1.11.2: Simple contour properties

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/)

In the previous section of this lesson we learned about the basics of contours. We learned how to *find* them in images, and we learned how to *draw* our contours: both all at once and at a time. We also explored how *masking* can aid us in accessing only single shapes of images at a time.

But outside of simply finding and drawing contours, there exists a fairly extensive set of properties that we can use to quantify and represent the shape of an object in an image.

In this lesson we'll be reviewing what I call "simple" contour properties. These properties are fairly intuitive to understand and are quite easily visually explained.

Objectives:

By the end of this lesson you should be able to compute various properties of objects using contours, including:

1. Centroid/Center of Mass
2. Area
3. Perimeter

4. Bounding boxes
5. Rotated bounding boxes
6. Minimum enclosing circles
7. Fitting an ellipse

Simple Contour Properties

Like I mentioned above, there is a fairly extensive list of properties we can compute for contours. Most of these properties are best explained via example, so let's go ahead and jump right in.

Centroid/Center of Mass

The "centroid" or "center of mass" is the center *(x, y)*-coordinate of an object in an image. This *(x, y)*-coordinate is actually calculated based on the *image moments*, which are based on the weighted average of the *(x, y)*-coordinates/pixel intensity along the contour.

While we haven't discussed moments just yet in this course (we'll get to them in **Module 10** (https://gurus.pyimagesearch.com/lessons/what-are-image-descriptors-feature-descriptors-and-feature-vectors/) when we start discussing image descriptors), just understand that moments allow us to use basic statistics to represent the structure and shape of an object in an image. However, if you're curious about reading more about image moments right now, I would suggest taking a look at this page (http://en.wikipedia.org/wiki/Image_moment).

The centroid calculation itself is actually very straightforward: it's simply the mean (i.e. average) position of all *(x, y)*-coordinates along the contour of the shape.

If you were to imagine computing the centroid yourself, you would simply walk along the points of the outline and average their *(x, y)*-coordinates together. Pretty easy, right?

To make this point more clear, let's take a peek at how we can compute the center *(x, y)*-coordinate of a shape in an image:

contour_properties_1.py — Python

```
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 # find external contours in the image
17 cnts = cv2.findContours(gray.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
18 cnts = cnts[0] if imutils.is_cv2() else cnts[1]
19 clone = image.copy()
20
21 # loop over the contours
22 for c in cnts:
23     # compute the moments of the contour which can be used to compute the
24     # centroid or "center of mass" of the region
25     M = cv2.moments(c)
26     cX = int(M["m10"] / M["m00"])
27     cY = int(M["m01"] / M["m00"])
28
29     # draw the center of the contour on the image
30     cv2.circle(clone, (cX, cY), 10, (0, 255, 0), -1)
31
32 # show the output image
33 cv2.imshow("Centroids", clone)
34 cv2.waitKey(0)
35 clone = image.copy()
```


Lines 2-9 are fairly standard code: we are just importing our necessary packages, setting up our argument parser, which will accept a single switch, `--image` , the path to our image on disk.

Our image is then loaded off disk on **Line 13** and converted to grayscale on **Line 14**. Given our grayscale image, we then find the contoured regions on **Line 17**.

Computing contour properties is done on *only a single contour at a time* — thus we start looping over our detected contours on **Line 22**.

