Using napari from Jupyter notebooks

Marcelo Leomil Zoccoler

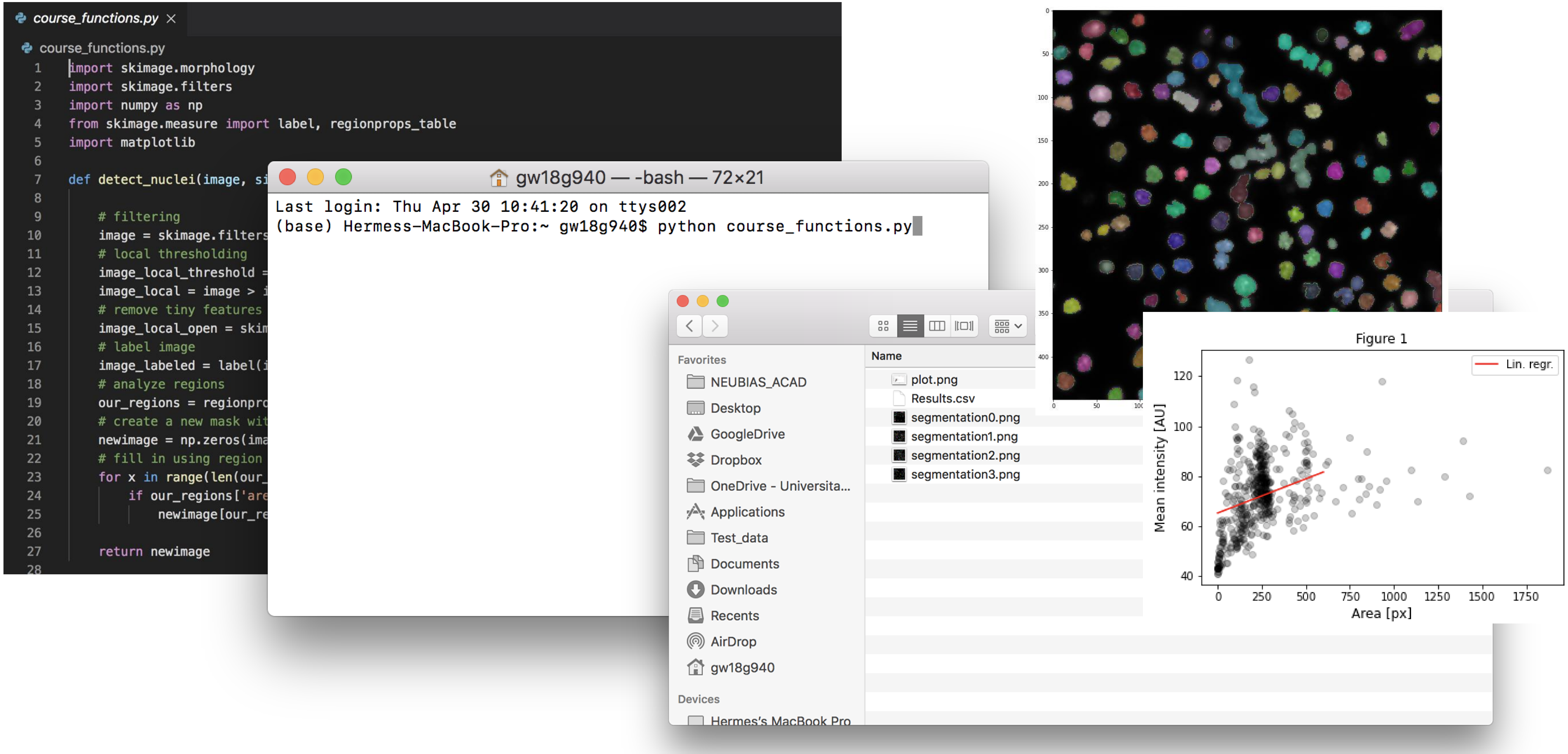
With material from

Robert Haase, BiAPoL, PoL TU Dresden

Guillaume Witz, Universität Bern

September 2022

“Classic” software vs. notebooks



course_functions.py ×

course_functions.py

```
import skimage.morphology
import skimage.filters
import numpy as np
from skimage.measure import label, regionprops_table
import matplotlib

def detect_nuclei(image, si

    # filtering
    image = skimage.filters
    # local thresholding
    image_local_threshold =
    image_local = image > i
    # remove tiny features
    image_local_open = skim
    # label image
    image_labeled = label(i
    # analyze regions
    our_regions = regionpro
    # create a new mask wit
    newimage = np.zeros(ima
    # fill in using region
    for x in range(len(our_
        if our_regions['are
            newimage[our_re

    return newimage
```

gw18g940 — -bash — 72×21

```
Last login: Thu Apr 30 10:41:20 on ttys002
(base) Hermess-MacBook-Pro:~ gw18g940$ python course_functions.py
```

Name

- plot.png
- Results.csv
- segmentation0.png
- segmentation1.png
- segmentation2.png
- segmentation3.png

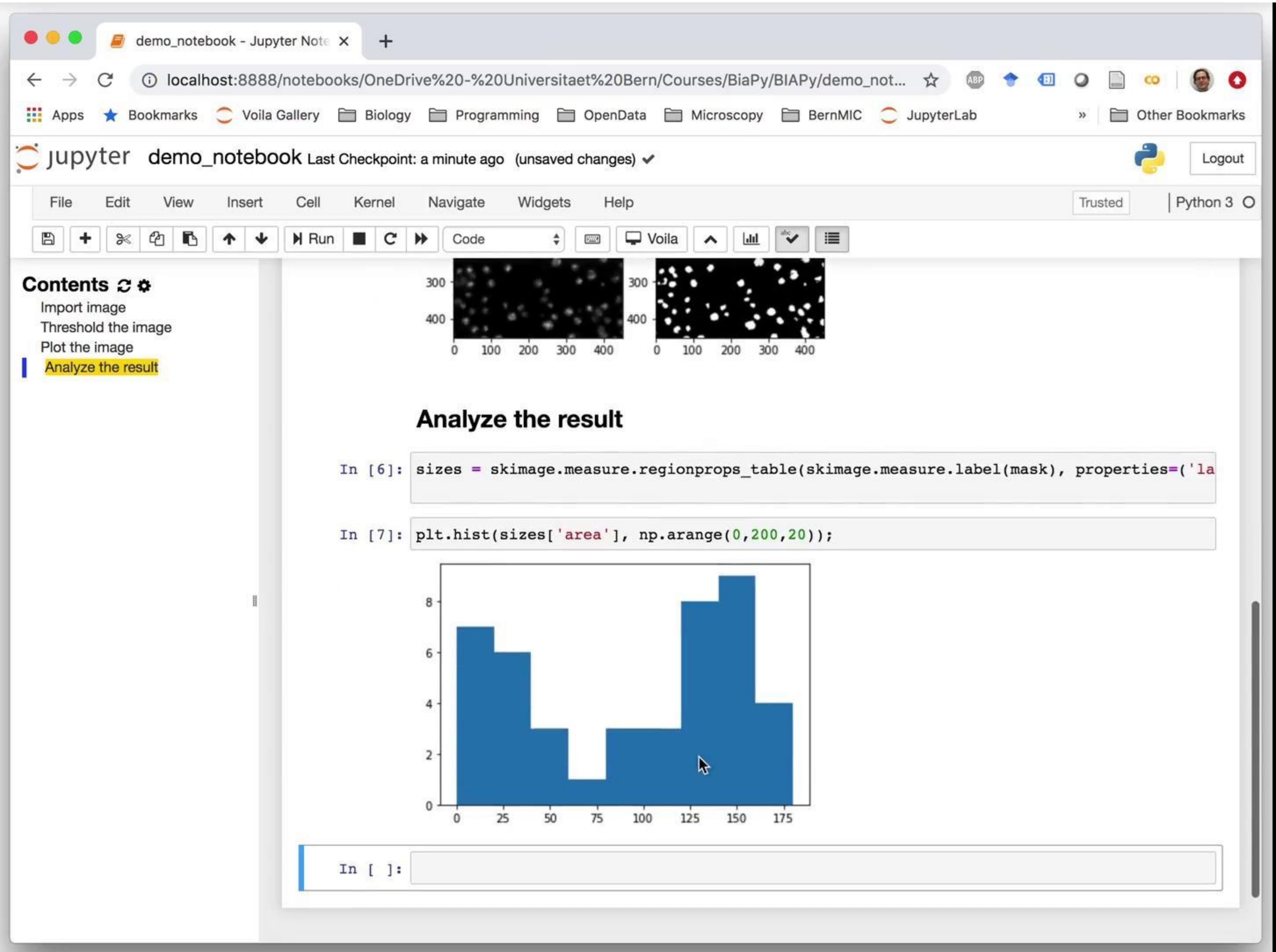
“Classic” software vs. notebooks

Code divided in parts

Dynamic: easy to test

Rich output: processing + visualisation + analysis in one place

Code + formatted text: easy documentation



Analyze the result

```
In [6]: sizes = skimage.measure.regionprops_table(skimage.measure.label(mask), properties=('la
```

```
In [7]: plt.hist(sizes['area'], np.arange(0,200,20));
```

```
In [ ]:
```

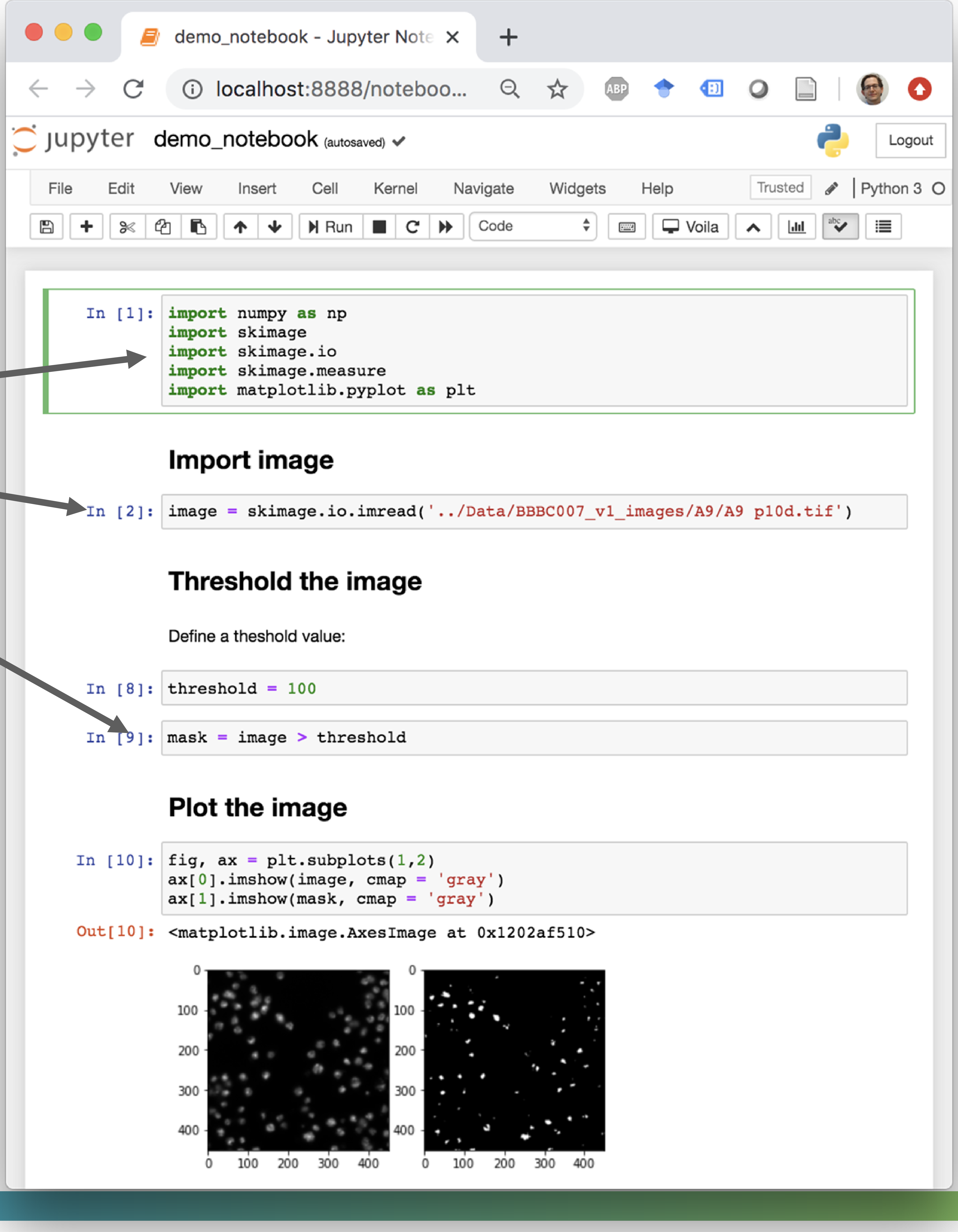
Illustration: load an image, threshold it, analyze object size

