 1])
ax_img.set_title('Contrast stretching')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_eq, axes[:, 2])
ax_img.set_title('Histogram equalization')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_adapteq, axes[:, 3])
ax_img.set_title('Adaptive equalization')

ax_cdf.set_ylabel('Fraction of total intensity')
ax_cdf.set_yticks(np.linspace(0, 1, 5))

# prevent overlap of y-axis labels
fig.tight_layout()
plt.show()
```



Manipulating image intensities directly

Once you start looking more into pixel intensities, it can be useful to know that it is easy to affect pixels of a given intensity, or a range of intensities directly. To do this, simply index with a boolean condition. Say for instance in our chest x-ray image example we want to take all pixels that are close to black, and set them to be exactly zero. Effectively nulling out these pixels. You could easily do this as follows

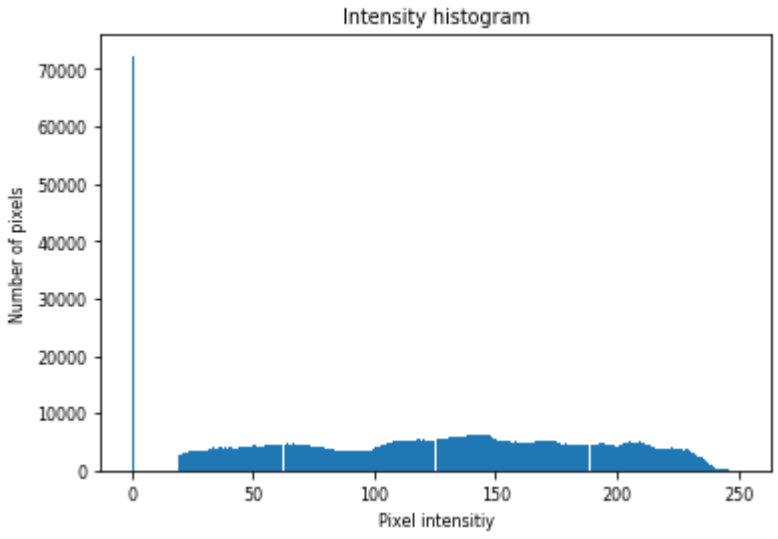
In [57]:
```
xray_masked_background = xray.copy()

# Set all pixels below 20 to 0
xray_masked_background[xray < 20] = 0

plt.subplot(1, 2, 1)
plt.imshow(xray, cmap='gray')

plt.subplot(1, 2, 2)
plt.imshow(xray_masked_background, cmap='gray')
plt.show()
```



Here we first use `.copy` to copy the original image (that way we can make a new version without changing the original). Then we select all pixels close to zero by indexing with `xray < 20`, and then we set these to zero.

Note that we don't really see a major difference between these images, as the values we put to zero were already quite close to 0 to begin with. But if we plot out the histogram, we can easily see that the image is indeed different.

In [58]:
```
plt.hist(xray_masked_background.ravel(), bins=255)
plt.xlabel('Pixel intensitiy')
plt.ylabel('Number of pixels')
plt.title('Intensity histogram')
plt.show()
```



Say we now for instance want to Find the average value of non-zero pixels, we could do that by again using a boolean conditional as an index:

In [59]:
```
print("Average pixel intensity of whole image:", xray.mean())
print("Average pixel intensity of pixels above 20:", xray[xray > 20].mean())
```

```
Average pixel intensity of whole image: 122.28851699829102
Average pixel intensity of pixels above 20: 131.3003431342333
```

Thresholding

The next thing we want to look at is another fundamental digital imaging technique, which we call *thresholding*. Thresholding is again often an important step in segmentation, but it can also be useful in defining masks and regions of interested to perform further analysis.

Thresholding means taking an image, and then comparing to a given threshold level to try to define what is considered *foreground pixels* and *background pixels*, often these correspond to a given object.

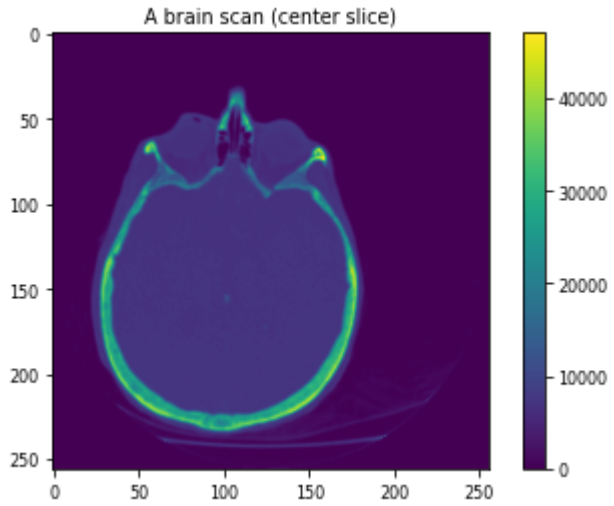
As an example, let us take an image of a brain scan. We pull this out of `skimage` 's example data

In [60]:
```
from skimage import data

