1. Introduction

This course assumes some familiarity with Python, Jupyter notebooks and python scientific packages such as Numpy. There are many great resources to learn Python, including within Jupyter environements. For example this (https://gitlab.erc.monash.edu.au/andrease/Python4Maths/tree/master/Intro-to-Python) is a great introduction that you can follow to refresh your memories if needed.

The course will mostly focus on image processing using the package scikit-image, which is 1) easy to install, 2) offers a huge choice of image processing functions and 3) has a simple syntax. Other tools that you may want to explore are OpenCV (https://opencv.org/) (focus on computer vision) and ITK (https://itkpythonpackage.readthedocs.io/en/latest/) (focus on medical image processing). Finally, it has recently become possible to "import" Fiji (ImageJ) (https://github.com/imagej/pyimagej) into Jupyter, which may be of interest if you rely on specific plugins that are not implemented in Python (this is however in very beta mode).

1.1 Installation

1.1.1 Running the course material remotely

To avoid loosing time at the beginning of the course with faulty installations, we provide every attendee access to a JupyterHub allowing to remotely run the notebooks (links will be provided in time). This possibility is only offered for the duration of the course. The notebooks can however be permanently accessed and executed through the mybinder (https://mybinder.org/) service that you can activate by clicking on the badge below that is also present on the repository. If you want to "full experience" you can also install all the necessary packages on your own computer (see below).

launch binder

(https://mybinder.org/v2/gh/guiwitz/PyImageCourse/master)

1.1.2 Local installation

Python and Jupyter can be installed on any operating system. Instead of manually installing all needed components, we highly recommend using the environment manager conda (https://conda.io/docs/user-guide/index.html) by installing either Anaconda or Miniconda (https://conda.io/docs/user-guide/install/index.html#) (follow instructions on the website). This will install Python, Python tools (e.g. pip), several important libraries (including e.g. Numpy) and finally the conda tool itself. For Mac/Linux users: Anaconda is quite big so we recommend installing Miniconda, and then installing additional packages that you need from the Terminal. For Windows users: Anaconda might be better for you as it installs a command prompt (Anaconda prompt) from which you can easily issue conda commands.

The point of using conda is that it lets you install various packages and even versions of Python within closed environments that don't interfere with each other. In such a way, once you have an environment that functions as intended, you don't have to fear messing it up when you need to install other tools for you next project.

Once conda is installed, you should create a conda environment for the course. We have automated this process and you can simply follow the instructions below:

- Clone or download (https://github.com/guiwitz/PyImageCourse/archive/master.zip) and unzip this repository.
- Open a terminal and `cd` to it.
- Create the conda environment by typing:

```
conda env create -f binder/environment.yml
```

- Activate the environment:

```
conda activate improc_env
```

- Several imaging datasets are used during the course. The download of these data is automated through the following command (the total size is 6Gb so make sure you have a good internet connection and enough disk space):

```
python installation/download_data.py
```

Note that if you need an additional package for that environment, you can still install it using conda or pip. **To make it accessible within the course environment don't forget to type:**

```
conda activate improc_env
```

before you conda or pip install anything. Alternatively you can type your instructions directly from a notebook e.g.:

```
! pip install mypackage
```

Whenever you close the terminal where notebooks are running, don't forget to first activate the environment before you want to run the notebooks next time:

```
conda activate improc_env
```

1.2 Some Python refresh

I give here a **very** short summary of basic Python, focusing on structures and operations that we will use during this lecture. So this is **not** an exhaustive Python introduction. There are many many operations that one can do on basic Python structures, however as we are mostly going to use Numpy arrays, those operations are **not** desribed here.