What is a jupyter notebook?

A text file (easily sent around)

Rendered by Jupyter in the browser

Split into sections called cells



```
In [1]: import numpy as np
        import skimage
        import skimage.io
        import skimage.measure
        import matplotlib.pyplot as plt
```

Import image

```
In [2]: image = skimage.io.imread('../Data/BBBC007_v1_images/A9/A9 p10d.tif')
```

Threshold the image

Define a theshold value:

```
In [8]: threshold = 100

In [9]: mask = image > threshold
```

Plot the image

```
In [10]: fig, ax = plt.subplots(1,2)
         ax[0].imshow(image, cmap = 'gray')
         ax[1].imshow(mask, cmap = 'gray')

Out[10]: <matplotlib.image.AxesImage at 0x1202af510>
```

What is a jupyter notebook?

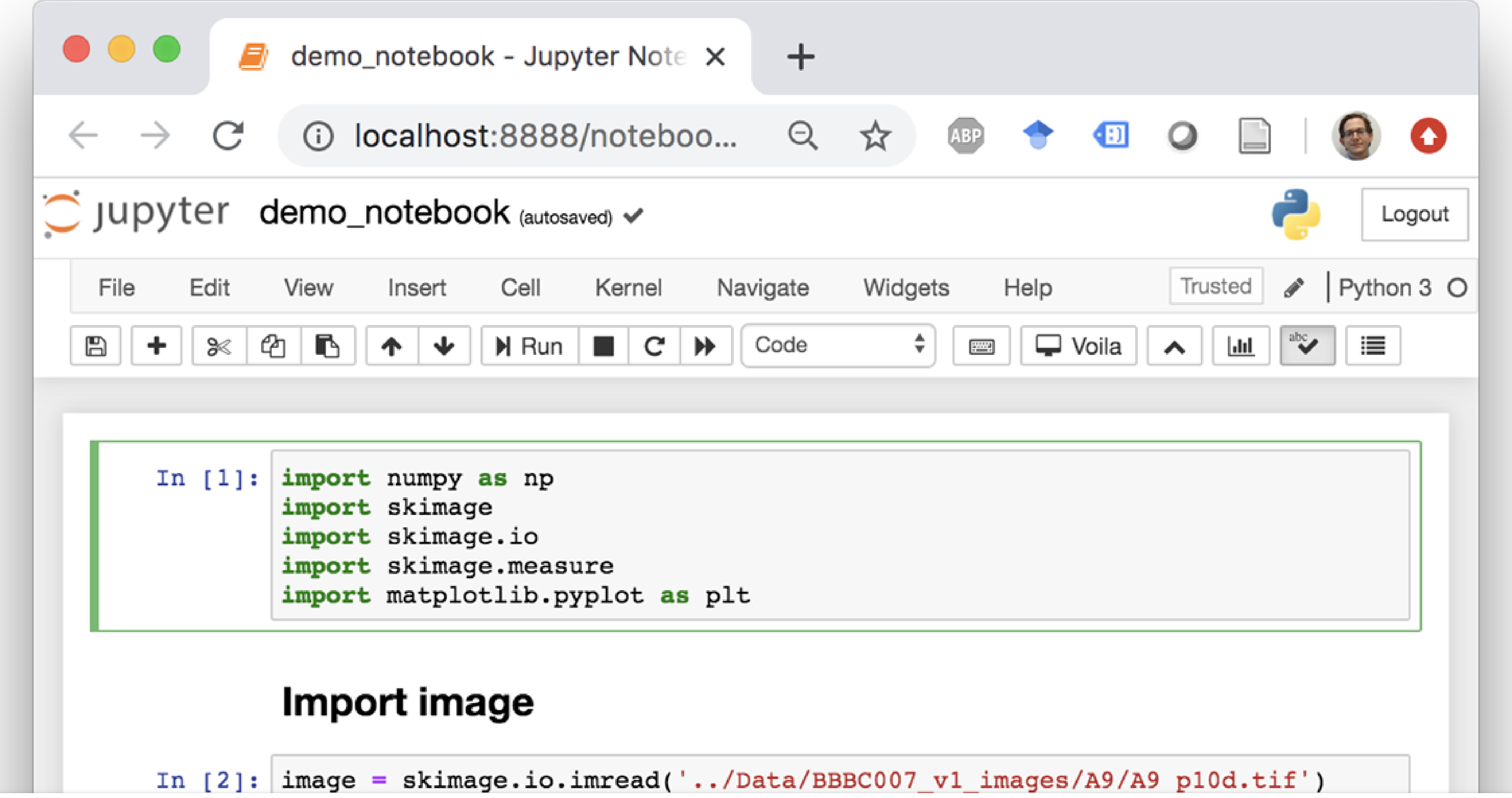
A text file (easily sent around)

Rendered by Jupyter in the browser

Split into sections called cells

Cells can contain:

- Code
- Formatted text
- Rich output



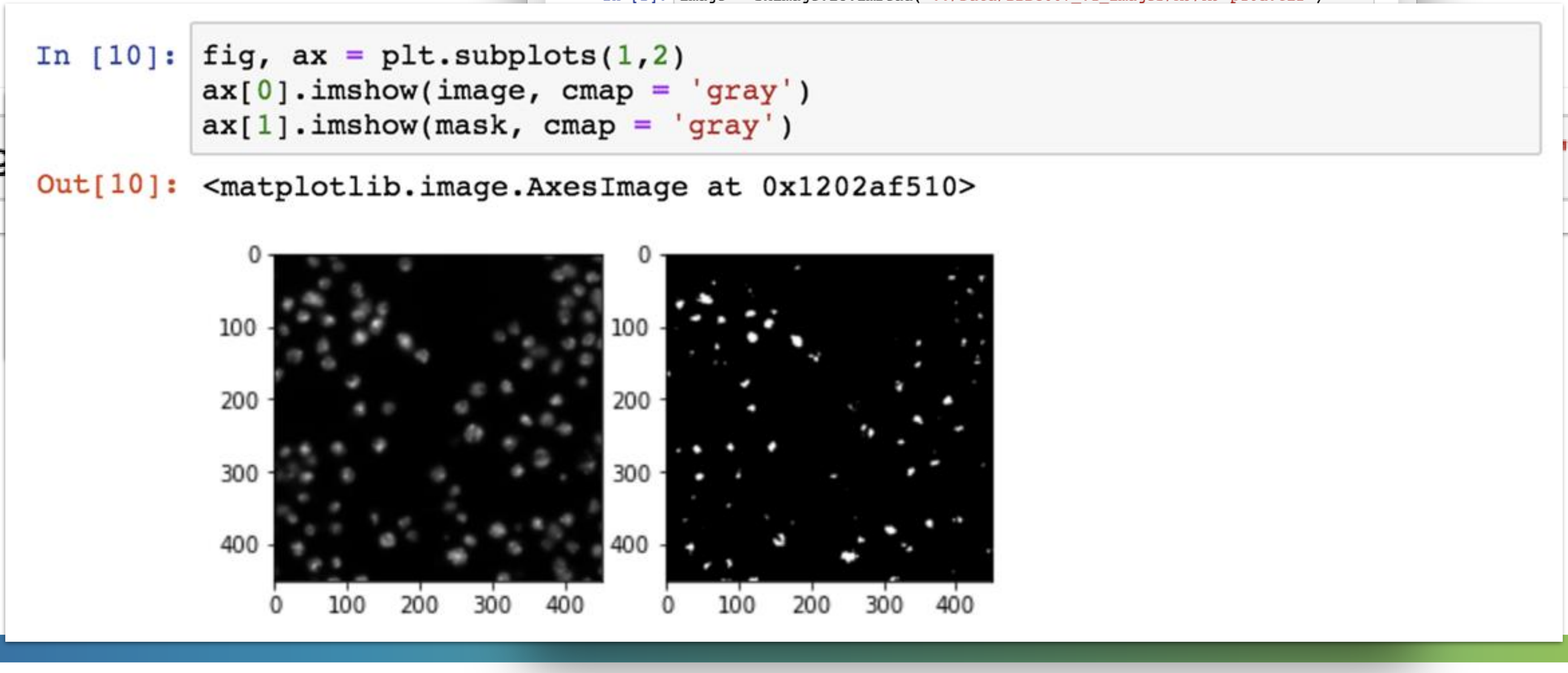
demo_notebook - Jupyter Note
localhost:8888/noteboo...
jupyter demo_notebook (autosaved)
Logout
File Edit View Insert Cell Kernel Navigate Widgets Help
Trusted
Python 3
Run
Code
Voila

```
In [1]: import numpy as np
        import skimage
        import skimage.io
        import skimage.measure
        import matplotlib.pyplot as plt
```

Import image

```
In [2]: image = skimage.io.imread('../Data/BBBC007_v1_images/A9/A9 p1d.tif')
```

```
In [2]: imag
```



```
In [10]: fig, ax = plt.subplots(1,2)
         ax[0].imshow(image, cmap = 'gray')
         ax[1].imshow(mask, cmap = 'gray')

Out[10]: <matplotlib.image.AxesImage at 0x1202af510>
```

Why using notebooks?

