Practical Python and OpenCV: An Introductory, Example Driven Guide to Image Processing and Computer Vision

Adrian Rosebrock

1

INTRODUCTION

The goal of computer vision is to understand the story unfolding in a picture. As humans, this is quite simple. But for computers, the task is extremely difficult.

So why bother learning computer vision?

Well, images are everywhere!

Whether it be personal photo albums on your smartphone, public photos on Facebook, or videos on YouTube, we now have more images than ever – and we need methods to analyze, categorize, and quantify the contents of these images.

For example, have you recently tagged a photo of yourself or a friend on Facebook lately? How does Facebook seem to "know" where the faces are in an image?

Facebook has implemented facial recognition algorithms into their website, meaning that they can not only *find* faces in an image, but they can also *identify* whose face it is as well! Facial recognition is an application of computer vision in the real-world.

What other types of useful applications of computer vision are there?

Well, we could build representations of our 3D world using public image repositories like Flickr. We could download thousands and thousands of pictures of Manhattan, taken by citizens with their smartphones and cameras, and then analyze them and organize them to construct a 3D representation of the city. We would then virtually navigate this city through our computers. Sound cool?

Another popular application of computer vision is surveillance.

While surveillance tends to have a negative connotation of sorts, there are many different types of surveillance. One type of surveillance is related to analyzing security videos, looking for possible suspects after a robbery.

But a different type of surveillance can be seen in the retail world. Department stores can use calibrated cameras to track how you walk through their stores and which kiosks you stop at.

On your last visit to your favorite clothing retailer, did you stop to examine the spring's latest jean trends? How long did you look at the jeans? What was your facial expression as you looked at the jeans? Did you then pickup a pair and head to the dressing room? These are all types of questions that computer vision surveillance systems can answer.

Computer vision can also be applied to the medical field. A year ago, I consulted with the National Cancer Institute to develop methods to automatically analyze breast histology images for cancer risk factors. Normally, a task like this would require a trained pathologist with years of experience – and it would be extremely time consuming!

Our research demonstrated that computer vision algorithms could be applied to these images and automatically analyze and quantify cellular structures – without human intervention! Now that we can analyze breast histology images for cancer risk factors much faster.

Of course, computer vision can also be applied to other areas of the medical field. Analyzing X-Rays, MRI scans, and cellular structures all can be performed using computer vision algorithms.

Perhaps the biggest success computer vision success story you may have heard of is the X-Box 360 Kinect. The Kinect can use a stereo camera to understand the depth of an image, allowing it to classify and recognize human poses, with the help of some machine learning, of course.

The list doesn't stop there.

Computer vision is now prevalent in many areas of your life, whether you realize it or not. We apply computer vision algorithms to analyze movies, football games, hand gesture recognition (for sign language), license plates (just in case you were driving too fast), medicine, surgery, military, and retail.

We even use computer visions in space! NASA's Mars Rover includes capabilities to model the terrain of the planet, detect obstacles in its path, and stitch together panorama images.

This list will continue to grow in the coming years.

Certainly, computer vision is an exciting field with endless possibilities.

With this in mind, ask yourself, what does your imagination want to build? Let it run wild. And let the computer vision techniques introduced in this book help you build it.

4

IMAGE BASICS

In this chapter 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.

4.1 SO, WHAT'S A PIXEL?

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

Normally, we think of a pixel as 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.

For example, let's pretend we have an image with a resolution of 500×300. This means that our image is represented as a grid of pixels, with 500 rows and 300 columns. Overall, there are $500 \times 300 = 150,000$ 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 lighter.

Color pixels are normally represented in the RGB color space – one value for the Red component, one for Green, and one for Blue. Other color spaces exist, but let's start with the basics and move our way up from there.

Each of the three colors are represented by an integer in the range 0 to 255, which 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)`.

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

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?

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.

4.2 OVERVIEW OF THE COORDINATE SYSTEM

As I mentioned 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 4.1 to make this point more clear.

Here we have the letter "I" on a piece of graph paper. We see that we have an 8×8 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.

Finally, the point $(3, 4)$ is the pixel three columns to the right, and four rows down, once again keeping in mind that we start counting from *zero* rather than *one*.

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 in mind and you'll avoid a lot of confusion later on.

4.3 ACCESSING AND MANIPULATING PIXELS

Admittedly, the example from Chapter 3 wasn't very exciting. All we did was load an image off disk, display it, and

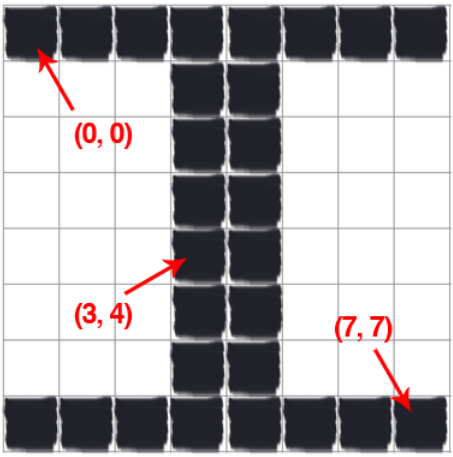

Figure 4.1: The letter "I" placed on a piece of graph paper. Pixels are accessed by their (x, y) coordinates, where we go x columns to the right and y rows down, keeping in mind that Python is zero-indexed: we start counting from zero rather than one.

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:

Listing 4.1: getting_and_setting.py

```
1 import argparse
2 import cv2
3
4 ap = argparse.ArgumentParser()
5 ap.add_argument("-i", "--image", required = True,
6     help = "Path to the image")
7 args = vars(ap.parse_args())
8
9 image = cv2.imread(args["image"])
10 cv2.imshow("Original", image)
```

Similar to our example in the previous chapter, **Lines 1-7** handle importing the packages we need along with setting up our argument parser. There is only one command line argument needed: the path to the image we are going to work with.

Lines 9 and 10 handle loading the actual image off of disk and displaying it to us.

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 Section 4.1 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. This is **important to note** since it could cause some confusion later.

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

Listing 4.2: getting_and_setting.py

```
11 (b, g, r) = image[0, 0]
12 print "Pixel at (0, 0) - Red: %d, Green: %d, Blue: %d" % (r, g, b
       )
13
14 image[0, 0] = (0, 0, 255)
15 (b, g, r) = image[0, 0]
16 print "Pixel at (0, 0) - Red: %d, Green: %d, Blue: %d" % (r, g, b
       )
```

On **Line 11**, 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 12** 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 14** 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 14**, we then grab the pixel value and print it back to console on **Lines 15 and 16**, just to demonstrate that we have indeed successfully changed the color of the pixel.

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:

Listing 4.3: getting_and_setting.py

```
17 corner = image[0:100, 0:100]
18 cv2.imshow("Corner", corner)
19
20 image[0:100, 0:100] = (0, 255, 0)
21
22 cv2.imshow("Updated", image)
23 cv2.waitKey(0)
```

On **line 17** we grab a 100×100 pixel region of the image. In fact, this is the top-left corner of the image! In order to grab chunks of an image, NumPy expects we provide four

indexes:

1. **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 $y = 0$.
2. **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 $y = 100$.
3. **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 $x = 0$.
4. **End x:** Lastly, we need to provide a x-axis value for our slice to stop. We stop when $x = 100$.

Once we have extracted the top-left corner of the image, **Line 18** shows us the result of the cropping. Notice how our image is just the 100×100 pixel region from the top-left corner of our original image.

The last thing we are going to do is use array slices to change the color of a region of pixels. On **Line 20**, 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 22 and 23 then show us the results of our work.

So how do we run our Python script?

