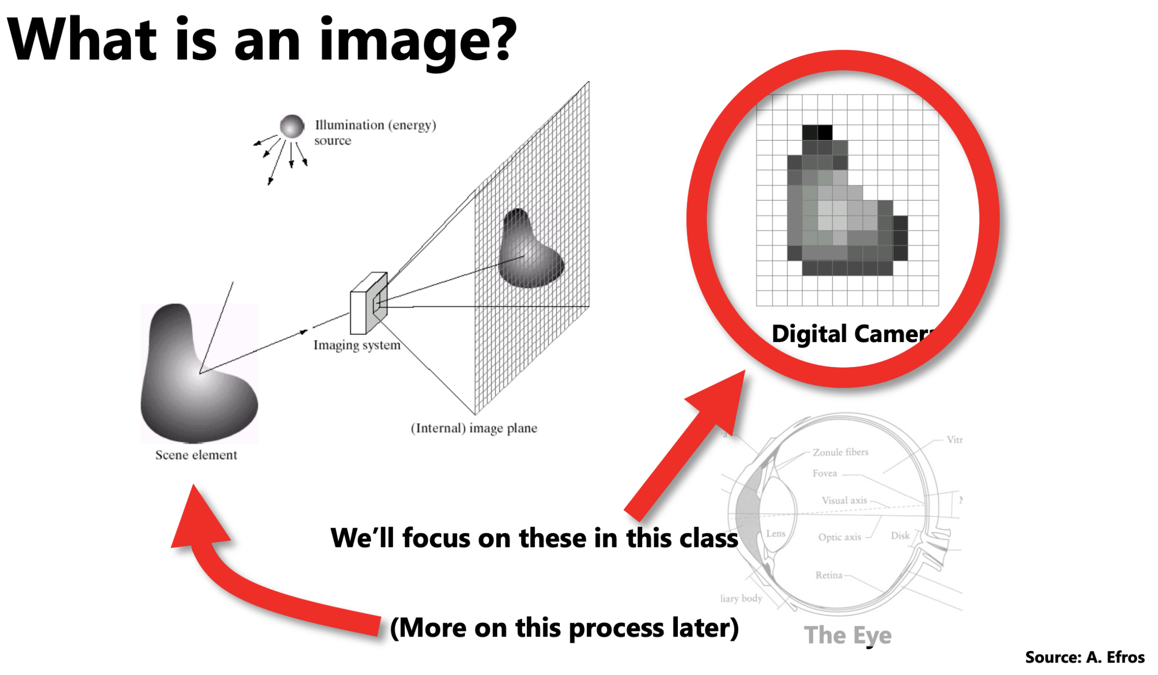
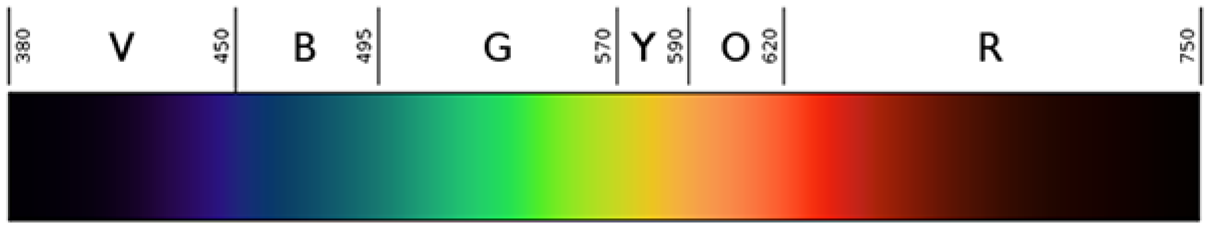
Module 1: Color, Pixel, & Image

"Only light generates color. Without light, no color exists" – Linda Holtzschue (Understanding Color)



Color

Our perception of color comes from our ability to detect a small range of the electromagnetic spectrum which we call the visible spectrum or more commonly visible light.



There are two ways to interpert color: **Additive** (RGB) or **Subtractive** (CMYK).

- **Additive** color is produced by adding together wavelengths of light. Additive color starts at black (absence of light) and moves to white as more light is added.
- **Subtractive** color is produced by removing, partially or totally, certain wavelengths of light while allowing the remaining wavelengths through. Subtractive color starts at white (pure light) and moves to black as wavelengths are filtered out.

Digital color is most commonly represnted in the **RGB** (additive) color system. If you are reading this on a computer monitor, tablet, or smart phone; the device is emmiting wavelengths of light that your eyes are perceiving as color. To make a brighter color the device emits more light. What happens when you turn your device off? It returns to black. The magic of additive color!