brain = data.brain()

print(brain.shape)
print(brain.dtype)
```

```
(10, 256, 256)
uint16
```

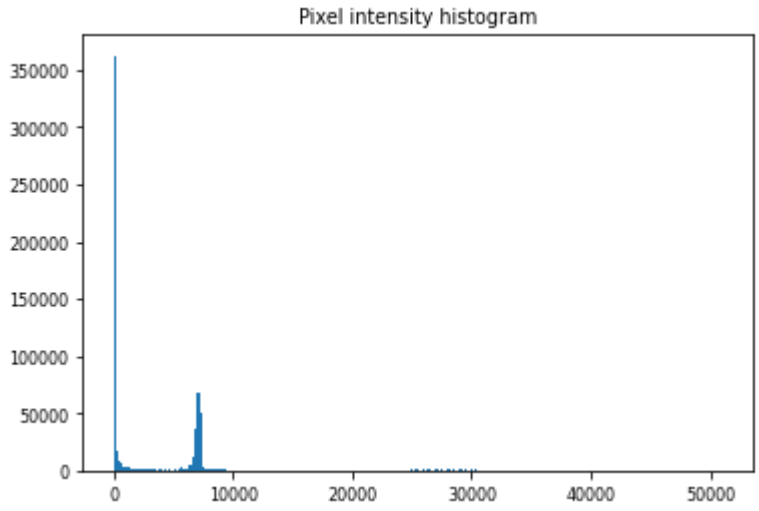
This brain scan is a stack of 10 slices, each 256x256 pixels. To simplify our example, let us pick out the center slice

In [61]:
```
nz, ny, nx = brain.shape

img = brain[nz//2]

plt.imshow(img)
plt.title("A brain scan (center slice)")
plt.colorbar()
plt.show()
```



From the image we see that the pixel intensities in this image are a lot bigger than in our earlier example, this is because the pixels in this case are `uint16` objects, meaning pixel intensities are in the range $[0, 2^{16})$ or $[0, 65536)$. The exact numbers are not the important part, but rather that different *regions* of this scan have different intensities. Let us show this by showing the histogram of this image.

In [62]:
```
plt.hist(brain.ravel(), bins=255)
plt.title('Pixel intensity histogram')
plt.show()
```



From the histogram we can see that there are a vast majority of pixels close to 0 in value, then there is a region of pixels around 10000, then some in the region 25000-30000. From the picture

we can see that the upper region in the histogram most likely corresponds to the skull itself.

From the histogram we can now clearly see that if we pick out pixels that have an intensity of 20 thousand and above, we *should* be able to threshold out the skull cleanly, let us see

In [63]:
```
1 skull = img > 20000
```

This codeline might look very strange, but it is a straight forward boolean operation in Python. Remember that image objects are actually `numpy` arrays, and when we write a conditional with a numpy array like this, what we get is a *new* array, that is *boolean*. Meaning we have a new image where each pixel is either *False* or *True* (or 0 and 1, or black and white). Let us show this

In [64]:
```
1 print(skull.shape)
2 print(skull.dtype)
3 print(skull.min())
4 print(skull.max())
```

