Image_as_Matrices

[2]:
```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
%matplotlib inline
```

0.1 Goals for this group activity:

This notebook will provide:

- intro to image as data
- intro to rgb color space
- test your knowledge of norms

0.2 Image as Data

A picture, in the real world, is a two-dimensional representation of something. That *something* can be three-dimensional or itself flat.

We will use the library `skimage` to import the image:

[3]:
```
from skimage.io import imread
```

[4]:
```
waldo_color_filename = 'waldo1.png'
waldo_color = imread( waldo_color_filename )
```

We can display the image using `plt.imshow`

[5]:
```
plt.figure(figsize = (10,10))
plt.imshow(waldo_color)
```

Now your image is represented in the form of a numpy array:

[6]:
```
type(waldo_color)
```

Try this!

And you can start performing computations just like you do with other data. For example, take a look at the shape your image data:

[7]:
```

```

Why do we have "4 layers" representing the above image, instead of 1? We will soon be talking about this! But let's first see how we can get images from just one layer.

0.3 Gray scale

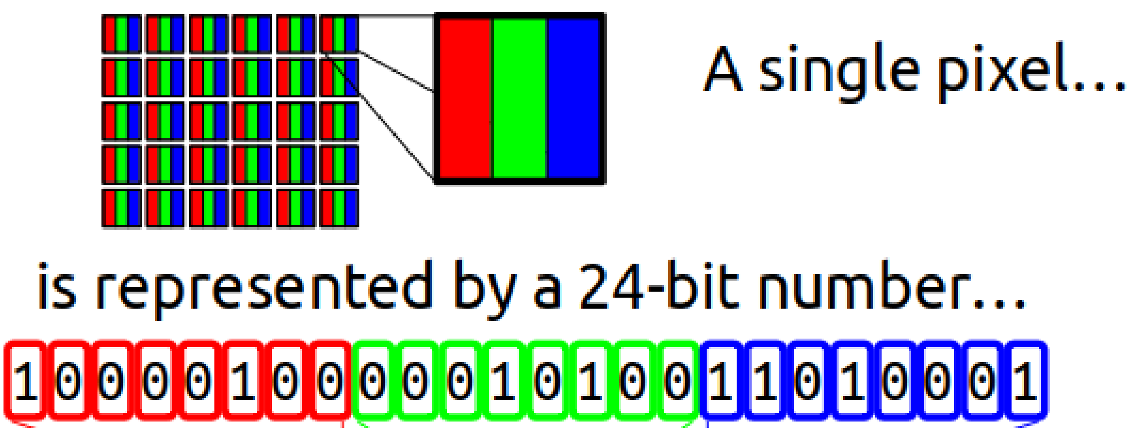
Here we create a 5x5 array of random numbers between 0 and 1. In the example below, we use a gray scale mapping for the colors, where 0 corresponds to black and 1 corresponds to white.

[8]:
```
random_colors = np.random.rand(25).reshape(5,5)
print(random_colors)
plt.imshow(random_colors, cmap='gray' )
```

0.4 RGB Colors

RGB color space constructs all the colors from the combination of the Red, Green and Blue colors (it is just a linear combination!)

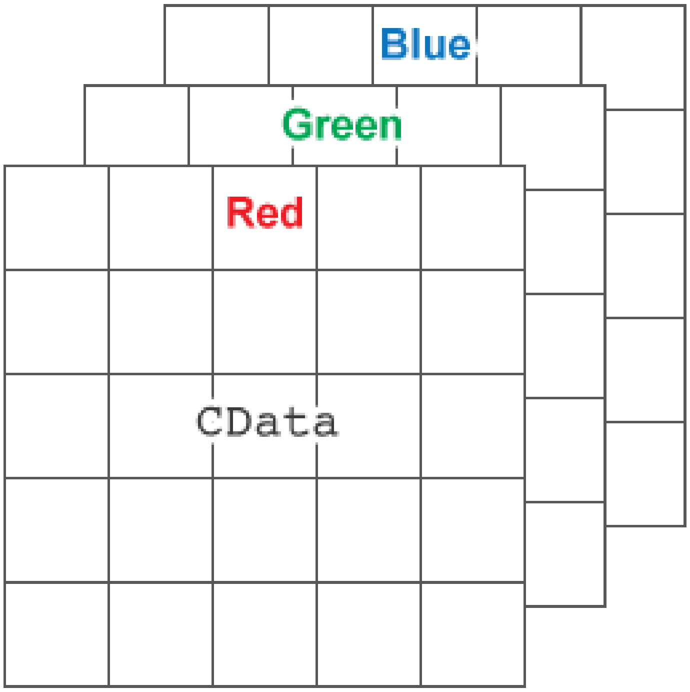
In our grayscale image, each pixel tone is defined by a value between 0 and 1. In a colored image, each pixel color is obtained by a given amount of red, given amount of green and a given amount of blue, and are stored in 24 bits format: 8 bits to describe the amount of each color (red, green, and blue). RBGA color space adds 8 bits to describe the *alpha* channel, or transparency of the pixel. The illustration below shows how different colors can be represented using 24 bits (not including the transparency channel):



Recall that $(11111111)_2 = (255)_{10}$, meaning that each color value will vary from 0 to 255.

