pyimagesearch gurus
(https://gurus.pyimagesearch.com/)

1.2: Image basics

So how are we feeling after the first lesson (https://gurus.pyimagesearch.com/lessons/loading-displaying-and-saving-images/)?

Not too bad right? Keep it up — you're doing great.

And if you need any help regarding command line arguments, I would suggest heading over to the PyImageSearch Gurus Community (https://community.pyimagesearch.com/) and asking for some advice and pointers from some of our more seasoned Unix experts.

Feedback

Objectives:

In this section we are going to review the building blocks of an image — the pixel. We'll discuss exactly what a pixel is, how pixels are used to form an image, and then how to access and manipulate pixels in OpenCV. By the end of this lesson you'll:

1. Have a full understanding of what a "pixel" is.
2. Understand the coordinate system of an image.
3. Have the ability to access the Red, Green, and Blue (RGB) values of a pixel...
4. ...Along with set the RGB values of a pixel.
5. Have a gentle introduction to extracting regions from an image.

So, what's a pixel?

Pixels are the raw building block of an image. Every image consists of a set of pixels. There is no smaller granularity than the pixel.

Normally, a pixel is considered the "color" or the "intensity" of light that appears in a given place in our image.

If we think of an image as a grid, each square in the grid contains a single pixel.

Let's take a look at the example image below:

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/florida_trip.jpg)

FIGURE 1: THIS IMAGE IS 600 WIDE AND 450 PIXELS TALL, FOR A TOTAL OF *600 X 450 = 270,000* PIXELS!

The image in **Figure 1** above has a resolution of *600 x 450*, meaning that it is 600 wide and 450 pixels tall. This means that our image is represented as a grid of pixels, with 600 columns and 450 rows. Overall, there are *600 x 450 = 270,000* pixels in our image.

Most pixels are represented in two ways: grayscale and color. In a grayscale image, each pixel has a value between *0* and *255*, where zero is corresponds to "black" and *255* being "white". The values in between *0* and *255* are varying shades of gray, where values closer to *0* are darker and values closer *255* are

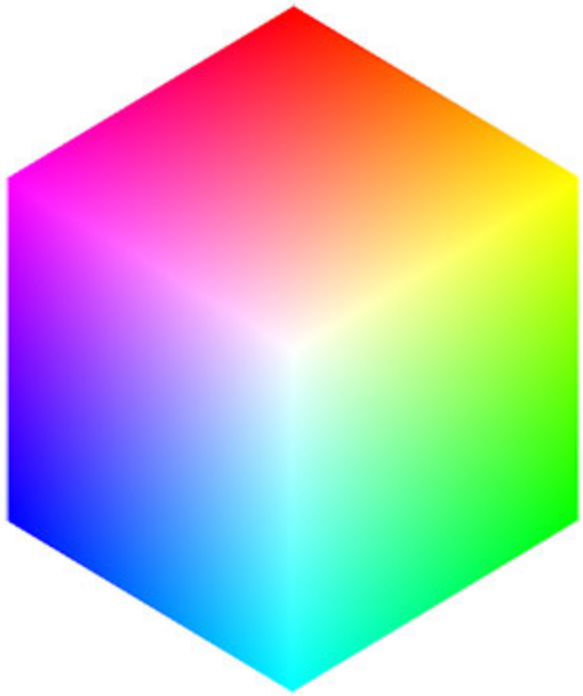
Feedback

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/image_gradient.jpg)

FIGURE 2: IMAGE GRADIENT DEMONSTRATING PIXEL VALUES GOING FROM BLACK (0) TO WHITE (255).

The grayscale gradient image in **Figure 2** above demonstrates *darker pixels* on the left-hand side and progressively *lighter pixels* on the right-hand side.

Color pixels, however, are normally represented in the RGB color space — one value for the Red component, one for Green, and one for Blue, leading to a total of *3 values per pixel*:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/rgb_cube.jpg)

FIGURE 3: THE RGB CUBE.

Other color spaces exist, but let's start with the basics and move our way up from there.

Each of the three Red, Green, and Blue colors are represented by an integer in the range *0 to 255*, which

Feedback

indicates how much of the color there is. Given that the pixel value only needs to be in the range [0, *255]* we normally use an 8-bit unsigned integer to represent each color intensity.

We then combine these values into a RGB tuple in the form `(red, green, blue)` . This tuple represents our color.

To construct a white color, we would fill each of the red, green, and blue buckets completely up, like this: `(255, 255, 255)` — since white is the presence of all color.

