1.11.1: Finding and drawing contours

Topic Progress: (https://gurus.pyimagesearch.com/topic/finding-and-drawing-contours/)

(https://gurus.pyimagesearch.com/topic/simple-contour-properties/)

(https://gurus.pyimagesearch.com/topic/advanced-contour-properties/)

(https://gurus.pyimagesearch.com/topic/contour-approximation/)

(https://gurus.pyimagesearch.com/topic/sorting-contours/)

← Back to Lesson (https://gurus.pyimagesearch.com/lessons/contours/)

So up until this point in the course we have been able to apply methods like thresholding (**Module 1.9** (https://gurus.pyimagesearch.com/lessons/thresholding/)) and edge detection (**Module 1.10** (https://gurus.pyimagesearch.com/lessons/gradients-and-edge-detection/)) to detect the outlines and structures of objects in images.

However, now that we have the outlines and structures of the objects in images, the big question becomes: *How do we find and access these outlines?*

The answer: *contours*.

I'll say this many times before this lesson is over, but contours are an ***invaluable*** tool to have in your tool belt. Being able to leverage simple contour properties enables you to solve complicated problems with ease.

Be sure to pay attention to this module — it will be a lot to digest, but the knowledge you gain will enable you to reach Guru status quicker.

All too often I see computer vision and image processing developers trying to leverage complicated machine learning techniques to solve problems *where contours would be better suited*.

Don't fall into this trap — pay attention to this module, as it's one of ***the most important*** techniques that you'll use over and over again in your computer vision career.

In the meantime, we'll get started with the basics of contours, which include *finding and detecting contours* in an image, along with *extracting objects from an image* using contours, masks, and cropping.

Objectives:

By the time you are done working through this lesson you will be able to:

1. Find and detect the contours of objects in images.
2. Extract objects from images using contours, masks, and cropping.

Finding and Drawing Contours

Before we can get utilize *contour properties* to identify X's and O's on a tic-tac-toe board or identify Tetris blocks, we first need to understand how to *find* contours in an image and *draw* them.

As I mentioned in the introduction to this lesson, contours are simply the *outlines* of an object in an image. If the image is simple enough, we might be able to get away with using the grayscale image as an input.

But for more complicated images, we must first find the object by using methods such as edge detection or thresholding — we are simply seeking a binary image where white pixels correspond to objects in an image and black pixels as the background. There are many ways to obtain a binary image like this, but the most used methods are edge detection and thresholding.

Key takeaway: For better accuracy you'll normally want to utilize a *binary* image rather than a *grayscale* image. We'll use both binary and grayscale images in this lesson. But once we start to get to some of the more advanced contour topics, you'll notice that we switch from grayscale to binary images to improve our contour accuracy.

Once we have this binary or grayscale image, we need to *find* the outlines of the objects in the image. This is actually a lot easier than it sounds thanks to the `cv2.findContours` function. The `cv2.findContours` function has many parameters to explore, so let's go ahead and start coding to figure out how to use this function:

```
1  # import the necessary packages
2  import numpy as np
3  import argparse
4  import cv2
5  import imutils
6
7  # construct the argument parser and parse the arguments
8  ap = argparse.ArgumentParser()
9  ap.add_argument("-i", "--image", required=True, help="Path to the image")
10 args = vars(ap.parse_args())
11
12 # load the image and convert it to grayscale
13 image = cv2.imread(args["image"])
14 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
15
16 # show the original image
17 cv2.imshow("Original", image)
18
19 # find all contours in the image and draw ALL contours on the image
20 cnts = cv2.findContours(gray.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
21 cnts = cnts[0] if imutils.is_cv2() else cnts[1]
22 clone = image.copy()
23 cv2.drawContours(clone, cnts, -1, (0, 255, 0), 2)
24 print("Found {} contours".format(len(cnts)))
25
26 # show the output image
27 cv2.imshow("All Contours", clone)
28 cv2.waitKey(0)
```

We'll start by importing our packages and setting up our argument parser. We'll need just a single argument here, `--image` , which is the path to where our image resides on disk.

We then take this image, load it off disk on **Line 13**, convert it to grayscale on **Line 14**, and display the original image to our screen on **Line 17**.