```
(256, 256)
bool
False
True
```

In [65]:
```
1 plt.imshow(skull, cmap='gray')
2 plt.title("Thresholded image")
3 plt.show()
```

We see that the thresholding operation picked out most pixels in the skull, but not *all*. If we lower the threshold a bit, we might get more of the skull.

Automatically picking the threshold

In our first example, we selected the threshold ourselves. However, often we want to automate this. This is of course to save ourselves time, but it is also a benefit that an automatically selected threshold is *unbiased*, which is often nice in science.

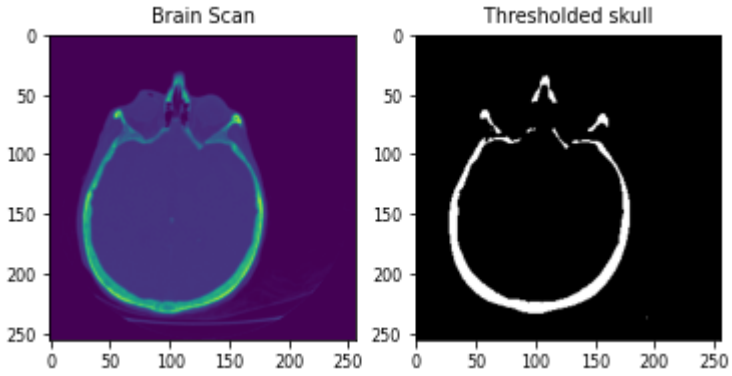
Many algorithms exist for selecting a threshold, but they all rely on analysing the histogram of an image, and in some way trying to calculate where the threshold should be. The most famous of these algorithms is called the *Otsu* threshold. Let us try it out.

In [66]:
```
from skimage import filters

threshold = filters.threshold_otsu(img)
print(threshold)
```

```
14641
```

Here we use the function `skimage.filters.threshold_otsu`, which takes our image in. It then analyses the image histogram, and computes what it considers to be a reasonable threshold value, in this case 14641. Let us use this value to threshold our image

In [67]:
```
skull = img > filters.threshold_otsu(img)

plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title('Brain Scan')

plt.subplot(1, 2, 2)
plt.imshow(skull, cmap='gray')
plt.title('Thresholded skull')
plt.show()
```



The image of the right is the thresholded skull image, which is a boolean image. At this point you might wonder what the use of such an image is? For one it is possible to analyse this image directly to get out at least one value that might be of interest, which is the number of pixels the skull is. As the image is boolean, simply summing the image will give us the number of pixels

In [68]:
```
print(skull.sum())
```

```
3077
```

Thus 3077 pixels of the image is the skull. If we know the pixel size of the image, we can then make an area estimate.

However, much more useful (at least for most applications) is that the boolean image can be used as a *mask* in further analysis. Let us talk briefly about that soon. First, you should try your hand at segmenting.

Note by the way, that Otsu is just one of many algorithms for automatically selecting the threshold level. You can read more about thresholds and see examples on this page:

- https://scikit-image.org/docs/stable/auto_examples/applications/plot_thresholding.html (https://scikit-image.org/docs/stable/auto_examples/applications/plot_thresholding.html)

Your turn: Segmenting a cell

- a) Read in the example image `skimage.data.cell`
- b) Check the `shape` and `dtype` of the image
- c) Plot the image
- d) Use thresholding to segment out the cell itself
- e) How big is the cell (in the number of pixels)?

Masks

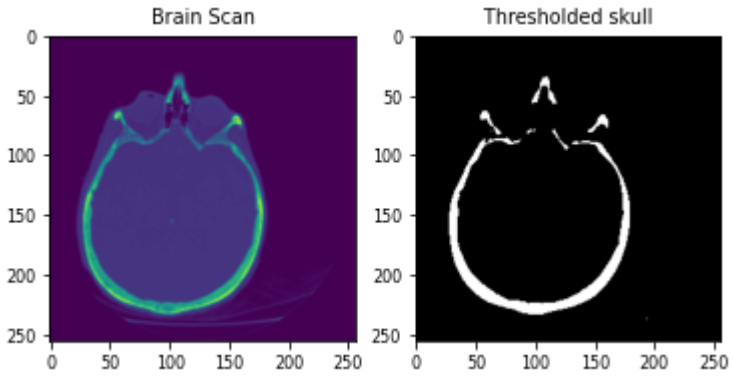
When performing digital image analysis, we are typically either analayzing, or manipulating a digital image. But in many cases we do not want to analyse or manipulate the *entire* image, but only parts of it.

Earlier we looked at cropping. Which is of course *one* option we can do, if we only want to focus on a smaller part of an image. However, due to the nature of numpy arrays, digital images always have to be *square*. But what if we only want to focus on some organ like a liver or a heart in a medical image - these organs are rarely square!

In these cases what we can do is create a *mask*, which is a boolean image saying which pixels of an image we are interested in and not. Many functions from both `skimage` and `cv2` can then take in such a mask as an optional argument, telling the algorithm to focus only on these parts of the image.

We can also use masks directly with the images, due to the nature of `numpy` arrays. Let us keep working on our skull example to show you. We had a slice of a brain scan, and we have a boolean threshold of the skull

In [69]:
```
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title('Brain Scan')

