CMM201_Week9_lab_solved

November 14, 2019

First, download image "logo.png" from Moodle, which correspond to a grayscale version of the Python logo. Save these file **in the same directory as this Jupyter notebook**.

To work with images in Python, we will use packages such as **numpy**, **PIL**, **skimage** and **matplotlib**. Execute the code in the following cell to install these packages using the *pip* command (remember that we add the ! since this command is used in the terminal):

```
In [ ]: !pip install pillow
        !pip install matplotlib
        !pip install scikit-image
```

Notice that we did not install numpy due to the following reasons: * You have installed it before during class (and there is no need to install it again). * Sometimes packages include sub-packages. Such is the case of the three previous ones, which already contain numpy.

0.1 Importing the modules

Now we will import the packages, functions and classes we will use for this activity:

- The first one is the **Image** class from the **PIL** module (which comes in the **pillow** package). For this, we will execute the following command:

```
In [1]: from PIL import Image
```

- To handle and perform operations on the images, we need **numpy**, which can be imported by executing the command we saw in class:

```
In [2]: import numpy as np
```

- To plot the images, we will use **matplotlib.pyplot** through the pseudonym **plt**:

```
In [3]: import matplotlib.pyplot as plt
```

0.2 Working with Grayscale Images

Now it's time to import the grayscale logo to Python. To do so, we can use the **open** function from the **Image** class, storing the image in a variable called **img**:

```
In [4]: img = Image.open("logo.png")
```

If we check the type of the image, we will see that it belongs to a weird **PIL.Image...** type.

```
In [5]: type(img)
```

```
Out[5]: PIL.PngImagePlugin.PngImageFile
```

To manipulate the image, we need to convert it into a numpy array. To do so, we can use the following command:

```
In [6]: img = np.array(img)
```

Now check the type of img in the following cell:

```
In [7]: type(img)
```

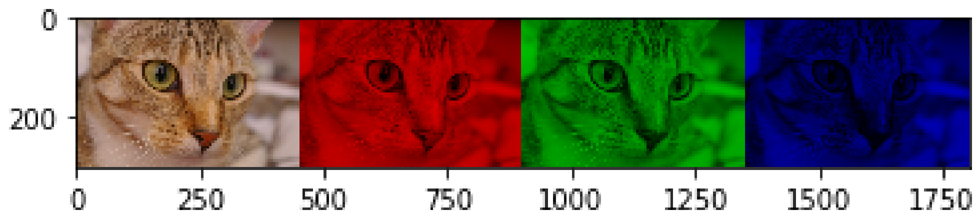
```
Out[7]: numpy.ndarray
```

One particularity of the Image.open() function is that, by default, it imports all images as colour ones! This means that although the file contains a grayscale image, the numpy array that we have imported has three dimensions (keep in mind that images within the computer are typically composed of three channels: red, green and blue). An example of this is shown by running the following code (don't get too much into it, just run it!):

```
In [8]: from skimage import data
        cat = np.asarray(data.chelsea())
        print(cat.shape)
        cat_R = cat.copy()
        cat_R[:, :, (1, 2)] = 0
        cat_G = cat.copy()
        cat_G[:, :, (0, 2)] = 0
        cat_B = cat.copy()
        cat_B[:, :, (0, 1)] = 0
        cat_RGB = np.concatenate((cat, cat_R, cat_G, cat_B), axis=1)
        plt.imshow(cat_RGB)
```

```
(300, 451, 3)
```

```
Out[8]: <matplotlib.image.AxesImage at 0xdc48350>
```



Therefore, if we check the dimensions of **img**, we will see that it has three (one per channel):

```
In [9]: print(img.ndim)

3
```

At this stage we don't want this, therefore we will use another module to import the image as a *grayscale image*. Therefore, we will use the **color** and **io** classes from **scikit-image** (referred to as **skimage**) to import the grayscale image into the **img** variable using the following code:

```
In [10]: from skimage import color
         from skimage import io

         img = color.rgb2gray(io.imread('logo.png'))
```

Now check the type and the dimensions of **img** to verify that now it stores a two-dimensional numpy array.

```
In [11]: ## Use this cell to print the type and the dimension of img
         print(type(img), img.ndim)

<class 'numpy.ndarray'> 2
```

You can also check the number of pixels that the image contains. Notice that the original file is a 75x75 pixel sized image, and therefore your **img** numpy array should have that same size.

```
In [12]: img.shape

Out[12]: (75, 75)
```

