Practical Python and OpenCV

An Introductory, Example Driven Guide to Image Processing and Computer Vision

3RD EDITION

Dr. Adrian Rosebrock

pyimagesearch

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

3rd Edition

Dr. Adrian Rosebrock

COPYRIGHT

This version of the book was published on 21 August 2016.

CONTENTS

COMPANION WEBSITE & SUPPLEMENTARY MATERIAL

Thank you for picking up a copy of the 3rd edition of *Practical Python and OpenCV*!

In this latest edition, I'm excited to announce the creation of a *companion website* which includes supplementary material that I could not fit inside the book.

At the end of nearly every chapter inside *Practical Python and OpenCV + Case Studies*, you'll find a link to a supplementary webpage that includes additional information, such as my commentary on methods to extend your knowledge, discussions of common error messages, recommendations on various algorithms to try, and optional quizzes to test your knowledge.

Registration to the companion website is **free** with your purchase of *Practical Python and OpenCV*.

To create your companion website account, just use this link:

http://pyimg.co/o1y7e

Take a second to create your account **now** so you'll have access to the supplementary materials as you work through the book.

PREFACE

When I first set out to write this book, I wanted it to be as hands-on as possible. I wanted lots of visual examples with lots of code. I wanted to write something that you could easily learn from, without all the rigor and detail of mathematics associated with college level computer vision and image processing courses.

I know from all my years spent in the classroom that the way I learned best was from simply opening up an editor and writing some code. Sure, the theory and examples in my textbooks gave me a solid starting point. But I never really "learned" something until I did it myself. I was very hands-on. And that's exactly how I wanted this book to be. Very hands-on, with all the code easily modifiable and well documented so you could play with it on your own. That's why I'm giving you the full source code listings and images used in this book.

More importantly, I wanted this book to be accessible to a wide range of programmers. I remember when I first started learning computer vision – it was a daunting task. But I learned a lot. And I had a lot of fun.

I hope this book helps you in your journey into computer vision. I had a blast writing it. If you have any questions, suggestions, or comments, or if you simply want to say hello, shoot me an email at adrian@pyimagesearch.com, or

you can visit my website at www.PyImageSearch.com and leave a comment. I look forward to hearing from you soon!

-Adrian Rosebrock

PREREQUISITES

In order to make the most of this, you will need to have a little bit of programming experience. All examples in this book are in the Python programming language. Familiarity with Python or other scripting languages is suggested, but not required.

You'll also need to know some basic mathematics. This book is hands-on and example driven: lots of examples and lots of code, so even if your math skills are not up to par, do not worry! The examples are very detailed and heavily documented to help you follow along.

CONVENTIONS USED IN THIS BOOK

This book includes many code listings and terms to aid you in your journey to learn computer vision and image processing. Below are the typographical conventions used in this book:

Italic

Indicates key terms and important information that you should take note of. May also denote mathematical equations or formulas based on connotation.

Bold

Important information that you should take note of.

`Constant width`

Used for source code listings, as well as paragraphs that make reference to the source code, such as function and method names.

USING THE CODE EXAMPLES

This book is meant to be a hands-on approach to computer vision and machine learning. The code included in this book, along with the source code distributed with this book, are free for you to modify, explore, and share as you wish.

In general, you do not need to contact me for permission if you are using the source code in this book. Writing a script that uses chunks of code from this book is totally and completely okay with me.

However, selling or distributing the code listings in this book, whether as information product or in your product's documentation, *does* require my permission.

If you have any questions regarding the fair use of the code examples in this book, please feel free to shoot me an email. You can reach me at adrian@pyimagesearch.com.

HOW TO CONTACT ME

Want to find me online? Look no further:

Website:	www.PyImageSearch.com
Email:	adrian@pyimagesearch.com
Twitter:	@PyImageSearch
Google+:	+AdrianRosebrock
LinkedIn:	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 cannot only *find* faces in an image, 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. 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 jeans trends? How long did you look at the jeans? What was your facial expression as you looked at the jeans? Did you then pick up 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 could automatically analyze and quantify cellular structures – without human intervention! Now, 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 panoramic 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.

