Task_01_Image_Processing_Output

August 6, 2022

0.1 Image Processing Basics in Python

As far as computers are concerned, images are simply numerical data representations. You can use statistical techniques to manipulate and analyze the numerical properties of images.

In this task, you are required to understand and get familiar with commonly-used syntax involved in image-processing. Further, you should implement some basic geometric transformations and verify them with outputs obtained from libraries.

[1]:
```
!git clone https://github.com/SanVik2000/EE5179-Final.git
```

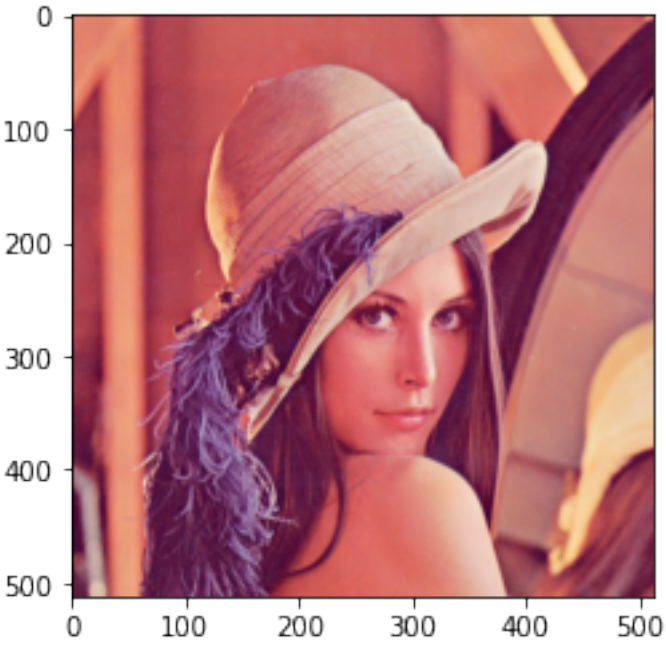
```
Cloning into 'EE5179-Final'…
remote: Enumerating objects: 21, done.
remote: Counting objects: 100% (21/21), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 21 (delta 4), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (21/21), done.
```

0.2 Loading and Displaying an Image

Let's start by loading a PNG file and examining its properties. Run the following cell to load and display an image using the **matplotlib.image** library. You are requested to execute the cells one-by-one after going through the syntax of each cell.

[3]:
```
from matplotlib import image as mpimg
from matplotlib import pyplot as plt
import numpy as np
%matplotlib inline

img1 = mpimg.imread('/content/EE5179-Final/Tutorial2/lena.png')
plt.imshow(img1)
type(img1)
```

[3]:
```
numpy.ndarray
```



So we can see the file is definitely an image, but note that the data type of the **img1** object is actually a multidimensional numpy array.

Let's take a closer look at this array by printing it's shape:

[4]:
```
print("Image Shape : " , img1.shape)
```

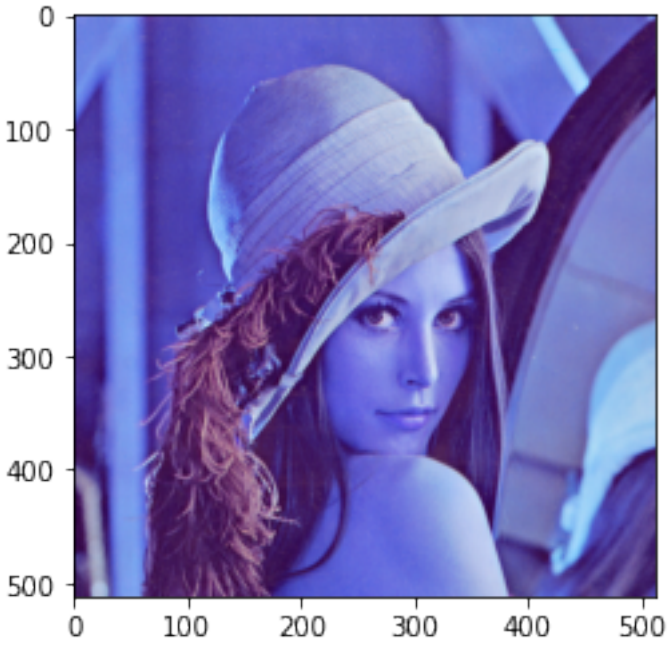
```
Image Shape :  (512, 512, 3)
```

The image is actually composed of three "layers, or *channels*, for red, green, and blue (RGB) pixel intensities. Each layer of the image represents 512 x 512 pixels (the dimensions of the image).