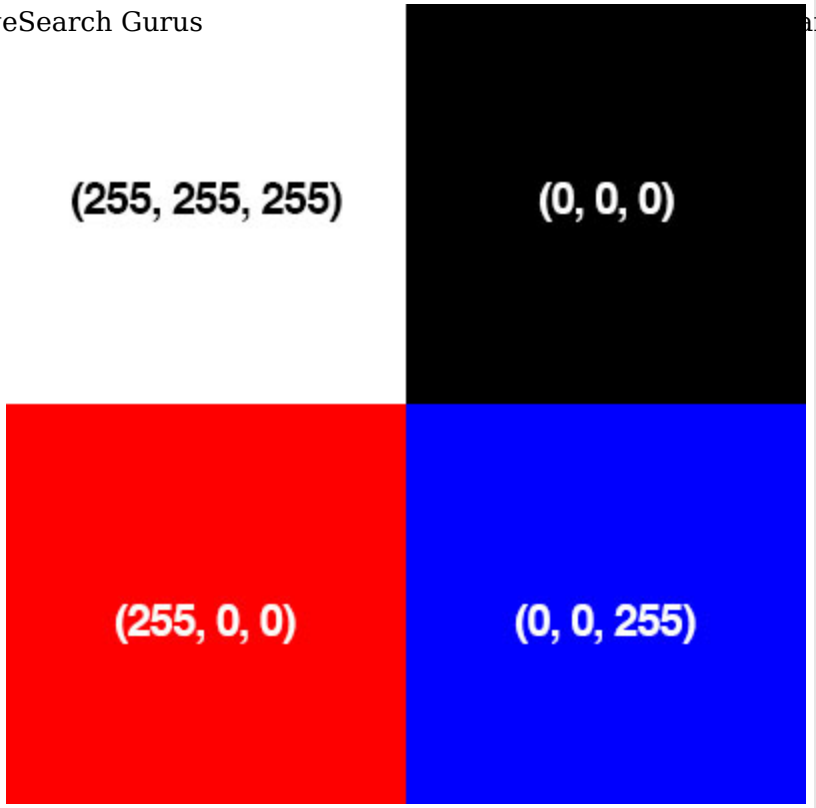
Then, to create a black color, we would empty each of the buckets out: `(0, 0, 0)` — since black is the absence of color.

To create a pure red color, we would fill up the red bucket (and only the red bucket) up completely: `(255, 0, 0)` .

Are you starting to see a pattern?

Take a look at the following image to make this concept more clear:

Feedback



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/color_examples.jpg)

FIGURE 4: HERE WE HAVE THREE EXAMPLES OF COLORS AND THE "BUCKET" AMOUNTS FOR EACH OF THE RED, GREEN, AND BLUE COMPONENTS.

Feedback

In the *Top-Left* example we have the color *white* — each of the Red, Green, and Blue buckets have been completely filled up to form the white color. And on the *Top-Right*, we have the color *black* — the Red, Green, and Blue buckets are now totally empty.

Similarly, to form the color red in the *Bottom-Left* we simply fill up the Red bucket completely, leaving the other Green and Blue buckets totally empty. Finally, blue is formed by filling up only the Blue bucket, as demonstrated in the *Bottom-Right*.

For your reference, here are some common colors represented as RGB tuples:

- **Black:** `(0, 0, 0)`
- **White:** `(255, 255, 255)`
- **Red:** `(255, 0, 0)`
- **Green:** `(0, 255, 0)`
- **Blue:** `(0, 0, 255)`
- **Aqua:** `(0, 255, 255)`
- **Fuchsia:** `(255, 0, 255)`

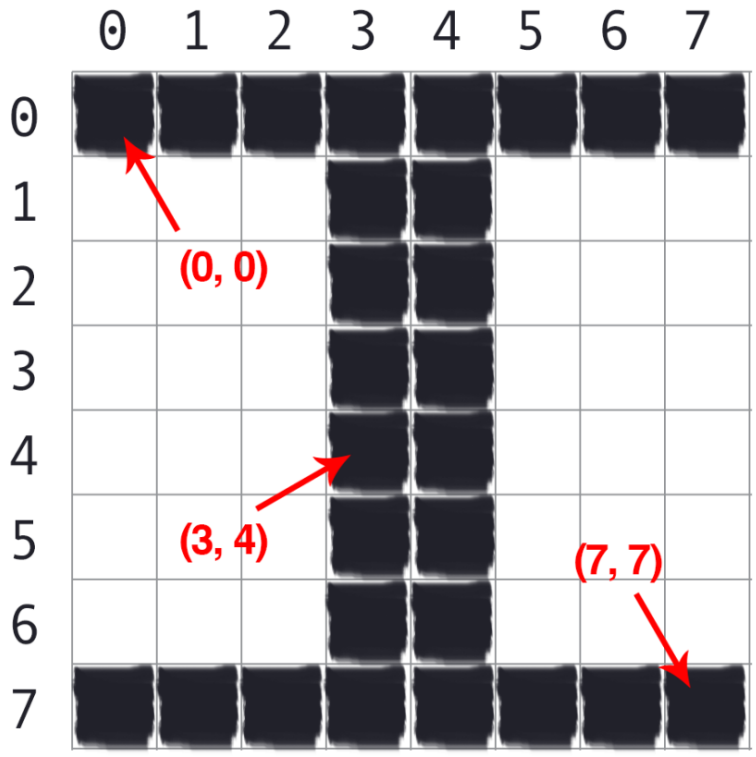
- **Maroon:** `(128, 0, 0)`
- **Navy:** `(0, 0, 128)`
- **Olive:** `(128, 128, 0)`
- **Purple:** `(128, 0, 128)`
- **Teal:** `(0, 128, 128)`
- **Yellow:** `(255, 255, 0)`

Now that we have a good understanding of pixels, let's have a quick review of the coordinate system.