Further Reading

Welcome to the supplementary material portion of the chapter! If you haven't already registered and created your account for the companion website, please do so using the following link:

http://pyimg.co/o1y7e

From there, you can find the Chapter 1 supplementary material page here:

http://pyimg.co/rhsgi

This page serves as an introduction to the companion website and details how to use it and what to expect as you work through the rest of *Practical Python and OpenCV*.

2

PYTHON AND REQUIRED PACKAGES

In order to explore the world of computer vision, we'll first need to install some packages and libraries. As a first-timer in computer vision, installing some of these packages (especially OpenCV) can be quite tedious, depending on what operating system you are using. I've tried to consolidate the installation instructions into a short how-to guide, but as you know, projects change, websites change, and installation instructions change! If you run into problems, be sure to consult the package's website for the most up-to-date installation instructions.

I highly recommend that you use either easy_install or pip to manage the installation of your packages. It will make your life much easier! You can read more about pip here: http://pyimg.co/9quup.

Finally, if you don't want to undertake installing these packages by hand, I have put together an Ubuntu virtual machine with all the necessary computer vision and image processing packages you need to run the examples in this book pre-installed! Using this virtual machine allows you to jump right in to the examples in this book, without having to worry about package managers, installation instruc-

tions, and compiling errors.

To find out more about this pre-configured virtual machine, head on over to: http://www.pyimagesearch.com/practical-python-opencv/.

In the rest of this chapter, I will discuss the various Python packages that are useful for computer vision and image processing. I'll also provide instructions on how to install each of these packages.

It is worth mentioning that I have collected OpenCV installation tutorials for various Python versions and operating systems on PyImageSearch: http://pyimg.co/vvlpy.

Be sure to take a look as I'm sure the install guides will be helpful to you! In the meantime, let's review some important Python packages that we'll use for computer vision.

2.1 A NOTE ON PYTHON & OPENCV VERSIONS

Over a year ago, when I wrote the first edition of *Practical Python and OpenCV + Case Studies*, the current version of OpenCV was 2.4.9, which only supported Python 2.7. While many scientific developers (myself included) are very much accustomed to using Python 2.7, newcomers to computer vision and machine learning were often confused and frustrated by the lack of Python 3 support – Python 3 is the future of the Python programming language, after all!

However, this all changed on June 4th, 2015, which marked a momentous date in the history of OpenCV:

OpenCV 3.0 was finally released!

The benefits of OpenCV 3.0 are numerous, including improved stability, performance, increases, and even transparent OpenCL support.

But *by far* the most exciting update to us in the Python world is:

Python 3 support!

After years of being stuck and sequestered to Python 2.7, **we can now finally use OpenCV with Python 3+!**

Inside this book, you'll find that all chapters, code samples, and datasets are compatible with **OpenCV 3+**. Furthermore, all code examples will run in both the **Python 2.7** *and* the **Python 3+** environments!

If you are looking for the OpenCV 2.4.X and Python 2.7 version of this book, please look in the download directory associated with your purchase – inside you will find the OpenCV 2.4.X + Python 2.7 edition.

2.2 NUMPY AND SCIPY

NumPy is a library for the Python programming language that (among other things) provides support for large, multi-dimensional arrays. Why is that important? Using NumPy, we can express images as multi-dimensional arrays. Representing images as NumPy arrays is not only computation-

ally and resource efficient, many other image processing and machine learning libraries use NumPy array representations as well. Furthermore, by using NumPy's built-in high-level mathematical functions, we can quickly and easily perform numerical analysis on an image.

Going hand-in-hand with NumPy, we also have SciPy. SciPy adds further support for scientific and technical computing.

2.2.1 *Windows*

By far, the easiest way to install NumPy and SciPy on your Windows system is to download and install the binary distribution from: http://www.scipy.org/install.html.

2.2.2 *OSX*

If you are running OSX 10.7.0 (Lion) or above, NumPy and SciPy come pre-installed.

You can also install NumPy and SciPy using pip:

Listing 2.1: Install NumPy and SciPy on OSX

```
$ pip install numpy
$ pip install scipy
```

2.2.3 *Linux*

On many Linux distributions, such as Ubuntu, NumPy comes pre-installed and configured.