Assuming you have downloaded the source code listings for this book, simply navigate to the `chapter-04` directory

and execute the command below:

Listing 4.4: getting_and_setting.py

```
$ python getting_and_setting.py --image ../images/trex.png
```

Once our script starts running, you should see some output printed to your console (**Line 12**). The first line of output tells us that the pixel located at $(0, 0)$ has a value of 254 for all three red, green, and blue channels. This pixel appears to be almost pure white.

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 14-16**).

Listing 4.5: getting_and_setting.py

```
Pixel at (0, 0) - Red: 254, Green: 254, Blue: 254
Pixel at (0, 0) - Red: 255, Green: 0, Blue: 0
```

We can see the results of our work in Figure 4.2. The *Top-Left* image is our original image we loaded off disk. The image on the *Top-Right* is the result of our array slicing and cropping out a 100×100 pixel region of the image. And, if you look closely, you can see that the top-left pixel located at $(0, 0)$ is red!

Finally, the *bottom* image shows that we have successfully drawn a green square on our image.

In this chapter 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

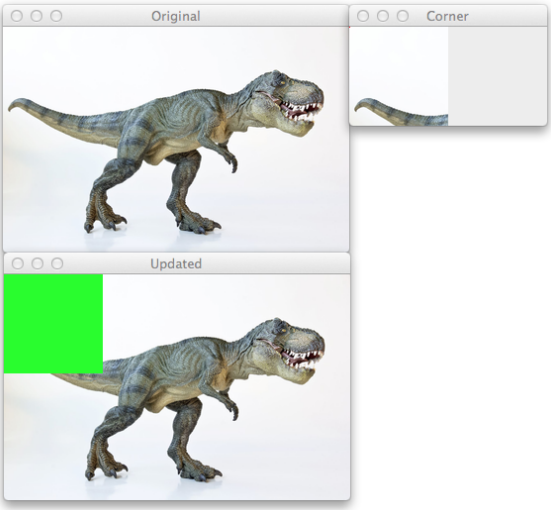

Figure 4.2: *Top-Left*: Our original image. *Top-Right*: Cropping our image using NumPy array slicing. *Bottom*: Drawing a 100×100 pixel green square on our image by using basic NumPy indexing.

square using nothing but NumPy array manipulation!

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

6

IMAGE PROCESSING

Now that you have a solid foundation to build upon, we can start to explore simple image processing techniques.

First, we'll start off with basic image transformations, such as translation, rotation, resizing, flipping, and cropping. Then, we'll explore other types of image processing techniques, including image arithmetic, bitwise operations, and masking.

Finally, we'll explore how to split an image into its respective channels and then merge them back together again. We'll conclude this chapter with a discussion of different color spaces that OpenCV supports and the benefits and limitations of each of them.

6.1 IMAGE TRANSFORMATIONS

In this section we'll cover basic image transformations. These are common techniques that you'll likely apply to images, including translation, rotation, resizing, flipping and cropping. We'll explore each of these techniques in detail.

Make sure you have a good grasp of these methods! They are important in nearly all areas of computer vision.

6.1.1 *Translation*

The first method we are going to explore is translation. Translation is the shifting of an image along the x and y axis. Using translation, we can shift an image up, down, left, or right, along with any combination of the above!

This concept is better explained through some code:

Listing 6.1: translation.py

```
1 import numpy as np
2 import argparse
3 import imutils
4 import cv2
5
6 ap = argparse.ArgumentParser()
7 ap.add_argument("-i", "--image", required = True,
8     help = "Path to the image")
9 args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 M = np.float32([[1, 0, 25], [0, 1, 50]])
15 shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape
       [0]))
16 cv2.imshow("Shifted Down and Right", shifted)
17
18 M = np.float32([[1, 0, -50], [0, 1, -90]])
19 shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape
       [0]))
20 cv2.imshow("Shifted Up and Left", shifted)
```

On **Lines 1-4** we simply import the packages we will make use of. At this point, using `numpy`, `argparse`, and `cv2`

should feel like commonplace. However, I am introducing a new package here: `imutils`. This isn't a package included in NumPy or OpenCV. Rather, it's a library that we are going to write ourselves and create "convenience" methods to do common tasks like translation, rotation, and resizing.

After we have the necessary packages imported, we construct our argument parser and load our image on **Lines 6-12**.

The actual translation takes place on **Lines 14-16**. We first define our translation matrix M. This matrix tells us how many pixels to the left or right our image will shifted, and then how many pixels up or down the image will be shifted.

Our translation matrix M is defined as a floating point array – this is important because OpenCV expects this matrix to be of floating point type. The first row of the matrix is $[1, 0, t_x]$, where t_x is the number of pixels we will shift the image left or right. Negative values of t_x will shift the image to the left and positive values will shift the image to the right.

Then, we define the second row of the matrix as $[0, 1, t_y]$, where t_y is the number of pixels we will shift the image up or down. Negative value of t_y will shift the image up and positive values will shift the image down.

Using this notation, we can see on **Line 14** that $t_x = 25$ and $t_y = 50$, implying that we are shifting the image 25 pixels to the right and 50 pixels down.

Now that we have our translation matrix defined, the actual translation takes place on **Line 15** using the `cv2.warpAffine` function. The first argument is the image we wish to shift and the second argument is our translation matrix M. Finally, we manually supply the dimensions (width and height) of our image as the third argument. **Line 16** shows the results of the translation.

Moving on to **Lines 18-20**, we perform another translation. Here, we set $t_x = -50$ and $t_y = -90$, implying that we are shifting the image 50 pixels to the left and 90 pixels up. The image is shifted *left* and *up* rather than *right* and *down* because we are providing a negative values for both t_x and t_y.

However, manually constructing this translation matrix and calling the `cv2.warpAffine` method takes a fair amount of code – and it's not pretty code either!

Let's create a new file: `imutils.py`. This file will store basic image processing methods, allowing us to conveniently call them without writing a lot of code.

The first method we are going to define is a `translate` function:

Listing 6.2: imutils.py

```
1 import numpy as np
2 import cv2
3
4 def translate(image, x, y):
5     M = np.float32([[1, 0, x], [0, 1, y]])
6     shifted = cv2.warpAffine(image, M, (image.shape[1], image.
          shape[0]))
```

```
7
8     return shifted
```

Our `translate` method takes three parameters: the `image` we are going to translate, the number pixels that we are going to shift along the x-axis, and the number of pixels we are going to shift along the y-axis.

This method then defines our translation matrix M on **Line 5** and then applies the actual shift on **Line 6**. Finally, we return the shifted image on **Line 8**.

Let's apply our translate method and compare to the methods discussed above:

Listing 6.3: translation.py

```
21 shifted = imutils.translate(image, 0, 100)
22 cv2.imshow("Shifted Down", shifted)
23 cv2.waitKey(0)
```

Using our convenience `translate` method, we are able to shift the image 100 pixels down using a single line of code. Furthermore, this `translate` method is much easier to use – less code is required and based on the function name, we conveniently know what image processing task is being performed.

To see our translation in action, take a look at Figure 6.1. Our original image is on the *top-left*. On the *top-right*, we shift our image 25 pixels to the right and 50 pixels down. Next, we translate our image 50 pixels to the left and 90 pixels up by using negative values for t_x and t_y. Finally, on the *bottom-right* we shift our T-Rex 100 pixels down using

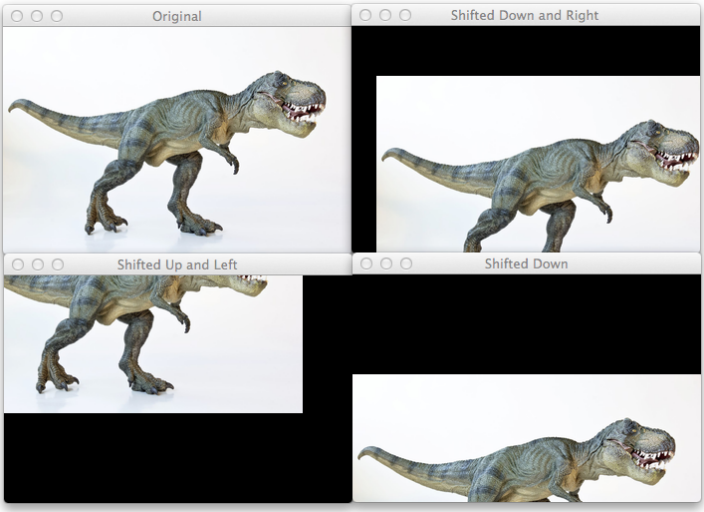

Figure 6.1: *Top-Left:* Our original T-rex image. *Top-Right:* Translating our image 25 pixels to the right and 50 pixels down. *Bottom-Left:* Shifting T-Rex 50 pixels to the left and 90 pixels up. *Bottom-Right:* Shifting the T-Rex down using our convenience method.

our convenient `translate` method defined above.