Overview of the Coordinate System

As I mentioned in **Figure 1** above, an image is represented as a grid of pixels. Imagine our grid as a piece of graph paper. Using this graph paper, the point *(0, 0)* corresponds to the upper left corner of the image. As we move down and to the right, both the *x* and *y* values increase.

Let's take a look at the image in **Figure 5** to make this point more clear:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/image_grid.png)

FIGURE 5: THE LETTER "I" PLACED ON A PIECE OF GRAPH PAPER. PIXELS ARE ACCESSED BY THEIR (X, Y) COORDINATES, WHERE WE GO X COLUMNS TO THE RIGHT AND Y ROWS DOWN, KEEPING IN MIND THAT PYTHON IS ZERO-INDEXED: WE START COUNTING FROM ZERO RATHER THAN ONE.

Feedback

image we have the region from a piece of graph paper. We see that we have an 8 x 8 grid with 64 total pixels.

The point at *(0, 0)* corresponds to the top left pixel in our image, whereas the point *(7, 7)* corresponds to the bottom right corner.

It is important to note that we are counting from *zero* rather than *one*. The Python language is *zero indexed*, meaning that we always start counting from zero. Keep this mind and you'll avoid a lot of confusion later on.

Finally, the pixel 4 columns to the right and 5 rows down is indexed by the point *(3, 4)*, keeping in mind that we are counting from *zero* rather than *one*.

Accessing and Manipulating Pixels

Admittedly, the example from **Module 1.1** (https://gurus.pyimagesearch.com/lessons/loading-displaying-and-saving-images/) wasn't very exciting. All we did was load an image off disk, display it, and then write it back to disk in a different image file format. Let's do something a little more exciting and see how we can access and manipulate the pixels in an image.

Feedback

But before we start coding, if you configured your development environment to use Python virtual environments (or if you're using the virtual machine I have supplied), be sure to access your `gurus` environment prior to executing any code:

Accessing the gurus virtual environment — Python

```
1 $ workon gurus
```

Executing this command will ensure that you are in the `gurus` Python virtual environment and your code will have access to the appropriate computer vision libraries. This is just a friendly reminder to check your Python environment before executing any code. It's an easy, subtle step to miss that can lead to some serious head scratching when OpenCV does not import!

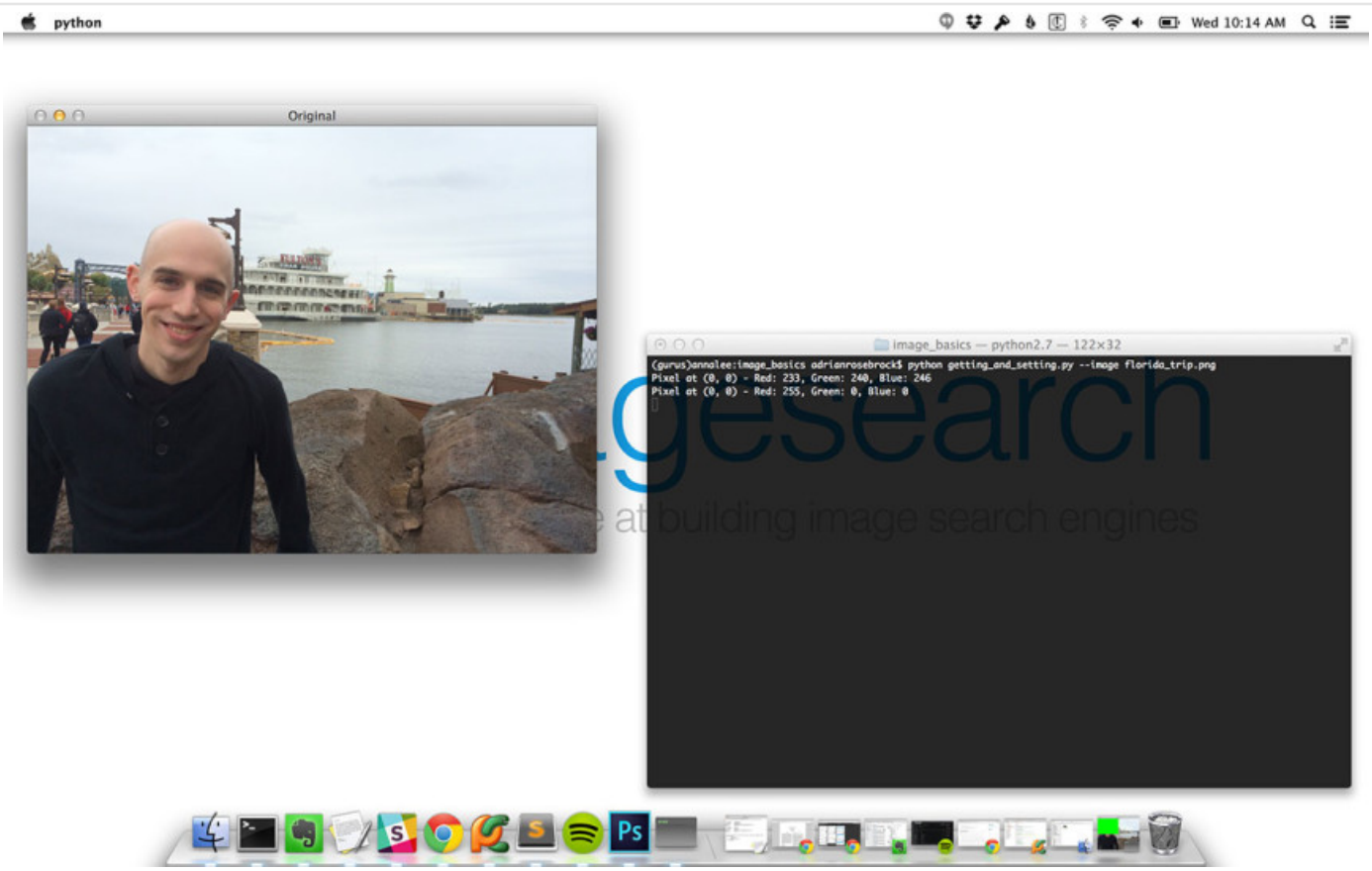
Anyway, let's get coding:

getting_and_setting.py — Python

```
1  # import the necessary packages
2  import argparse
3  import cv2
4
5  # construct the argument parser and parse the arguments
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required=True, help="Path to the image")
8  args = vars(ap.parse_args())
9
10 # load the image, grab its dimensions, and show it
11 image = cv2.imread(args["image"])
12 (h, w) = image.shape[:2]
13 cv2.imshow("Original", image)
```

Similar to our example in the previous chapter, **Lines 1-8** handle importing the packages we need along with setting up our argument parser. There is only one command line argument needed, `--image` , which is the path to the image we are going to work with.

Line 11 handles loading the image from, **Line 12** grabs the dimensions of the image (i.e. the width and height), and finally **Line 13** displays our image to the screen:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/getting_and_setting_original.jpg)

FIGURE 6: LOADING OUR IMAGE AND DISPLAYING IT TO OUR SCREEN.

Feedback

So now that we have the image loaded, how can we access the actual pixel values?

Remember, OpenCV represents images as NumPy arrays. Conceptually, we can think of this representation as a matrix, as discussed in ***Overview of the Coordinate System*** section above. In order to access a pixel value, we just need to supply the *x* and *y* coordinates of the pixel we are interested in. From there, we are given a tuple representing the Red, Green, and Blue components of the image.

However, it's important to note that OpenCV stores RGB channels in *reverse order*. While we normally think in terms of Red, Green, and Blue, OpenCV actually stores them in the order of Blue, Green, and Red.

So why in the world does OpenCV store images in BGR rather than RGB order?

The answer is *efficiency*. Multi-channel images in OpenCV are stored in row order — meaning that each of the Blue, Green, and Red components are concatenated together in sub-columns to form the entire image. Furthermore, each row of the matrix should be aligned to a 4-byte boundary, one byte for each of the Red, Green, Blue, and Alpha channels. Given our image, the last row of the image comes first in memory, thus we store the components in reverse order.

Note: The terms "multi-channel" images and "alpha channel" may seem confusing right now, so don't focus too much on these terms at the present moment. My suggestion is to finish going through this module and pay close attention to the **Lighting and color spaces (https://gurus.pyimagesearch.com/lessons/lighting-and-color-spaces/)** lesson, then come back to this section of the lesson if you need further clarification.

Feedback

Again, I'll make sure I say this again: It's **important to note that OpenCV stores images in *BGR order* rather than *RGB order*** since this caveat could cause some confusion later.

Alright, let's explore some code that can be used to access and manipulate pixels:

getting_and_setting.py Python

```
15 # images are just NumPy arrays. The top-left pixel can be found at (0, 0)
16 (b, g, r) = image[0, 0]
17 print("Pixel at (0, 0) - Red: {r}, Green: {g}, Blue: {b}".format(r=r, g=g, b=b))
18
19 # now, let's change the value of the pixel at (0, 0) and make it red
20 image[0, 0] = (0, 0, 255)
21 (b, g, r) = image[0, 0]
22 print("Pixel at (0, 0) - Red: {r}, Green: {g}, Blue: {b}".format(r=r, g=g, b=b))
```

On **Line 16**, we grab the pixel located at *(0, 0)* — the top-left corner of the image. This pixel is represented as a tuple. Again, OpenCV stores RGB pixels in *reverse order*, so when we unpack and access

each element in the tuple, we are actually viewing them in BGR order. Then, **Line 17** then prints out the values of each channel to our console.