If you want the latest versions of NumPy and SciPy, you can build the libraries from source, but the easiest method is to use a pip:

Listing 2.2: Install NumPy and SciPy on Linux

```
$ pip install numpy
$ pip install scipy
```

2.3 MATPLOTLIB

Simply put, matplotlib is a plotting library. If you've ever used MATLAB before, you'll probably feel very comfortable in the matplotlib environment. When analyzing images, we'll make use of matplotlib. Whether plotting image histograms or simply viewing the image itself, matplotlib is a great tool to have in your toolbox.

2.3.1 *All Platforms*

Matplotlib is available from http://matplotlib.org/. The matplotlib package is also pip-installable:

Listing 2.3: Install matplotlib

```
$ pip install matplotlib
```

Otherwise, a binary installer is provided for Windows.

2.4 OPENCV

If NumPy's main goal is large, efficient, multi-dimensional array representations, then, the main goal of OpenCV is real-time image processing. This library has been around since 1999, but it wasn't until the 2.0 release in 2009 that we saw the incredible NumPy support. The library itself is written in C/C++, but Python bindings are provided when running the installer. OpenCV is hands down my favorite computer vision library, and we'll use it a lot in this book.

In June 2015, OpenCV 3.0 was **officially** released. This update is definitely one of the most extensive overhauls to the library in recent years and boasts increased stability, performance increases, and OpenCL support.

But *by far*, the most exciting update for us in the Python world is: **Python 3 support!**

After years of being stuck in Python 2.7, **we can now finally use OpenCV in Python 3.0!** Awesome news, indeed!

The installation for OpenCV is constantly changing. Since the library is written in C/C++, special care has to be taken when compiling and ensuring that the prerequisites are installed. Be sure to check the OpenCV website at http://opencv.org/ for the latest installation instructions since they do (and will) change in the future.

2.4.1 *Linux and OSX*

Installing OpenCV in Linux and OSX has been a pain in previous years, but has luckily gotten much easier. I have accumulated OpenCV installation instructions on the PyImageSearch blog for Debian-based Linux distributions (such as Ubuntu) and OSX here:

http://pyimg.co/vvlpy

Just scroll down the "Install OpenCV 3 and Python" section, select the operating system and Python version that you want to install OpenCV 3 for, and you'll be on your way!

Alternatively, you can install the previous version of OpenCV 2.4.X on OSX using these instructions from Jeffrey Thompson:

http://pyimg.co/kdwts

2.4.2 *Windows*

The OpenCV Docs provide fantastic tutorials on how to install OpenCV in Windows using binary distributions. You can check out the installation instructions here:

http://pyimg.co/l2q6s

2.5 MAHOTAS

Mahotas, just like OpenCV, relies on NumPy arrays. Much of the functionality implemented in Mahotas can be found in OpenCV, but in some cases, the Mahotas interface is just easier to use. We'll use Mahotas to complement OpenCV.

2.5.1 *All Platforms*

Installing Mahotas is extremely easy on all platforms. Assuming you already have NumPy and SciPy installed, all you need is a single call to the pip command:

Listing 2.4: Install Mahotas

```
$ pip install mahotas
```

2.6 SCIKIT-LEARN

Alright, you got me, scikit-learn isn't an image processing or computer vision library – it's a machine learning library. That said, you can't have advanced computer vision techniques without some sort of machine learning, whether it be clustering, vector quantization, classification models, etc. Scikit-learn also includes a handful of image feature extraction functions as well. We don't use the scikit-learn library in *Practical Python and OpenCV*, but it's heavily used in *Case Studies*.

2.6.1 *All Platforms*

Installing scikit-learn on all platforms is dead-simple using pip:

Listing 2.5: Install scikit-learn

```
$ pip install scikit-learn
```

2.7 SCIKIT-IMAGE

The algorithms included in scikit-image (I would argue) follow closer to the state-of-the-art in computer vision. New algorithms right from academic papers can be found in scikit-image, but in order to (effectively) use these algorithms, you need to have developed some rigor and understanding in the computer vision field. If you already have some experience in computer vision and image processing, definitely check out scikit-image; otherwise, I would continue working with OpenCV to start. Again, scikit-image won't be used in of *Practical Python and OpenCV*, but it will be used in *Case Studies*, especially when we perform handwritten digit recognition.

Assuming you already have NumPy and SciPy installed, you can install scikit-image using pip:

Listing 2.6: Install scikit-image

```
$ pip install -U scikit-image
```

Now that we have all our packages installed, let's start exploring the world of computer vision!