plt.subplot(1, 2, 2)
plt.imshow(skull, cmap='gray')
plt.title('Thresholded skull')
plt.show()
```



We can now use the boolean skull image as a mask, to select only the pixels of the skull. For example, we can compute the average of all the skull pixels in the original image, as follows:

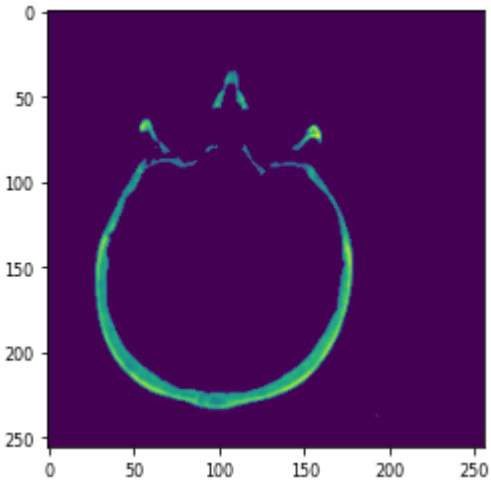
In [70]:
```
print("Average pixel intensity in full image:", np.mean(img))
print("Average pixel intensity only in skull:", np.mean(img, where=skull))
```

```
Average pixel intensity in full image: 3683.947494506836
Average pixel intensity only in skull: 27009.91257718557
```

So here we first use `np.mean(img)` to compute the average pixel intensity, which is around 3.7 thousand. But then we add the `where` keyword. And then we see that the average intensity *only in the skull pixels* is around 27 thousand, almost 8 times higher.

We can also use a boolean mask array to *index*. Let us for example say we want to set all pixels in the original image that aren't skull pixels to 0. Effectively we say everything else is background. We can do this in two ways. We can either multiply the image with the mask:

In [71]:
```
1 img_skull = img*skull
2
3 plt.imshow(img_skull)
4 plt.show()
```



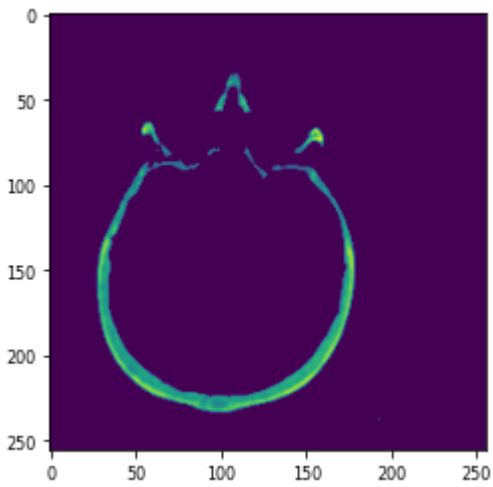
This image is quite similar to the skull *mask* image, in that only the skull pixels are non-zero. However, in this case the pixel intensities are not just 0 and 1, like they are in the mask.

The other option is to directly index the image with the mask. But now some care should be taken, as what we are trying to do is to select everything that is *not* the skull to set to zero. So we need to *inverted* mask, this can be done with `np.invert` or by simply writing `~` in front of the mask:

In [72]:
```
1 img[~skull] = 0
```

This line is very short, but it does quite a lot of cool things. It says: Take the image `img`, select all pixels that are *not skull* (`~skull`) and set these to 0.

In [73]:
```
plt.imshow(img)
plt.show()
```



These are just some examples of using masks. Sometimes we can find masks by thresholding, or something automated task. Other times we need to assemble them ourselves. for example, often we might want to make a circular mask to look at the center of an image or something like that. You can read more about making such masks here:

- https://datacarpentry.org/image-processing/04-drawing/ (https://datacarpentry.org/image-processing/04-drawing/)

Your turn: Finding average pixels in a water phantom

Let us return to the water phantom we had you read in earlier. You can find this image data as follows

In [74]:
```
import pydicom as dcm

data = dcm.dcmread("img/vannfantom.dcm")
img = data.pixel_array
```

- a) Print out the pixel type, and the shape of this image.
- b) Plot the image
- c) Compute the average pixel intensity and the standard deviation (`np.mean` and `np.std`)
- d) Use `filters.threshold_otsu` on the image to segment it, then compute again the mean and std of just the thresholded parts of the image

In this case the Otsu thresholded image gets the small gray segment at the top of the image. Let us instead create a circular mask towards the center. The code lines below do this:

In [75]:
```
import numpy as np
from skimage import draw