Now let's load and display the same image but this time we'll use another popular Python library for working with images - **cv2**.

[6]:
```
import cv2
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

img2 = cv2.imread('/content/EE5179-Final/Tutorial2/lena.png')
plt.imshow(img2)
type(img2)
```

[6]:
```
numpy.ndarray
```

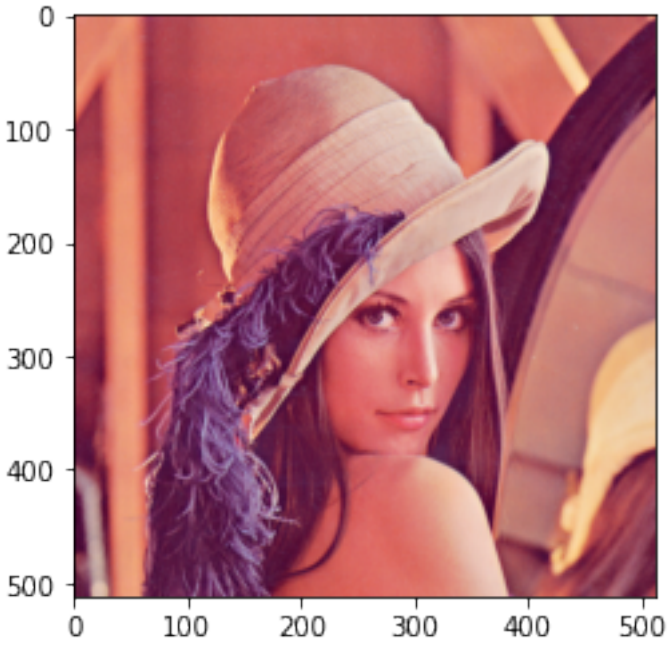


Once again we've got a multidimensional array; but the image appears with a blue-tint

The problem is that cv2 loads the array of image data with the channels ordered as blue, green, red (BGR) instead of red, green blue (RGB). This can be fixed as follows:

[7]:
```
img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2RGB)
plt.imshow(img2)
```

[7]:
```
<matplotlib.image.AxesImage at 0x7fe154b789d0>
```

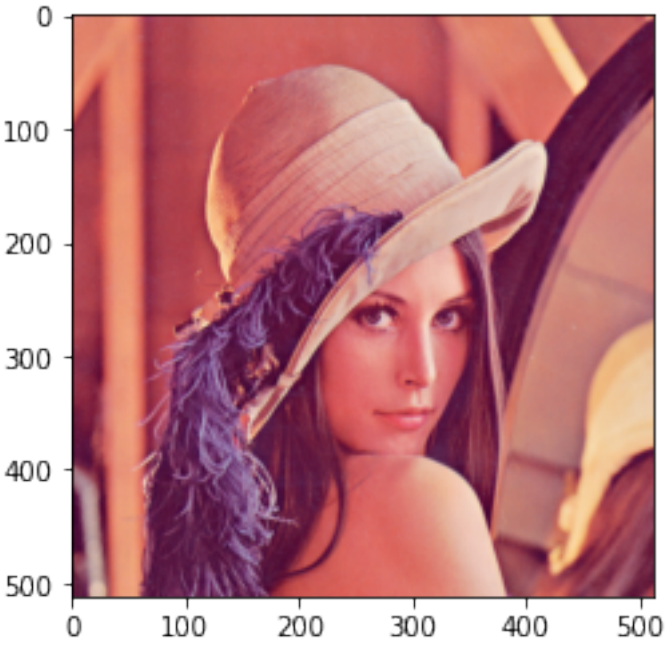


That's better - now the image is a numpy array with 3 dimensions for red, green, and blue.

There's one more commonly used library for image processing in Python we should consider - **PIL**:

[8]:
```
from PIL import Image
import matplotlib.pyplot as plt
%matplotlib inline

img3 = Image.open('/content/EE5179-Final/Tutorial2/lena.png')
plt.imshow(img3)
type(img3)
```

[8]:
```
PIL.PngImagePlugin.PngImageFile
```



This time, the data type is a JpegImageFile - not a numpy array. That's great if we only want to manipulate it using PIL methods; but sometimes we'll want to be flexible and process images using mulitple libraries; so we need a consistent format.