2.8 SKIP THE INSTALLATION

As I've mentioned above, installing all these packages can be time consuming and tedious. If you want to skip the installation process and jump right into the world of image processing and computer vision, I have set up a pre-configured Ubuntu virtual machine with all of the above libraries mentioned already installed.

If you are interested in downloading this virtual machine (and saving yourself a lot of time and hassle), you can head on over to http://www.pyimagesearch.com/practical-python-opencv/.

Further Reading

To learn more about installing OpenCV, Python virtual environments, and choosing a code editor, please see the Chapter 2 supplementary material webpage:

http://pyimg.co/fosxq

In particular, I think you'll be interested in learning how the PyCharm IDE can be utilized with Python virtual environments to create the perfect computer vision development environment.

3

LOADING, DISPLAYING, AND SAVING

This book is meant to be a hands-on, how-to guide to getting started with computer vision using Python and OpenCV. With that said, let's not waste any time. We'll get our feet wet by writing some simple code to load an image off disk, display it on our screen, and write it to file in a different format. When executed, our Python script should show our image on screen, like in **Figure 3.1**.

First, let's create a file named `load_display_save.py` to contain our code. Now we can start writing some code:

Listing 3.1: load_display_save.py

```
1 from __future__ import print_function
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())
```

The first thing we are going to do is import the packages we will need for this example.

Figure 3.1: Example of loading and displaying a *Tyrannosaurus Rex* image on our screen.

Throughout this book you'll see us importing the `print_function` from the `__future__` package. We'll be using the actual `print()` *function* rather than the `print` *statement* so that our code will work with **both** Python 2.7 and Python 3 – just something to keep in mind as we work through the examples!

We'll use `argparse` to handle parsing our command line arguments. Then, `cv2` is imported – `cv2` is our OpenCV library and contains our image processing functions.

From there, **Lines 5-8** handle parsing the command line arguments. The only argument we need is `--image`: the path to our image on disk. Finally, we parse the arguments and store them in a dictionary.

Listing 3.2: load_display_save.py

```
9  image = cv2.imread(args["image"])
10 print("width: {} pixels".format(image.shape[1]))
11 print("height: {} pixels".format(image.shape[0]))
12 print("channels: {}".format(image.shape[2]))
13
14 cv2.imshow("Image", image)
15 cv2.waitKey(0)
```

Now that we have the path to the image, we can load it off the disk using the `cv2.imread` function on **Line 9**. The `cv2.imread` function returns a NumPy array representing the image.

Lines 10-12 examine the dimensions of the image. Again, since images are represented as NumPy arrays, we can simply use the `shape` attribute to examine the width, height, and the number of channels.

Finally, **Lines 14 and 15** handle displaying the actual image on our screen. The first parameter is a string, the "name" of our window. The second parameter is a reference to the image we loaded off disk on **Line 9**. Finally, a call to `cv2.waitKey` pauses the execution of the script until we press a key on our keyboard. Using a parameter of `0` indicates that any keypress will un-pause the execution.

The last thing we are going to do is write our image to file in JPG format:

Listing 3.3: load_display_save.py

```
16     cv2.imwrite("newimage.jpg", image)
```

All we are doing here is providing the path to the file (the first argument) and then the image we want to save

(the second argument). It's that simple.

To run our script and display our image, we simply open up a terminal window and execute the following command:

Listing 3.4: load_display_save.py

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

If everything has worked correctly, you should see the *T-Rex* on your screen as in **Figure 3.1**. To stop the script from executing, simply click on the image window and press any key.

Examining the output of the script, you should also see some basic information on our image. You'll note that the image has a width of 350 pixels, a height of 228 pixels, and 3 channels (the RGB components of the image). Represented as a NumPy array, our image has a shape of `(228,350,3)`.

The NumPy shape may seem reversed to you (specifying the height before the width), but in terms of a *matrix definition*, it actually makes sense. When we define matrices, it is common to write them in the form *(# of rows* $\times$ *# of columns)*. Here, our image has a *height* of 228 pixels (the number of *rows*) and a *width* of 350 pixels (the number of *columns*) – thus, the NumPy shape makes sense (although it may seen a bit confusing at first).