mask = np.zeros(img.shape, dtype=bool)
rr, cc = draw.disk((256, 256), 20, shape=img.shape)
mask[rr, cc] = 1
```

- e) Read the code lines and *attempt* to understand what is happening.
- f) Plot the mask image
- g) Use this mask image to compute the mean and standard deviation.

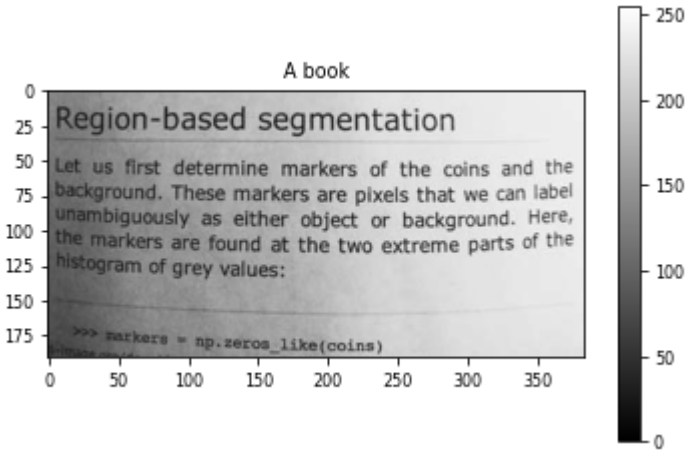
Local vs Global thresholding

We have so far shown a few examples of thresholding, and used `filters.threshold_otsu` to do this. Otsu works well when regions of interest are clearly seperate in an image histogram, and for this to be the case an image has to be evenly illuminated. In some cases however, this is clearly not the case. In such cases we might need to turn to a *local* threshold variant.

Let us now show an example to illustrate this. In this case we do not use a medical image, but use `skimage` 's own example, as I think it is quite illustrative. The main goal of this example is simply to highlight the difference between *global* approaches, which work on an entire image, and *local* approaches, which work on parts of an image. All algorithms can be divided into these two categories.

In this example we have an image of a book page, and we want to segment out the letters of ink using thresholding

In [76]:
```
from skimage import data
import matplotlib.pyplot as plt

img = data.page()

plt.imshow(img, cmap='gray')
plt.title('A book')
plt.colorbar()
plt.show()
```



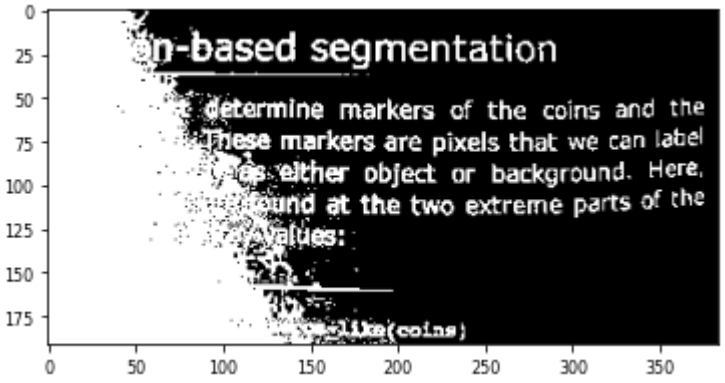
Let us check the image values

In [77]:
```
print(img.dtype)
print(img.shape)
print(img.min())
print(img.max())
```

```
uint8
(191, 384)
0
255
```

So we see that the image consists of values between 0 and 255. Let us try to threshold this page using the Otsu algorithm

In [78]:
```
from skimage import filters
import numpy as np

letters = img > filters.threshold_otsu(img)
letters = np.invert(letters)