In this section we explored how to shift an image up, down, left, and right. Next up, we'll explore how to rotate an image.

6.1.2 *Rotation*

Rotation is exactly what it sounds like: rotating an image by some angle θ. In this section, we'll explore how to rotate an image. We'll use θ to represent by how many degrees we are rotating the image. Later, I'll provide another convenience method, `rotate` to make performing rotations on images easier.

Listing 6.4: rotate.py

```
1  import numpy as np
2  import argparse
3  import imutils
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 (h, w) = image.shape[:2]
15 center = (w / 2, h / 2)
16
17 M = cv2.getRotationMatrix2D(center, 45, 1.0)
18 rotated = cv2.warpAffine(image, M, (w, h))
19 cv2.imshow("Rotated by 45 Degrees", rotated)
20
21 M = cv2.getRotationMatrix2D(center, -90, 1.0)
```

```
22 rotated = cv2.warpAffine(image, M, (w, h))
23 cv2.imshow("Rotated by -90 Degrees", rotated)
```

Lines 1-4 again import the packages we need. You should take note of `imutils`. Once again we will be defining a convenience method to make our lives easier.

Lines 6-12 construct our argument parser. We only need one argument: the path to the image we are going to use. We then load our image off disk and display it.

When we rotate an image, we need to specify which point we want to rotate about. In most cases, you will want to rotate around the *center* of an image; however, OpenCV allows you to specify any arbitrary point you want to rotate around. Let's just go ahead and rotate about the center of the image. **Lines 14 and 15** grabs the width and height of the image, then divides each by 2 to determine the center of the image.

Just as we defined a matrix to translate an image, we also define a matrix to rotate the image. Instead of manually constructing the matrix using NumPy, we'll just make a call to the `cv2.getRotationMatrix2D` method on **Line 17**.

The `cv2.getRotationMatrix2D` function takes three arguments: the point in which we want to rotate the image about (in this case, the `center` of the image). We then specify θ, the number of degrees we are going to rotate the image by. In this case, we are going to rotate the image 45 degrees. The last argument is the scale of the image. We haven't discussed resizing an image yet, but here you can specify a floating point value, where 1.0 means the same dimensions of the image are used. However, if you specified

a value of 2.0 the image would be doubled in size. Similarly, a value of 0.5 halve the size of the image.

Once we have our rotation matrix M from the `cv2.getRotationMatrix2D` function, we can apply the rotation to our image using the `cv2.warpAffine` method on **Line 18**. The first argument to this function is the image we want to rotate. We then specify our rotation matrix M along with the output dimensions (width and height) of our image. **Line 19** then shows our image rotated by 45 degrees. Check out Figure 6.2 *Top-Right* to see our rotated image.

Let's not waste anytime. We'll go ahead and jump into some code to perform rotations:

On **Lines 21-23** we perform another rotation. The code is identical to that **Lines 17-19**, only this time we are rotating by -90 degrees rather than 45. Figure 6.2 *Bottom-Left* shows our T-Rex rotated by -90 degrees.

Just as in translating an image, the code to rotate an image isn't the most pretty and Pythonic. Let's change that and define our own custom `rotate` method:

Listing 6.5: imutils.py

```
27 def rotate(image, angle, center = None, scale = 1.0):
28     (h, w) = image.shape[:2]
29
30     if center is None:
31         center = (w / 2, h / 2)
32
33     M = cv2.getRotationMatrix2D(center, angle, scale)
34     rotated = cv2.warpAffine(image, M, (w, h))
35
36     return rotated
```

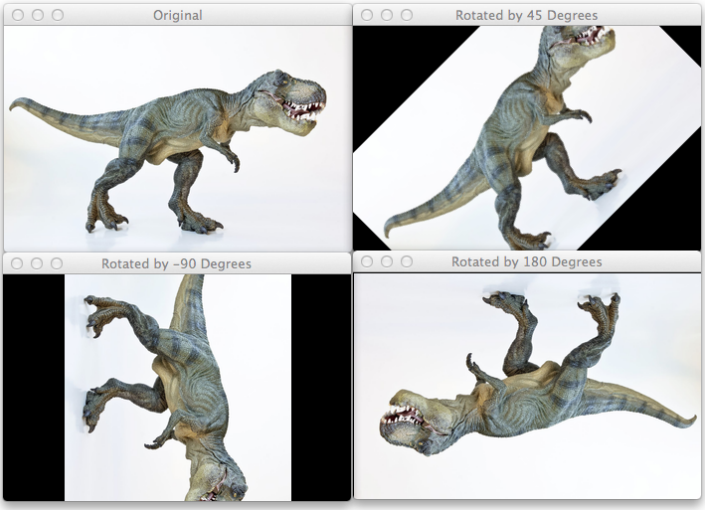

Figure 6.2: *Top-Left:* Our original T-Rex image. *Top-Right:* Rotating the image by 45 degrees. *Bottom-Left:* Rotating the image by -90 degrees. *Bottom-Right*: Flipping T-Rex upside-down by rotating the image by 180 degrees.

Our `rotate` method takes four arguments. The first is our image. The second is the angle θ in which we want to rotate the image. We provide two optional keyword arguments, `center` and `scale`. The `center` parameter is the point in which we wish to rotate our image about. If a value of `None` is provided, the method automatically determines the center of the image on **Lines 30-31**. Finally, the `scale` parameter is used to handle if the size of the image should be changed during the rotation. The `scale` parameter has a default value of 1.0, implying that no resizing should be done.

The actual rotation of the image takes place on **Lines 33 and 34**, were we construct our rotation matrix M and apply it to the image. Finally, our image is returned on **Line 36**.

Now that we have defined our `rotate` method, let's apply it:

Listing 6.6: rotate.py

```
24 rotated = imutils.rotate(image, 180)
25 cv2.imshow("Rotated by 180 Degrees", rotated)
26 cv2.waitKey(0)
```

Here we are rotating our image by 180 degrees. Figure 6.2 *Bottom-Right* shows that our T-Rex has indeed been flipped upside-down. The code for our rotate method is much easier to read and maintain that making calls to `cv2.getRotationMatrix2D` and `cv2.warpAffine` each time we want to rotate an image.

6.1.3 *Resizing*

So far we've covered two image transformations: translation and rotation. Now, we are going to explore how to resize an image. We'll also define one last method for our `imutils.py` file, a convenience method to help us resize images with ease.

Perhaps, not surprisingly, we will be using the `cv2.resize` function to resize our images. But we need to keep in mind the aspect ratio of the image when we are using this function. But before we get too deep into the details, let's jump right into an example:

Listing 6.7: resize.py

```
1  import numpy as np
2  import argparse
3  import imutils
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 r = 150.0 / image.shape[1]
15 dim = (150, int(image.shape[0] * r))
16
17 resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
18 cv2.imshow("Resized (Width)", resized)
```

Lines 1-12 should start to feel quite redundant at this point. We are importing our packages, setting up our argument parser, and finally loading our image and displaying

it.