Now we are ready for the fun part: *actually detecting the contours in the image.*

We make a call to `cv2.findContours` on **Line 20**, passing in our grayscale image. The `cv2.findContours` function is ***destructive*** to the input image (meaning that it manipulates it) so if you intend on using your input image again, be sure to clone it using the `copy()` method prior to passing it into `cv2.findContours` .

We'll instruct `cv2.findContours` to return a list of ***all*** contours in the image by passing in the `cv2.RETR_LIST` flag. This flag will ensure that all contours are returned. Other methods exist, such as returning only the external most contours, which we'll explore later in this article.

Finally, we pass in the `cv2.CHAIN_APPROX_SIMPLE` flag. If we did not specify this flag and instead used `cv2.CHAIN_APPROX_NONE` , we would be storing ***every single*** (*x*, *y*)-coordinate along the contour. In general, this not advisable. It's substantially slower and takes up significantly more memory. By compressing our horizontal, vertical, and diagonal segments into only end-points we are able to reduce memory consumption significantly without any substantial loss in contour accuracy. You can read more about the compression of contours in the OpenCV documentation (http://docs.opencv.org/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#findcontou

Finally, the `cv2.findContours` function returns a tuple of values. The first value is the image itself (OpenCV 2.4 does not place the image in the first value of the tuple).

The 2nd value is the contours themselves (for OpenCV 2.4, it is first value). These contours are simply the boundary points of the outline along the object.

The third value is the hierarchy of the contours (for OpenCV 2.4, it is the second value), which contains information on the topology of the contours. Often we are only interested in the contours themselves and not their actual hierarchy (i.e. one contour being contained in another) so this second value is usually ignored. However, in the case that you would like to examine the contour hierarchy, this variable is available to you.

(10/19/2017) Update for OpenCV 3: *On **Line 21** OpenCV version differences mentioned above are handled. See this blog post (https://www.pyimagesearch.com/2015/08/10/checking-your-opencv-version-using-python/) for further details. This is changed throughout all 1.11 lessons and will eventually be updated throughout the whole course. Take note!*

Feedback

We then draw our found contours on **Line 23**. The first argument we pass in is the image we want to draw the contours on. The second parameter is our list of contours we found using the `cv2.findContours` function.

The third parameter is the *index* of the contour inside the `cnts` list that we want to draw. If we wanted to draw **only** the first contour, we could pass in a value of *0*. If we wanted to draw **only** the second contour, we would supply a value of *1*. Passing in a value of *-1* for this argument instructs the `cv2.drawContours` function to draw ***all*** contours in the `cnts` list. Personally, I like to always supply a value of *-1* and just supply the single contour manually. Doing this is slightly more Pythonic and easier to read — I'll make this point clear later in this lesson.

Finally, the last two arguments to the `cv2.drawContours` function is the *color* of the contour (in this case green), and the thickness of the contour line (2 pixels).