Finally, note the contents of your directory. You'll see a new file there: `newimage.jpg`. OpenCV has automatically converted our PNG image to JPG for us! No further effort is needed on our part to convert between image formats.

Next up, we'll explore how to access and manipulate the pixel values in an image.

Further Reading

You can find the Chapter 3 supplementary material, resources, and quizzes here:

http://pyimg.co/xh73h

Specifically, I discuss some common "gotchas" that may trip you up when utilizing OpenCV for the first time – these tips and tricks are *especially useful* if this is your first exposure to OpenCV.

Be sure to take the quiz to test your knowledge after reading this chapter!

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 corresponds to "black" and 255 corresponds to "white". The values in between 0 and 255 are varying shades of gray, where values closer to 0 are darker and values closer to 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 is 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 an RGB tuple in the form `(red, green, blue)`. This tuple represents our color.

To construct a white color, we would fill up each of the red, green, and blue buckets completely, 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 clearer.

Here we have the letter "I" on a piece of graph paper. We see that we have an 8×8 grid with a total of 64 pixels.

The point $(0, 0)$ 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*.

The Python language is *zero indexed*, meaning that we always start counting from zero. Remember this 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

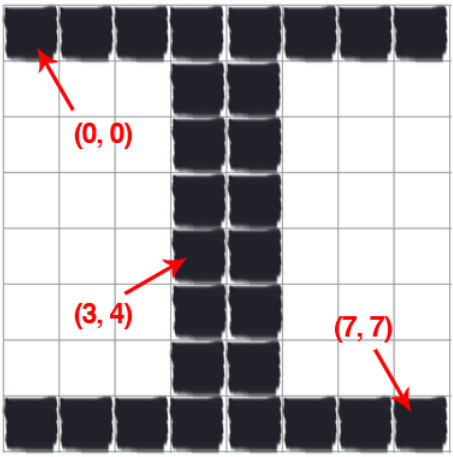

Figure 4.1: The letter "I" placed on a piece of graph paper. Pixels are accessed by their (x, y) coordinates, where we go x columns to the right and y rows down, keeping in mind that Python is zero-indexed: we start counting from zero rather than one.

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 from __future__ import print_function
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)
```

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: the path to the image we are going to work with.

Lines 10 and 11 handle loading the actual image off 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

```
12 (b, g, r) = image[0, 0]
13 print("Pixel at (0, 0) - Red: {}, Green: {}, Blue: {}".format(r,
       g, b))
14
15 image[0, 0] = (0, 0, 255)
16 (b, g, r) = image[0, 0]
17 print("Pixel at (0, 0) - Red: {}, Green: {}, Blue: {}".format(r,
       g, b))
```

On **Line 12**, 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 13** 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 is 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 15** 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 15**, we then grab the pixel value and print it back to console on **Lines 16 and 17**, 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

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

On **Line 18** 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:** Finally, we need to provide an 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 19** 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 21**, 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 23 and 24 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 13**). 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 15-17**).

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

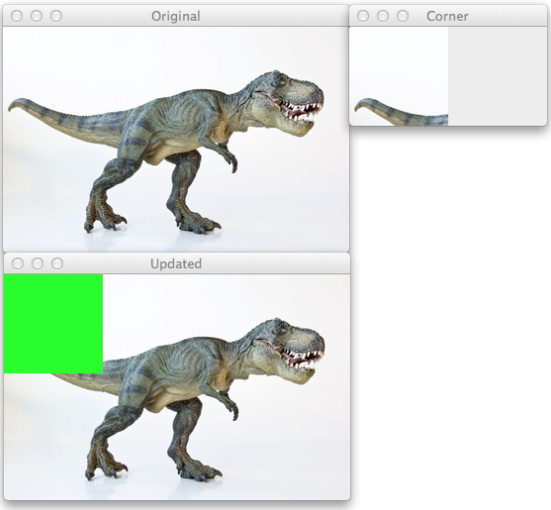

Figure 4.2: *Top-Left*: Our original image. *Top-Right*: Cropping our image using NumPy array slicing. *Bottom*: Drawing a 100×100 pixel green square on our image by using basic NumPy indexing.

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.

Further Reading

One of the *most common* errors I see with developers just starting to learn OpenCV is the *(x, y)*-coordinate ordering passed into images. I also tend to see a lot of confusion regarding the BGR versus RGB channel ordering.

To learn more about these common errors (and how you can avoid) then, be sure to refer to the Chapter 4 supplementary material:

http://pyimg.co/mtemn

I've also included a quiz that you can use to test your knowledge on image basics.

5

DRAWING

Using NumPy array slices in Chapter 4, we were able to draw a green square on our image. But what if we wanted to draw a single line? Or a circle? NumPy does not provide that type of functionality – it's only a numerical processing library after all!

Luckily, OpenCV provides convenient, easy-to-use methods to draw shapes on an image. In this chapter, we'll review the three most basic methods to draw shapes: `cv2.line`, `cv2.rectangle`, and `cv2.circle`.

While this chapter is by no means a complete, exhaustive overview of the drawing capabilities of OpenCV, it will nonetheless provide a quick, hands-on approach to get you started drawing immediately.

5.1 LINES AND RECTANGLES

Before we start exploring the the drawing capabilities of OpenCV, let's first define our canvas in which we will draw our masterpieces.

Up until this point, we have only loaded images off disk. However, we can also define our images manually using NumPy arrays. Given that OpenCV interprets an image as a NumPy array, there is no reason why we can't manually define the image ourselves!

In order to initialize our image, let's examine the code below:

Listing 5.1: drawing.py