plt.imshow(letters, cmap='gray')
plt.show()
```



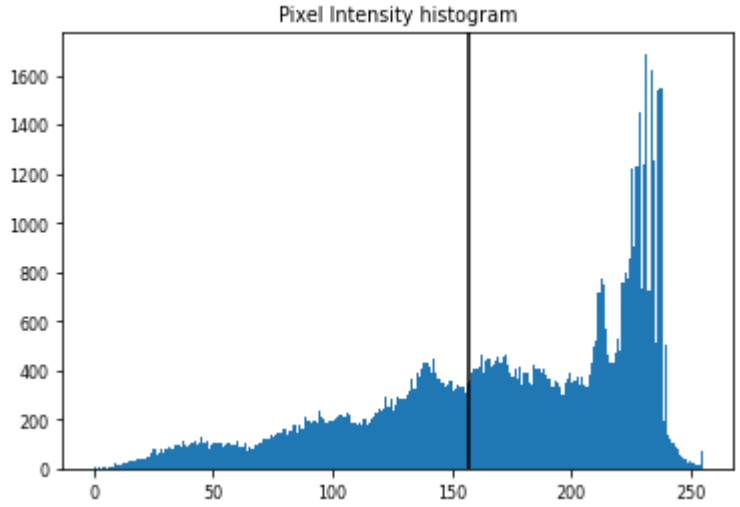
Note that we use `np.invert` to flip the page after we threshold it. This is simply because the letters are dark and the page is bright, so technically what the thresholding does is pick out the background. An easier way to do this would simply be to flip the `>` into a `<` in the boolean itself. However, we felt the invert method might be more intuitive.

We see that the Otsu algorithm manages to pick out most of the letters quite cleanly on the right hand side of the image, but on the left much of the page we get the background as well, this is simply because the page is not evenly lit in the source image.

This might feel like a short-coming in the Otsu thresholding algorithm, but the main issue isn't with the algorithm itself, but rather the fact that the page is not evenly illuminated

In [79]:

```
1 plt.hist(img.ravel(), bins=255)
2 threshold = filters.threshold_otsu(img)
3 plt.axvline(threshold, color='black')
4 plt.title('Pixel Intensity histogram')
5 plt.show()
```

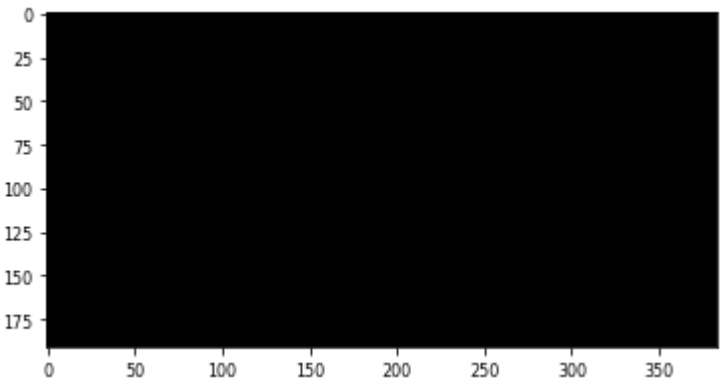


It simply doesn't matter exactly where we place the threshold. As the fact is the pixels of the background on the left, are *darker* than the letter pixels on the right. To make this easier to see, let us quickly whip together a Jupyter widget we can use to change the threshold in real time.

The use of Jupyter widgets is again out of scope of the course. So don't worry to much about the details here.

In [80]:
```
from ipywidgets import interact
import ipywidgets as widgets

def plot_img_with_given_threshold(threshold):
    plt.imshow(img < threshold, cmap='gray')
    plt.show()