0.5 3D array for color images

We can now think of your colored image as a three dimensional array, where each layer of the array corresponds to a different color channel.



0.6 Back to the Waldo Image

Try this!

We can inspect the entries of the color Waldo image `waldo_color`. Take a look at the maximum value and the minimum value.

[9]:
```

```

and observe that indeed the values for each pixel vary between 0 and 255. We now know that the "4 layers" correspond to each RGB channel, and the transparency channel. We can use Python slicing to get the color for a given pixel:

[10]:
```
waldo_color[4,5,:]
```

0.7 Help us finding Waldo!

Waldo's gone missing again. Time to solve the problem once and for all with the power of norms. Norms provide a measure of 'magnitude' for a vector or 'distance' given the difference between two vectors. For arrays that represent images (as in this problem), a small distance indicates a high degree of similarity between images.

To make this problem simpler, we will first convert the image from rgb scale to grayscale, such that our image can be represented by a matrix (only one laywer). We will again use the library `skimage`:

[11]:
```
from skimage.color import rgb2gray
from skimage.color import rgba2rgb
```

[12]:
```
waldo_gray = rgb2gray(rgba2rgb(waldo_color))
```

[13]:
```
plt.figure(figsize = (10,10))
plt.imshow(waldo_gray,cmap='gray')
```

Check the shape of your new image `waldo_gray`:

[14]:
```
waldo_gray.shape
```

Great! We now have the image represented as a matrix.

For this problem, we want to use the **2-norm (or Euclidean norm)**. A vector 2-norm $||\mathbf{x}||_2$ assigns to every vector $\mathbf{x} \in \mathbb{R}^n$ a nonnegative number by

$$||\mathbf{x}||_2 = \left(\sum^n x_i^2 \right)^{1/2}.$$

The function numpy.linalg.norm computes by default the vector 2-norm.

We could flatten the image, to transform it from a 2d array into a 1d array, and then apply the vector 2-norm above.

[15]:
```
waldo_gray.flatten().shape
```

This procedure corresponds to the Frobenius norm (and not the induced matrix 2-norm) which is given by:

$$||X||_F = \left(\sum^n \sum_{j=1}^m X_{ij}^2 \right)^{1/2},$$

which is the default norm computed when we use the function numpy.linalg.norm with a 2d array as input argument.

Try this!

Compute the norm of the array `waldo_gray` passing the 2d array as argument, and compare with the result when you pass the flatten 1d array as argument

[17]:
```

```

Waldo face is given in the following numpy array:

[18]:
```
waldo_face = np.load('face.npy')
waldo_face.shape
```

Write a code snippet that finds the image of Waldo's face (given by `waldo_face`) inside the complete image `waldo_gray`.

- Look through each subimage of `waldo_gray`, with the same dimensions as `waldo_face`, and find the one where their difference has the minimum **vector 2-norm**. You are in essence finding the subimage the has the minimum error when comparing to the image given by `waldo_face`.

- Each subimage should be uniquely identified by the position of its top-left-most pixel. In other words, the left-top corner of the image is the reference point, or you can think of location (0,0). All other pixels have location with respect to the top-left corner. A position (i,j) means that the pixel is located i rows below the top-left corner and j columns to the right of the top-left corner.
- Once you've found Waldo's likeliest hiding spot, save that position in top_left, which is a tuple with the (i,j) position.
- Also save the resulting norm of the difference in min_diff.

The grading code will be checking the variables `top_left` and `min_diff` defined inside the `#grade` cell below:

[19]:
```
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

[20]:
```
print( top_left, min_diff )
```

Define the image `waldo_found` that "highlights" Waldo face in the complete image. Here is how you should do this:

- Create a copy of `waldo_gray` as `waldo_found`
- Multiplying all pixel values in waldo_found that are **outside** of waldo_face by 0.3. This will darken the other portion of the image.

[21]:
```
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

[22]:
```
# view image
plt.figure(figsize = (10,10))
plt.imshow(waldo_found, cmap="gray")
```

[ ]:
```

```