As you can see, accessing pixel values is quite easy! NumPy takes care of all the hard work for us. All we are doing are providing indexes into the array.

Just as NumPy makes it easy to *access* pixel values, it also makes it easy to *manipulate* pixel values.

On **Line 20** we manipulate the top-left pixel in the image, which is located at coordinate *(0, 0)* and set it to have a value of `(0, 0, 255)` . If we were reading this pixel value in RGB format, we would have a value of *0* for red, *0* for green, and *255* for blue, thus making it a pure blue color.

However, as I mentioned above, we need to take special care when working with OpenCV. Our pixels are actually stored in BGR format, **not** RGB format.

We actually read this pixel as *255* for red, *0* for green, and *0* for blue, making it a red color, *not* a blue color.

After setting the top-left pixel to have a red color on **Line 20**, we then grab the pixel value and print it back to console on **Lines 21 and 22**, just to demonstrate that we have indeed successfully changed the color of the pixel.

Feedback

Accessing and setting a single pixel value is simple enough, but what if we wanted to use NumPy's array slicing capabilities to access larger rectangular portions of the image? The code below demonstrates how we can do this:

getting_and_setting.py Python

```
24 # compute the center of the image, which is simply the width and height
25 # divided by two
26 (cX, cY) = (w // 2, h // 2)
27
28 # since we are using NumPy arrays, we can apply slicing and grab large chunks
29 # of the image -- let's grab the top-left corner
30 tl = image[0:cY, 0:cX]
31 cv2.imshow("Top-Left Corner", tl)
```

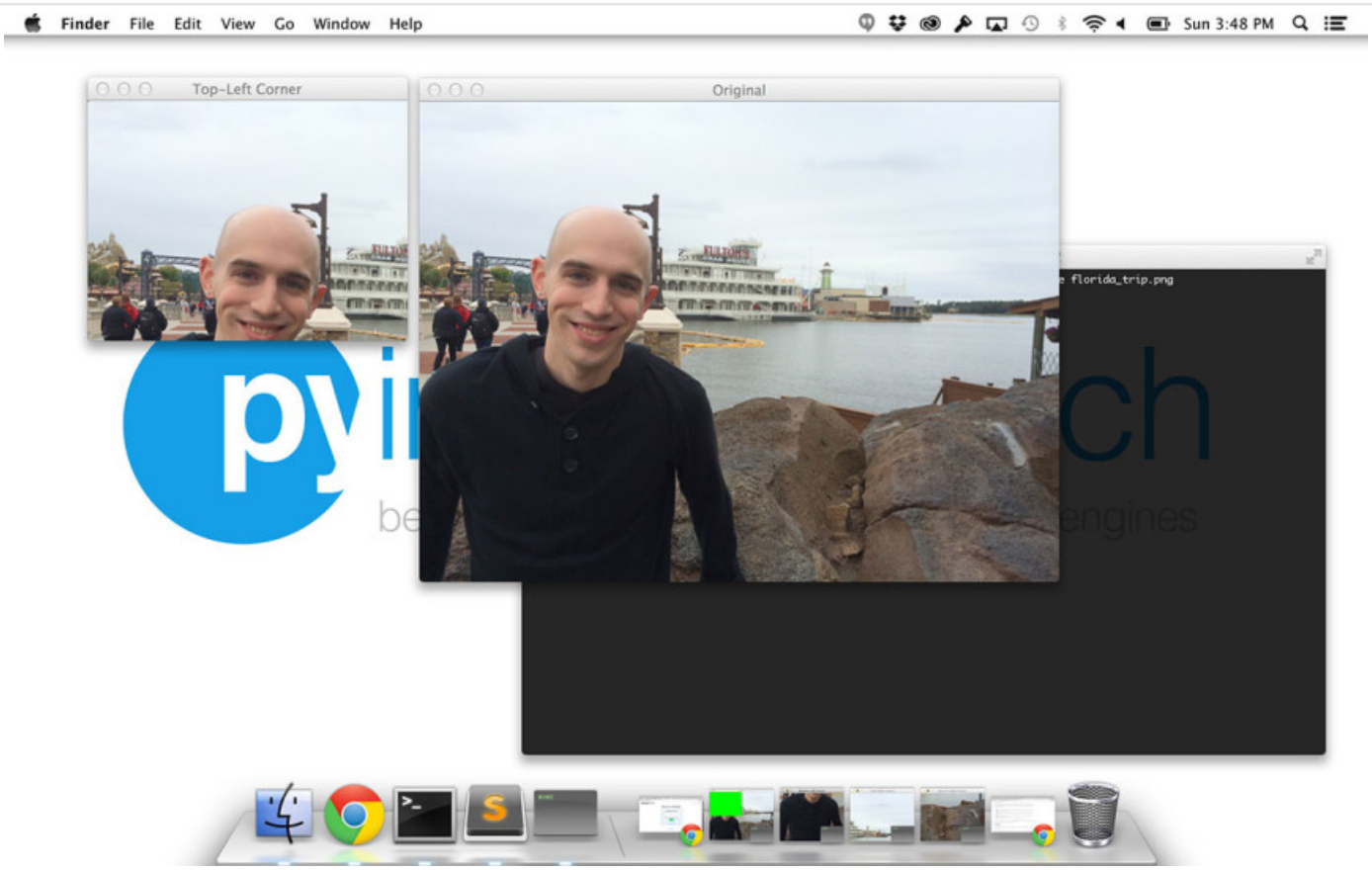
On **Line 26** we compute the center *(x, y)*-coordinates of the image. This is simply accomplished by dividing the width and height by *2*.