The actual interesting code doesn't start until **Lines 14 and 15**. When resizing an image, we need to keep in mind the aspect ratio of the image. The aspect ratio is the proportional relationship of the width and the height of the image. If we aren't mindful of the aspect ratio, our resizing will return results that don't look correct.

Computing the aspect ratio is handled on **Line 14**. In this line of code we define our new image width to be 150 pixels. In order to compute the ratio of the new height to the old height, we simply define our ratio `r` to be the new width (150 pixels) divided by the old width, which we access using `image.shape[1]`.

Now that we have our ratio, we can compute the new dimensions of the image on **Line 15**. Again, the width of the new image will be 150 pixels. The height is then computed by multiplying the old height by our ratio and converting it to an integer.

The actual resizing of the image takes place on **Line 17**. The first argument is the image we wish to resize and the second is our computed dimensions for the new image. Last parameter is our `interpolation` method, which is the algorithm working behind the scenes to handle how the actual image is resized. In general, I find that using `cv2.INTER_AREA` obtains the best results when resizing; however, other appropriate choices include `cv2.INTER_LINEAR`, `cv2.INTER_CUBIC`, and `cv2.INTER_NEAREST`.

Finally, we show our resized image on **Line 18**

In the example we just explored, we only resized the image by specifying the width. But what if we wanted to resize the image by specifying the height? All that requires is a change to computing the aspect ratio:

Listing 6.8: resize.py

```
19 r = 50.0 / image.shape[0]
20 dim = (int(image.shape[1] * r), 50)
21
22 resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
23 cv2.imshow("Resized (Height)", resized)
24 cv2.waitKey(0)
```

On **Line 19** we define our ratio `r`. Our new image will have a height of 50 pixels. To determine the ratio of the new height to the old height, we divide 50 by the old height.

Then, we define the dimensions of our new image. We already know that the new image will have a height of 50 pixels. The new width is obtained by multiplying the old width by the ratio.

We then perform the actual resizing of the image on **Line 22** and show it on **Line 23**.

Resizing an image is simple enough, but having to compute the aspect ratio, define the dimensions of the new image, and then perform the resizing takes three lines of code. This looks like the perfect time to define a `resize` method in our `imutils.py` file:

Listing 6.9: resize.py

```
25 resized = imutils.resize(image, width = 100)
```

```
26 cv2.imshow("Resized via Function", resized)
27 cv2.waitKey(0)
```

In this example you can see that the resizing of the image is handled by a single function: `imutils.resize`. The first argument we pass in is our image we want to resize. Then, we specify the keyword argument `width`, which is the width of our new image. The function then handles the resizing for us.

Of course, we could also resize via the height of the image by changing the function call to:

Listing 6.10: resize.py

```
1     resized = imutils.resize(image, height = 50)
```

Let's take this function apart and see what's going on under the hood:

Listing 6.11: imutils.py

```
9  def resize(image, width = None, height = None, inter = cv2.
       INTER_AREA):
10     dim = None
11     (h, w) = image.shape[:2]
12
13     if width is None and height is None:
14         return image
15
16     if width is None:
17         r = height / float(h)
18         dim = (int(w * r), height)
19
20     else:
21         r = width / float(w)
22         dim = (width, int(h * r))
23
24     resized = cv2.resize(image, dim, interpolation = inter)
25
```

```
26     return resized
```

As you can see, we have defined our `resize` function. The first argument is the image we want to resize. Then, we define two keyword arguments, `width` and `height`. Both of these arguments cannot be `None`, otherwise we won't know how to resize the image. We also provide `inter`, which is our interpolation method and defaults to `cv2.INTER_AREA`.

On **Lines 10 and 11** we define the dimensions of our new, resized image and grab the dimensions of the original image.

We perform a quick check on **Lines 13-14** to ensure that a numerical value has been provided for either the width or the height.

The computation of the ratio and new, resized image dimensions are handled on **Lines 16-22**, depending on whether we are resizing via width or via height.

Line 24 handles the actual resizing of the image, then **Line 26** returns our resized image to the user.

To see the results of our image resizings, check out Figure 6.3. On the *Top-Left* we have our original T-Rex image. Then, on the *Top-Right* we have our T-Rex resized to have a width of 150 pixels. The *Middle-Right* image then shows our image resized to have a height of 50 pixels. Finally, *Bottom-Right* shows the output of our `resize` function – the T-Rex is now resized to have a width of 100 pixels using only a single line of code.

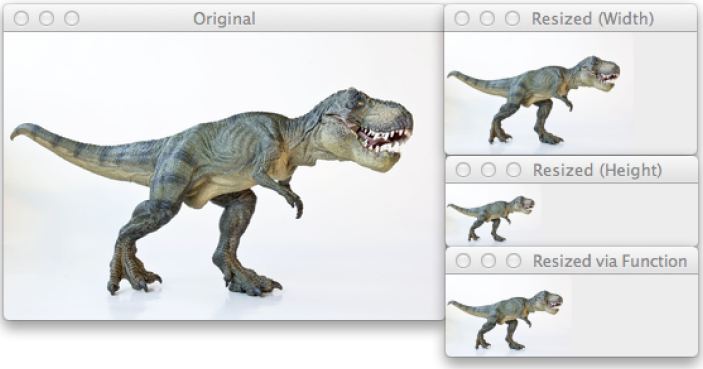

Figure 6.3: *Top-Left:* Our original T-Rex image. *Top-Right:* The T-Rex resized to have a width of 150 pixels. *Middle-Right:* Our image resized to have a height of 50 pixels. *Bottom-Right:* Resizing our image to have a width of 100 pixels using our helper function. In all cases, the aspect ratio of the image is maintained.

Translation, rotation, and resizing are certainly the more challenging and involved image transformation tasks. The next two we will explore, flipping and cropping, are substantially easier.

6.1.4 *Flipping*

Next up on our image transformations to explore is flipping an image. We can flip an image around either the x or y axis, or even both.

In fact, I think explaining how to flip an image is better explained by viewing the output of an image flip, before we get into the code. Check out Figure 6.4 to see our T-Rex image flipped horizontally, vertically, and both horizontally and vertically at the same time.

Now that you see what an image flip looks like, we can explore the code:

Listing 6.12: flipping.py

```
1 import argparse
2 import cv2
3
4 ap = argparse.ArgumentParser()
5 ap.add_argument("-i", "--image", required = True,
6     help = "Path to the image")
7 args = vars(ap.parse_args())
8
9 image = cv2.imread(args["image"])
10 cv2.imshow("Original", image)
11
12 flipped = cv2.flip(image, 1)
13 cv2.imshow("Flipped Horizontally", flipped)
14
```

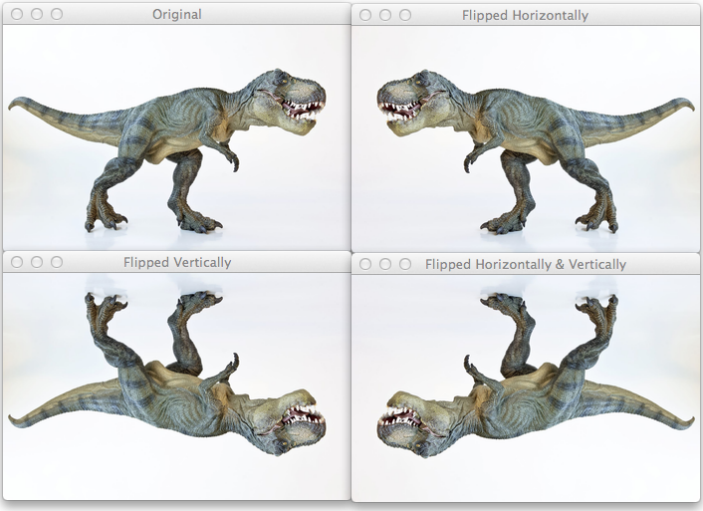

Figure 6.4: *Top-Left:* Our original T-Rex image. *Top-Right:* Flipping the T-Rex image horizontally. *Bottom-Left:* Flipping the T-Rex vertically. *Bottom-Right*: Flipping the image both horizontally and vertically.

```
15 flipped = cv2.flip(image, 0)
16 cv2.imshow("Flipped Vertically", flipped)
17
18 flipped = cv2.flip(image, -1)
19 cv2.imshow("Flipped Horizontally & Vertically", flipped)
20 cv2.waitKey(0)
```

lines 1-10 handle our standard procedure of importing our packages, parsing arguments, and loading our image from disk.