interact(plot_img_with_given_threshold, threshold=widgets.IntSlider(min=img.
```

threshold 0



Out[80]: <function __main__.plot_img_with_given_threshold(threshold)>

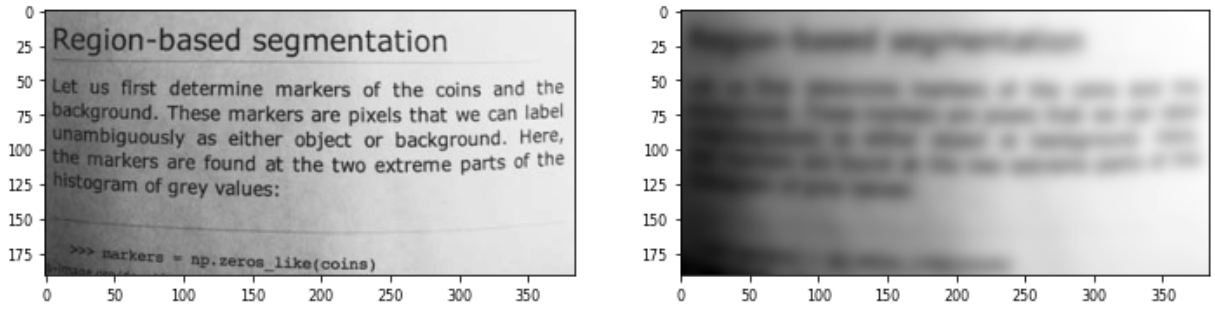
The point of this widget is to illustrate that by the time we increase the threshold sufficiently to properly segment out the letters, we start to pick up background. This is again because of the uneven illumination of the page.

A *local* threshold limit

Because the image is unevely illuminated, we cannot find a singular *global* threshold to use. However, note that every single letter in the image is darker than its immediate surroundings. This means that we can probably create a *local* threshold to use.

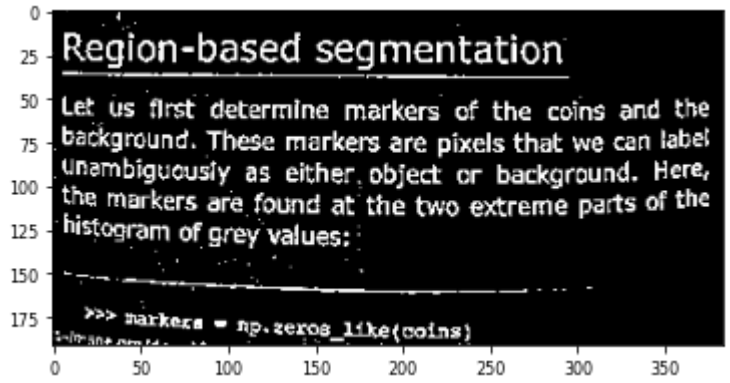
To do this we will first create a *blurred* copy of the image. This will give us a sort of reference about how light or dark a portion of the image is. We can then compare each pixel to this blurred image, effectively comparing each pixel to its surroundings.

In [81]:
```
1 blurred_img = filters.gaussian(img, sigma=5)
2
3 plt.figure(figsize=(12, 10))
4 plt.subplot(121)
5 plt.imshow(img, cmap='gray')
6
7 plt.subplot(122)
8 plt.imshow(blurred_img, cmap='gray')
9 plt.show()
```



The `sigma` keyword is a parameter adjusting how large the gaussian to be used in the blurring is. The higher the Gaussian, the larger of an area we effectively average out. Now that we have a blurred copy, we can compare the image to the blurred copy. To get a proper segmentation, it is useful to also add in and offset

In [82]:
```
1 sigma = 3
2 offset = 10
3
4 # Blur image to create background
5 blurred_img = filters.gaussian(img, sigma=sigma, preserve_range=True)
6
7 # Threshold image locally
8 letters = img < blurred_img - offset
9
10
11 plt.imshow(letters, cmap='gray')
12 plt.show()
```



By playing around with the two parameters `sigma` and `offset`, we can get quite a nice segmentation out.

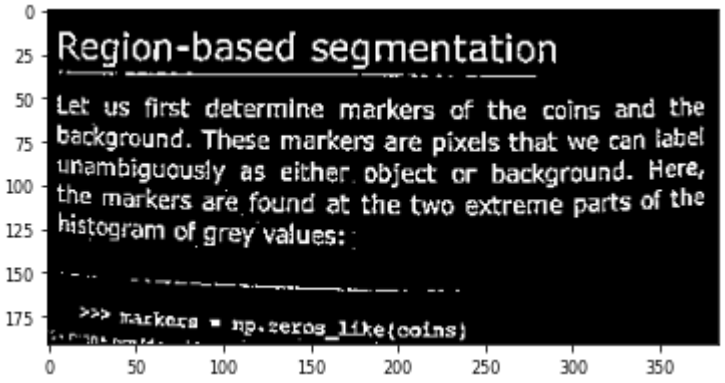
Here we have effectively created our own thresholding algorithm by combining *gaussian blurring* and *binary thresholding*. And this is often how digital image analysis is done, by combining many small steps to produce a final analysis or outcome.

Please note that `skimage` actually supports local threhsolding out of the box, and so we mostly implemented it as a learning exercise. You can read about the function here:

- https://scikit-image.org/docs/stable/api/skimage.filters.html#skimage.filters.threshold_local (https://scikit-image.org/docs/stable/api/skimage.filters.html#skimage.filters.threshold_local)

So we could use it directly as follows:

In [83]:
```
from skimage.filters import threshold_local