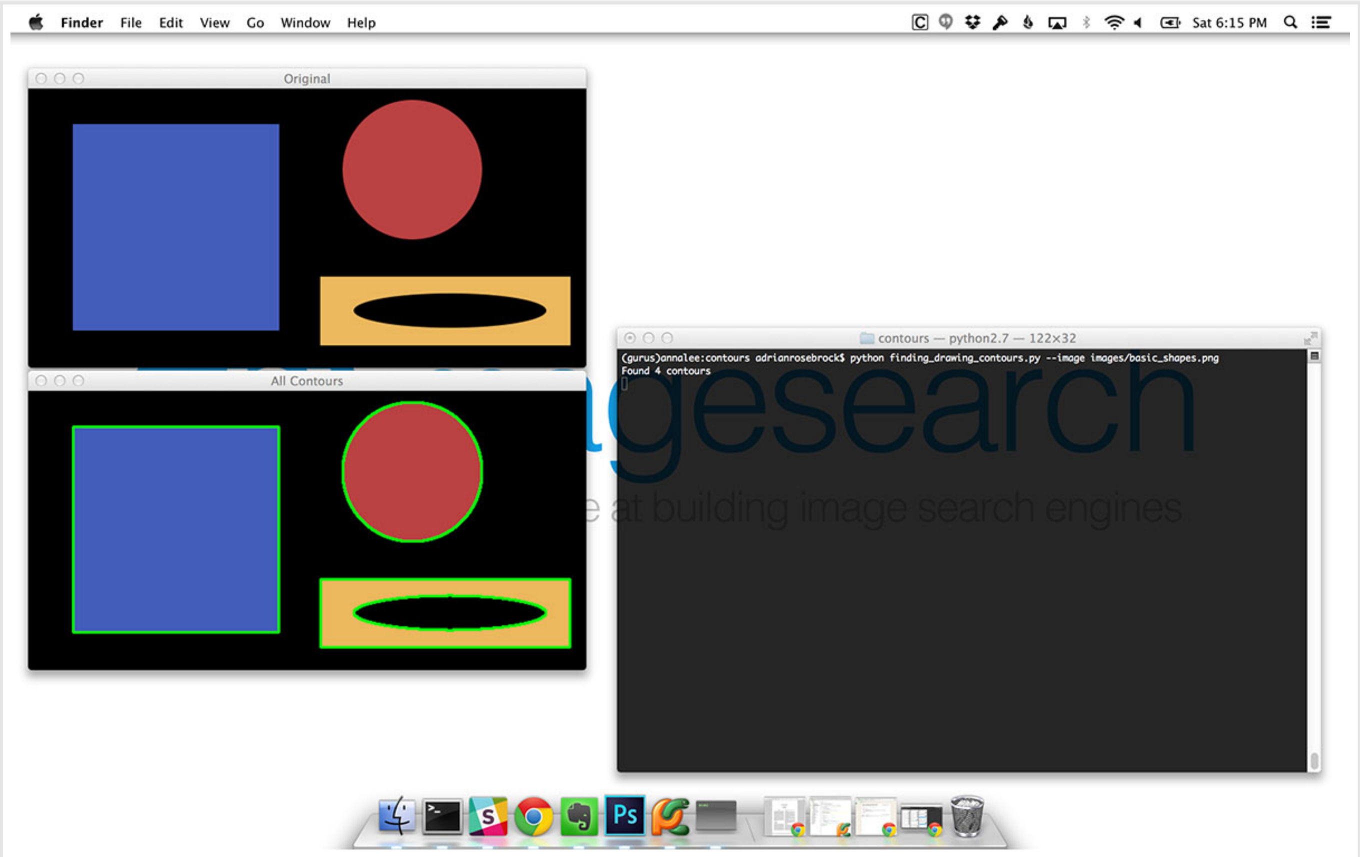
We then print out the number of contours we found on **Line 24** — the `cnts` variable is simply a Python list, so we can call `len` on it to get the number of contour entries in the list.

Then we display our image with detected contours to our screen on **Line 27**.

If we were to open up our command line and execute our script:

```
finding_and_drawing_contours.py                                                    Shell
$ python finding_drawing_contours.py --image images/basic_shapes.png
```

We would see something like this:



Finder File Edit View Go Window Help
Sat 6:15 PM
Original
All Contours
contours — python2.7 — 122×32

```
(gurus)annalee:contours adrianrosebrock$ python finding_drawing_contours.py --image images/basic_shapes.png
Found 4 contours
```

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

FIGURE 1: FINDING AND DRAWING ALL CONTOURS IN AN IMAGE.

On the *top* we have our original input image. We have three shapes: a square, a circle, and a rectangle with an ovular region cut out from the center of it.

And on the *bottom* we have our detected contours drawn in *green*. Notice how the rectangle has been detected. And the circle. And the rectangle.

But perhaps most interestingly, the **cutout region inside the rectangle has been detected as well!**

You may be wondering, is this the intended behavior of `cv2.drawContours` ? I mean, the oval region is *black*, so shouldn't it be part of the background?

Perhaps. But also consider the *contrast* of the orange rectangle. Clearly we see the orange rectangle on the black background. But we could also interpret the oval as a *black shape* on an *orange background*. We may perceive the rectangle as having its center cutout — but there is still a black oval on an orange background according to the `cv2.findContours` function.

Feedback

Also take a second to inspect the output on our terminal — our script has indeed found 4 contours. One contour for the square. One contour for the circle. One for the rectangle. And one contour for the oval inside the rectangle.