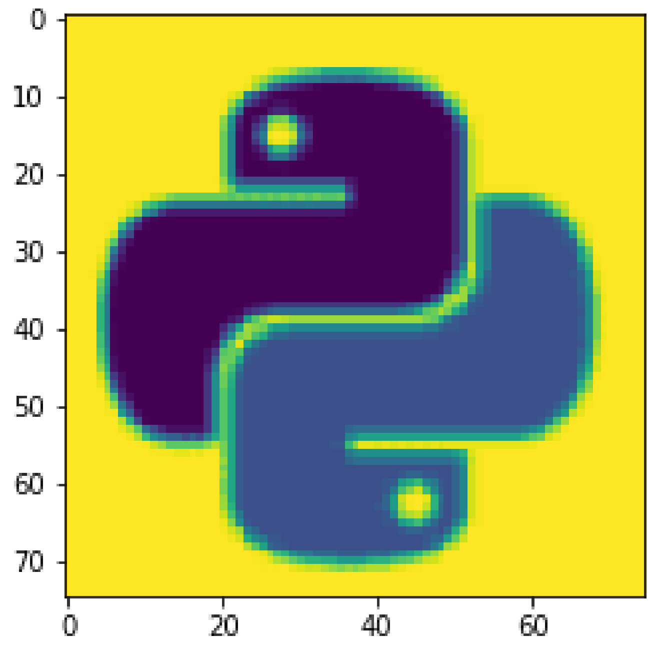
What would happen if we print **img**? Are we able to see the image? Run the following cell to find out!

```
In [13]: print(img)

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

Notice that we only see a bunch of "ones" in lists. This is because for now, the image is a numpy array with values from 0 to 1, where 0 represents black and 1 represents white. Since the print command only shows us the "corners" of the numpy array, we are only able to see the white corners of the images (hence the ones).

To show the actual image, we can use the **plt.imshow** function from the **matplotlib.pyplot** library as follows:

```
In [14]: plt.imshow(img)

Out[14]: <matplotlib.image.AxesImage at 0x1483630>
```



It is likely that the image has a yellow background. This will help us to better identify the background pixels.

0.2.1 Locating the value of pixels

Notice that the image is shown with a coordinate plane which gives us an idea of the positions of the pixels within the logo. Given that this is a 75x75 image, we can access different positions of the image to see their value. For instance, to see the value of the left-top pixel we can use the following command:

```
In [15]: print(img[0,0])

1.0
```

The first zero corresponds to the y-axis (rows) and the second zero corresponds to the x-axis (columns). We can use the same command to see different positions of the image. For instance, you can see the value of the other corners or some values from the two pythons.

```
In [16]: ## Use this cell to print the values of different pixels
         print(img[20,40])

0.196078431372549
```

Write down the numerical values corresponding to each of the pythons' bodies. We will use this information later!

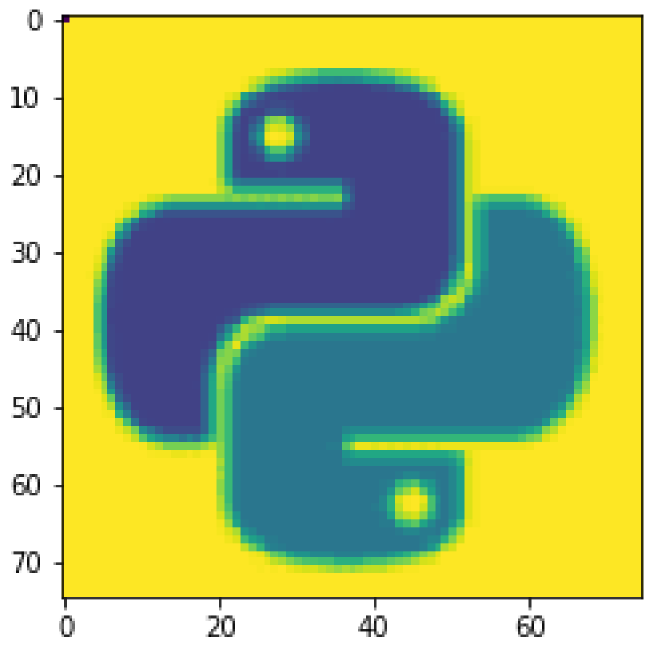
0.2.2 Changing the value of pixels

It is also possible to change the value of a pixel. To do so, you can assign a new value to a certain location. For instance, we can paint the top-left pixel (coordinate 0,0) in black with the following code:

```
In [17]: img[0,0]=0
         print(img)
         plt.imshow(img)
```

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

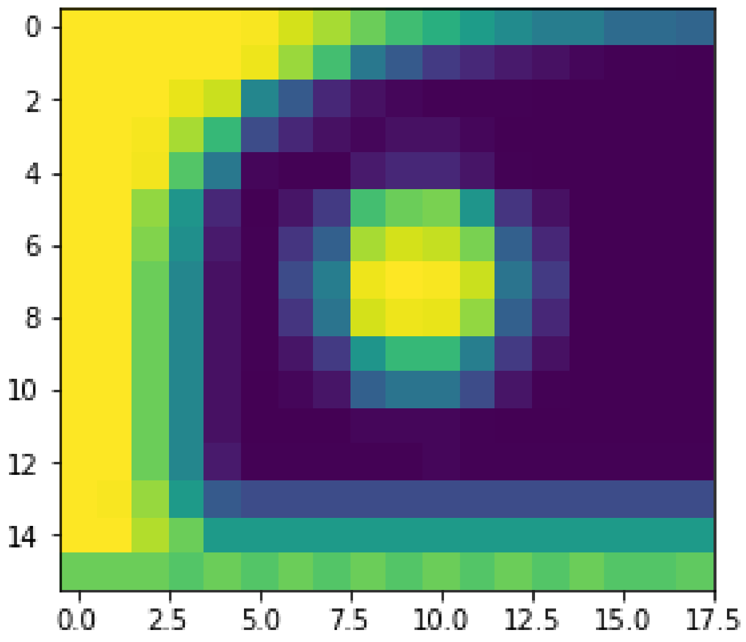
```
Out[17]: <matplotlib.image.AxesImage at 0x14c0e10>
```



0.2.3 Trimming the image

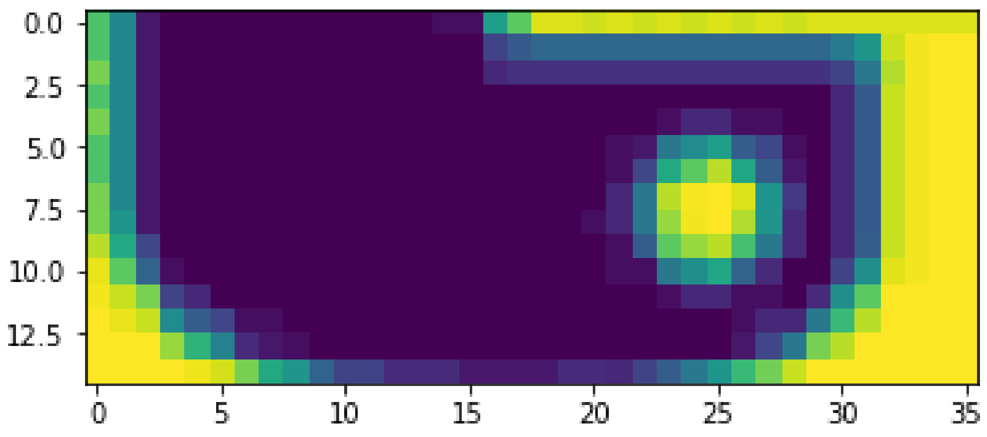
We can also slice the image and create a new, smaller one. For instance, say that we want to produce a new image only with the face of the dark-coloured Python. To do so, we first need to analyse the image and calculate the approximate position of the **top-left** corner of the square we want to cut. In this case, it could be the coordinate in the **row 8** and in the **column 18**. Then, we need to see where we want that square to end (i.e. **bottom-right** corner). In this case, it is around **row 24** and **column 36**. Now, we can use the following code to produce a trimmed image called **img_trimdark** with the face of the dark-coloured Python.

```
In [18]: img_trimdark = img[8:24,18:36] # rows go first, columns go after
         plt.imshow(img_trimdark)

Out[18]: <matplotlib.image.AxesImage at 0x15070b0>
```



Now it's your turn to create a trimmed image called **img_trimlight** with the face of the light-coloured python.