Pixel

In digital images we use pixels as containers for light information. Each pixel will either store a brightness value when you have a black and white image or red, green, or blue values when you have a color image. Each value will always be a whole number (0,1,2,3,4,5,...) so we store them as the data type **Integer**.

- An integer is a data type to store whole numbers.
- A single integer pixel value in a black and white image represents the brightness value of that pixel.
- A three integer pixel value in a color image represent the red, green, and blue values (component colors of that pixel).

RGB: Red, Green, Blue

Three 8-bit integers are most commonly used to store RGB values. (10, 12, 14, 16, ... bit color can be used for higher fidelity.)

8-bit: 256 values from 0-255. (zero-based index)

256 x 256 x 256 = 16,777,216 colors can be represented in 8-bit color.

(red value, green value, blue value)

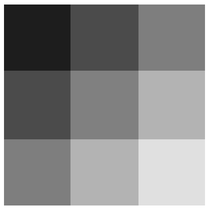
- (255,0,0) = Red
- (0,255,0) = Green
- (0,0,255) = Blue
- (0,0,0) = Black
- (255,255,255) = White

Notice, when the RGB values are all 0 you have black and when they are all 255 you have white. When Red, Green, and Blue is at it's maximum value (255) while the others are set to 0 you have a pure color.

Pixel Grid (Matrix)

Every image is made up of a grid (matrix) of pixels. The grid holds all the pixel values in their correct location.

The	Pixel	Grid
29	75	126
75	128	179
126	179	224



You can think of a (grayscale) image as a function:

$$f : \mathbb{R}^2 \to \mathbb{R}$$

where

$$f(x, y)$$

gives the intensity at position

$$(x, y)$$

A digital image is a discrete (sampled, quantized) version of this function.

Storing Image Information

How can we store these values in the computer's memory?

First, let's store them each individualy.

Seperate vaiables:

pixel0 = 29
pixel1 = 75
pixel2 = 126
pixel3 = 75
pixel4 = 128
pixel5 = 179
pixel6 = 126
pixel7 = 179
pixel8 = 224

Each variable is stored but we have lost the context that they belong to one image. We need a way to place all the seperate variables into one container and tell the computer they make up our image.

Array

We collect the individual pixel values in a structure called an **Array**. The array will hold all the pixel values in a single container.

To simplify this example instead of using RGB values in our array we are going to use a brightness values since they are just a single variable.

1D Array: [ ]

image = [55,74,89,124,174,201,191,215,245]

The array gives us the container to store all our pixel values.

1D	-	-	-	Pixel	-	-	-	Array
29	75	126	75	128	179	126	179	224

The previous 1D array can be rendered as a 3x3 image:

	0	1	2
0	29	75	126
1	75	128	179
2	126	179	224
	<--	**width=3**	-->

To find the location of our pixel value in the 1D array we use the formula: LOCATION = X + Y*WIDTH

For example, if we wanted to find the array location of the pixel with the value 224. Location = 2 + (2 * 3) = 8.

Our rows and columns use zero-based indexing while the width uses one-based indexing. This is why the width is 3 but the x and y locations are 2.

In [ ]:

```
#Import Libraries
import numpy as np

# Print the variable at position 8.

#1D array python
image = [29,75,126,75,128,179,126,179,224]
print(image[8])

#1D array numpy
image = np.array([29,75,126,75,128,179,126,179,224])
print(image[8])
```

```
224
224
```

2D Array: [[ ]]

image = [[29,75,126],[75,128,179],[126,179,224]]

Going one step further we can group each row in our pixel grid into a seperate array. This creates a 2D array, which is a set of 1D arrays of the same data type contained in another 1D array. The easist way to visualize a 2D array is to think of our pixel grid.

	0	**1**	**2**
0	29	75	126
1	75	128	179
2	126	179	224

- The first dimension of the array is giving us access to each row in the image. The rows location corresponds with the Y axis of the grid.
- The second dimension of the array is giving us access to each pixel value in the selected row. The individual pixel values correspond with the X axis of the grid.

For example, if we wanted to find the array location of the pixel with the value 179. We would select the value at row 1 position 2.

In [ ]:

```
# Print the variable at row 1 position 2

#2D array python
image = [[29,75,126],[75,128,179],[126,179,224]]
print(image[1][2])

#2D array numpy
image = np.array([[29,75,126],[75,128,179],[126,179,224]])
print(image[1][2])
```

```
179
179
```

Image

Let's install OpenCv in our Juptyer notebook.