Documenting (for your-(future)-self and for others) and enhanced reproducibility

“First, we applied a Gaussian filter from scikit-image with sigma = 1. Then, we applied another Gaussian filter to the original image with sigma = 10. After that, we subtracted the first result from the second…

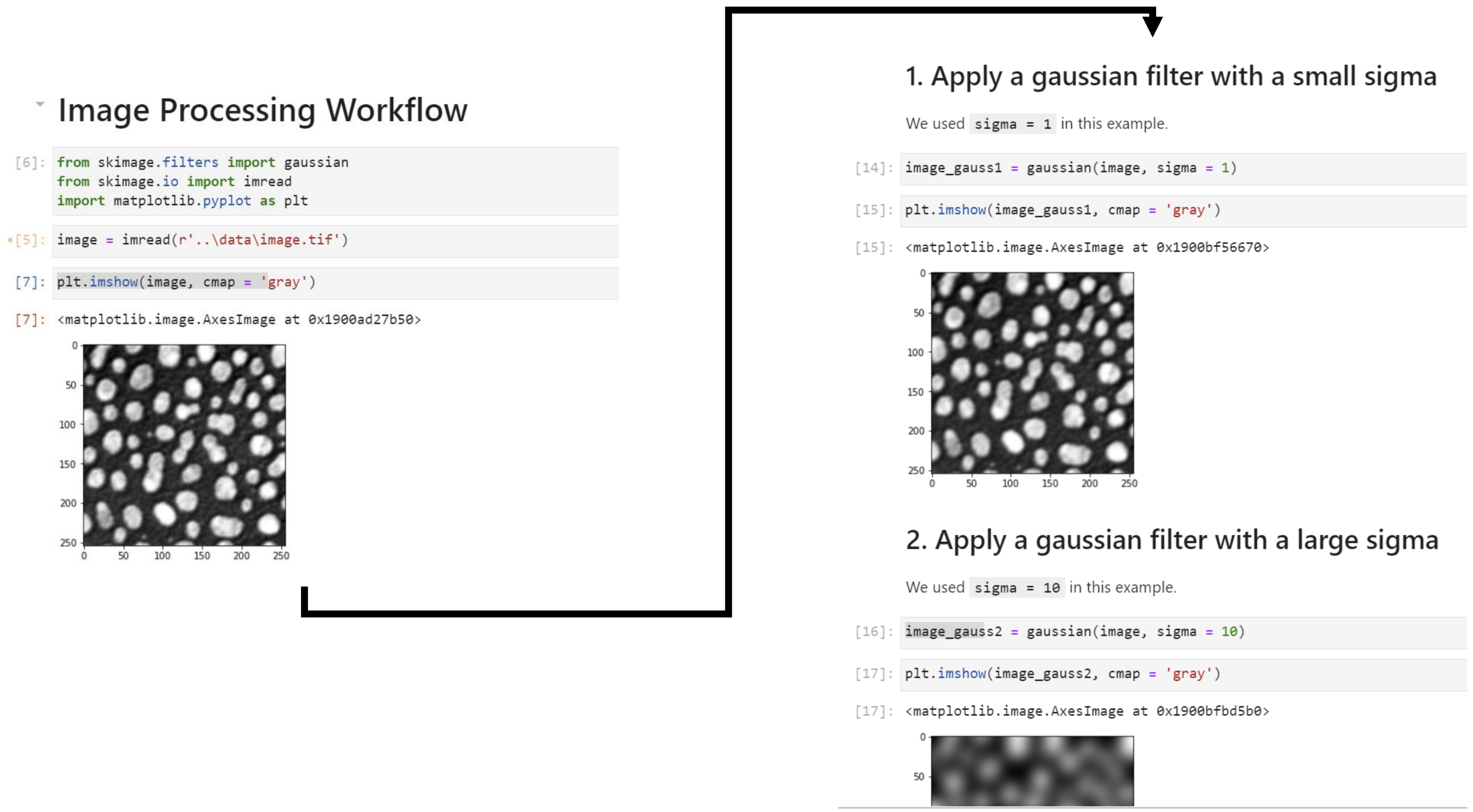


Image Processing Workflow

```
[6]: from skimage.filters import gaussian
     from skimage.io import imread
     import matplotlib.pyplot as plt

•[5]: image = imread(r'..\data\image.tif')

[7]: plt.imshow(image, cmap = 'gray')

[7]: <matplotlib.image.AxesImage at 0x1900ad27b50>
```

1. Apply a gaussian filter with a small sigma

We used `sigma = 1` in this example.

```
[14]: image_gauss1 = gaussian(image, sigma = 1)

[15]: plt.imshow(image_gauss1, cmap = 'gray')

[15]: <matplotlib.image.AxesImage at 0x1900bf56670>
```

2. Apply a gaussian filter with a large sigma

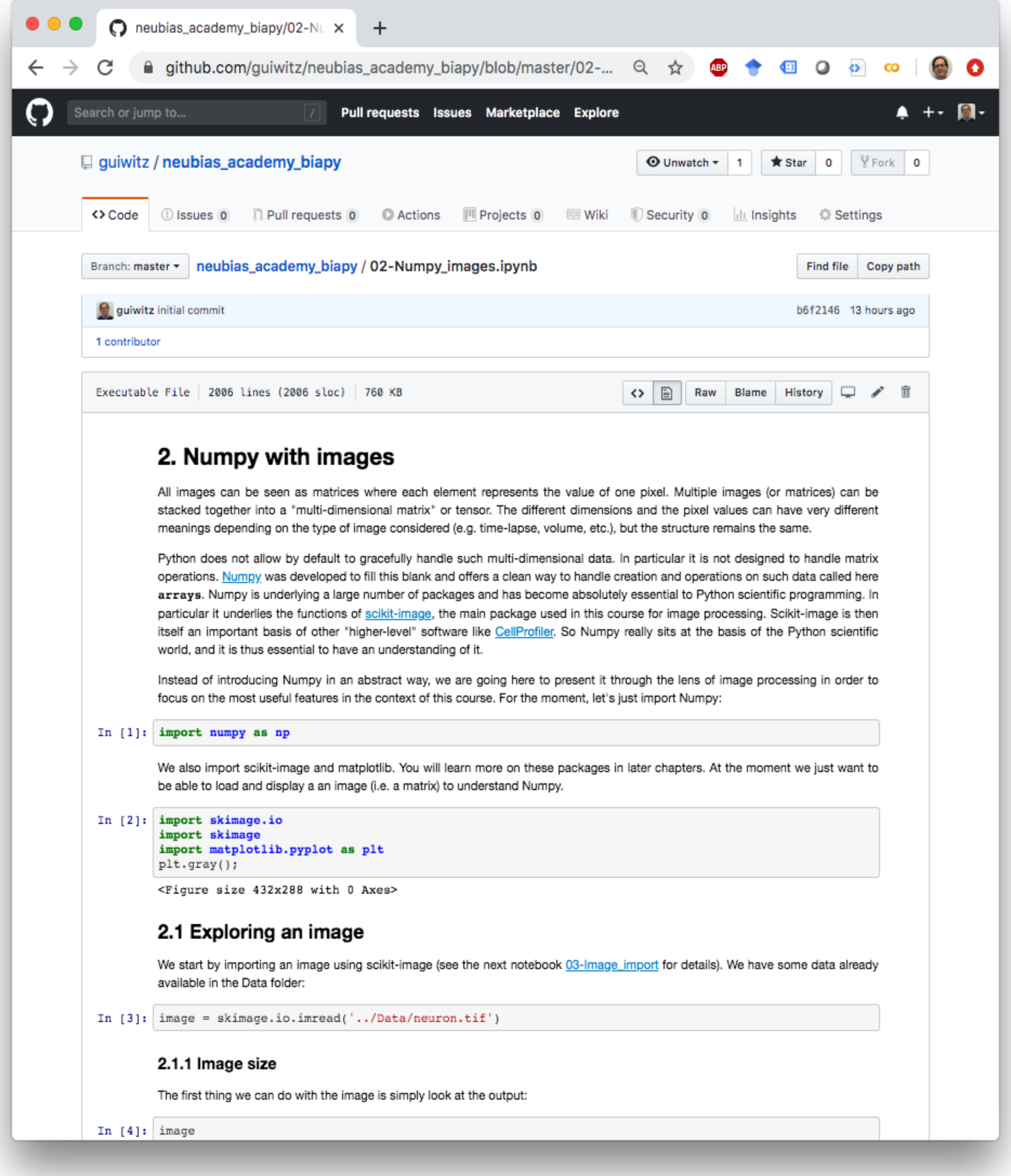
We used `sigma = 10` in this example.

```
[16]: image_gauss2 = gaussian(image, sigma = 10)

[17]: plt.imshow(image_gauss2, cmap = 'gray')

[17]: <matplotlib.image.AxesImage at 0x1900bfbd5b0>
```

Why using notebooks?

Sharing

- Send a single file with code, intermediary outputs and rich text explanations



github.com/guiwitz/neubias_academy_biapy/blob/master/02-...
Pull requests Issues Marketplace Explore
guiwitz / neubias_academy_biapy
Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security 0 Insights Settings
Branch: master
neubias_academy_biapy / 02-Numpy_images.ipynb
Find file Copy path
guiwitz initial commit
1 contributor
Executable File 2086 lines (2086 sloc) 760 KB
Raw Blame History

2. Numpy with images

All images can be seen as matrices where each element represents the value of one pixel. Multiple images (or matrices) can be stacked together into a "multi-dimensional matrix" or tensor. The different dimensions and the pixel values can have very different meanings depending on the type of image considered (e.g. time-lapse, volume, etc.), but the structure remains the same.

Python does not allow by default to gracefully handle such multi-dimensional data. In particular it is not designed to handle matrix operations. Numpy was developed to fill this blank and offers a clean way to handle creation and operations on such data called here **arrays**. Numpy is underlying a large number of packages and has become absolutely essential to Python scientific programming. In particular it underlies the functions of scikit-image, the main package used in this course for image processing. Scikit-image is then itself an important basis of other "higher-level" software like CellProfiler. So Numpy really sits at the basis of the Python scientific world, and it is thus essential to have an understanding of it.