But what if we *didn't* want to detect that oval region? What if we are only interested in the *external* shapes? Can we do it?

Of course we can! We'll discuss detecting external contours soon, but first I think it's important to explore how to access each *individual* contour:

finding_and_drawing_contours.py　　Python

```python
30 # re-clone the image and close all open windows
31 clone = image.copy()
32 cv2.destroyAllWindows()
33
34 # loop over the contours individually and draw each of them
35 for (i, c) in enumerate(cnts):
36     print("Drawing contour #{}".format(i + 1))
37     cv2.drawContours(clone, [c], -1, (0, 255, 0), 2)
38     cv2.imshow("Single Contour", clone)
39     cv2.waitKey(0)
```

Lines 31 and 32 are not very interesting, but I'll go ahead and explain them. Here we are simply re-cloning our input image so that we can draw on it without destroying the original. And as the name implies, the `cv2.destroyAllWindows` function closes all open windows generated by OpenCV.

To access each individual contour all we need is a **for** loop on **Line 35** where we loop over each of the contours. By using the built-in Python `enumerate` function we are also able to get the *index* of each contour along with the contour itself.

Line 36 then prints the contour number that we are drawing and **Line 37** takes care of the actual drawing.

Take a second to really examine **Line 37**.

Notice that I am supplying a value of *-1* for my contour index value (indicating that I want to draw ***all*** contours) and then wrapping the contour *c* as a list. Why am I doing this? Wouldn't `cv2.drawContours(clone, cnts, i, (0, 255, 0), 2)` be equivalent?

In fact, it would.

So why am I wrapping my contour as a list when I only want to draw one contour?

The reason is because specifying the contour index value other than *-1* inside the `cv2.drawContours` function is counter-productive.

Consider what would happen if I wanted to draw the ***first two*** contours and ***nothing else***. I would either have to manually make two calls to `cv2.findContours` or to use a `for` loop and supply the contour index value *i*.

Simply put, that's not very Pythonic when all I need to do is supply Python array slices:

```python
Drawing the first two contours                                                    Python
cv2.drawContours(clone, cnts[:2], -1, (0, 255, 0), 2)
```

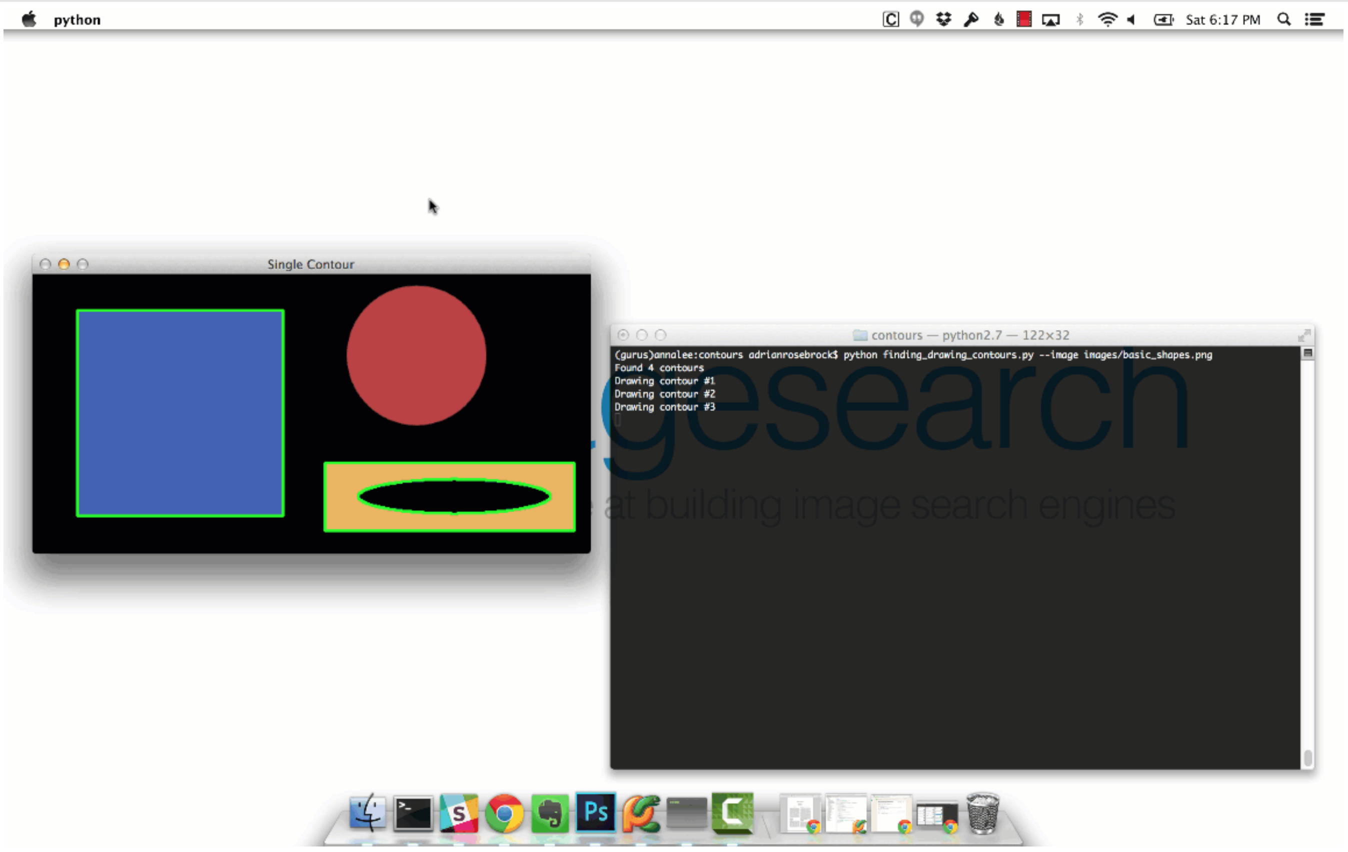
Now I have reduced what would have been 2-3 lines of code to only one — and furthermore, it's much easier to read.