```
In [19]: ## Use this cell to create img_trimlight
         img_trimlight = img[55:70,20:56] # rows go first, columns go after
         plt.imshow(img_trimlight)

Out[19]: <matplotlib.image.AxesImage at 0x1543a10>
```

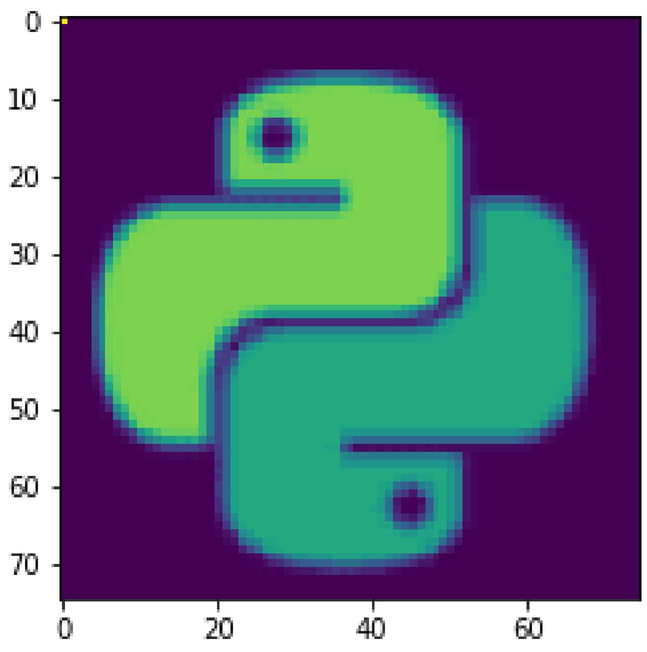


0.2.4 Inverting the image

We can also create the *negative* of an image by doing the following:

```
In [20]: img_neg = 1 - img
         plt.imshow(img_neg)
```

```
Out[20]: <matplotlib.image.AxesImage at 0x1588030>
```



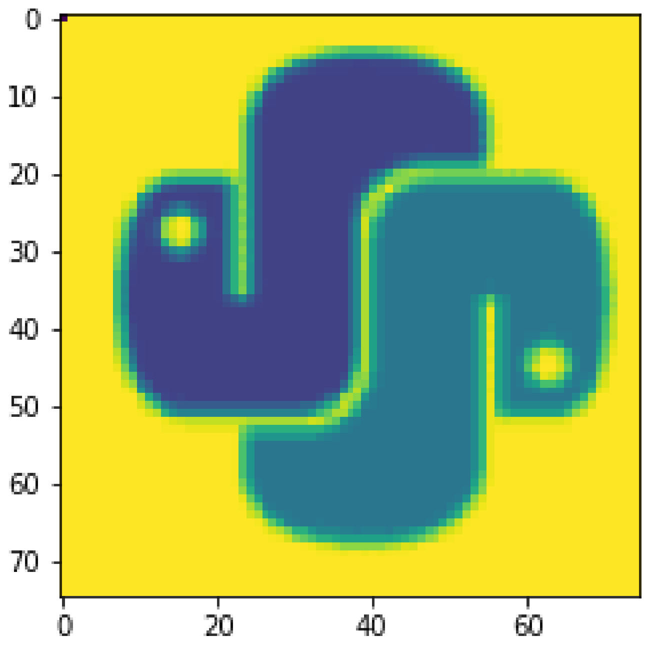
By subtracting 1 minus the original image, basically what we are doing is converting all ones to zeros, all zeros to ones, and all values in between to their opposite value in terms of how far they are to 0 or 1.

0.2.5 Transposing the image

There is a mathematical operation called **transpose** which lets you exchange the columns and the rows of a matrix. This function is also included in the numpy module, and it can be applied to image arrays as well:

```
In [21]: img_trans = img.transpose()
         plt.imshow(img_trans)
```

```
Out[21]: <matplotlib.image.AxesImage at 0x15c4270>
```



Notice that the new image is turned 90 degrees counter-clockwise and has also been flipped vertically.

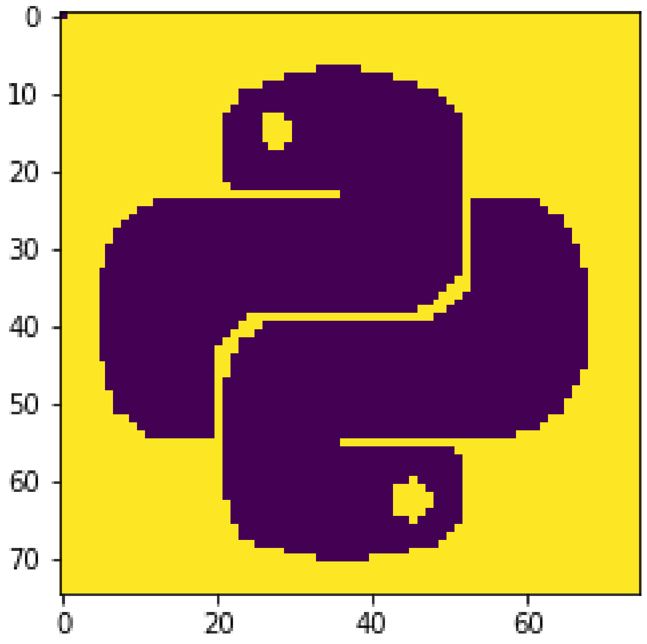
0.2.6 Binarising the image

In computer vision, there are times where you want the image to be just black and white instead of grayscale. This is useful when you want to clearly recognise shapes or when you want to get rid of noise. In this case, image binarisation could be used with two purposes: * Make both pythons black by converting all gray pixels into black. * Make the light python disappear by converting all light gray pixels into white.

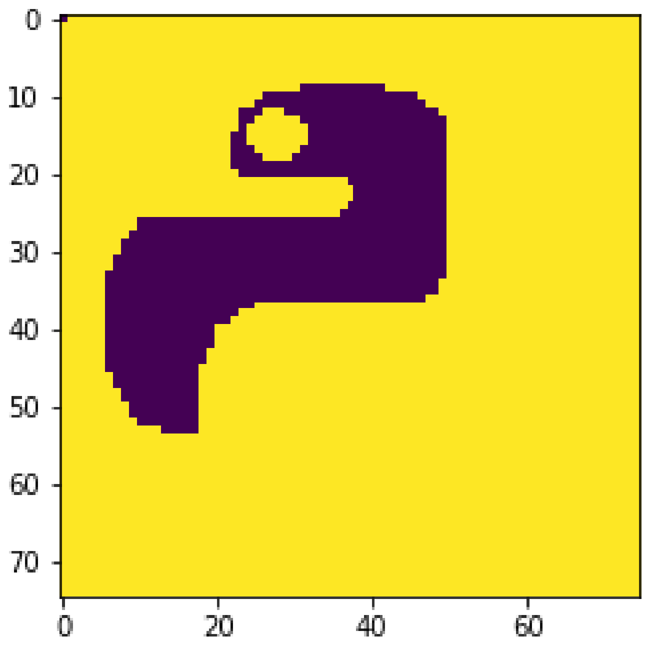
Remember you wrote down the values of the pixels of both pythons? Now it's time to put this information into use! To design our binarisation function, we need to consider the following: 1. The function has two inputs: the **image to binarise** and the **threshold**, which is a cut-off value between 0 and 1. 2. The function has to iterate the numpy array for all rows and columns. 3. For each position, the system has to compare the value of the pixel with the threshold. If the value of the pixel is *smaller* than the threshold, then we change the value of the pixel to zero. Else, we change it to one. 4. To avoid losing the original image, you can create a copy of it and change the copy, not the original one! (hint: use the .copy() function to do so). 5. Notice that depending on the threshold value selected, you will get different outputs!

```
In [22]: ## Use this cell to create the binarisation function
         def binarise(img, threshold):
             '''This function biniarises an image in a numpy array according to the specified
             img_bin = img.copy()
             for x in range(img.shape[0]):
                 for y in range(img.shape[1]):
                     if img[x,y]<threshold:
                         img_bin[x,y]=0
                     else:
                         img_bin[x,y]=1
             plt.imshow(img_bin)
             return img_bin
```

```
In [23]: ## Use this cell to execute the binarise function and show both pythons black.
         img_bin1 = binarise(img, 0.7)
```



```
In [24]: ## Use this cell to execute the binarise function and show the light python white.
         img_bin2 = binarise(img, 0.25)
```



0.2.7 Saving the image

You can save any of the images produced in this tutorial by using the **plt.imsave** command. For instance, if we want to save the transpose image, we can run the following command:

```
In [25]: plt.imsave('logotrans.png',img_trans)
```

You will notice that the image get saved with the yellow background! Is there any other way to save them?

```
In [26]: ## Use this cell to look for other ways to save the images with better quality.
         import imageio
         imageio.imwrite('logotranshd.png', img_trans)
```

```
D:\Anaconda3\lib\site-packages\imageio\core\util.py:78: UserWarning: Lossy conversion from floa
  dtype_str, out_type.__name__))
```