Instead of introducing Numpy in an abstract way, we are going here to present it through the lens of image processing in order to focus on the most useful features in the context of this course. For the moment, let's just import Numpy:

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

We also import scikit-image and matplotlib. You will learn more on these packages in later chapters. At the moment we just want to be able to load and display a an image (i.e. a matrix) to understand Numpy.

```
In [2]: import skimage.io
        import skimage
        import matplotlib.pyplot as plt
        plt.gray();

<Figure size 432x288 with 0 Axes>
```

2.1 Exploring an image

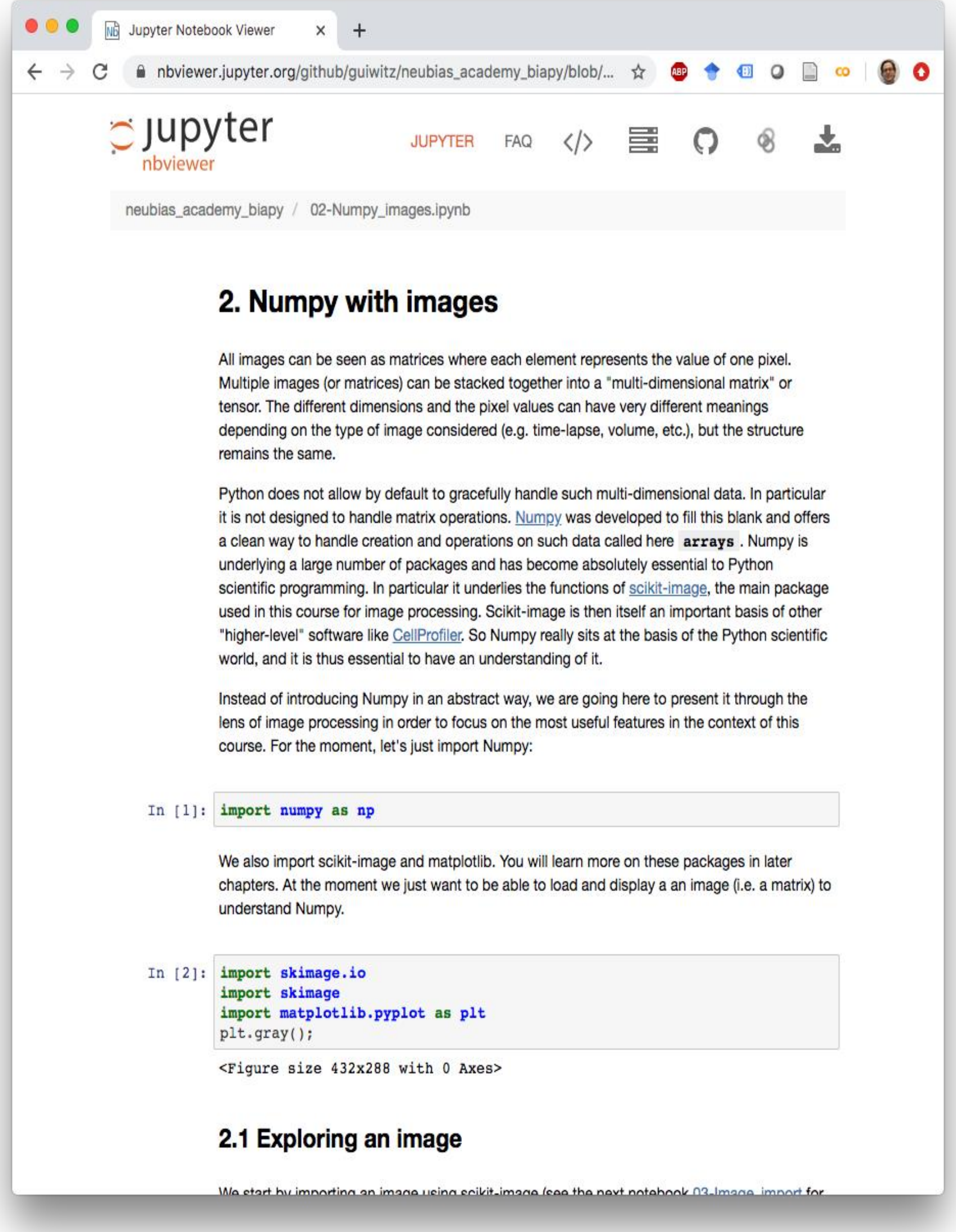
We start by importing an image using scikit-image (see the next notebook 03-Image_import for details). We have some data already available in the Data folder:

```
In [3]: image = skimage.io.imread('../Data/neuron.tif')
```

2.1.1 Image size

The first thing we can do with the image is simply look at the output:

```
In [4]: image
```



Jupyter Notebook Viewer
nbviewer.jupyter.org/github/guiwitz/neubias_academy_biapy/blob/...
jupyter nbviewer
JUPYTER FAQ
neubias_academy_biapy / 02-Numpy_images.ipynb

2. Numpy with images

All images can be seen as matrices where each element represents the value of one pixel. Multiple images (or matrices) can be stacked together into a "multi-dimensional matrix" or tensor. The different dimensions and the pixel values can have very different meanings depending on the type of image considered (e.g. time-lapse, volume, etc.), but the structure remains the same.

Python does not allow by default to gracefully handle such multi-dimensional data. In particular it is not designed to handle matrix operations. Numpy was developed to fill this blank and offers a clean way to handle creation and operations on such data called here arrays. Numpy is underlying a large number of packages and has become absolutely essential to Python scientific programming. In particular it underlies the functions of scikit-image, the main package used in this course for image processing. Scikit-image is then itself an important basis of other "higher-level" software like CellProfiler. So Numpy really sits at the basis of the Python scientific world, and it is thus essential to have an understanding of it.

Instead of introducing Numpy in an abstract way, we are going here to present it through the lens of image processing in order to focus on the most useful features in the context of this course. For the moment, let's just import Numpy:

In [1]: import numpy as np

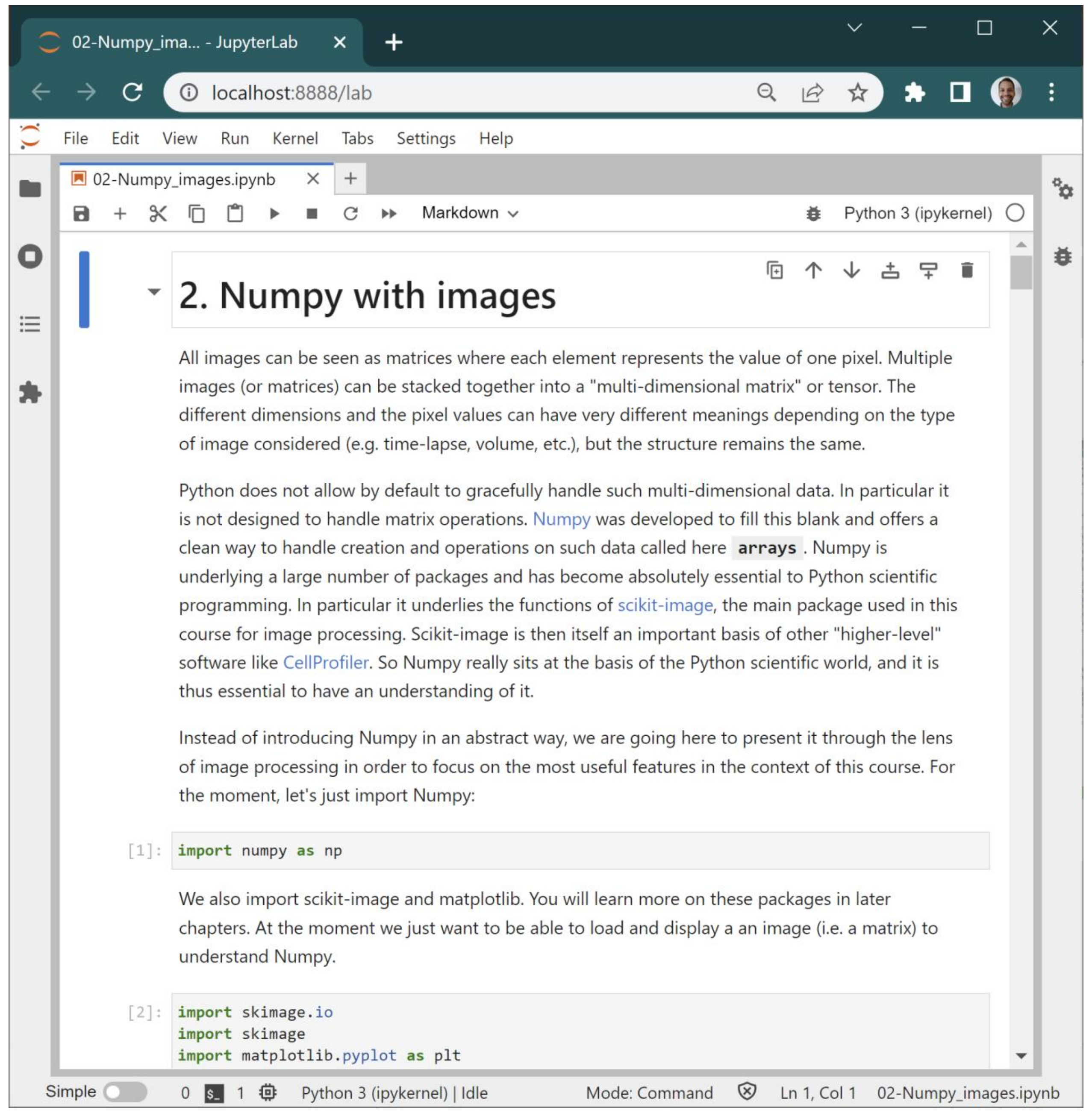
We also import scikit-image and matplotlib. You will learn more on these packages in later chapters. At the moment we just want to be able to load and display a an image (i.e. a matrix) to understand Numpy.

In [2]: import skimage.io
import skimage
import matplotlib.pyplot as plt
plt.gray();

<Figure size 432x288 with 0 Axes>

2.1 Exploring an image

We start by importing an image using scikit-image (see the next notebook 03-Image_import for



02-Numpy_ima... - JupyterLab
localhost:8888/lab
File Edit View Run Kernel Tabs Settings Help
02-Numpy_images.ipynb
Markdown
Python 3 (ipykernel)
Simple 0 1 Python 3 (ipykernel) | Idle Mode: Command Ln 1, Col 1 02-Numpy_images.ipynb

2. Numpy with images

All images can be seen as matrices where each element represents the value of one pixel. Multiple images (or matrices) can be stacked together into a "multi-dimensional matrix" or tensor. The different dimensions and the pixel values can have very different meanings depending on the type of image considered (e.g. time-lapse, volume, etc.), but the structure remains the same.