Then, on **Line 30** we use simple NumPy array slicing to extract the *[0, cX)* and *[0, cY)* region of the image.

image, this region corresponds to the top-left corner of the image. In order to grab chunks of an image, NumPy expects we provide four indexes:

- **Start y:** The first value is the starting *y* coordinate. This is where our array slice will start along the y-axis. In our example above, our slice starts at *y=0*.
- **End y:** Just as we supplied a starting *y* value, we must provide an ending *y* value. Our slice stops along the y-axis when *y=cY*.
- **Start x:** The third value we must supply is the starting *x* coordinate for the slice. In order to grab the top-left region of the image, we start at *x=0*.
- **End x:** Lastly, we need to provide the x-axis value for our slice to stop. We stop when *x=cX*.

Once we have extracted the top-left corner of the image, **Line 31** shows us the result of the cropping. Notice how our image is just the top-left corner of our original image:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/top_left_corner.jpg)

FIGURE 7: EXTRACTING THE TOP-LEFT CORNER OF THE IMAGE USING ARRAY SLICING.

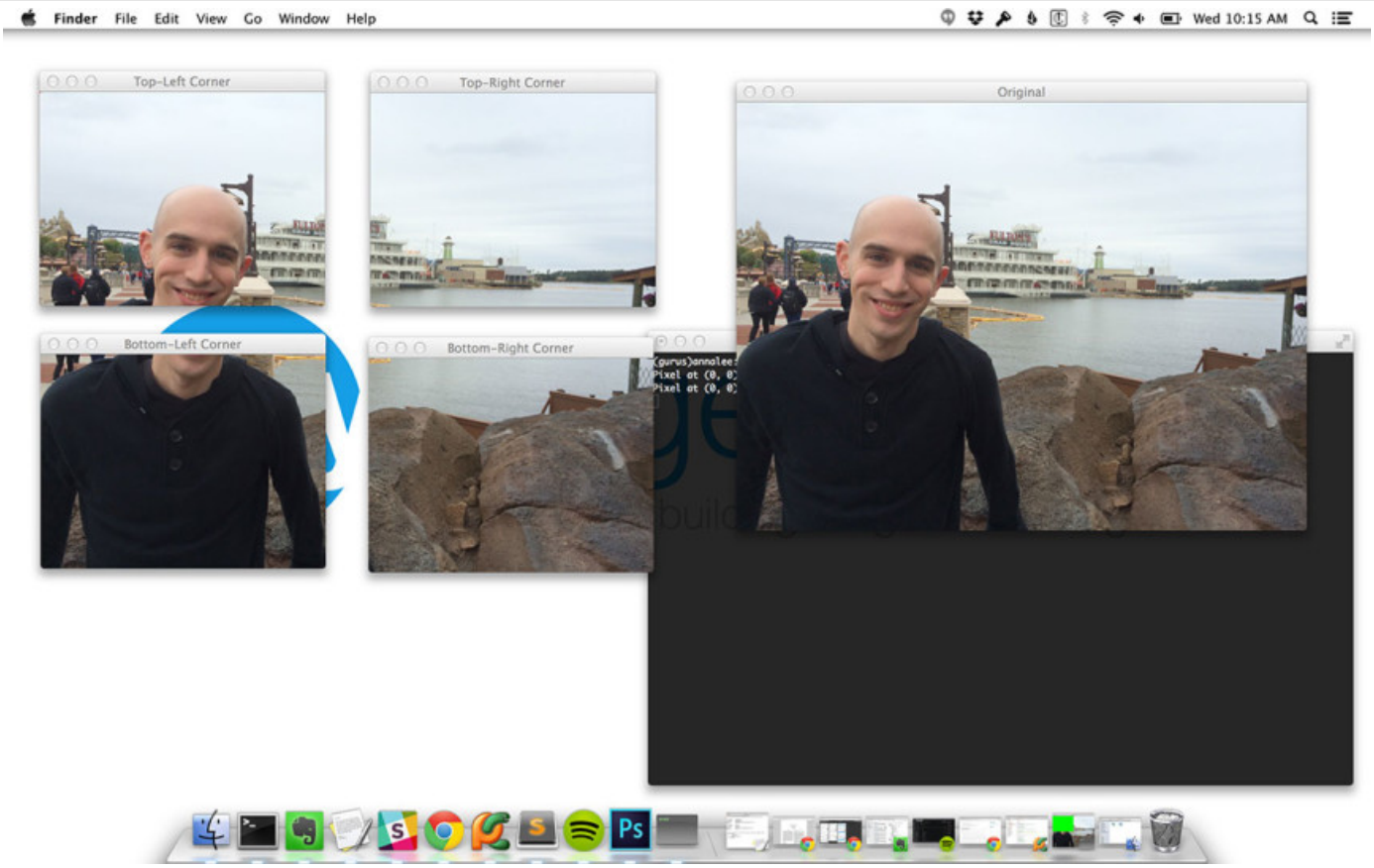
Let's extend this example a little further so we can get some practice using NumPy array slicing to extract regions from images:

Feedback

getting_and_setting.py Python

```
33 # in a similar fashion, let's grab the top-right, bottom-right, and bottom-left
34 # corners and display them
35 tr = image[0:cY, cX:w]
36 br = image[cY:h, cX:w]
37 bl = image[cY:h, 0:cX]
38 cv2.imshow("Top-Right Corner", tr)
39 cv2.imshow("Bottom-Right Corner", br)
40 cv2.imshow("Bottom-Left Corner", bl)
```

In a similar fashion to the example above, **Line 35** extracts the top-right corner of the image, **Line 36** extracts the bottom-right corner, and **Line 37** the bottom-left. Finally, all four corners of the image are displayed on screen on **Lines 38-40**, like this:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/getting_and_settign_corners.jpg)

FIGURE 8: EXTRACTING THE FOUR CORNERS OF AN IMAGE.

Understanding NumPy array slicing is a very important skill that we will be using heavily throughout this course. If you are unfamiliar with NumPy array slicing, I would suggest taking a few minutes and reading this page (http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html) on the basics of NumPy indexes, arrays, and slicing.

The last thing we are going to do is use array slices to change the color of a region of pixels:

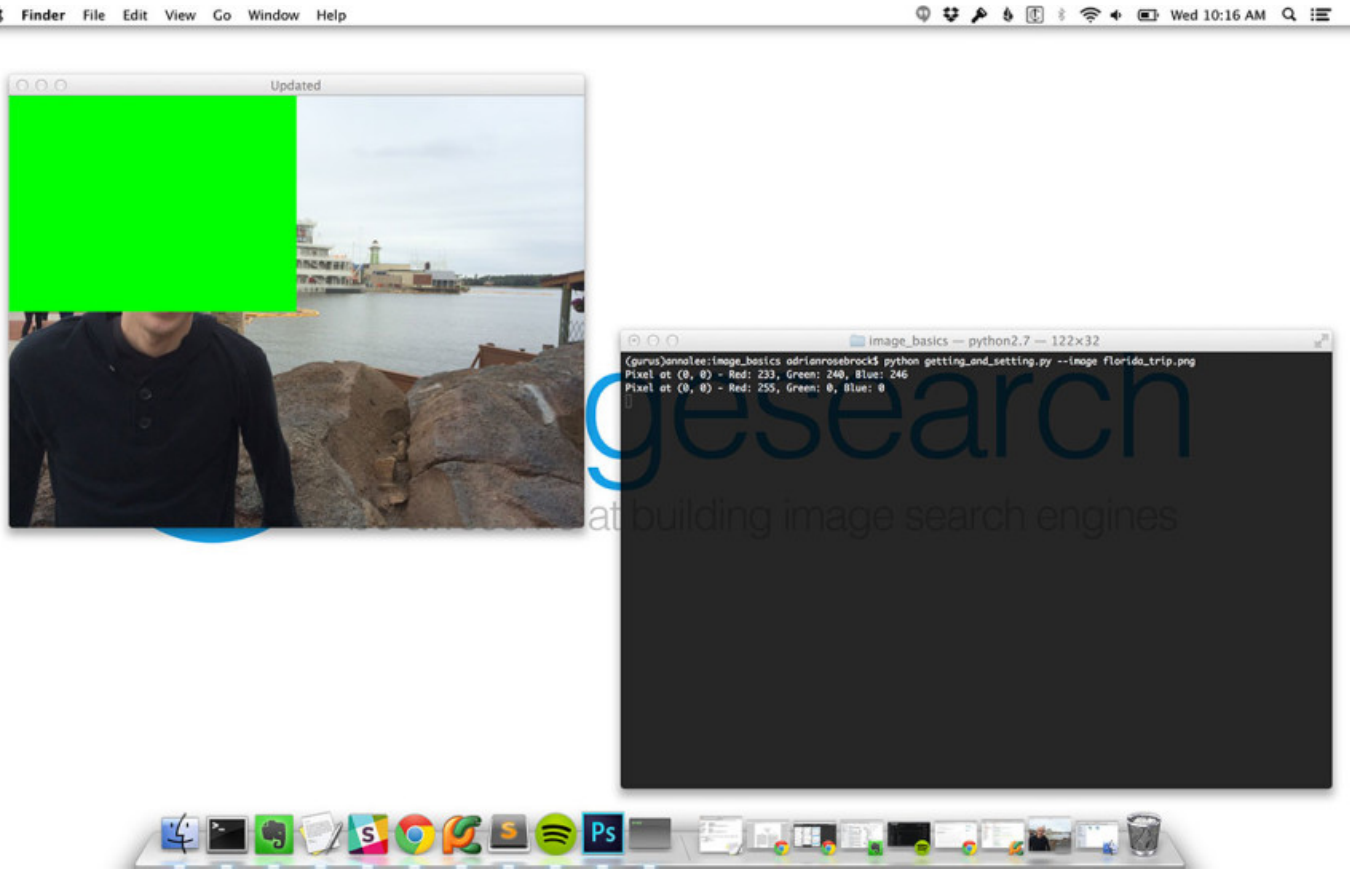
```
getting_and_setting.py Python
```


Feedback

```
42 # now, let's make the top-left corner of the original image green
43 image[0:cY, 0:cX] = (0, 255, 0)
44
45 # Show our updated image
46 cv2.imshow("Updated", image)
47 cv2.waitKey(0)
```

On **Line 43**, you can see that we are again accessing the top-left corner of the image; however, this time we are setting this region to have a value of `(0, 255, 0)` (green).

Lines 46 and 47 then show us the results of our work:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/getting_and_setting_green.jpg)

FIGURE 9: SETTING THE TOP-LEFT CORNER OF THE IMAGE TO BE A GREEN COLOR.

So now that we are done coding, how do we run our Python script?

Assuming you have downloaded the source code for this section (available at the bottom of this article), simply navigate to your downloaded source code directory and execute the command below:

```
getting_and_setting.py                                                Shell
1 $ python getting_and_setting.py --image florida_trip.png
```

Once our script starts running, you should see some output printed to your console (**Line 17**). The first

Feedback