Flipping an image is accomplished by making a call to the `cv2.flip` function on **Line 12**. The `cv2.flip` method requires two arguments: the image we want to flip and a flip code that is used to determine how we are going to flip the image.

Using a flip code value of 1 indicates that we are going to flip the image horizontally, around the y-axis (**Line 12**). Specifying a flip code of 0 indicates that we want to flip the image vertically, around the x-axis (**Line 15**). Finally, using a negative flip code (**Line 18**) flips the image around both axes.

Again, to see the output of our flipping example, take a look at Figure 6.4. Here we can see the image flipped horizontally, vertically, and around both axes.

Flipping an image is very simple, perhaps one of the most simple examples in this book! Next up, we'll go over cropping an image and how to extract regions of an image using NumPy array slices.

Figure 6.5: *Top:* Our original T-Rex image. *Bottom:* Cropping the face of the T-Rex using NumPy array slices.

6.1.5 *Cropping*

When we crop an image, we want to remove the outer parts of the image that we are not interested in. We can accomplish image cropping by using NumPy array slicing. In fact, we already performed image cropping in Chapter 4!

However, let's review it again and make sure we understand what is going on:

Listing 6.13: crop.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
```

```
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 cropped = image[30:120 , 240:335]
14 cv2.imshow("T-Rex Face", cropped)
15 cv2.waitKey(0)
```

Lines 1-11 handle importing our packages, parsing our arguments, and loading our images. For our cropping example, we will use our T-Rex image.

The actual cropping takes place on a single line of code: **Line 13**. We are supply NumPy array slices to extract a rectangular region of the image, starting at $(240, 30)$ and ending at $(335, 120)$. The order in which we supply the indexes to the crop may seem counterintuitive; however, remember that OpenCV represents images as NumPy arrays with the the height first and the width second. This means that we need to supply our y-axis values before our x-axis.

In order to perform our cropping, NumPy expects four indexes:

1. **Start y:** The starting y coordinate. In this case, we start at $y = 30$.
2. **End y:** The ending y coordinate. We will end our crop at $y = 120$.
3. **Start x:** The starting x coordinate of the slice. We start the crop at $x = 240$.

4. **End x:** The ending x-axis coordinate of the slice. Our slice ends at $x = 335$.

Executing our code detailed above, we will see from Figure 6.5 that we have cropped out the face of our T-Rex! While the T-Rex might seem a little scary, cropping sure isn't! In fact, it's quite simple when you consider all we are doing is performing array slices on NumPy arrays!

6.2 IMAGE ARITHMETIC

We all know basic arithmetic operations like addition and subtraction. But when working with images, we need to keep in mind the limits of our color space and data type.

For example, RGB images have pixels that fall within the range $[0, 255]$. What happens if we are examining a pixel with intensity 250 and we try to add 10 to it?

Under normal arithmetic rules, we would end up with a value of 260. However, since RGB images are represented as 8-bit unsigned integers, 260 is not a valid value.

So what should happen? Should we perform a check of some sorts to ensure no pixel falls outside the range of $[0, 255]$, thus clipping all pixels to have a minimum value of 0 and a maximum value of 255?

Or do we apply a modulus operation, and "wrap around"? Under modulus rules, adding 10 to 250 would simply wrap around to a value of 4.

Which way is the "correct" way to handle images additions and subtractions that fall outside the range of $[0, 255]$?

The answer is there is no correct way – it simply depends on how you manipulating your pixels and what you want the desired results to be.

However, be sure to keep in mind that there is a difference between OpenCV and NumPy addition. NumPy will perform modulus arithmetic and "wrap around". OpenCV on the other hand will perform clipping and ensure pixel values never fall outside the range $[0, 255]$.

But don't worry! These nuances will become more clear as we explore some code below.

Listing 6.14: arithmetic.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 print "max of 255: " + str(cv2.add(np.uint8([200]), np.uint8
       ([100])))
14 print "min of 0: " + str(cv2.subtract(np.uint8([50]), np.uint8
       ([100])))
15
16 print "wrap around: " + str(np.uint8([200]) + np.uint8([100]))
17 print "wrap around: " + str(np.uint8([50]) - np.uint8([100]))
```

We are going to perform our standard procedure on **Lines 1-11** by importing our packages, setting up our argument parser, and loading our image.

Remember how I mentioned the difference between OpenCV and NumPy addition above? Well now we are going to explore it further and provide a concrete example to ensure we fully understand it.

On **Line 13** we define two NumPy arrays that are 8-bit unsigned integers. The first array has one element: a value of 200. The second array also has only one element, but a value of 100. We then use OpenCV's `cv2.add` method to add the values together.

What do you think the output is going to be?

Well according, to standard arithmetic rules, we would think the result should be 300, *but*, remember that we are working with 8-bit unsigned integers that only have a range between $[0, 255]$. Since we are using the `cv2.add` method, OpenCV takes care of clipping for us, and ensures that the addition produces a maximum value of 255. When we execute this code, we can see the result on the first line of Listing 6.15. Sure enough, the addition returned a value of 255.

Line 14 then performs subtraction using `cv2.subtract`. Again, we define two NumPy arrays, each with a single element, and of the 8-bit unsigned integer data type. The first array has a value of 50 and the second a value of 100.

According to our arithmetic rules, the subtraction should return a value of -50; however, OpenCV once again performs clipping for us. We find that the value is clipped to a value of 0. The second line of Listing 6.15 verifies this: subtracting 100 from 50 using `cv2.subtract` returns a value of 0.

Listing 6.15: arithmetic.py

```
max of 255: [[255]]
min of 0: [[0]]
```

But what happens if we use NumPy to perform the arithmetic instead of OpenCV?

Line 16 and 17 explore this question.

First, we define two NumPy arrays, each with a single element, and of the 8-bit unsigned integer data type. The first array has a value of 200 and the second has a value of 100. Using the `cv2.add` function, our addition would be clipped and a value of 255 returned.

However, NumPy does not perform clipping – it instead performs modulo arithmetic and "wraps around". Once a value of 255 is reached, NumPy wraps around to zero, and then starts counting up again, until 100 steps have been reached. You can see this is true via the first line of output on Listing 6.16.

Then, we define two more NumPy arrays: one has a value of 50 and the other 100. Using the `cv2.subtract` method, this subtraction would be clipped to return a value of 0. However, we know that NumPy performs modulo arith-

metic rather than clipping. Instead, once 0 is reached during the subtraction, the modulo operations wraps around and starts counting backwards from 255 – thus the result on the second line of output on Listing 6.16.

Listing 6.16: arithmetic.py

```
wrap around: [44]
wrap around: [206]
```

When performing integer arithmetic it is important to keep in mind your desired output.

Do you want all values to be clipped if they fall outside the range $[0, 255]$? Then use OpenCV's built in methods for image arithmetic.

Do you want modulus arithmetic operations and have values wrap around if they fall outside the range of $[0, 255]$? Then simply add and subtract the NumPy arrays as you normally would.