In general, if you want to draw only a single contour, I would get in the habit of always supplying a value of *-1* for your contour index and then wrapping your single contour *c* as a list. If you get into this habit now, you'll be reinforcing the power of the Python programming language and side-stepping an OpenCV caveat that might trip you up.

Anyway, back to the example code.

Once we have drawn the individual contour, we display it on screen and wait for a keypress.

Our output would look something like this:



```
(gurus)annalee:contours adrianrosebrock$ python finding_drawing_contours.py --image images/basic_shapes.png
Found 4 contours
Drawing contour #1
Drawing contour #2
Drawing contour #3
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours-drawing-individual.gif)

FIGURE 2: DRAWING EACH *INDIVIDUAL* CONTOUR ONE-AT-A-TIME RATHER THAN ALL CONTOURS AT ONCE.

While drawing each individual contour one at a time doesn't seem very interesting or helpful, I simply wanted to introduce the notion of *accessing* single contours. When we start to explore contour properties, we'll be examining them one at a time, so it's good to expose ourselves to this concept now.

That wasn't so complicated, was it?

Let's move on and see if we can find only *external* contours and ignore the ovular region inside the orange rectangle:

finding_and_drawing_contours.py Python

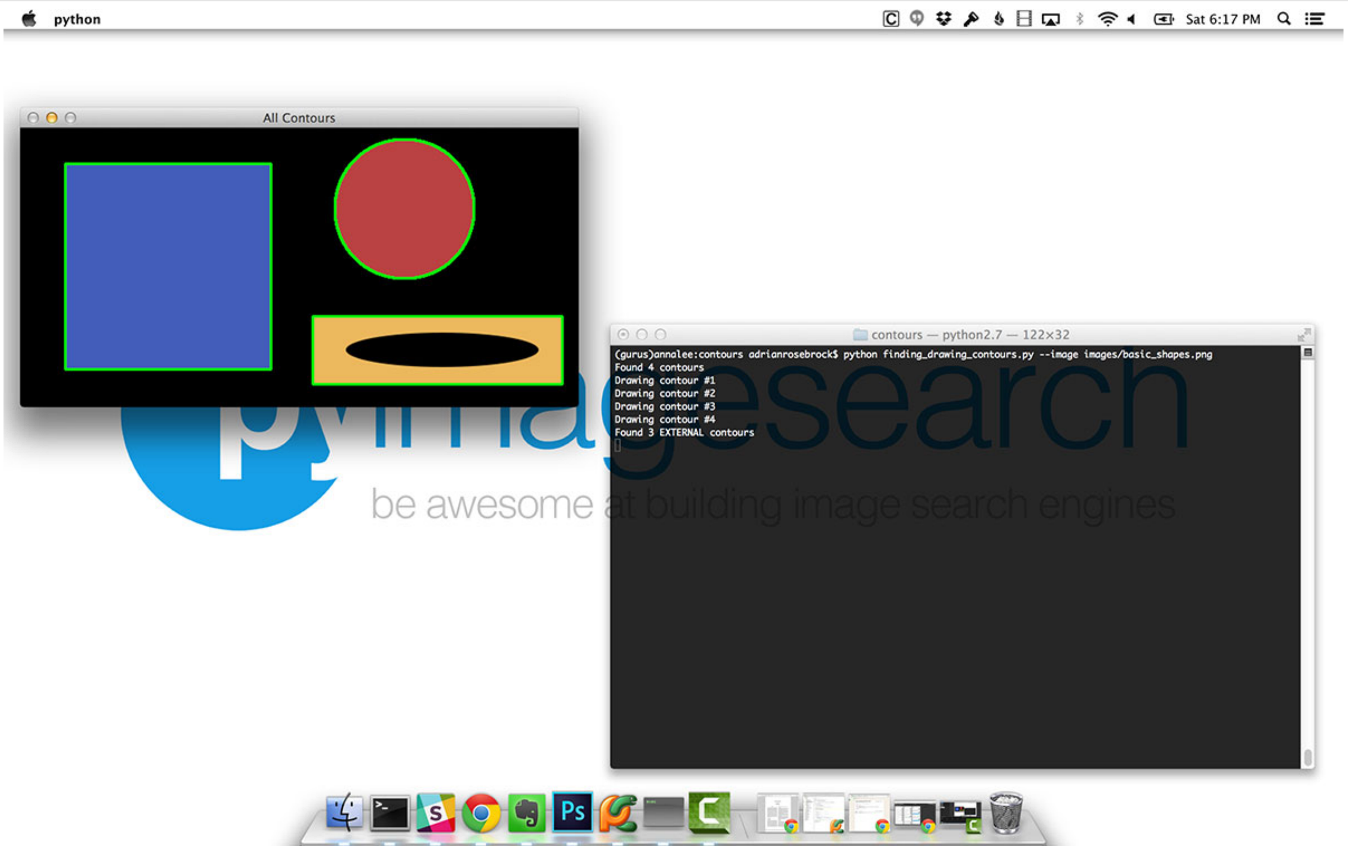
```python
# re-clone the image and close all open windows
clone = image.copy()
cv2.destroyAllWindows()

# find contours in the image, but this time keep only the EXTERNAL
# contours in the image
cnts = cv2.findContours(gray.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
cnts = cnts[0] if imutils.is_cv2() else cnts[1]
cv2.drawContours(clone, cnts, -1, (0, 255, 0), 2)
print("Found {} EXTERNAL contours".format(len(cnts)))

# show the output image
cv2.imshow("All Contours", clone)
cv2.waitKey(0)
```

Feedback

Our call to `cv2.findContours` on **Line 47** looks almost identical to our call to `cv2.findContours` on **Line 20** — but there is one major, important difference. This time we are supplying a value of `cv2.RETR_EXTERNAL` for the contour detection mode. Specifying this flag instructs OpenCV to return only the *external most* contours of each shape in the image, meaning that if one shape is enclosed in another, then the contour is ignored.

Returning only external contours is a technique that we'll exploit heavily in this course, so it's important for us to understand this concept.

Let's take a look at the output of detecting and drawing ***only*** external contours:


python
Sat 6:17 PM
All Contours
contours — python2.7 — 122×32
pyimagesearch
be awesome at building image search engines