image output tells us that the pixel located at *(0, 0)* has a value of R=233, G=240, and B=246. The buckets for all three channels are nearly white, indicating that the pixel is very bright.

The second line of output shows us that we have successfully changed the pixel located at *(0, 0)* to be red rather than white (**Lines 20-22**).

```
getting_and_setting.py                                                    Shell
1 Pixel at (0, 0) - Red: 233, Green: 240, Blue: 246
2 Pixel at (0, 0) - Red: 255, Green: 0, Blue: 0
```

From there, your output should match the screenshots provided earlier in this article.

Summary

In this section we have explored how to access and manipulate the pixels in an image using NumPy's built-in array slicing functionality. We were even able to draw a green square using nothing but NumPy array manipulation!

However, we won't get very far using only NumPy functions. The next section will show you how to draw lines, rectangles, and circles using OpenCV methods.

Feedback

Downloads:

Download the Code (https://gurus.pyimagesearch.com/protected/code/computer_vision_basics/image_basics.zip)

Quizzes		Status
1	Image Basics Quiz (https://gurus.pyimagesearch.com/quizzes/image-basics-quiz/)	

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/loading-displaying-and-saving-images/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/drawing/)

Course Progress

Ready to continue the course?

Click the button below to **continue your journey to computer vision guru**.

I'm ready, let's go! (/pyimagesearch-gurus-course/)

Resources & Links

- PyImageSearch Gurus Community (https://community.pyimagesearch.com/)
- PyImageSearch Virtual Machine (https://gurus.pyimagesearch.com/pyimagesearch-virtual-machine/)
- Setting up your own Python + OpenCV environment (https://gurus.pyimagesearch.com/setting-up-your-python-opencv-development-environment/)
- Course Syllabus & Content Release Schedule (https://gurus.pyimagesearch.com/course-syllabus-content-release-schedule/)
- Member Perks & Discounts (https://gurus.pyimagesearch.com/pyimagesearch-gurus-discounts-perks/)
- Your Achievements (https://gurus.pyimagesearch.com/achievements/)
- Official OpenCV documentation (http://docs.opencv.org/index.html)

Your Account

- Account Info (https://gurus.pyimagesearch.com/account/)
- Support (https://gurus.pyimagesearch.com/contact/)
- Logout (https://gurus.pyimagesearch.com/wp-login.php?action=logout&redirect_to=https%3A%2F%2Fgurus.pyimagesearch.com%2F&_wpnonce=5736b21cae)

Search

Feedback

© 2018 PyImageSearch. All Rights Reserved.