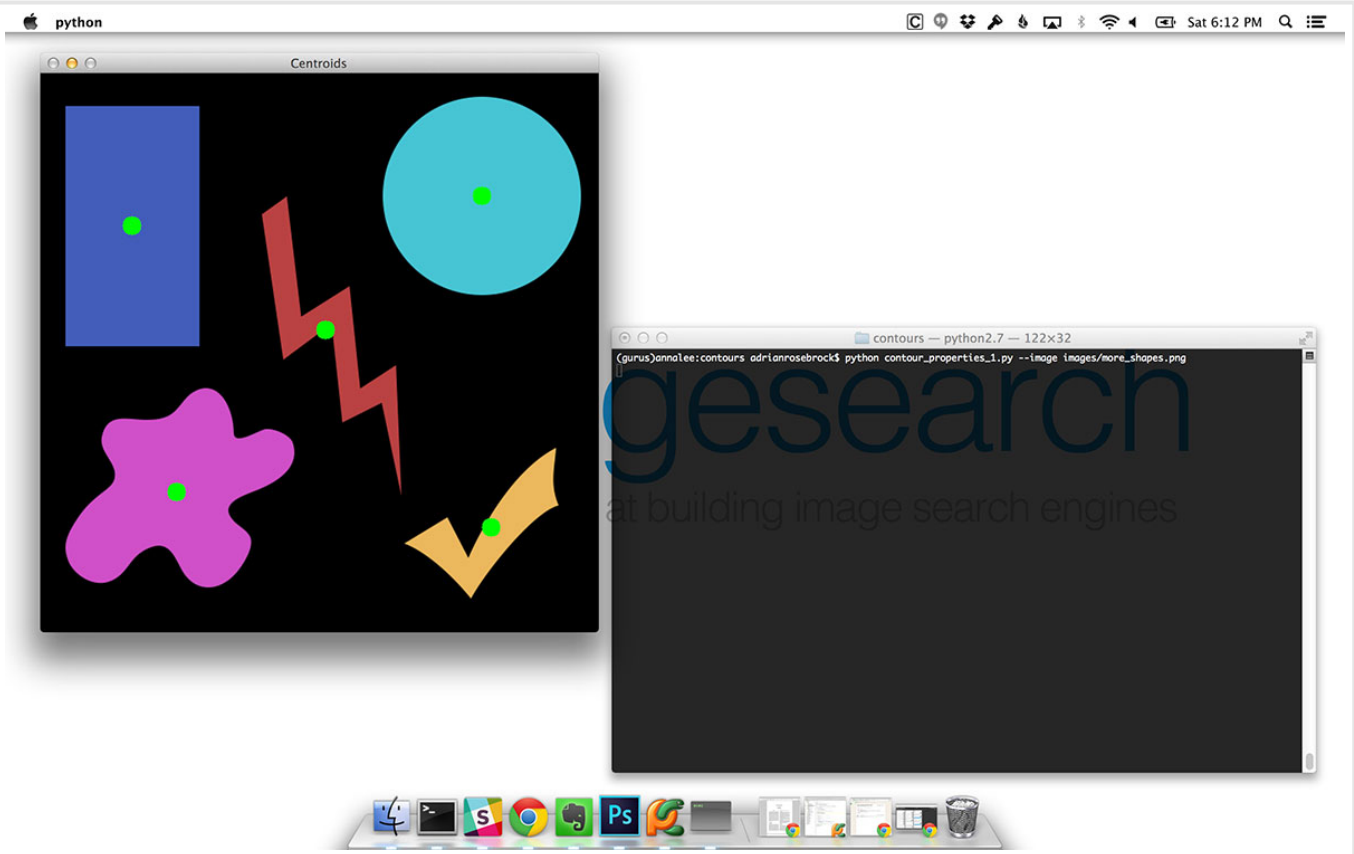
Using the `cv2.moments` (**Line 25**) function we are able to compute the center *(x, y)*-coordinate of the shape the contour represents (**Lines 26 and 27**). This function returns a dictionary of moments with the *keys* of the dictionary as the moment number and the *values* as the actual calculated moment. Again, we'll be covering moments in more detail in **Module 10** of this course. For the time being, simply understand that we are using image moments to compute the center region of the shape.

Finally, we draw the center *(x, y)*-coordinate on our image and display it on screen.

We can execute our Python script thus far using this command:

```
contour_properties_1.py                                                    Shell
1 $ python contour_properties_1.py --image images/more_shapes.png
```

Assuming there are no errors, we can see our output image:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_props1_centroids.jpg)

FIGURE 1: COMPUTING THE CENTROID/CENTER OF MASS FOR EACH SHAPE IN THE IMAGE.

For each shape in the image above we have been able to successfully compute and draw the center (*x*, *y*)-coordinates.