We'll start by reading in an image, examining it's properties, and then drawing it to the screen.

In [ ]:

```
#Install Open CV and tutorial on getting Jupyter notebook up and running.
# %pip install opencv-contrib-python
# %pip install matplotlib

#Import Libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt

#Saving an image. 0 for grayscale. 1 for color. -1 for color with a alpha channe
img = cv2.imread('../Graphics/image1.png',0)

# #How many elements are in that data.
print(img.size)

# #What is the shape of the array.
```

```
print(img.shape)

# #What type are the elements.
print(img.dtype)

# #What is the first element
print(img[0][0])

#Print my data
print(img)
```

```
225
(15, 15)
uint8
255
[[255 255 255 255 255   0   0   0   0   0 255 255 255 255 255]
 [255 255 255 255   0 255 255 255 255 255   0 255 255 255 255]
 [255 255 255   0 255 255 255 255 255 255 255   0 255 255 255]
 [255 255   0 255 255 255 255 255 255 255 255 255   0 255 255]
 [255   0 255 255 255 255 255 255 255 255 255 255 255   0 255]
 [  0 255 255 255   0 255 255 255 255 255   0 255 255 255   0]
 [  0 255 255 255   0 255 255 255 255 255   0 255 255 255   0]
 [  0 255 255 255 255 255 255 255 255 255 255 255 255 255   0]
 [  0 255 255 255   0 255 255 255 255 255   0 255 255 255   0]
 [  0 255 255 255   0 255 255 255 255 255   0 255 255 255   0]
 [255   0 255 255 255   0   0   0   0   0 255 255 255   0 255]
 [255 255   0 255 255 255 255 255 255 255 255 255   0 255 255]
 [255 255 255   0 255 255 255 255 255 255 255   0 255 255 255]
 [255 255 255 255   0 255 255 255 255 255   0 255 255 255 255]
 [255 255 255 255 255   0   0   0   0   0 255 255 255 255 255]]
```

In [ ]:

```
if isinstance(img, np.ndarray):
    print("my_array is a numpy array")
else:
    print("my_array is not a numpy array")
```

```
my_array is a numpy array
```

Results

- We have an image with 225 elements contained in a 2D array with a length of 15 in both dimensions.
- The data type of each element is uint8 which stands for unsigned 8-bit integer. Unsigned denotes that it is non-negative number (zero or positive).
- The first element of the first row has a value of 255.
- The image is saved in a numpy array.

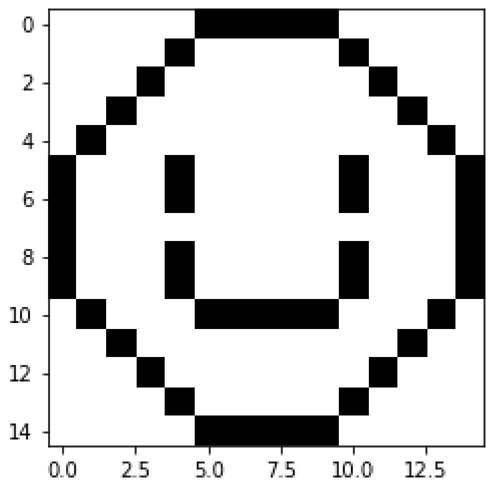
We also printed the data to the console to see what values it contained. The data is made up of rows of pixels set to either 255 or 0 which we just learned represent black and white. Your probably can guess what our image is just from this output.

In [ ]:

```
plt.imshow(img, cmap='gray')
```

Out[ ]:

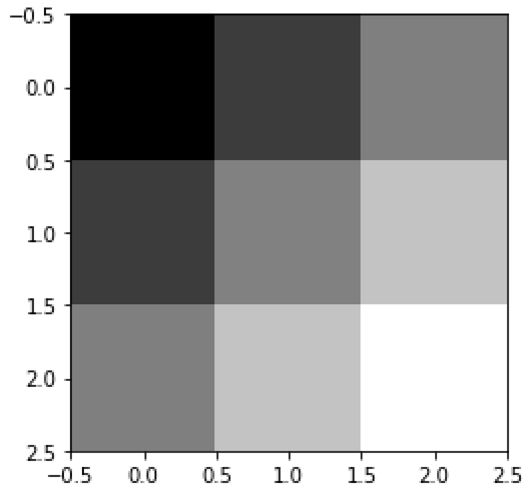
```
<matplotlib.image.AxesImage at 0x2245dec2580>
```



```
In [ ]:
#Create your own grayscale image

grayscaleImage = [[29,75,126],[75,128,179],[126,179,224]]
plt.imshow(binaryimage, cmap='gray')
```

```
Out[ ]: <matplotlib.image.AxesImage at 0x7fd67d02ef40>
```