1.2.1 Variables and structures

There are multiple types of Python variables:

In [56]:
```
myint = 4
myfloat = 4.0
mystring ='Hello'
print(myint)
print(myfloat)
print(mystring)
```

```
4
4.0
Hello
```

The type of your variable can be found using type():

In [57]:
```
type(myint)
```

Out[57]: int

In [58]:
```
type(myfloat)
```

Out[58]: float

These variables can be assembled into various Python structures:

In [59]:
```
mylist = [7,5,9]
mydictionary = {'element1': 1, 'element2': 2}
print(mylist)
print(mydictionary)
```

```
[7, 5, 9]
{'element2': 2, 'element1': 1}
```

Elements of those structures can be accessed through **zero-based** indexing:

In [60]:
```
mylist[1]
```

Out[60]: 5

In [61]:
```
mydictionary['element2']
```

Out[61]: 2

One can append elements to a list:

In [62]:
```
mylist.append(1)
print(mylist)
```

```
[7, 5, 9, 1]
```

Measure its length:

In [63]:
```
len(mylist)
```

Out[63]: 4

Ask if some value exists in a list:

In [64]:
```
5 in mylist
```

Out[64]: True

In [65]:
```
4 in mylist
```

Out[65]: False

1.2.2 Basic operations

A lot of operations are included by default in Python. You can do arithmetic:

In [66]:
```
a = 2
b = 3
#addition
print(a+b)
#multiplication
print(a*b)
#powers
print(a**2)
```

```
5
6
4
```

Logical operations returning booleans (True/False)

In [67]:
```
a>b
```

Out[67]: False

In [68]:
```
a<b
```

Out[68]: True

In [69]:
```
a<b and 2*a<b
```

Out[69]: False

In [70]:
```
a<b and 1.4*a<b
```

Out[70]: True

In [71]:
```
a<b or 2*a<b
```

Out[71]: True

Operations on strings:

In [72]:
```
mystring = 'This is my string'
mystring
```

Out[72]: 'This is my string'

In [73]:
```
mystring+ ' and an additional string'
```

Out[73]:
```
'This is my string and an additional string'
```

In [74]:
```
mystring.split()
```

Out[74]:
```
['This', 'is', 'my', 'string']
```

1.2.2 Functions and methods

In Python one can get information or modify any object using either functions or methods. We have already seen a few examples above. For example when we asked for the length of a list we used the len() function:

In [75]:
```
len(mylist)
```

Out[75]:
```
4
```

Python variables also have so-called methods, which are functions associated with particular object types. Those methods are written as variable.method(). For example we have seen above how to append an element to a list:

In [76]:
```
mylist.append(20)
print(mylist)
```

```
[7, 5, 9, 1, 20]
```

The two examples above involve only one argument, but any number can be used. All Python objects, inculding those created by other packages like Numpy function on the same scheme.

There are two ways to ask for help on funtions and methods. First, if you want to know how a specific function is supposed to work you can simply type:

In [77]:
```
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

This shows you that you can pass any container to the function len() (list, dictionary *etc.*) and it tells you what comes out. We will see later some more advanced examples of help information.

Second, if you want to know what methods are associated with a particular object you can just type:

In [111]:
```
#¼dir(mylist)
```

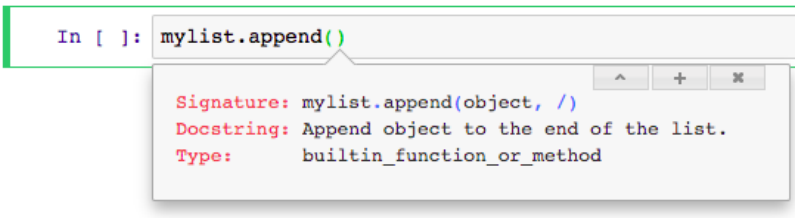
This returns a list of all possible methods. At the moment, only consider those **not** starting with an underscore. If you need help on one of those methods, you can type

In [79]:
```
help(mylist.append)
```

```
Help on built-in function append:

append(...) method of builtins.list instance
    L.append(object) -> None -- append object to end
```

Finally, whenever writing a function you can place the cursor in the empty function parenthesis and hit Command+Shift which will open a window with the help information looking like this:



1.2.2 For, if

Loops and conditions are classical programming features. In python, one can write them in a very natural way. A for loop:

In [80]:
```
for i in [1,2,3,4]:
    print(i)
```