Fortunately, it's easy to convert a PIL JpegImageFile to a numpy array:

[9]:
```
import numpy as np

img3 = np.array(img3)
img3.shape
```

[9]:
```
(512, 512, 3)
```

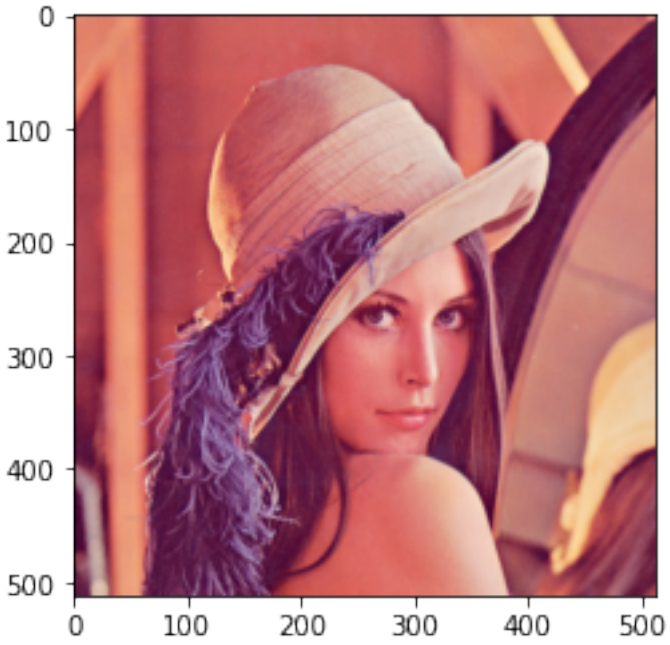
So fundamentally, the common format for image libraries is a numpy array. Using this as the standard format for your image processing workflow, converting to and from other formats as required, is the best way to take advantage of the particular strengths in each library while keeping your code consistent.

You can even save a numpy array in an optimized format, should you need to persist images into storage:

[10]:
```
import numpy as np

# Save the image
np.save('img.npy', img3)

#Load the image
```

```
img3 = np.load('img.npy')

plt.imshow(img3)
```

[10]: <matplotlib.image.AxesImage at 0x7fe1541a54d0>

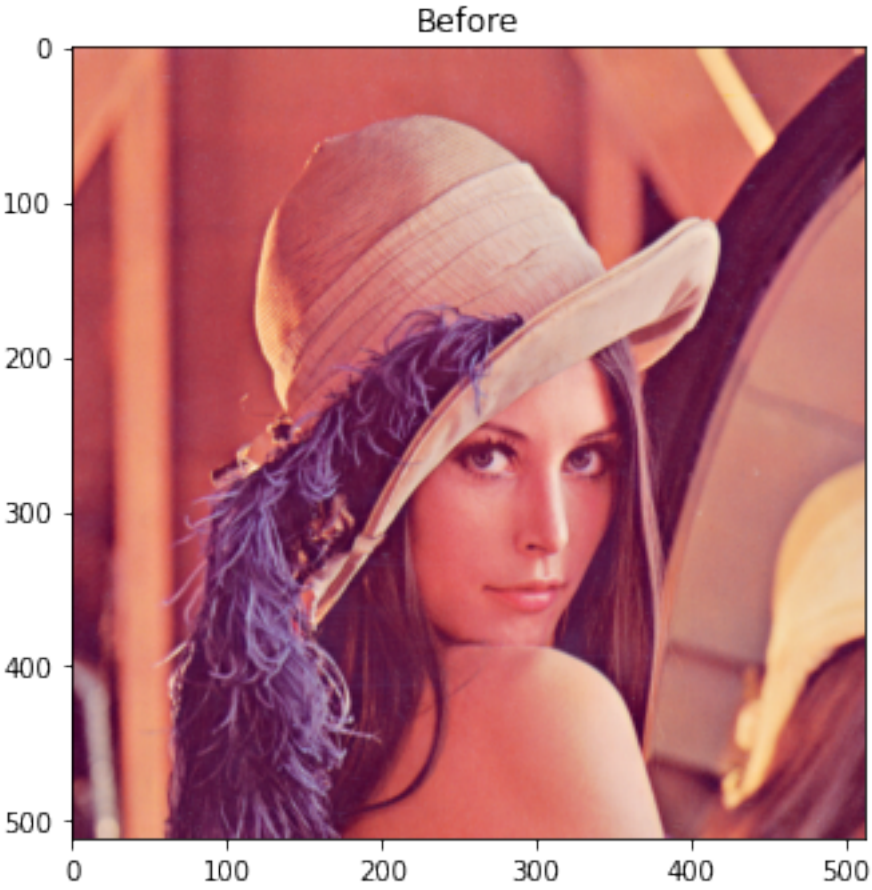


0.3 Resizing an Image

One of the most common manipulations of an image is to resize it. This can be particularly important when you're preparing multiple images to train a machine learning model, as you'll generally want to ensure that all of your training images have consistent dimensions.