Now let's read a color image and examine the results.

```
In [ ]:
#Saving an image. 0 for grayscale. 1 for color. -1 for color with a alpha channe
imgFace = cv2.imread('Graphics/face.png',1)

#How many elements are in that data.
print(imgFace.size)

#What is the shape of the array.
print(imgFace.shape)

#What type are the elements.
print(img.dtype)

#RGB pixel values at location x=0 y=0
print(imgFace[0][0])

#Individual RGB values.
print(imgFace[0][0][0])
print(imgFace[0][0][1])
print(imgFace[0][0][2])
```

```
120000
(200, 200, 3)
uint8
[ 85 101  80]
85
101
80
```

Results

- We have an image with 120,000 elements contained in a 3D array with a length of 200 in the first two dimensions and 3 in the last dimension.
- The data type of each element is uint8.

We have only discussed 1D and 2D arrays so far but this image is contained in a 3D array. Our previous example showed a 2D array with single variable brightness values in each array position. These brightness values will create a black & white (grayscale) image. If we want a color image we need to store and access the RGB values for each pixel. Our one brightness value has turned into three color values. We know what tool to use when we have collections of data that are the same data type. We add another array for the RGB values of each pixel.

An array for our image, containing an array for each row of pixels, and an array for each pixel's RGB values. [[[]]]

3D Array [[[ ]]]

	0	1	2
0	55,100,200	74,124,100	89,210,10
1	124,74,191	174,43,34	201,142,60
2	191,50,10	215,111,84	245,139,81

In [ ]:
```
print(imgFace[0][0])
print(imgFace[0][0][0])
print(imgFace[0][0][1])
print(imgFace[0][0][2])
```

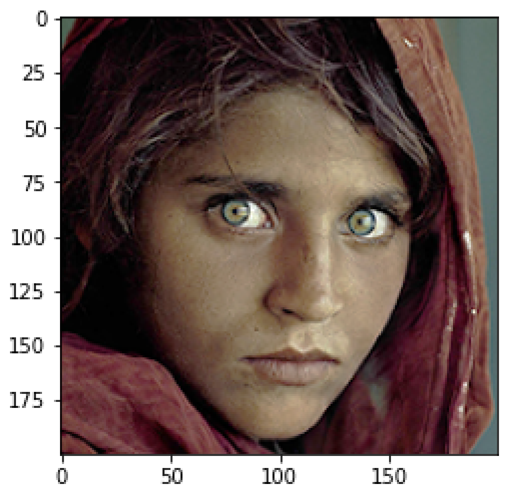
```
[ 85 101  80]
85
```

When we print the element at array position [0][0] we get another array containing [85 101 80]. When we access the 3rd dimension, print the element at [0][0][0], we get the single variable 85. The array structure is a path to the single RGB color values of our image.

In [ ]:
```
#OpenCV by defualt uses BGR instead of RGB.  BGR is an old digital camera standa
imgFace = cv2.cvtColor(imgFace, cv2.COLOR_BGR2RGB)

#Use matplotlib while using a python notebook
plt.imshow(imgFace)
```

Out[ ]: `<matplotlib.image.AxesImage at 0x2246277dc40>`



Exercises

(1) 50,75,100,125,150,175,200,225. Show this sequence of values in a 1D and 2D array.

(2) 50,75,100,125,150,175,200,225,250. In a 1D array, what is the array position of the value 150?

(3) 50,75,100,125,150,175,200,225,250. In a 3x3 2D array, what is the array position of the value 150?

(4) Find a photo from Google Images, save it as a variable, and draw it using matplotlib.

(5) How would you extract the following subarrays using slicing? (Code Below)

In [ ]:
```
'''
(A)
[100,175,250]

(B)
[125, 150, 175]

(C)
[[50,75]
[125,150]
[200,225]]

(D)
[[125 150]
[200 225]]
'''

import numpy as np

# Original 3x3 array
arr_2d = np.array([[50, 75, 100],
                    [125, 150, 175],
                    [200, 225, 250]])

# Slicing to get the subarray
subarray = arr_2d["YOUR CODE HERE"]

print(subarray)
```

(6) Create a 2-D and 3-D array with Integers 0-47.