```
(gurus)annalee:contours adrianrosebrock$ python finding_drawing_contours.py --image images/basic_shapes.png
Found 4 contours
Drawing contour #1
Drawing contour #2
Drawing contour #3
Drawing contour #4
Found 3 EXTERNAL contours
```

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

FIGURE 3: DETECTING ONLY *EXTERNAL SHAPES* IN OUR IMAGE. NOTICE HOW THE BLACK OVAL HAS BEEN SKIPPED SINCE IT IS FULLY ENCLOSED BY THE ORANGE RECTANGLE.

As you can see, we have only detected ***3 contours*** this time — not 4. And more importantly, notice how the black oval inside the rectangle has been ignored. The black oval was ignored because we specified that we wanted *external* contours only — and since the black oval is *fully enclosed* by the rectangle, the `cv2.findContours` function completely ignored it.

Pretty useful, right?

Finally, there is one more topic that I want to introduce — and one that we'll be using in nearly every single advanced concept inside the PyImageSearch Gurus: ***using both contours and masks together***.

For example, what if we wanted to access *just* the blue rectangle and ignore all other shapes? How would we do that?

The answer is that we loop over the contours individually, draw a mask for the contour, and then apply a bitwise AND:

```python
finding_and_drawing_contours.py                                         Python
56 # re-clone the image and close all open windows
57 clone = image.copy()
58 cv2.destroyAllWindows()
59
60 # loop over the contours individually
61 for c in cnts:
62     # construct a mask by drawing only the current contour
63     mask = np.zeros(gray.shape, dtype="uint8")