Now that we have explored the caveats of image arithmetic in OpenCV and NumPy, let's perform the arithmetic on actual images and view the reuslts:

Listing 6.17: arithmetic.py

```
18 M = np.ones(image.shape, dtype = "uint8") * 100
19 added = cv2.add(image, M)
20 cv2.imshow("Added", added)
21
22 M = np.ones(image.shape, dtype = "uint8") * 50
23 subtracted = cv2.subtract(image, M)
24 cv2.imshow("Subtracted", subtracted)
25 cv2.waitKey(0)
```

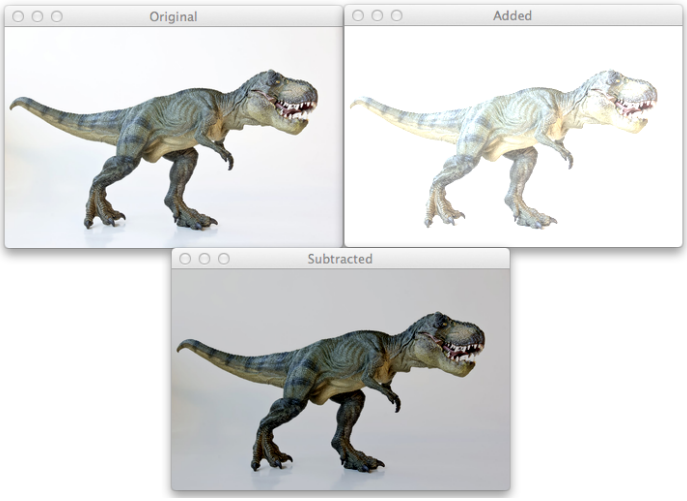

Figure 6.6: *Top-Left:* Our original T-Rex image. *Top-Right:* Adding 100 to every pixel in the image. Notice how the image looks more "washed out" and is substantially brighter than the original. *Bottom*: Subtracting 50 from every pixel in the image. Notice that the image is now darker than the original.

Line 18 defines an NumPy array of ones, with the same size as our image. Again, we are sure to use 8-bit unsigned integers as our data type. In order to fill our matrix with values of 100's rather than 1's, we simply multiply our matrix of 1's by 100. Finally, we use the `cv2.add` function to add our matrix of 100's to the original image – thus increasing every pixel intensity in the image by 100, but ensuring all values are clipped to the range $[0, 255]$ if they attempt to exceed 255.

The result of our operation can be found in Figure 6.6 *Top-Right*. Notice how the image looks more "washed out" and is substantially brighter than the original. This is because we are increasing the pixel intensities by adding 100 to them and pushing them towards brighter colors.

We then create another NumPy array filled with 50's on **Line 23** and use the `cv2.subtract` function to subtract 50 from each pixel intensity of the image. The *Bottom* image in Figure 6.6 shows the results of this subtraction. Our image now looks considerably darker than the original T-Rex. Pixels that were once white now look gray. This is because we are subtracting 50 from the pixels and pushing them towards the darker regions of the RGB color space.

In this section we explored the peculiarities of image arithmetic using OpenCV and NumPy. These caveats are important to keep in mind, otherwise you may get unwanted results when performing arithmetic operations on your images.

6.3 BITWISE OPERATIONS

In this section we will review four bitwise operations: AND, OR, XOR, and NOT. These four operations, while very basic and low level, are paramount to image processing, especially when we start working with masks in Section 6.4.

Bitwise operations operate in a binary manner and are represented as grayscale images. A given pixel is turned "off" if it has a value of zero and it is turned "on" if the pixel has a value greater than zero.

Let's go ahead and jump into some code:

Listing 6.18: bitwise.py

```
1 import numpy as np
2 import cv2
3
4 rectangle = np.zeros((300, 300), dtype = "uint8")
5 cv2.rectangle(rectangle, (25, 25), (275, 275), 255, -1)
6 cv2.imshow("Rectangle", rectangle)
7
8 circle = np.zeros((300, 300), dtype = "uint8")
9 cv2.circle(circle, (150, 150), 150, 255, -1)
10 cv2.imshow("Circle", circle)
```

The first two lines of code import the packages we will need: `numpy` and `cv2`. We initialize our rectangle image as a 300×300 NumPy array on **Line 4**. We then draw a 250×250 white rectangle at the center of the image.

Similarly, on **Line 8** we initialize another image to contain our circle, which we draw on **Line 9**, again centered at the center of the image, with a radius of 150 pixels.

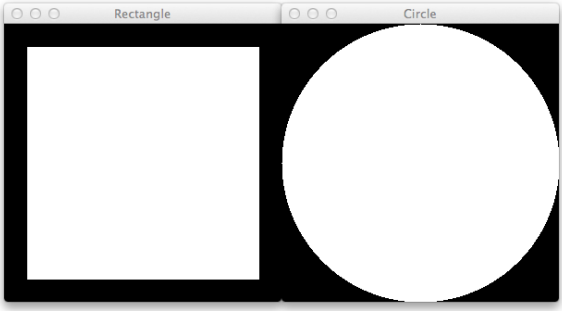

Figure 6.7: *Left*: Our rectangle image. *Right:* Our circle image. We will explore how these two images can be combined using bitwise operations.

Figure 6.7 shows our two shapes. We will make use of these shapes to demonstrate our bitwise operations:

Listing 6.19: bitwise.py

```
11 bitwiseAnd = cv2.bitwise_and(rectangle, circle)
12 cv2.imshow("AND", bitwiseAnd)
13 cv2.waitKey(0)
14
15 bitwiseOr = cv2.bitwise_or(rectangle, circle)
16 cv2.imshow("OR", bitwiseOr)
17 cv2.waitKey(0)
18
19 bitwiseXor = cv2.bitwise_xor(rectangle, circle)
20 cv2.imshow("XOR", bitwiseXor)
21 cv2.waitKey(0)
22
23 bitwiseNot = cv2.bitwise_not(circle)
24 cv2.imshow("NOT", bitwiseNot)
25 cv2.waitKey(0)
```

As I mentioned above, a given pixel is turned "on" if it has a value greater than zero and it is turned "off" if it has a value of zero. Bitwise functions operate on these binary conditions.

In order to utilize bitwise functions we assume (in most cases) that we are comparing two pixels (the only exception is the NOT function). We'll compare each of the pixels and then construct our bitwise representation.

Let's quickly review our binary operations:

1. **AND:** A bitwise AND is true if and only if both pixels are greater than zero.
2. **OR:** A bitwise OR is true if either of the two pixels are greater than zero.
3. **XOR:** A bitwise XOR is true if and only if *either* of the two pixels are greater than zero, but not both.
4. **NOT:** A bitwise NOT inverts the "on" and "off" pixels in an image.

On **Line 11** we apply a bitwise AND to our rectangle and circle images using the `cv2.bitwise_and` function. As the list above mentions, a bitwise AND is true if and only if both pixels are greater than zero. The output of our bitwise AND can be seen in Figure 6.8 *Top-Left*. We can see that edges of our square are lost – this makes sense because our rectangle does not cover as large of an area as the circle, and thus both pixels are not "on".

We then apply a bitwise OR on **Line 15** using the `cv2.bitwise_or` function. A bitwise OR is true if either of the two pixels are greater than zero. Figure 6.8 *Top-Right* shows the output of our bitwise OR. In this case, our square and rectangle have been combined together.

Next up is the bitwise XOR function, applied on **Line 19** using the `cv2.bitwise_xor` function. A XOR operation is true if both pixels are greater than zero, *but*, both pixels cannot be greater than zero. The output of the XOR operation is displayed on Figure 6.8 *Bottom-Right*. Here we see that the center of the square has been removed. Again, this makes sense because an XOR operation cannot have both pixels greater than zero.