```
1
2
3
4
```

An if condition:

In [81]:
```
a=5
if a>6:
    print('large')
else:
    print('small')
```

```
small
```

A mix of those:

In [82]:
```
for i in [1,2,3,4]:
    if i>3:
        print(i)
```

```
4
```

Note that **indentation of blocks is crucial in Python.**

1.2.3. Mixing lists, for's and if's

A very useful feature of Python is the very simple way it allows one to create lists. For exampel to create a list containing squares of certain values, in a classical programming languange one would do something like:

In [83]:
```
my_initial_list = [1,2,3,4]

my_list_to_create = []#initialize list

for i in my_initial_list:
    my_list_to_create.append(i*i)
print(my_list_to_create)
```

```
[1, 4, 9, 16]
```

Python allows one to do that in one line through a comprehension list, which is basically a compressed for loop:

In [84]:
```
[i*i for i in my_initial_list]
```

Out[84]:
```
[1, 4, 9, 16]
```

In a lot of cases, the list that the for loop goes through is not an explicit list but another function, typically range() which generate either numbers from 0 to N (range(N)) or from M to N in steps of P (range(M,N,P)):

In [85]:
```
[i for i in range(10)]
```

Out[85]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [86]:
```
[i for i in range(0,10,2)]
```

Out[86]:
```
[0, 2, 4, 6, 8]
```

If statements can be introduced in comprehension lists:

In [87]:
```
[i for i in range(0,10,2) if i>3]
```

Out[87]:
```
[4, 6, 8]
```

In [88]:
```
[i if i>3 else 100 for i in range(0,10,2)]
```

Out[88]:
```
[100, 100, 4, 6, 8]
```

A last very useful trick offered by Python is the function enumerate. Often when traversing a list, one needs both the actual value and the index of that value:

In [89]:
```
for ind, val in enumerate([8,4,9]):
    print('index: '+str(ind))
    print('value: ' + str(val))
```

```
index: 0
value: 8
index: 1
value: 4
index: 2
value: 9
```

1.2.4 Using packages

Python comes with a default set of data structures and operations. For particular applications like matrix calculations (image processing) or visulaization, we are going to need additional resources. Those exist in the form of python packages, ensembles of functions and data structures whose defintiions can be simply imported in any Python program.

For example to do matrix operations, we are going to use Numpy, so we run:

In [90]:
```
import numpy
```

All functions of a package can be called by using the package name followed by a dot and a parenthesis `numpy.xxx()`. Most functions are used with an argument and either "act" on the argument e.g. to find the maximum in a list:

In [91]:
```
numpy.max([1,2])
```

Out[91]:
```
2
```

or use the arguments to create a new object e.g. a 4x3 matrix of zeros:

In [92]:
```
mymat = numpy.zeros((4,3))
```

In [93]:
```
mymat
```

Out[93]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

To avoid lengthy typing, package names are usually abbreviated by giving them another name when loading them:

In [94]:
```
import numpy as np
```

Within packages, some additional tools are grouped as submodules and are typically called e.g for numpy as `numpy.submodule_name.xxx()`. For example, generating random numbers can be done using the numpy.random submodule. An array of ten uniform random numbers can be for example generated using:

In [95]:
```
np.random.rand(10)
```

Out[95]:
```
array([0.00738174, 0.82510957, 0.59643586, 0.92919436, 0.46570716,
       0.92526076, 0.17081481, 0.03715798, 0.12744829, 0.35009797])
```

To avoid lengthy typing, specific functions can be directly imported, which allows one to call them without specifying their source module:

In [96]:
```
from numpy.random import rand

rand(10)
```

Out[96]:
```
array([0.81812159, 0.97452756, 0.4383594 , 0.91854004, 0.37517642,
       0.11077294, 0.66271078, 0.8482131 , 0.70100188, 0.44337187])
```

This should be used very cautiously, as it makes it more difficult to debgug code, once it is not clear anymore that a given function comes from a module.

1.3 Matplotlib

To quickly look at images, we are mostly going to use the package Matplotlib. We review here the bare minimum function calls needed to do a simple plot. First let's import the pyplot submodule:

```
In [97]: import matplotlib.pyplot as plt
```

1.3.1 Plotting images

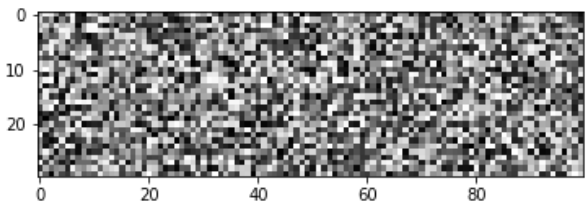
Using numpy we create a random 2D image of integers of 30x100 pixels (we will learn more about Numpy in the next chapters):

```
In [98]: image = numpy.random.randint(0,255,(30,100))
```

The variable image is a Numpy array, and we'll see in the next chapter what that exactly is. For the moment just consider it as a 2D image.

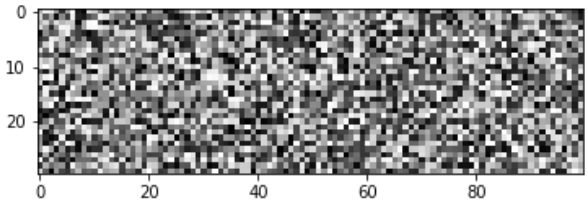
To show this image we are using the `plt.imshow()` command which takes an Numpy array as argument:

```
In [99]: plt.imshow(image)
Out[99]: <matplotlib.image.AxesImage at 0x7f7496c27160>
```



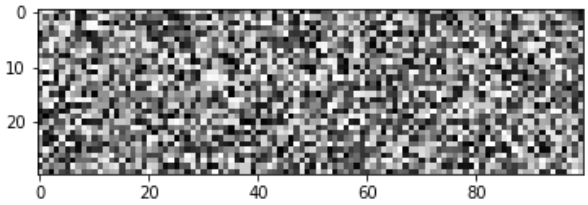
In order to suppress the matplotlib figure reference, you can end the line with `;`:

```
In [100]: plt.imshow(image);
```



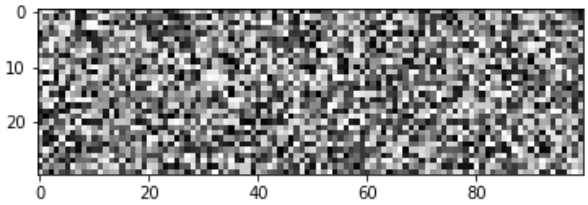
When plotting outside of an interactive environment like a notebook you will also have to use the show() command. If you use it in a notebook you won't have to use `;`:

In [101]:
```
plt.imshow(image)
plt.show()
```



The rows and number indices are indicates on the left and the bottom and **actually** correspond to pixel indices. The image is just a gray-scale image, and Matplotlib used its default lookup table (or color map) to color it (LUT in Fiji). We can change that by specifiy another LUT (you can find the list of LUTs here (https://matplotlib.org/examples/color/colormaps_reference.html) by using the argument cmap (color map):

In [102]:
```
plt.imshow(image, cmap = 'gray');
```



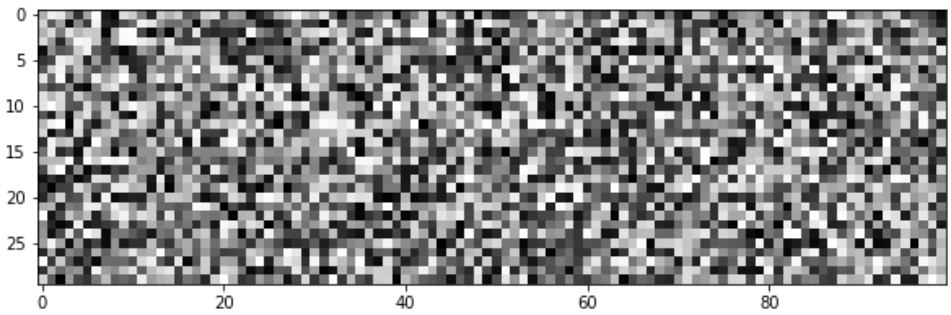
Note that you can change the default color map used by matplotlib using a command of the type plt.yourcolor, *e.g.* for gray scale:

In [103]:
```
plt.gray()
```

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

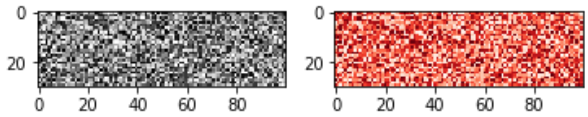
Sometimes we want to see a slightly larger image. To do that we have to add another line that specifies options for the figure.

In [104]:
```
plt.figure(figsize=(10,10))
plt.imshow(image);
```



Sometimes we want to show an array of figures to compare for example an original image and its segmentations. We use the subplot() function and pass three arguments: number of rows, number of columns and index of plot. We use it for each element and increment the plot index. There are multiple ways of creating complex figures and you can refer to the Matplotlib documentation for further information:

In [105]:
```
plt.subplot(1,2,1)
plt.imshow(image, cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(image, cmap = 'Reds');
```

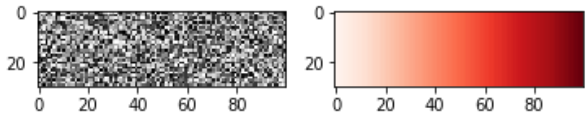


The imshow() function takes basically two types of data. Either single planes as above, or images with three planes. In the latter case, imshow() assumes that the image is in RGB format (Red, Green, Blue) and uses those colors.

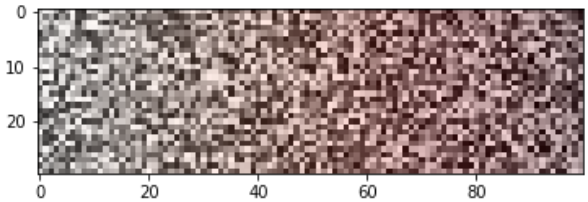
Finally, one can superpose various plot elements on top of each other. One very useful option in the frame of this course, is the possibility to ovelay an image in transparency on top of another using the `alpha` argument. We create a gradient image and then superpose it:

In [106]:
```
image_grad = np.ones((30,100))*np.linspace(0, 1, 100)[None, :]

plt.subplot(1,2,1)
plt.imshow(image, cmap = 'gray')
plt.subplot(1,2,2)
plt.imshow(image_grad, cmap = 'Reds');
```



In [107]:
```
plt.imshow(image, cmap = 'gray')
plt.imshow(image_grad, cmap = 'Reds', alpha = 0.2);
```



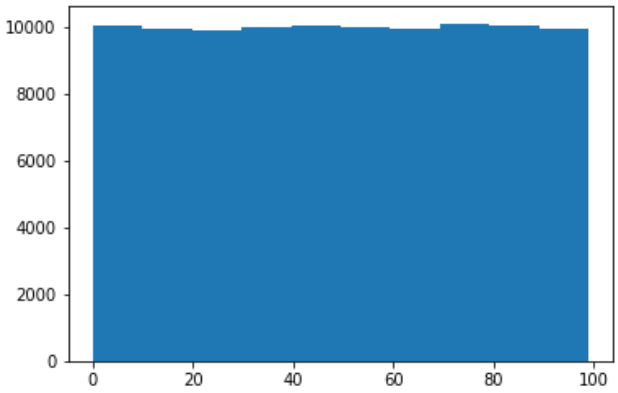
1.3.2 Plotting histograms

One thing that we are going to do very often is looking at histograms, typically of pixel values, for example to determine a threshold from background to signal. For that we can use the plt.hist() command.

If we have a list of numbers we can simply called the `plt.hist()` function on it (we will see more options later). We crate again a list of random numbers:

In [108]:
```
list_number = np.random.randint(0,100,100000)
```

In [109]:
```
plt.hist(list_number);
```



Once we have an idea of the distribution of values, we can refine the binning:

In [110]:
```
plt.hist(list_number, bins = np.arange(0,255,2));
```