```
1 import numpy as np
2 import cv2
3
4 canvas = np.zeros((300, 300, 3), dtype = "uint8")
```

Lines 1 and 2 imports the packages we will be using. As a shortcut, we'll create an alias for `numpy` as `np`. We'll continue this convention throughout the rest of the book. In fact, you'll commonly see this convention in the Python community as well! We'll also import `cv2`, so we can have access to the OpenCV library.

Initializing our image is handled on **Line 4**. We construct a NumPy array using the `np.zeros` method with 300 rows and 300 columns, yielding a 300×300 pixel image. We also allocate space for 3 channels – one for Red, Green, and Blue, respectively. As the name suggests, the `zeros` method fills every element in the array with an initial value of zero.

It's important to draw your attention to the second argument of the `np.zeros` method: the data type, `dtype`. Since we are representing our image as an RGB image with pixels in the range $[0, 255]$, it's important that we use an 8-bit unsigned integer, or `uint8`. There are many other data types

that we can use (common ones include 32-bit integers, and 32-bit or 64-bit floats), but we'll mainly be using `uint8` for the majority of the examples in this book.

Now that we have our canvas initialized, we can do some drawing:

Listing 5.2: drawing.py

```
5 green = (0, 255, 0)
6 cv2.line(canvas, (0, 0), (300, 300), green)
7 cv2.imshow("Canvas", canvas)
8 cv2.waitKey(0)
9
10 red = (0, 0, 255)
11 cv2.line(canvas, (300, 0), (0, 300), red, 3)
12 cv2.imshow("Canvas", canvas)
13 cv2.waitKey(0)
```

The first thing we do on **Line 5** is define a tuple used to represent the color "green". Then, we draw a green line from point $(0, 0)$ (the top-left corner of the image) to point $(300, 300)$, the bottom-right corner of the image on **Line 6**.

In order to draw the line, we make use of the `cv2.line` method. The first argument to this method is the image we are going to draw on. In this case, it's our `canvas`. The second argument is the starting point of the line. We choose to start our line from the top-left corner of the image, at point $(0, 0)$. We also need to supply an ending point for the line (the third argument). We define our ending point to be $(300, 300)$, the bottom-right corner of the image. The last argument is the color of our line, which, in this case, is green. **Lines 7 and 8** show our image and then wait for a keypress.

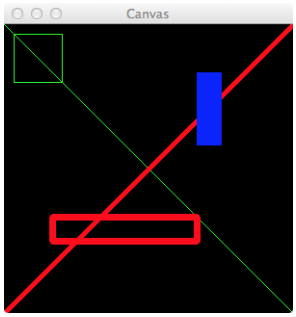

Figure 5.1: Examples of drawing lines and rectangles using OpenCV.

As you can see, drawing a line is quite simple! But there is one other important argument to consider in the `cv2.line` method: the thickness.

On **Lines 10-13** we define a red color as a tuple (again, in BGR rather than RGB format). We then draw a red line from the top-right corner of the image to the bottom left. The last parameter to the method controls the thickness of the line – we decide to make the thickness 3 pixels. Again, we show our image and wait for a keypress.

Drawing a line was simple enough. Now we can move on to drawing rectangles. Check out the code below for more details:

Listing 5.3: drawing.py