Finally, we apply the NOT function on **Line 23** using the `cv2.bitwise_not` function. Essentially, the bitwise NOT function flips pixel values. All pixels that are greater than zero are set to zero, and all pixels that are set to zero are set to 255. Figure 6.8 *Bottom-Right* flips our white circle to a black circle.

Overall, bitwise functions are extremely simple, yet very powerful. And they are absolutely essentially when we start to discuss masking in Section 6.4.

6.4 MASKING

In the previous section we explored bitwise functions. Now we are ready to explore masking, an extremely powerful and useful technique in computer vision and image pro-

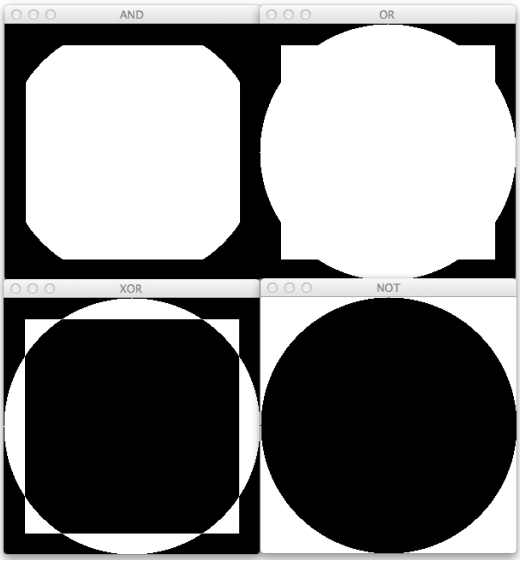

Figure 6.8: *Top-Left:* Applying a bitwise AND to our rectangle and circle image. *Top-Right:* A bitwise OR applied to our square and circle. *Bottom-Left:* An XOR applied to our shapes. *Bottom-Right:* Flipping pixel values of our circle using a bitwise NOT.

cessing.

Using a mask allows us to focus only on the portions of the image that interests us.

For example, let's say that we were building a computer vision system to recognize faces. The only part of the image we are interested in finding and describing are the parts of the image that contain faces – we simply don't care about the rest of the content of the image. Provided that we could find the faces in the image, we might construct a mask to show *only* the faces in the image.

Let's make this example a little more concrete.

In Figure 6.9 we have an image of a beach on the *Top-Left*. But I'm not interested in the beach in the image. I'm only interested in the sky and the palm tree. We could apply a cropping to extract that region of the image. Or, we could apply a mask to the image.

The image on the *Top-Right* is our mask – a white rectangle at the center of the image. By applying our mask to our beach image, we arrive at the image on the *Bottom*. By using our rectangle mask, we have focused only on the sky and palm tree in the image.

Let's examine the code to accomplish the masking in Figure 6.9:

Listing 6.20: masking.py

```
1 import numpy as np
```

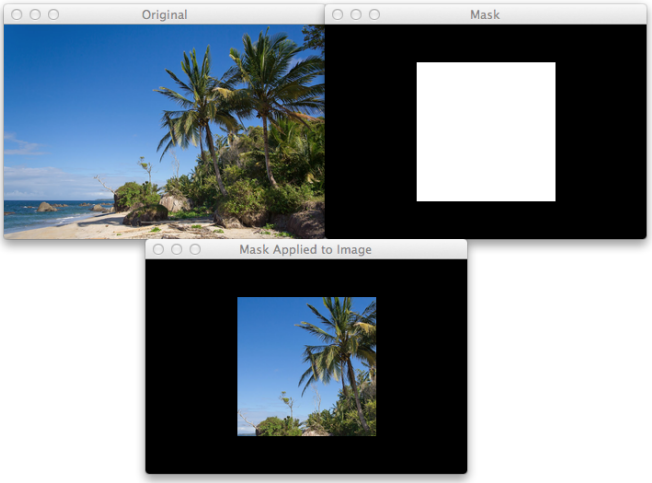

Figure 6.9: *Top-Left:* Our image of a peaceful beach scene. *Top-Right:* Our mask image – a white rectangle at the center of the image. *Bottom:* Applying the rectangular mask to the beach image. Only the parts of the image where the mask pixels are greater than zero are shown.

```
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 mask = np.zeros(image.shape[:2], dtype = "uint8")
14 (cX, cY) = (image.shape[1] / 2, image.shape[0] / 2)
15 cv2.rectangle(mask, (cX - 75, cY - 75), (cX + 75 , cY + 75), 255,
       -1)
16 cv2.imshow("Mask", mask)
17
18 masked = cv2.bitwise_and(image, image, mask = mask)
19 cv2.imshow("Mask Applied to Image", masked)
20 cv2.waitKey(0)
```

On **Lines 1-11** we import the packages we need, parse our arguments, and load our image.

We then construct a NumPy array, filled with zeros, with the same width and height as our beach image on **Line 13**. In order to draw the white rectangle, we first compute the center of the image on **Line 14** by dividing the width and height by two. Finally, we draw our white rectangle on **Line 15**.

Remember reviewing the `cv2.bitwise_and` function in the previous section? It's a function that is used extensively when applying masks to images.

We apply our mask on **Line 18** using the `cv2.bitwise_and` function. The first two parameters are the image itself. Obviously, the AND function will be `True` for all pixels in the image; however, the important part of this func-

tion is the `mask` keyword argument. By supplying a mask, the `cv2.bitwise_and` function only examines pixels that are "on" in the mask. In this case, only pixels that are part of the white rectangle.

Let's look at another example:

Listing 6.21: masking.py

```
21 mask = np.zeros(image.shape[:2], dtype = "uint8")
22 cv2.circle(mask, (cX, cY), 100, 255, -1)
23 masked = cv2.bitwise_and(image, image, mask = mask)
24 cv2.imshow("Mask", mask)
25 cv2.imshow("Mask Applied to Image", masked)
26 cv2.waitKey(0)
```

On **Line 21** we re-initialize our mask to be filled with zeros and the same dimensions as our beach image. Then, we draw a white circle on our mask image, starting at the center of the image and a radius of 100 pixels. Applying the circular mask is then performed on **Line 23**, again using the `cv2.bitwise_and` function.

The results of our circular mask can be seen in Figure 6.10. Our beach image is shown on the *Top-Left*, our circle mask on the *Top-Right*, and the application of the mask on the *Bottom*. Instead of a rectangular region of the beach being shown, we now have a circular region.

Right now masking may not seem very interesting. But we'll return to it once we start computing histograms in Chapter 7. Again, the key point of masks is that they allow us to focus our computation only on regions of the image that interests us.

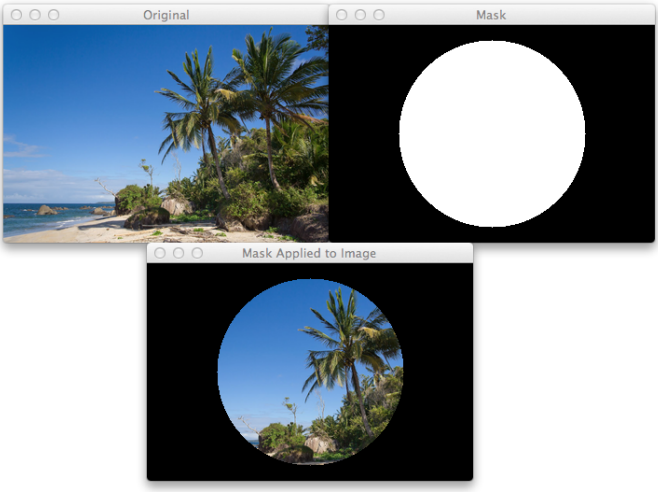

Figure 6.10: Applying the circular mask to the beach image. Only pixels within the circular white region are shown.