[11]:
```
from PIL import Image, ImageOps
import matplotlib.pyplot as plt
%matplotlib inline

img2_resized = cv2.resize(img2, (1050, 1610))

# Show the original and resized images
# Create a figure
fig = plt.figure(figsize=(12, 12))

# Subplot for original image
a=fig.add_subplot(2,1,1)
imgplot = plt.imshow(img2)
```

```
a.set_title('Before')

# Subplot for resized image
a=fig.add_subplot(2,1,2)
imgplot = plt.imshow(img2_resized)
a.set_title('After')

plt.show()
```


Before
0
100
200
300
400
500
0
100
200
300
400
500


After
0
200
400
600
800
1000
1200
1400
1600
0
200
400
600
800
1000

0.4 Examining Numerical Properties of the Image Array

So we've seen that an image is inherently an array of values. Let's examine that in more detail. What type of values are in the array?

[12]:
```
img3.dtype
```

[12]:
```
dtype('uint8')
```

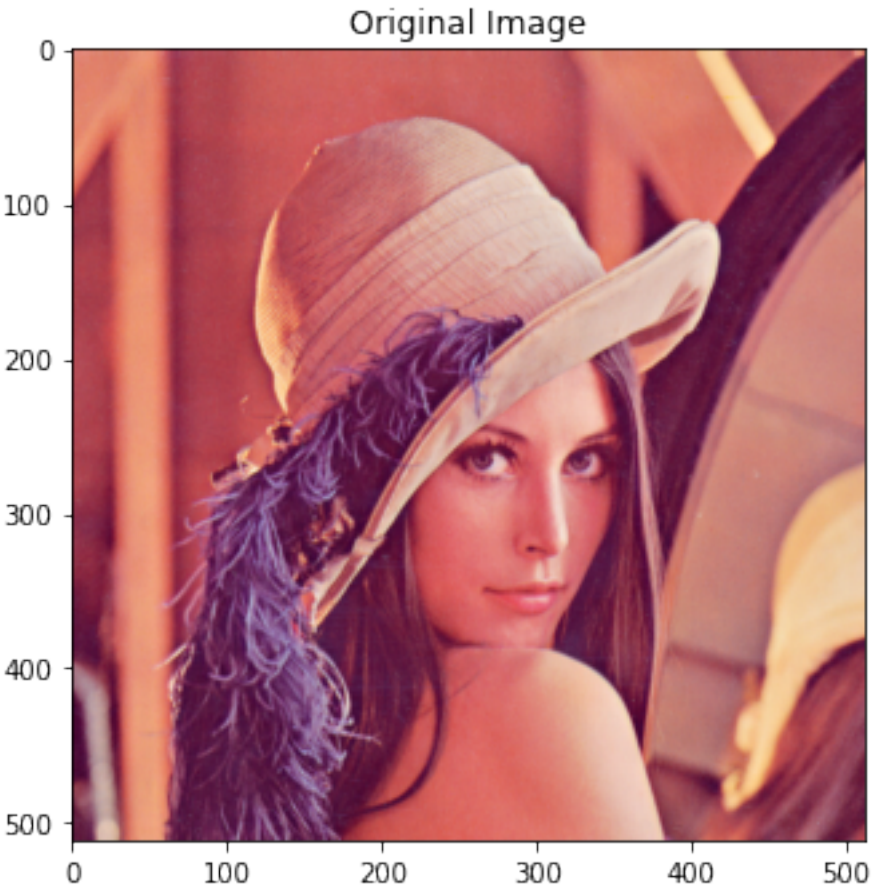
OK, so the array consists of 8-bit integer values. In other words, whole numbers between 0 and 255. These represent the possible pixel intensities for the RGB color channels in each pixel of the image.