64     cv2.drawContours(mask, [c], -1, 255, -1)
65
66     # show the images
67     cv2.imshow("Image", image)
68     cv2.imshow("Mask", mask)
69     cv2.imshow("Image + Mask", cv2.bitwise_and(image, image, mask=mask))
70     cv2.waitKey(0)
```

On **Line 61** we again start looping over each of the (external) contours individually.

We then create an empty NumPy array with the same dimensions of our original image. This empty NumPy array will serve as the `mask` for the current shape that we want to examine.

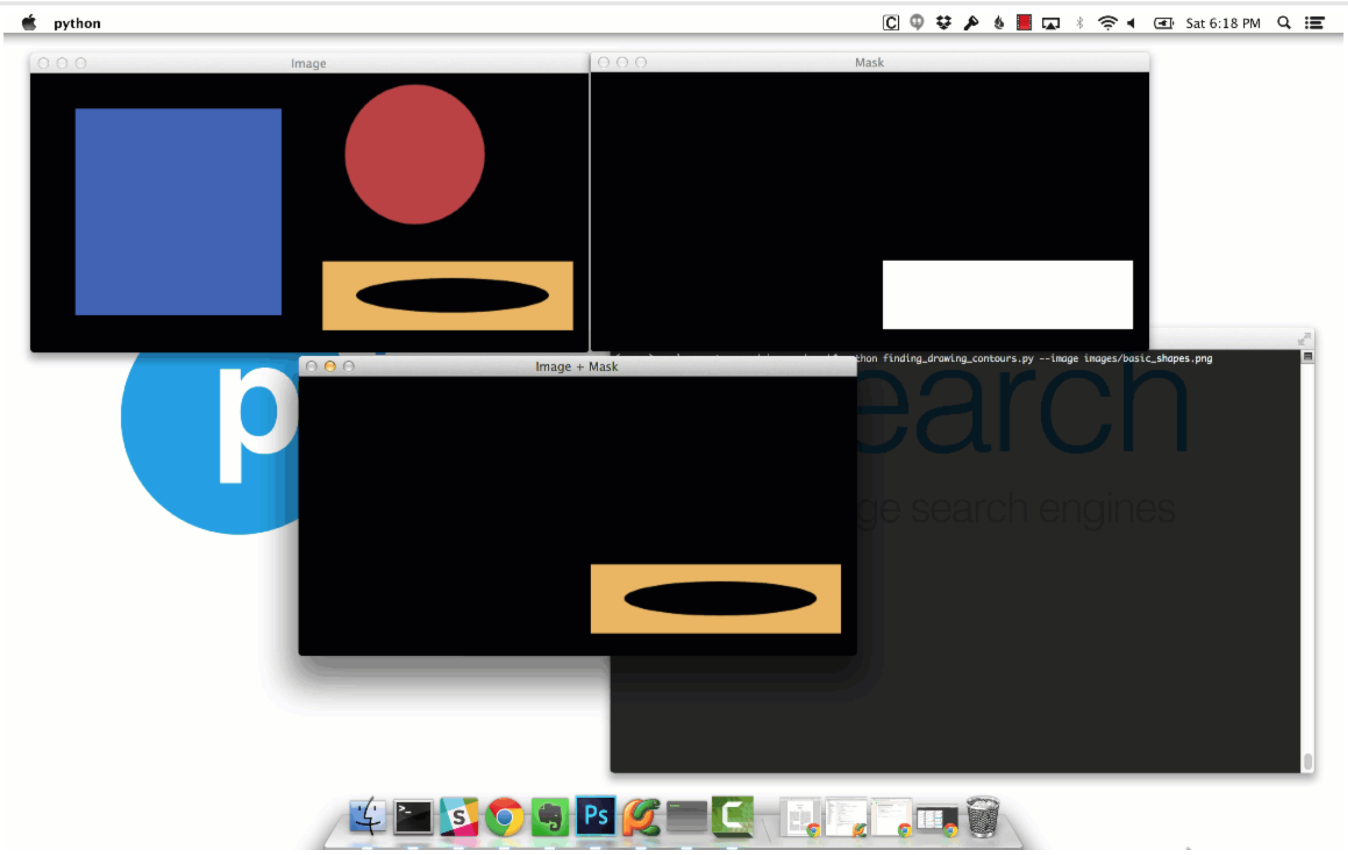
Now that we have initialized our `mask`, we can draw the contour on the mask on **Line 64**. Notice how I only supplied a value of *255* (white) for the color here — but isn't this incorrect? Isn't white represented as *(255, 255, 255)*?

Yep. You're right. White is represented by *(255, 255, 255)*, but ***only*** if we are working with a RGB image. In this case we are working with a mask that has only a single (grayscale) channel — thus only need to supply a value of *255* to get white.

Lines 67 and 68 show our original image and mask. **Line 69** takes the bitwise AND between the input image and the mask to *hide* all other shapes but the one we are interested in.

If this seems confusing to you, recall **Module 1.4.8 (https://gurus.pyimagesearch.com/topic/masking/)** where we learned about bitwise operations. A bitwise AND is true only if both input pixels are greater than zero. By supplying a `mask` to the `cv2.bitwise_and` function we are telling OpenCV to only investigate regions of the image where the corresponding mask is non-zero.

This point becomes particularly clear when we look at our output:



```
thon finding_drawing_contours.py --image images/basic_shapes.png
```

[(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contour_with_masks.gif)](https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contour_with_masks.gif)

FIGURE 4: ACCESSING ONLY A SINGLE SHAPE AT A TIME, HIDING THE REST.

On the *top-left* we have our original input image. And on the *top-right* we have the mask of the current shape. By applying a bitwise AND to the input `image` using our `mask` region, we can see that *all other* shapes are hidden *except* for the current shape.

This is an **incredibly useful** technique that we'll be utilizing heavily in the PyImageSearch Gurus course.

Just take a second to really contemplate how powerful masking and accessing single contours at a time really is: we'll need it for accessing the individual letters and numbers on license plates to build our Automatic License Plate Identification system. We'll need contours and masks to access handwritten characters and digits to recognize them. And as you'll see later in this module, we'll be utilizing contours and masks to identify Tetris blocks.

And with that in mind, let's start exploring various *properties* of contours which allow us to accomplish the projects detailed in the previous paragraph.

Summary

In this introductory lesson to contours we learned how to *find and detect contours* in an image, along with how to actually *extract contoured regions* from an image.

Feedback

In the next topic we'll discuss basic, simple properties of contours.

Downloads:

Download the Code
(https://gurus.pyimagesearch.com/protected/code/computer_vision_basics/fin

Quizzes		Status
1	Finding and Drawing Contours Quiz (https://gurus.pyimagesearch.com/quizzes/finding-and-drawing-contours-quiz/)	

Next Topic → (https://gurus.pyimagesearch.com/topic/simple-contour-properties/)

Feedback

Course Progress

Ready to continue the course?

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

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

Resources & Links

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

Your Account

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

Feedback

Search