```
14 cv2.rectangle(canvas, (10, 10), (60, 60), green)
15 cv2.imshow("Canvas", canvas)
```

```
16 cv2.waitKey(0)
17
18 cv2.rectangle(canvas, (50, 200), (200, 225), red, 5)
19 cv2.imshow("Canvas", canvas)
20 cv2.waitKey(0)
21
22 blue = (255, 0, 0)
23 cv2.rectangle(canvas, (200, 50), (225, 125), blue, -1)
24 cv2.imshow("Canvas", canvas)
25 cv2.waitKey(0)
```

On **Line 14** we make use of the `cv2.rectangle` method. The signature of this method is identical to the `cv2.line` method above, but let's explore each argument anyway.

The first argument is the image we want to draw our rectangle on. We want to draw on our `canvas`, so we pass it into the method. The second argument is the starting (x, y) position of our rectangle – here, we are starting our rectangle at point $(10, 10)$. Then, we must provide an ending (x, y) point for the rectangle. We decide to end our rectangle at $(60, 60)$, defining a region of 50×50 pixels. Finally, the last argument is the color of the rectangle we want to draw.

Just as we can control the thickness of a line, we can also control the thickness of a rectangle. **Line 18** provides one added argument: the thickness. Here, we draw a red rectangle that is 5 pixels thick, starting from point $(50, 200)$ and ending at $(200, 225)$.

At this point, we have only drawn the *outline* of a rectangle. How do we draw a rectangle that is "filled in", like when using NumPy array slices in Chapter 4?

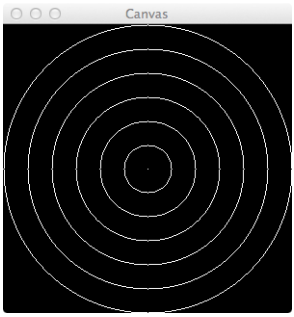

Figure 5.2: Drawing a simple bullseye with the `cv2.circle` function.

Simple. We just pass in a negative value for the thickness argument.

Line 23 demonstrates how to draw a rectangle of a solid color. We draw a blue rectangle, starting from $(200, 50)$ and ending at $(225, 125)$. By specifying `-1` as the thickness, our rectangle is drawn as a solid blue.

Congratulations! You now have a solid grasp of drawing rectangles. In the next section, we'll move on to drawing circles.

5.2 CIRCLES

Drawing circles is just as simple as drawing rectangles, but the function arguments are a little different. Let's go ahead and get started:

Listing 5.4: drawing.py

```
26 canvas = np.zeros((300, 300, 3), dtype = "uint8")
27 (centerX, centerY) = (canvas.shape[1] // 2, canvas.shape[0] // 2)
28 white = (255, 255, 255)
29
30 for r in range(0, 175, 25):
31     cv2.circle(canvas, (centerX, centerY), r, white)
32
33 cv2.imshow("Canvas", canvas)
34 cv2.waitKey(0)
```

On **Line 26** we re-initialize our `canvas` to be blank. The rectangles are gone! We need a fresh canvas to draw our circles.

Line 27 calculates two variables: `centerX` and `centerY`. These two variables represent the (x, y) coordinates of the center of the image. We calculate the center by examining the shape of our NumPy array, and then dividing by two. The height of the image can be found in `canvas.shape[0]` and the width in `canvas.shape[1]`. Finally, **Line 28** defines a white pixel.

Now, let's draw some circles!

On **Line 30** we loop over a number of radius values, starting from 0 and ending at 150 (since the `range` function is *exclusive*), incrementing by 25 at each step.

Line 31 handles the actual drawing of the circle. The first parameter is our `canvas`, the image we want to draw the circle on. We then need to supply the point in which our circle will be drawn around. We pass in a tuple of `(centerX, centerY)` so that our circles will be centered at the middle of the image. The third argument is the radius of the circle we wish to draw. Finally, we pass in the color of our circle,