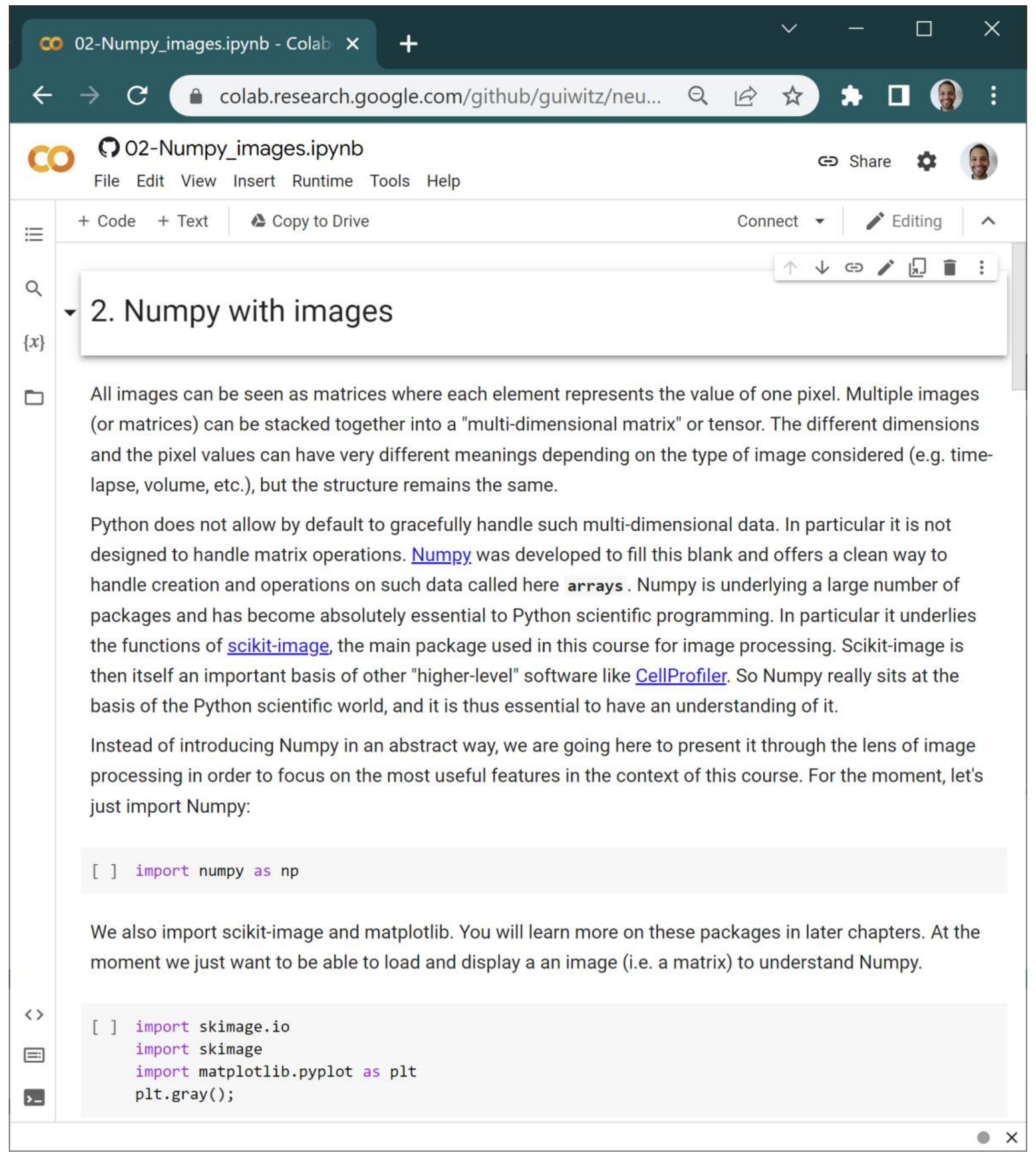
Python does not allow by default to gracefully handle such multi-dimensional data. In particular it is not designed to handle matrix operations. Numpy was developed to fill this blank and offers a clean way to handle creation and operations on such data called here arrays. Numpy is underlying a large number of packages and has become absolutely essential to Python scientific programming. In particular it underlies the functions of scikit-image, the main package used in this course for image processing. Scikit-image is then itself an important basis of other "higher-level" software like CellProfiler. So Numpy really sits at the basis of the Python scientific world, and it is thus essential to have an understanding of it.

Instead of introducing Numpy in an abstract way, we are going here to present it through the lens of image processing in order to focus on the most useful features in the context of this course. For the moment, let's just import Numpy:

[1]: import numpy as np

We also import scikit-image and matplotlib. You will learn more on these packages in later chapters. At the moment we just want to be able to load and display a an image (i.e. a matrix) to understand Numpy.

[2]: import skimage.io
import skimage
import matplotlib.pyplot as plt



02-Numpy_images.ipynb - Colab
colab.research.google.com/github/guiwitz/neu...
02-Numpy_images.ipynb
File Edit View Insert Runtime Tools Help
Share
+ Code + Text Copy to Drive
Connect Editing

2. Numpy with images

All images can be seen as matrices where each element represents the value of one pixel. Multiple images (or matrices) can be stacked together into a "multi-dimensional matrix" or tensor. The different dimensions and the pixel values can have very different meanings depending on the type of image considered (e.g. time-lapse, volume, etc.), but the structure remains the same.

Python does not allow by default to gracefully handle such multi-dimensional data. In particular it is not designed to handle matrix operations. Numpy was developed to fill this blank and offers a clean way to handle creation and operations on such data called here arrays. Numpy is underlying a large number of packages and has become absolutely essential to Python scientific programming. In particular it underlies the functions of scikit-image, the main package used in this course for image processing. Scikit-image is then itself an important basis of other "higher-level" software like CellProfiler. So Numpy really sits at the basis of the Python scientific world, and it is thus essential to have an understanding of it.

Instead of introducing Numpy in an abstract way, we are going here to present it through the lens of image processing in order to focus on the most useful features in the context of this course. For the moment, let's just import Numpy:

[ ] import numpy as np

We also import scikit-image and matplotlib. You will learn more on these packages in later chapters. At the moment we just want to be able to load and display a an image (i.e. a matrix) to understand Numpy.

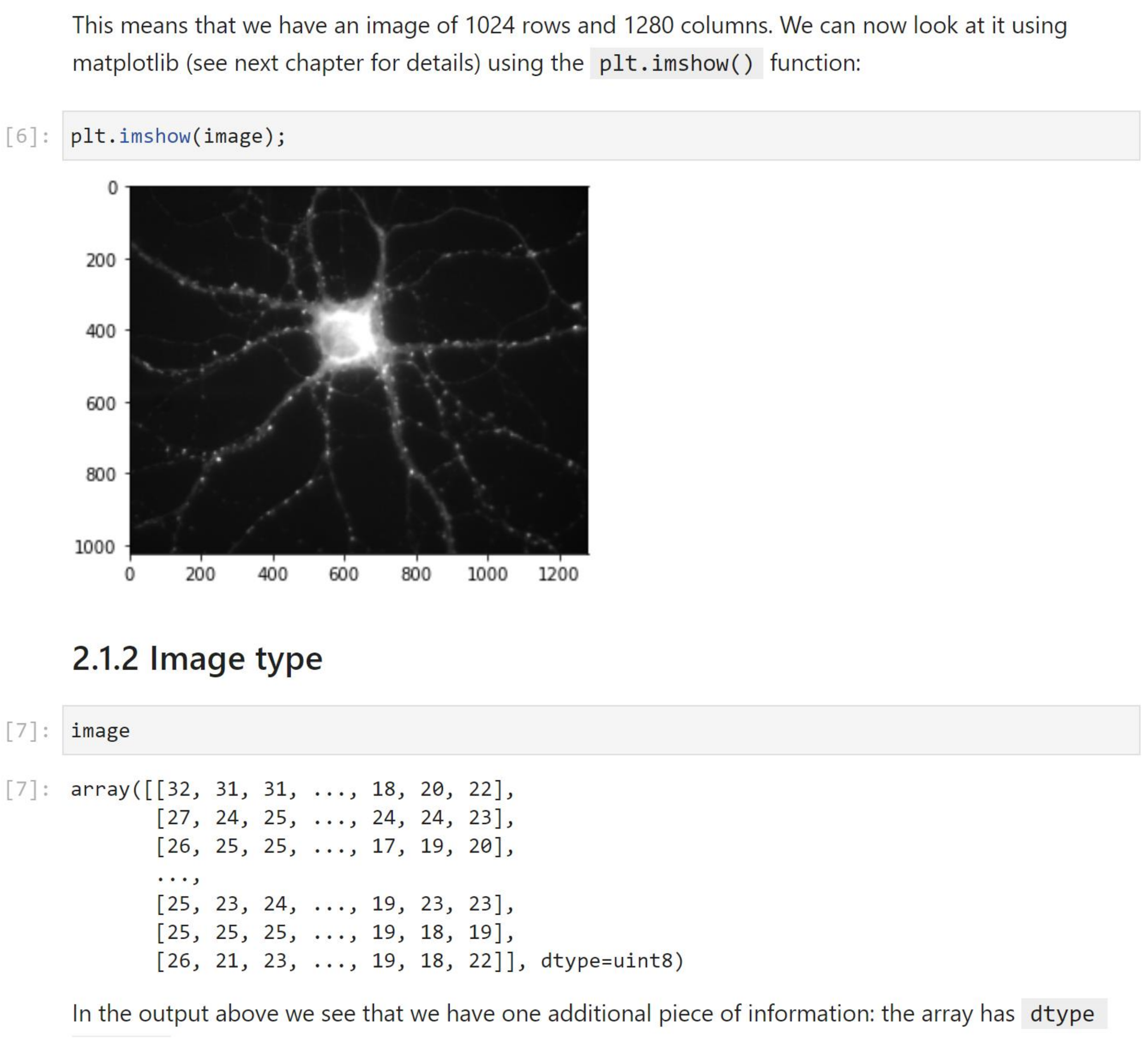
[ ] import skimage.io
import skimage
import matplotlib.pyplot as plt
plt.gray();

Github rendering **Nbviewer rendering** **Jupyter lab (local or Binder)** **Google Colab**

Why using notebooks?

Teaching

- Small blocks of code with intermediary results are easier to understand than scripts that spit tons of outputs in sequence



This means that we have an image of 1024 rows and 1280 columns. We can now look at it using matplotlib (see next chapter for details) using the `plt.imshow()` function:

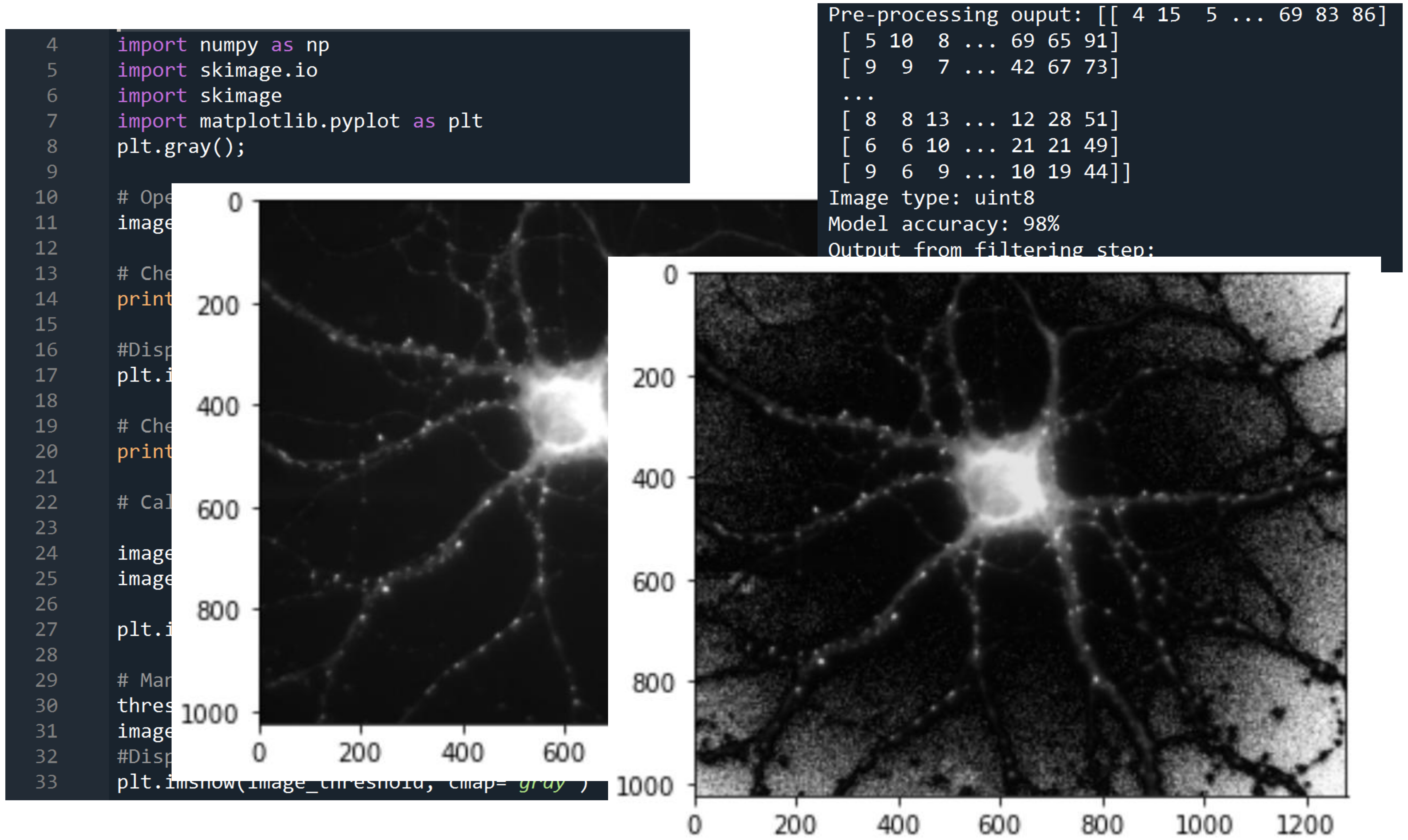
```
[6]: plt.imshow(image);
```

2.1.2 Image type

```
[7]: image
```

```
[7]: array([[32, 31, 31, ..., 18, 20, 22],
            [27, 24, 25, ..., 24, 24, 23],
            [26, 25, 25, ..., 17, 19, 20],
            ...,
            [25, 23, 24, ..., 19, 23, 23],
            [25, 25, 25, ..., 19, 18, 19],
            [26, 21, 23, ..., 19, 18, 22]], dtype=uint8)
```

In the output above we see that we have one additional piece of information: the array has `dtype`



```
import numpy as np
import skimage.io
import skimage
import matplotlib.pyplot as plt
plt.gray();

# Ope
image

# Che
print

#Disp
plt.i

# Che
print

# Cal

image
image

plt.i

# Mar
thres
image
#Disp
plt.imshow(image_threshold, cmap='gray')
```

```
Pre-processing ouput: [[ 4 15  5 ... 69 83 86]
 [ 5 10  8 ... 69 65 91]
 [ 9  9  7 ... 42 67 73]
 ...
 [ 8  8 13 ... 12 28 51]
 [ 6  6 10 ... 21 21 49]
 [ 9  6  9 ... 10 19 44]]
Image type: uint8
Model accuracy: 98%
Output from filtering step:
```

Why using notebooks?

Keep all the benefits from using code:

- Batch processing

- Running python functions/tools that do not have UI

Why using notebooks with napari?

- Easy data interaction and visualization with napari:
    - Great for visualizing 3D (and more) data
    - Each processing step result can be displayed as a separate layer
- Data annotation

Example: Trying out a workflow in napari

1. Start napari from the command line

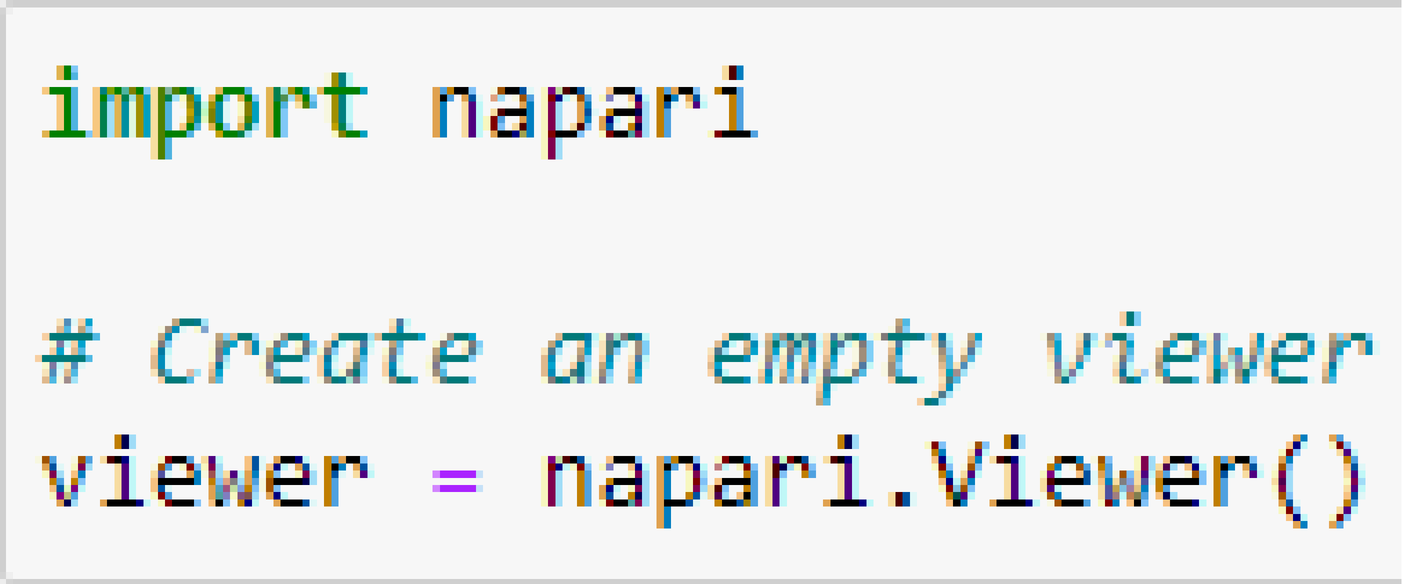
2. Open an image

3. Establish a segmentation workflow

What were the steps I did again?

Scripting napari in notebooks

Start napari from a jupyter notebook

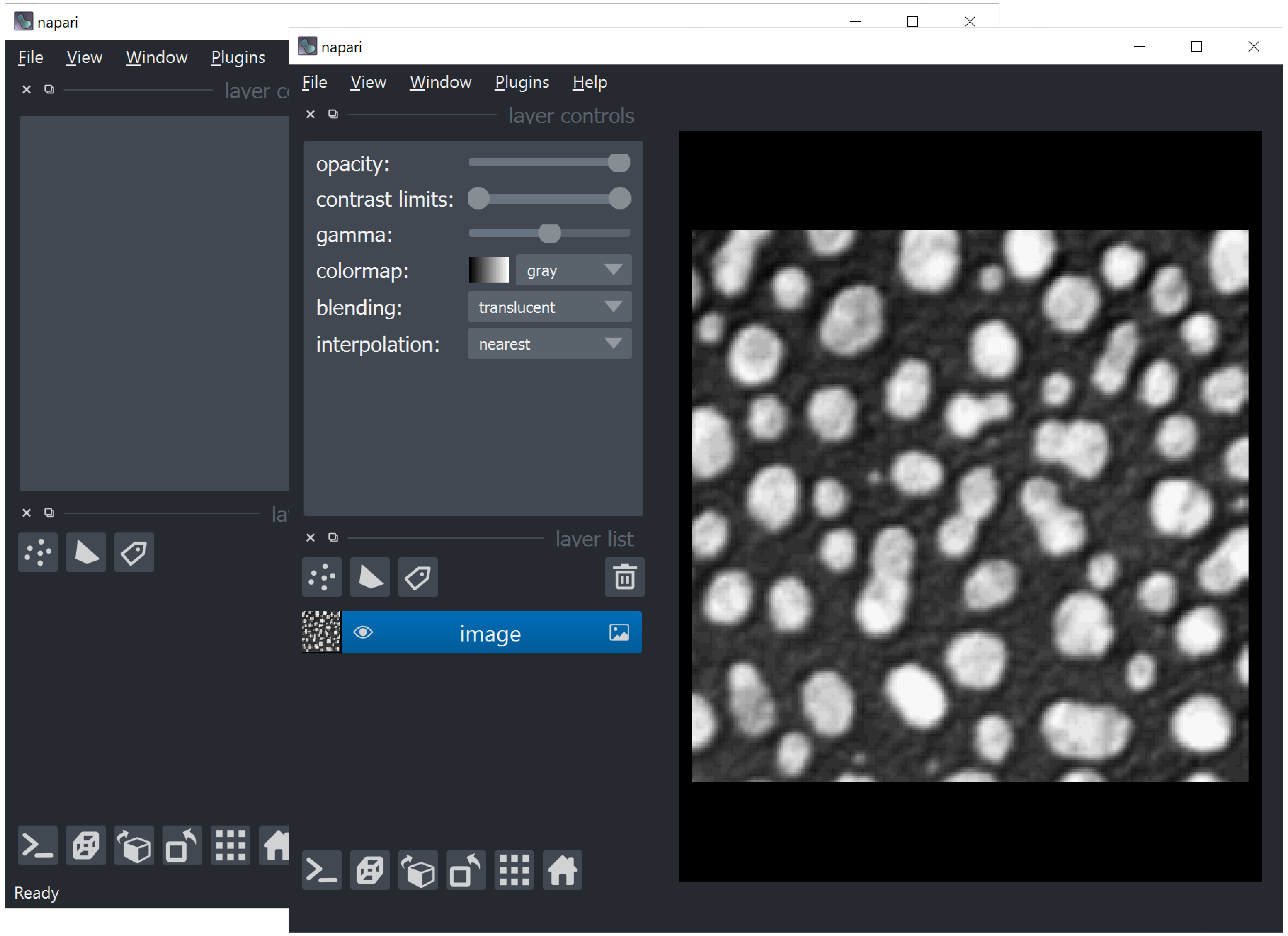


```
import napari

# Create an empty viewer
viewer = napari.Viewer()
```

```
viewer.add_image(image, name = ‘image’)
```

Load image from notebook to napari (and back): flexibility!

```
image = viewer.layers[‘image’].data
```