Area and Perimeter

The area and perimeter of a contour is also very much straightforward.

The **area** of the contour is the number of pixels that reside inside the contour outline. Similarly, the **perimeter** (sometimes called **arc length**) is the length of the contour.

Let's see how we can compute the area and perimeter:

```
contour_properties_1.py                                                    Python
```

Feedback

```
37 # loop over the contours again
38 for (i, c) in enumerate(cnts):
39     # compute the area and the perimeter of the contour
40     area = cv2.contourArea(c)
41     perimeter = cv2.arcLength(c, True)
42     print("Contour #{} -- area: {:.2f}, perimeter: {:.2f}".format(i + 1, area, perimeter))
43 
44     # draw the contour on the image
45     cv2.drawContours(clone, [c], -1, (0, 255, 0), 2)
46 
47     # compute the center of the contour and draw the contour number
48     M = cv2.moments(c)
49     cX = int(M["m10"] / M["m00"])
50     cY = int(M["m01"] / M["m00"])
51     cv2.putText(clone, "#{}".format(i + 1), (cX - 20, cY), cv2.FONT_HERSHEY_SIMPLEX,
52         1.25, (255, 255, 255), 4)
53 
54 # show the output image
55 cv2.imshow("Contours", clone)
56 cv2.waitKey(0)
```

On **Line 38** we start to loop over the contours.

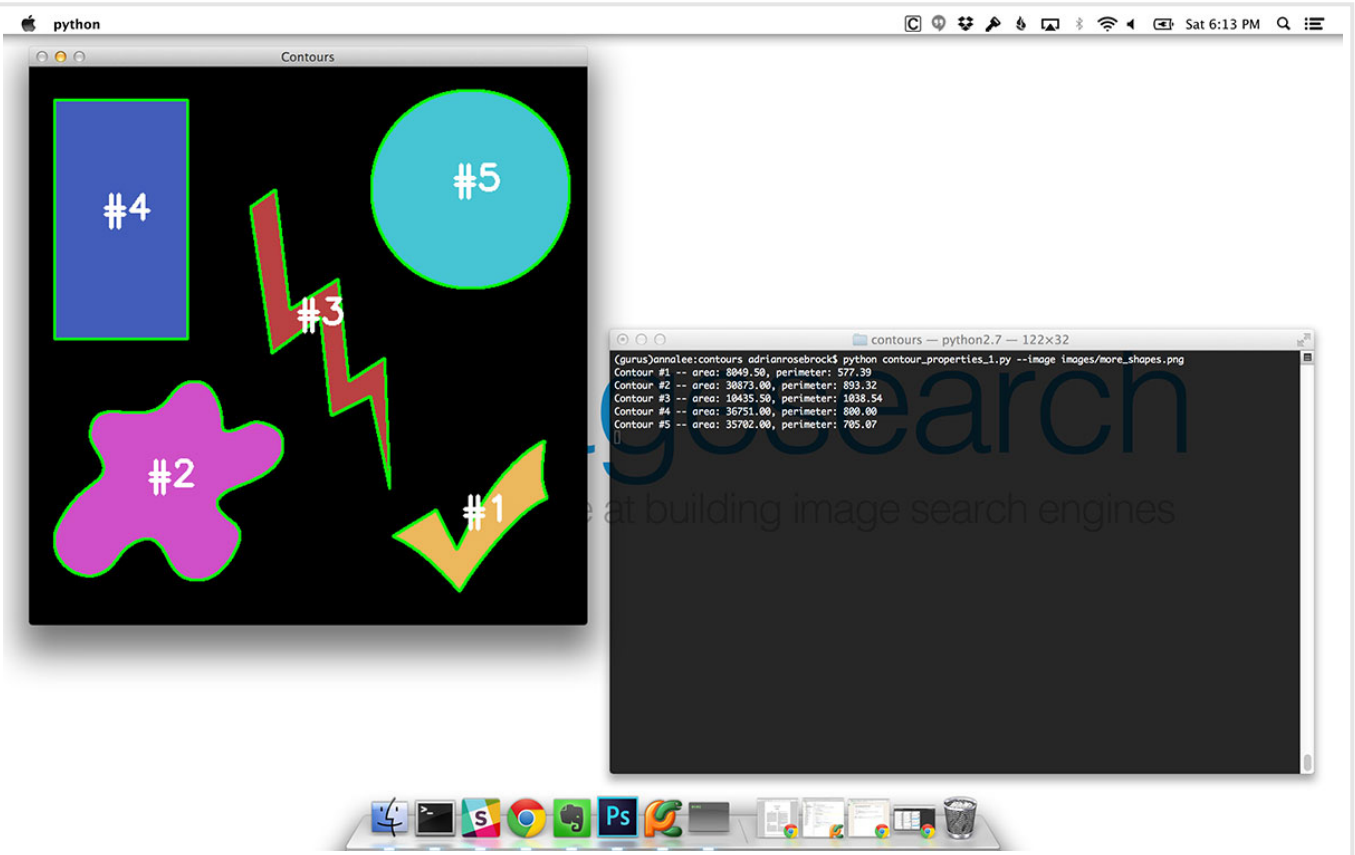
Computing the area of the contour is accomplished using the `cv2.contourArea` function on **Line 40**. This function takes only a single argument: the contour that we want to compute the area for.