plt.imshow(img < threshold_local(img, block_size=15, offset=15), cmap='gray'
plt.show()
```



In the function `filters.threshold_local` there are two parameters: `block_size` and `offset`. And by implementing our own version first, it might be easier to understand what these two parameters represent. Either way, *most* digital image analysis have such adjustable parameters. You often need to play around with them to get an algorithm to work on your use case. However, you also need to take some care if you are using batch processing, and make sure whatever script or pipeline you are working on is reliable enough to work on several images, not just an individual image.

Denoising

As a final example I did want to cover a bit about *denoising* images. Which is a slightly more advanced topic. However, with just 3 hours I ran out of time.

If you want to learn more about denoising images and look at some examples you can take a look at the `skimage` example gallery:

- https://scikit-image.org/docs/stable/auto_examples/index.html (https://scikit-image.org/docs/stable/auto_examples/index.html)

Here there are examples using different denoising algorithms, including non-local means and wavelet denoising.

If you are working with MRI-data, you can also look at the `dipy` package, they have tutorials on a few different kinds of denoising (such as non-local means and PCA. You can find these examples here:

- https://dipy.org/tutorials/ (https://dipy.org/tutorials/)

I will include the code from the tutorial on non-local means in `dipy` here to round out the notebook:

- Source: https://dipy.org/documentation/1.5.0/examples_built/denoise_nlmeans/#example-denoise-nlmeans (https://dipy.org/documentation/1.5.0/examples_built/denoise_nlmeans/#example-denoise-nlmeans)

In [84]:
```
import numpy as np
import matplotlib.pyplot as plt
from time import time
from dipy.denoise.nlmeans import nlmeans
from dipy.denoise.noise_estimate import estimate_sigma
from dipy.data import get_fnames
from dipy.io.image import load_nifti, save_nifti


t1_fname = get_fnames('stanford_t1')
data, affine = load_nifti(t1_fname)

mask = data > 1500

print("vol size", data.shape)
```

```
vol size (81, 106, 76)
```

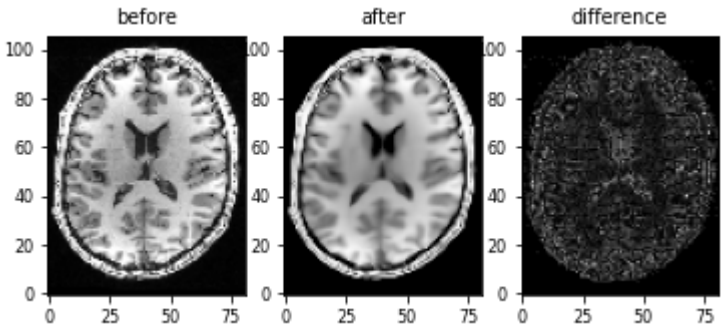
This code imports necessary packages, reads in the data and makes a simple mask from thresholding (as we covered above)

In [85]:
```
sigma = estimate_sigma(data, N=32)

t = time()

den = nlmeans(data, sigma=sigma, mask=mask, patch_radius=1,
              block_radius=2, rician=True)

print("total time", time() - t)
```

```
total time 0.0987863540649414
```

Here is the code that does the denosining itself. And now we can plot the results

In [86]:
```
axial_middle = data.shape[2] // 2

before = data[:, :, axial_middle].T
after = den[:, :, axial_middle].T

difference = np.abs(after.astype(np.float64) - before.astype(np.float64))

difference[~mask[:, :, axial_middle].T] = 0


fig, ax = plt.subplots(1, 3)
ax[0].imshow(before, cmap='gray', origin='lower')
ax[0].set_title('before')
ax[1].imshow(after, cmap='gray', origin='lower')
ax[1].set_title('after')
ax[2].imshow(difference, cmap='gray', origin='lower')
ax[2].set_title('difference')

plt.savefig('denoised.png', bbox_inches='tight')
```



And finally we can save the results

In [87]:
```
save_nifti('results/denoised.nii.gz', den, affine)
```

Additional Resources and Tutorials:

Scikit-image user guide

- https://scikit-image.org/docs/stable/user_guide.html (https://scikit-image.org/docs/stable/user_guide.html)

Scikit-image example gallery

- https://scikit-image.org/docs/stable/auto_examples/index.html (https://scikit-image.org/docs/stable/auto_examples/index.html)

Data Carpentry image processing course module

- https://datacarpentry.org/image-processing/ (https://datacarpentry.org/image-processing/)

Dipy package tutorial (for Diffusion MRI images)

- https://dipy.org/tutorials/ (https://dipy.org/tutorials/)

Pydicom package tutorial

- https://pydicom.github.io/pydicom/stable/auto_examples/index.html (https://pydicom.github.io/pydicom/stable/auto_examples/index.html)