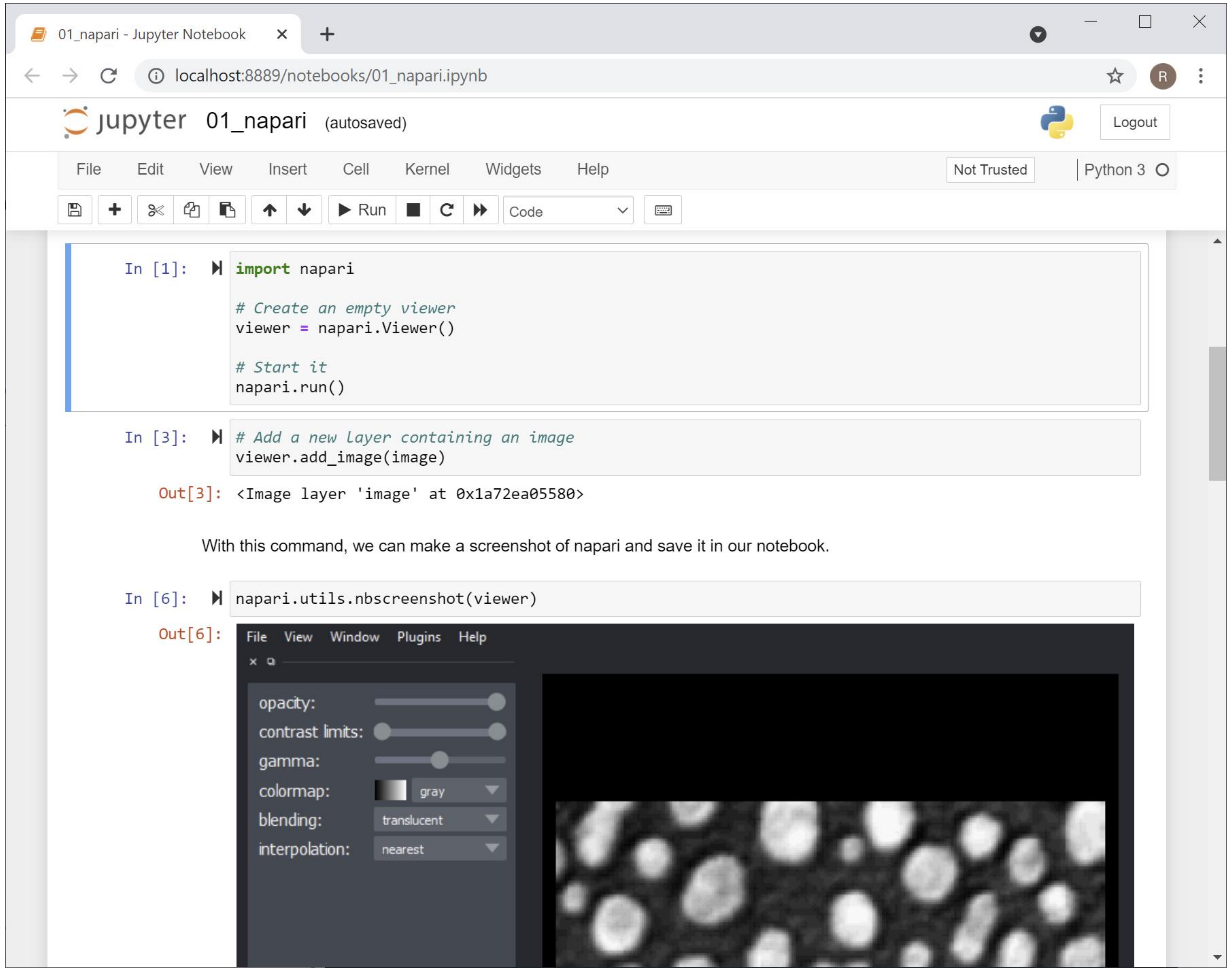


opacity:
contrast limits:
gamma:
colormap: gray
blending: translucent
interpolation: nearest

Scripting napari in notebooks

Make screenshots from napari and put them in your jupter notebook

```
napari.utils.nbscreenshot(viewer)
```



01_napari - Jupyter Notebook
localhost:8889/notebooks/01_napari.ipynb
jupyter 01_napari (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3
Run
Code

```
In [1]: import napari

        # Create an empty viewer
        viewer = napari.Viewer()

        # Start it
        napari.run()

In [3]: # Add a new Layer containing an image
        viewer.add_image(image)

Out[3]: <Image layer 'image' at 0x1a72ea05580>
```

With this command, we can make a screenshot of napari and save it in our notebook.

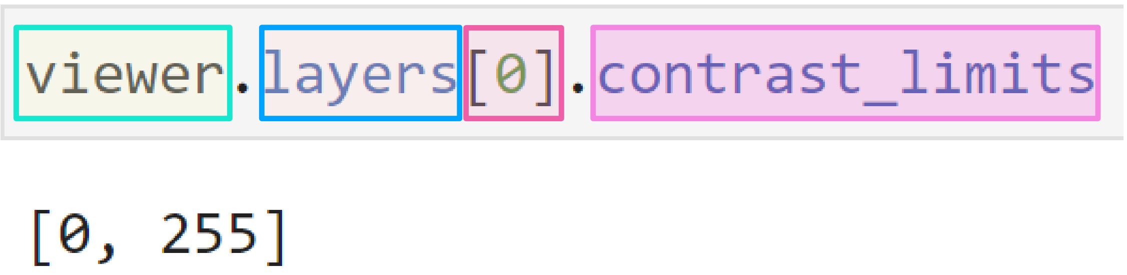
```
In [6]: napari.utils.nbscreenshot(viewer)

Out[6]:
```

Scripting napari in notebooks

- Add layers to napari to visualize intermediate processing results on top of each other or side by side.

- Change layer visualization within napari…

… or via code in a jupyter notebook:



```
viewer.layers[0].contrast_limits
```

```
[0, 255]
```

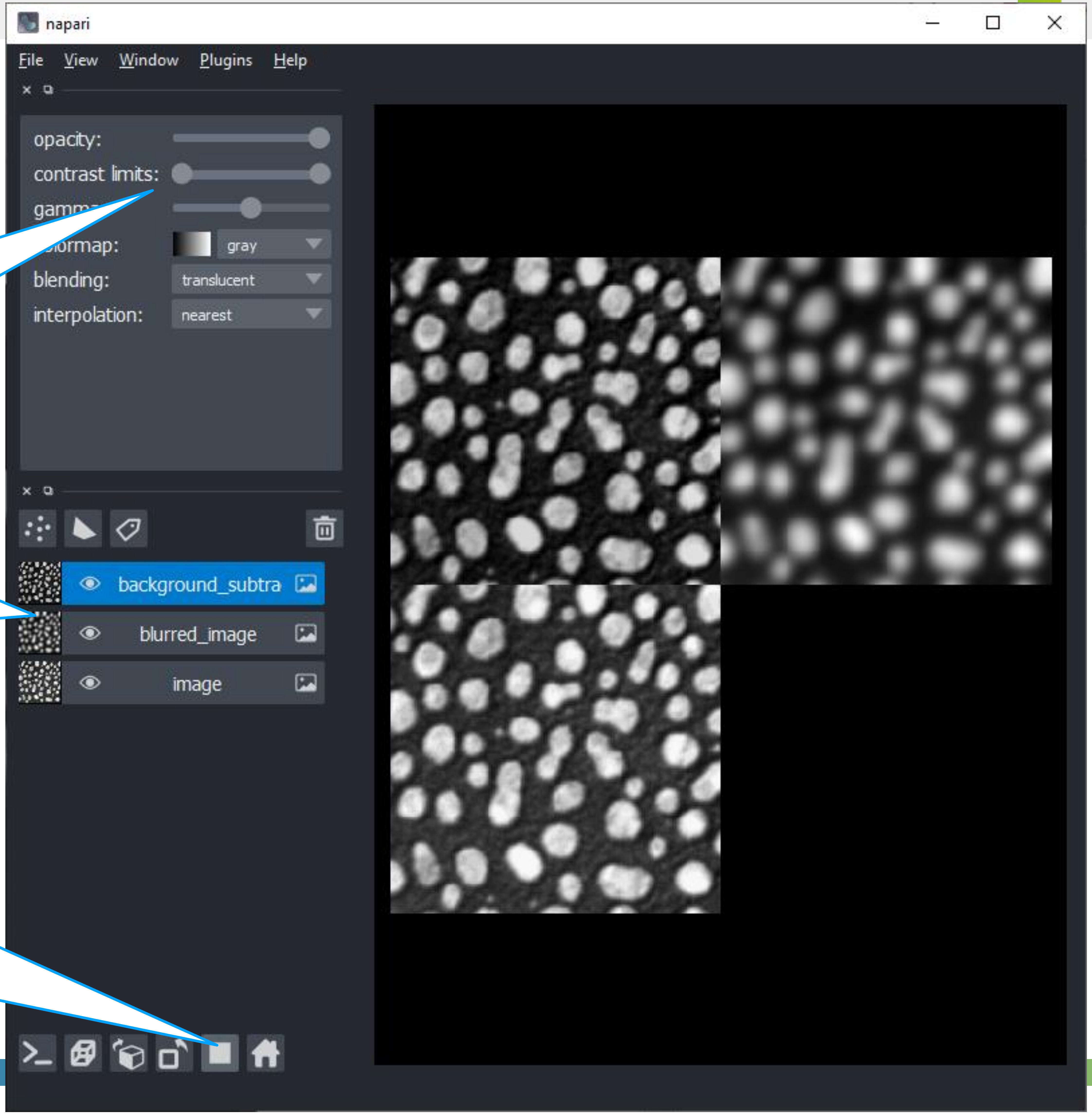
1. Access the viewer
2. Access the layers
3. Choose a layer (by index or name)
4. “Press TAB” and check out available properties

```
viewer.layers[0].contrast_limits = [30,170]
```

Change Brightness and contrast here

Toggle visualization of layers here

Gallery view



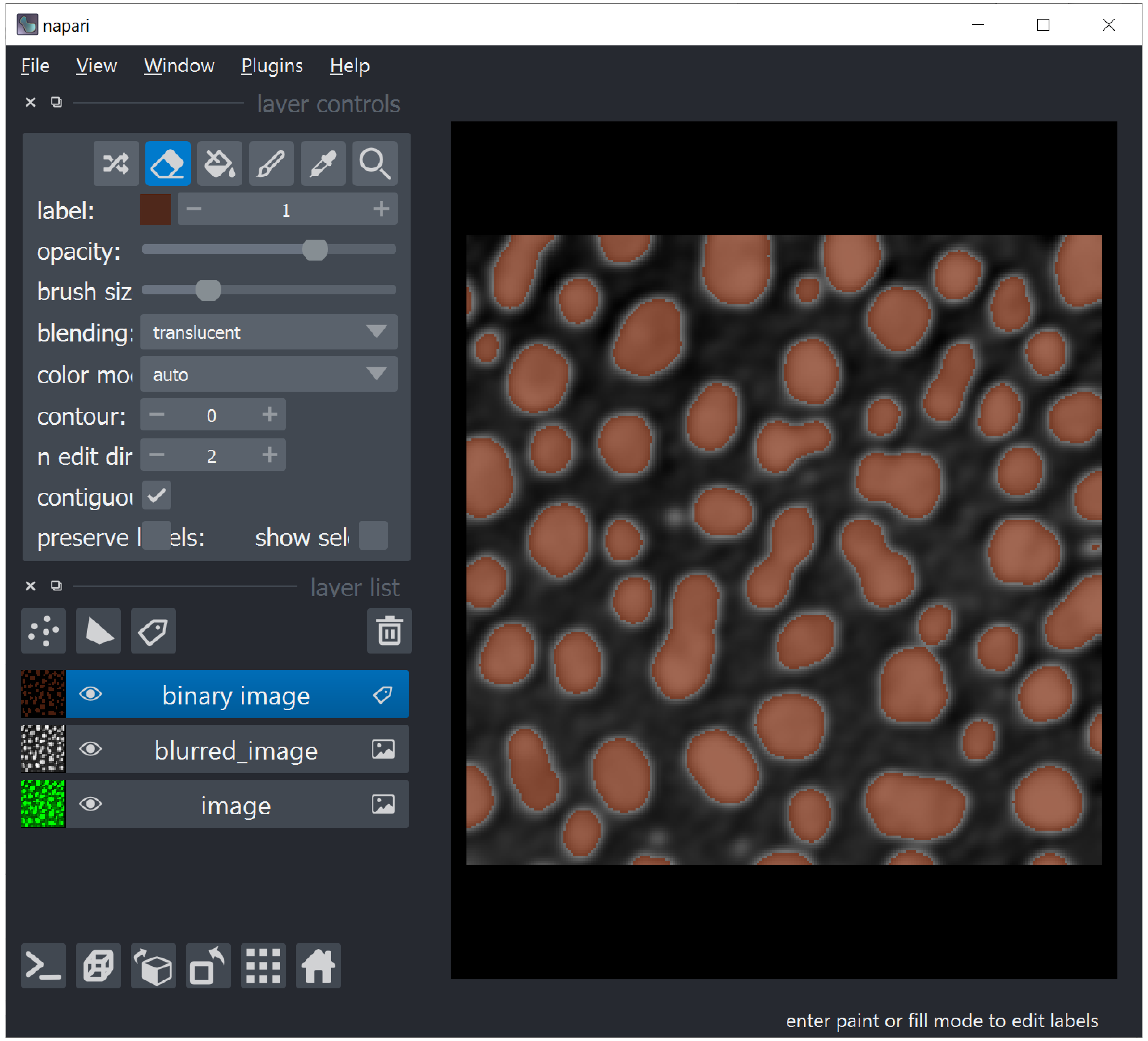
opacity:
contrast limits:
gamma:
colormap: gray
blending: translucent
interpolation: nearest