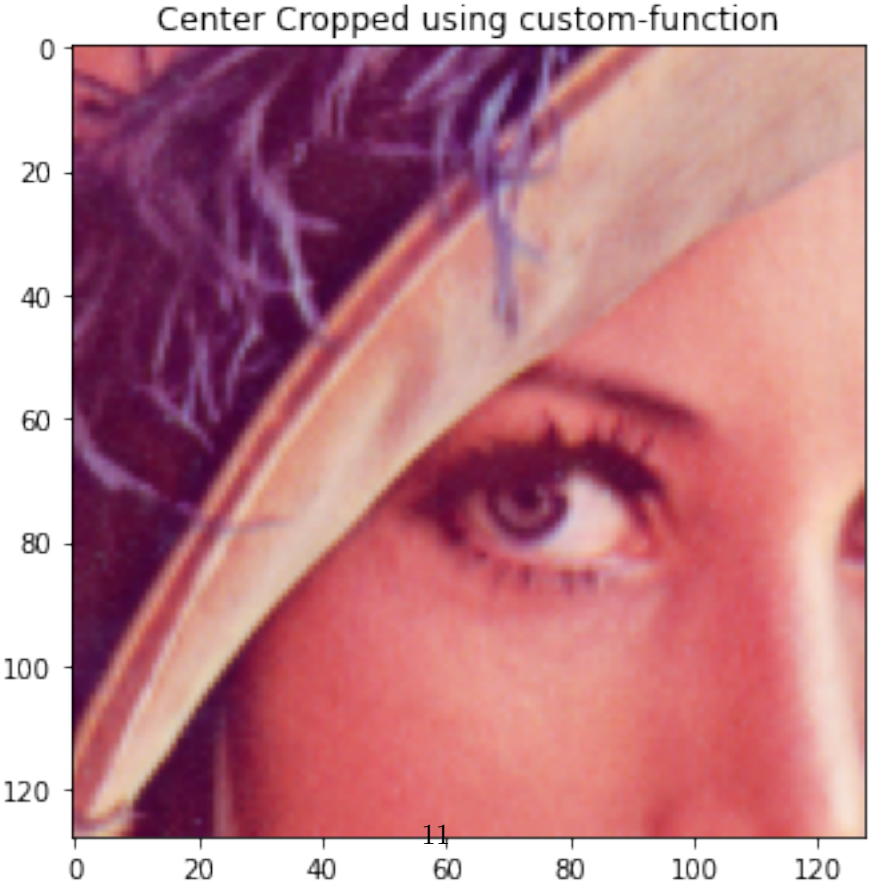
0.5 Center Cropping

In this task, you are expected to complete the following function, that takes in an image in numpy format and the target_crop_size, and returns the center cropped image as output.

[13]:
```
def center_crop(img_arr, target_size):
    center_w = int(img_arr.shape[0]/2)
    center_h = int(img_arr.shape[1]/2)

    width = int(target_size[0]/2)
    height = int(target_size[1]/2)

    cropped_img = img_arr[center_w-width:center_w+width, center_h-height:
 ↪center_h+height, :]
    return cropped_img

img2_cropped = center_crop(img2, (128, 128))

# Show the original and resized images
# Create a figure
fig = plt.figure(figsize=(12, 12))

# Subplot for original image
a=fig.add_subplot(2,1,1)
imgplot = plt.imshow(img2)
a.set_title('Original Image')

# Subplot for center-crop-custom image
a=fig.add_subplot(2,1,2)
imgplot = plt.imshow(img2_cropped)
a.set_title('Center Cropped using custom-function')
```

```
plt.show()
```


Original Image
0
100
200
300
400
500
0
100
200
300
400
500


Center Cropped using custom-function
0
20
40
60
80
100
120
0
20
40
60
80
100
120

0.6 RGB to GrayScale

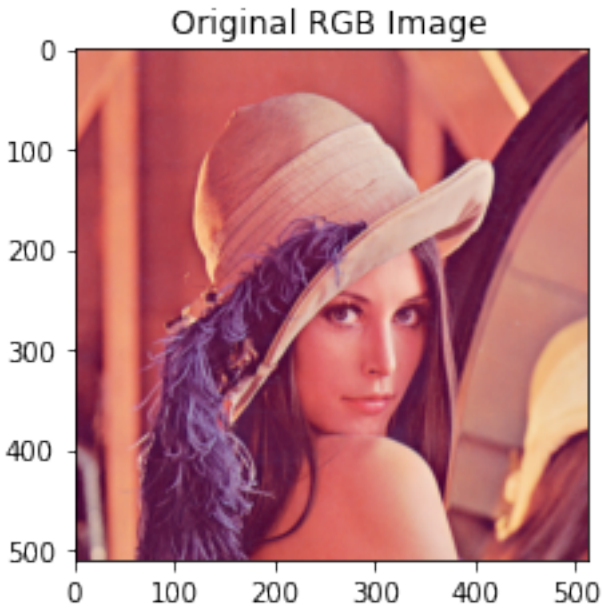
In this task, you are expected to complete the following function, that takes in an RGB-image in numpy format and returns the gray-scale version of the image as output.