6.5 SPLITTING AND MERGING CHANNELS

A color image consists of multiple channels: a Red, a Green, and a Blue component. We have seen that we can access these components via indexing into NumPy arrays. But what if we wanted to split an image into its respective components?

As you'll see, we'll make use of the `cv2.split` function.

But for the time being, let's take a look at an example image in Figure 6.11.

We have an image of a wave crashing down. This image is very "blue", due to the ocean. How do we interpret the different channels of the image?

The Red channel (*Top-Left*) is very dark. This makes sense because an ocean scene has very little red colors in it. The red colors present are either very dark, and thus not represented, or very light, and likely part of the white foam of the wave as it crashes down.

The Green channel (*Top-Right*) is more represented in the image, since ocean water does contain greenish hues.

Finally, the Blue channel (*Bottom-Left*) is extremely light, and near pure white in some locations. This is because shades of blue are heavily represented in our image.

Now that we have visualized our channels, let's examine some code to accomplish this for us:

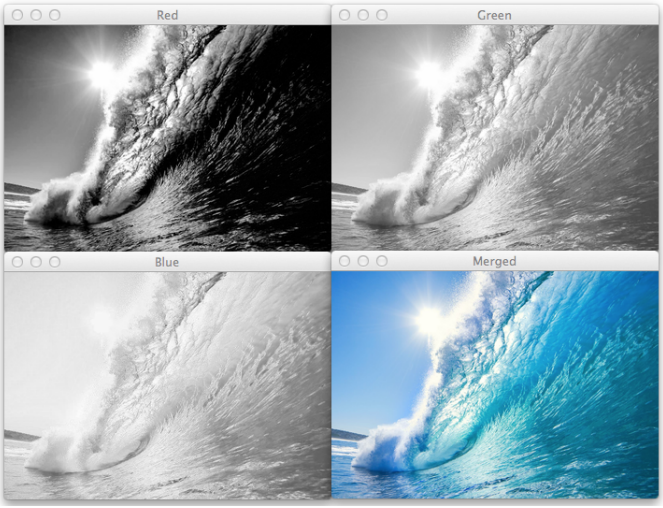

Figure 6.11: The three RGB channels of our wave image are shown on the *Bottom-Right*. The Red channels is on the *Top-Left*, the Green channel on the *Top-Right*, and the Blue channel on the *Bottom-Left*.

Listing 6.22: splitting_and_merging.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 (B, G, R) = cv2.split(image)
12
13 cv2.imshow("Red", R)
14 cv2.imshow("Green", G)
15 cv2.imshow("Blue", B)
16 cv2.waitKey(0)
17
18 merged = cv2.merge([B, G, R])
19 cv2.imshow("Merged", merged)
20 cv2.waitKey(0)
21 cv2.destroyAllWindows()
```

Lines 1-10 imports our packages, sets up our argument parser, and then loads our image. Splitting the channels is done using a call to `cv2.split` on **Line 11**.

Normally, we think of images in the RGB color space – the red pixel first, the green pixel second, and the blue pixel third. However, OpenCV stores RGB images as NumPy arrays in reverse channel order. Instead of storing an image in RGB order, it instead stores the image in BGR order, thus we unpack the tuple in reverse order.

Lines 13-16 then show each channel individually, as in Figure 6.11.

We can also merge the channels back together again using the `cv2.merge` function. We simply specify our chan-

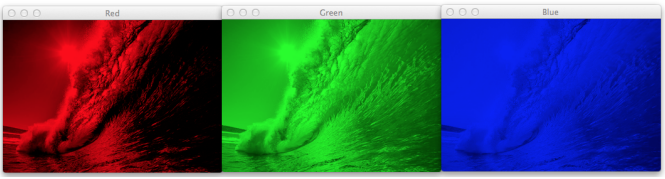

Figure 6.12: Representing the Red, Green, and Blue channels of our wave image.

nels, again in BGR order, and then `cv2.merge` takes care of the rest for us (**Line 18**)!

Listing 6.23: splitting_and_merging.py

```
22 zeros = np.zeros(image.shape[:2], dtype = "uint8")
23 cv2.imshow("Red", cv2.merge([zeros, zeros, R]))
24 cv2.imshow("Green", cv2.merge([zeros, G, zeros]))
25 cv2.imshow("Blue", cv2.merge([B, zeros, zeros]))
26 cv2.waitKey(0)
```

An alternative method to visualize the channels of an image can be seen in Figure 6.12. In order to show the actual "color" of the channel, we first need to take apart the image using `cv2.split`. Then, we need to re-construct the image, but this time, having all pixels *but the current channel* as zero.

On **Line 22** we construct a NumPy array of zeros, with the same width and height as our original image. Then, in order to construct the Red channel representation of the image, we make a call to `cv2.merge`, but specifying our `zeros` array for the Green and Blue channels. We take similar approaches to the other channels in **Line 24 and 25**.

6.6 COLOR SPACES

In this book, we have only explored the RGB color space; however, there are many other color spaces that we can utilize.

The Hue-Saturation-Value (HSV) color space is more similar to how humans think and conceive of color. Then there is the L*a*b* color space which is more tuned to how humans *perceive* color.

OpenCV provides support for many, many different color spaces. And understanding how color is perceived by humans and represented by computers occupies an entire library of literature itself.

In order to not get bogged down in the details, I'll just show you how to convert color spaces. If you think your application of image processing and computer vision might need a different color space than RGB, I will leave that as an exercise to the reader to explore the peculiarities of each color space.

Let's explore some code to change color spaces:

Listing 6.24: colorspaces.py

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
```

```
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 cv2.imshow("Gray", gray)
15
16 hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
17 cv2.imshow("HSV", hsv)
18
19 lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)
20 cv2.imshow("L*a*b*", lab)
21 cv2.waitKey(0)
```

Lines 1-11 imports the packages we need, parses our arguments, and loads our image. Then, on **Line 13** we convert our image from the RGB color space to grayscale by specifying the `cv2.COLOR_BGR2GRAY` flag.

Converting our image to the HSV color space is performed on **Line 16** by specifying the `cv2.COLOR_BGR2HSV` flag. Finally, on **Line 19** we convert to the L*a*b* color space by using the `cv2.COLOR_BGR2LAB` flag.

We can see the results of our color space conversions in Figure 6.13.

The role of color spaces in image processing and computer vision is important, yet complicated at the same time. If you are just getting started in computer vision, it's likely a good idea to stick to the RGB color space for the time being. However, I have included this section as a matter of completeness – it's good to show an example of how to convert color spaces for when you decide the time is right!

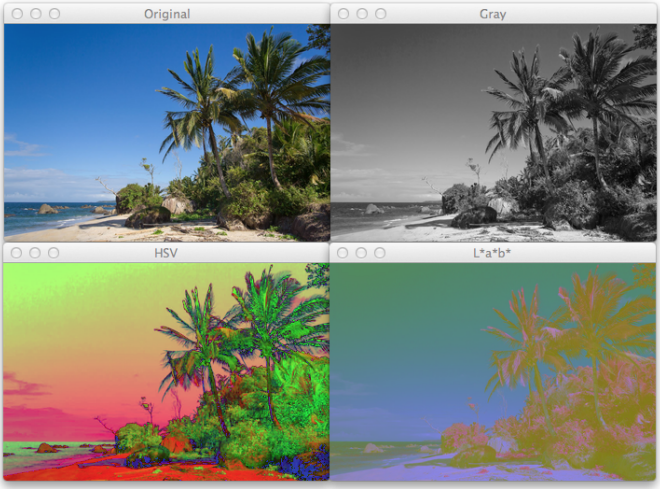

Figure 6.13: *Top-Left:* An image of beach scenery. *Top-Right:* The grayscale representation of the beach image. *Bottom-Left:* Converting the beach image to the HSV color space. *Bottom-Right:* Converting our image to the L*a*b* color space.