Visualizing image segmentation

Binary images and label images visualized as label layers

```
from skimage.filters import threshold_otsu
threshold = threshold_otsu(blurred_image)
binary_image = blurred_image > threshold

# Add a new labels layer containing an image
viewer.add_labels(binary_image,
                  name="binary image")
```

Name your layers to keep track of what they contain



Visualizing image segmentation

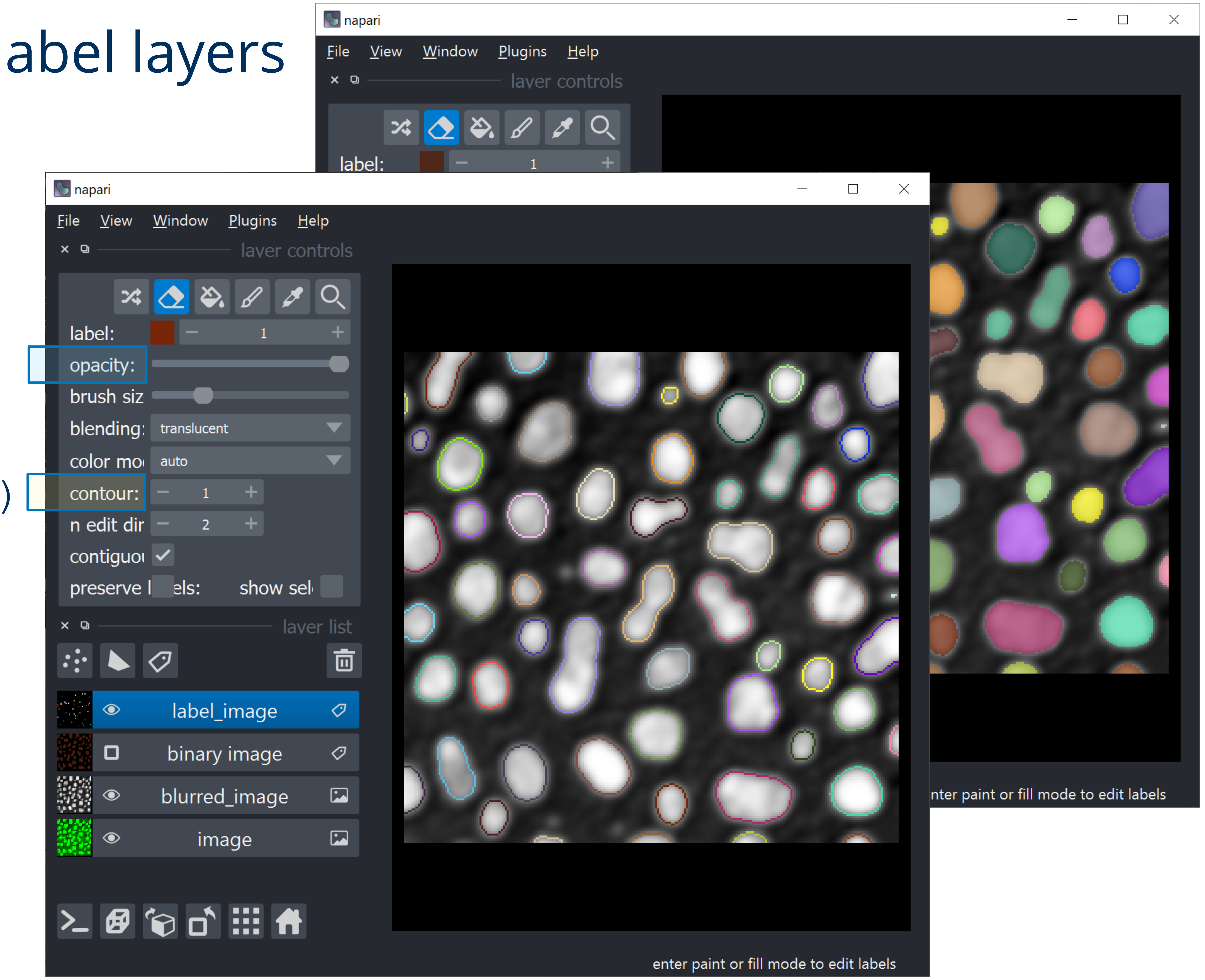
PoL Physics of Life TU Dresden

Binary images and label images visualized as label layers

```
from skimage.measure import label
label_image = label(binary_image)

# add labels to viewer
label_layer = viewer.add_labels(label_image)
```

Visualize contours instead of the overlay

```
label_layer.contour = 1
label_layer.opacity = 1
```



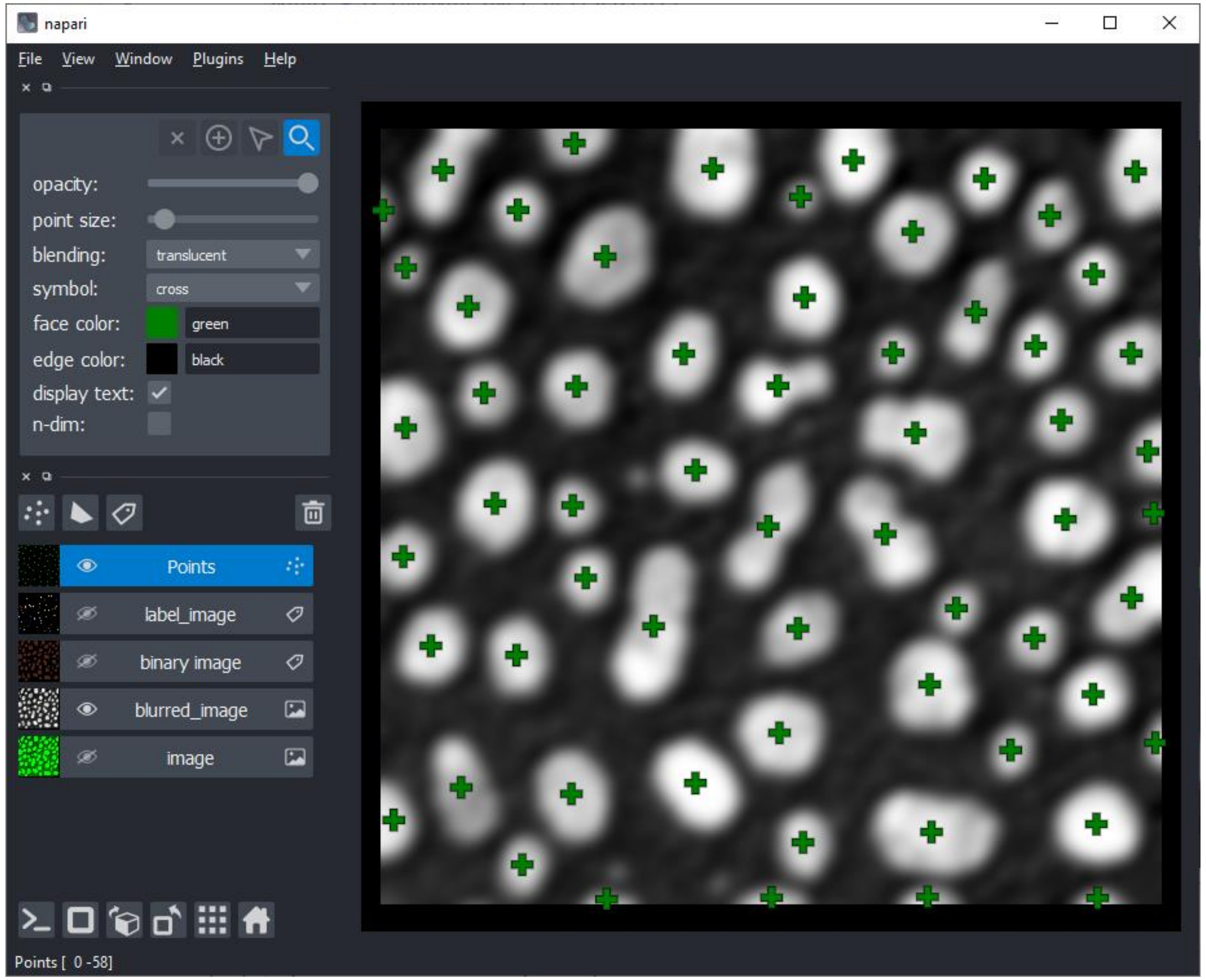
http://nbviewer.jupyter.org/github/BiAPoL/on_the_fly_image_processing_napari/blob/master/01_napari.ipy
nb

Points layers

There is also other layer types

- Shapes
- Points
- Surfaces
- Tracks
- Vectors

```
from skimage.measure import regionprops

statistics = regionprops(label_image)

points = [s.centroid for s in statistics]

# add points to viewer
label_layer = viewer.add_points(points, face_color='green', symbol='cross', size=5)
```



opacity:
point size:
blending: translucent
symbol: cross
face color: 🟩 green
edge color: ■ black
display text: ✓
n-dim: □

Points [0 -58]

Exercise: Interact with napari from notebook

1. Open napari viewer and images from notebook and change their viualization

2. Run a 3D segmentation workflow notebook and visualize processing steps in napari.
   Put workflow inside a Python function.

3. Run a batch processing notebook for the established segmentation workflow