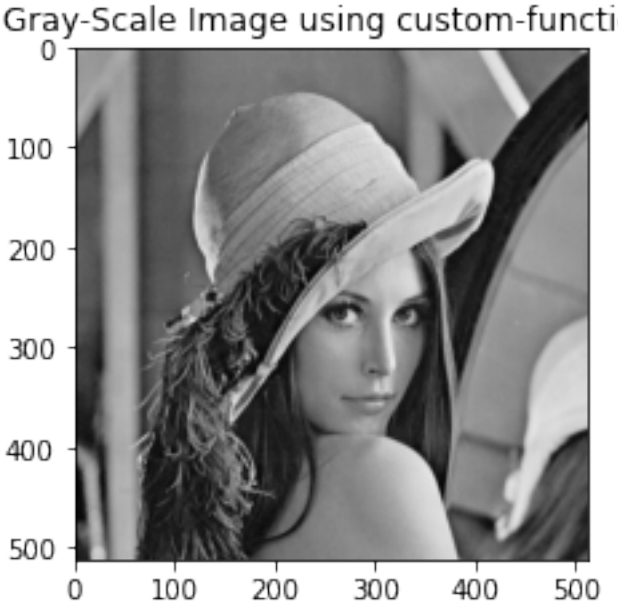
An RGB image can be converted to a gray-scale image using the formula below:

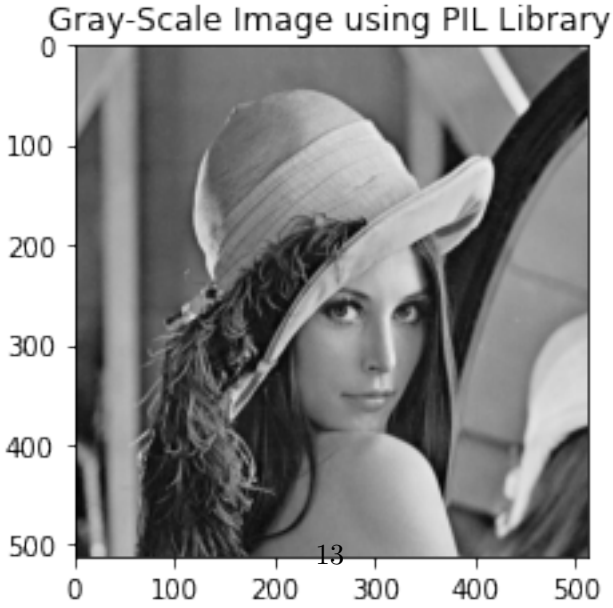
$Gray = 0.2989 * R + 0.5870 * G + 0.1140 * B$

where R,G and B represent the Red, Green and Blue channels of the RGB image.

[14]:
```
def rgb_2_grayscale(img):
    R, G, B = img[:,:,0], img[:,:,1], img[:,:,2]
    imgGray = 0.2989 * R + 0.5870 * G + 0.1140 * B
    return imgGray

img2_gray = rgb_2_grayscale(img2)
img2_pil = Image.fromarray(img2)
img2_gray_pil = np.array(img2_pil.convert('L'))

# Show the original and gray-scale images
# Create a figure
fig = plt.figure(figsize=(12, 12))

# Subplot for original image
a=fig.add_subplot(3,1,1)
imgplot = plt.imshow(img2)
a.set_title('Original RGB Image')

# Subplot for gray-scale-custom image
a=fig.add_subplot(3,1,2)
imgplot = plt.imshow(img2_gray, cmap='gray')
a.set_title('Gray-Scale Image using custom-function')

# Subplot for gray-scale-custom image
a=fig.add_subplot(3,1,3)
imgplot = plt.imshow(img2_gray_pil, cmap='gray')
a.set_title('Gray-Scale Image using PIL Library')

plt.show()
```


Original RGB Image
0
100
200
300
400
500
0
100
200
300
400
500


Gray-Scale Image using custom-function
0
100
200
300
400
500
0
100
200
300
400
500


Gray-Scale Image using PIL Library
0
100
200
300
400
500
0
100
200
300
400
500