We then compute the perimeter of the contour on **Line 41** using the `cv2.arcLength` function. This function takes two arguments: the contour itself along with a flag that indicates whether or not the contour is "closed." A contour is considered closed if the shape outline is *continuous* and there are no "holes" along the outline. In most cases, you'll be setting this flag to `True`, indicating that your contour has no gaps.

We then print both contour number, area, and perimeter to our terminal on **Line 42** so we can investigate the area and perimeter for various shapes.

Line 45 draws the current contour for us, while **Lines 48-52** compute the centroid of the image and displays the contour number on the image so we can associate the shape with the terminal output.

Our output ends up looking like this:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_props1_contours.jpg)

FIGURE 2: COMPUTING THE AREA AND PERIMETER OF OUR SHAPES.

Take a second to match up the shapes with the terminal output. Notice how the rectangle (shape #4) has the largest area. This makes sense, as it is clearly the largest shape in the image. The contour area can be used to quantify the shape of an object in an image — but it's mostly used as a component in the calculation of more advanced contour properties, which we'll explore later in this lesson.

However, what I find most interesting is shape #3, the lightning bolt. While the area itself is quite small, it has the largest perimeter, indicating that while the surface area of the shape is small, the distance around the entire shape is the largest of all the shapes in the image. Just like the contour area, we can use the perimeter as a method to quantify the shape of an object. However, the perimeter has a more important role to play when we explore *contour approximation* in a few sections.

Bounding Boxes

A bounding box is exactly what it sounds like — an upright rectangle that "bounds" and "contains" the entire contoured region of the image. However, it does not consider the *rotation* of the shape, so you'll want to keep that in mind.

A bounding box consists of four components: the starting *x*-coordinate of the box, then the starting *y*-coordinate of the box, followed by the width and height of the box.

Let's go ahead and compute the bounding box of each of the shapes in our image:

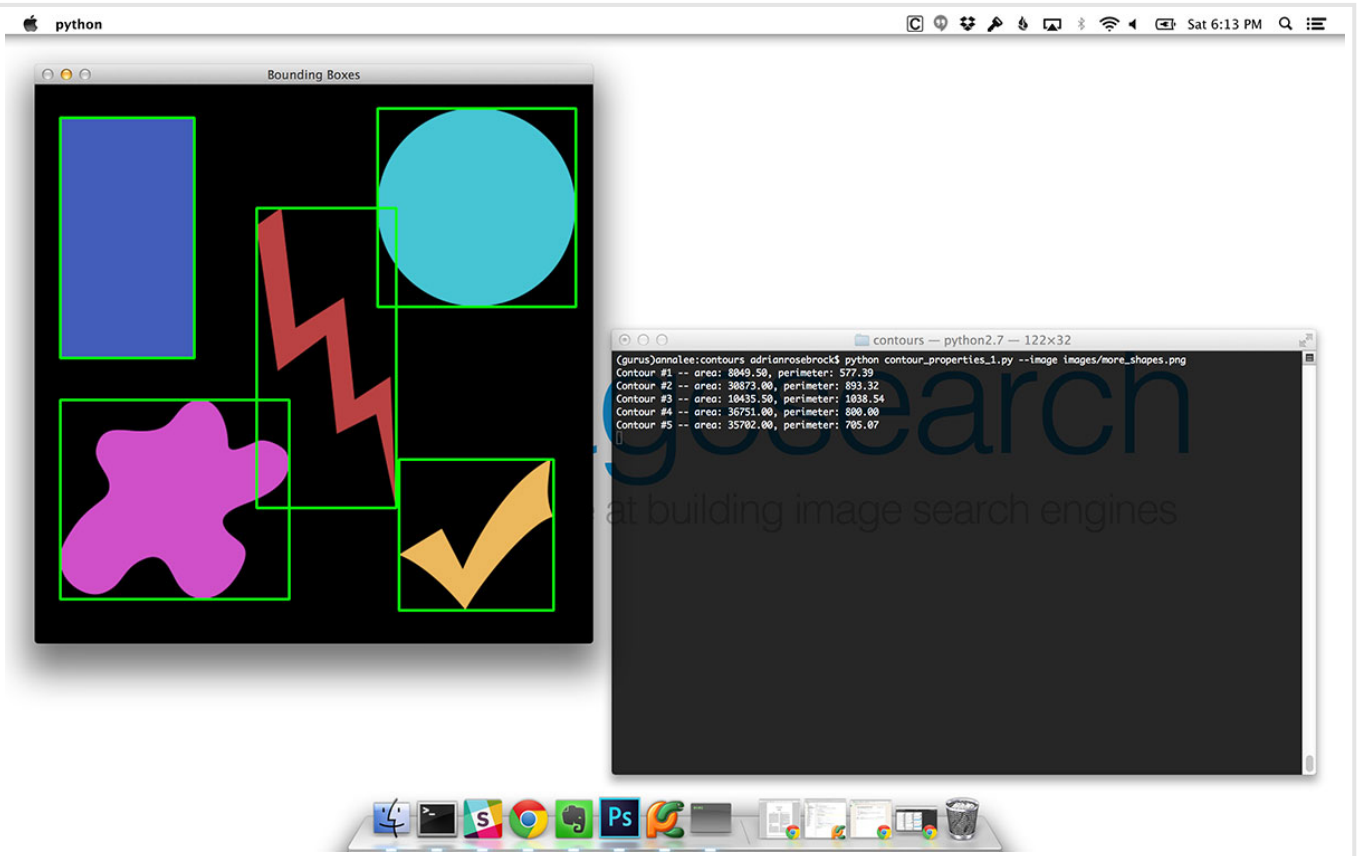
contour_properties_1.py Python

```
58 # clone the original image
59 clone = image.copy()
60
61 # loop over the contours
62 for c in cnts:
63     # fit a bounding box to the contour
64     (x, y, w, h) = cv2.boundingRect(c)
65     cv2.rectangle(clone, (x, y), (x + w, y + h), (0, 255, 0), 2)
66
67 # show the output image
68 cv2.imshow("Bounding Boxes", clone)
69 cv2.waitKey(0)
70 clone = image.copy()
```

For each of our contours in our list, we'll make a call to the `cv2.boundingRect` function on **Line 64**. This function expects only a single parameter: the contour *c* that we want to compute the bounding box for.

We then take the returned starting *(x, y)*-coordinates, the width, and the height of the bounding box, and use the `cv2.rectangle` function to draw our bounding box.

The output of drawing the bounding boxes can be seen below:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_props1_bounding_boxes.jpg)

FIGURE 3: DRAWING THE BOUNDING BOXES SURROUNDING EACH SHAPE.

As you can see, bounding boxes are simple and straightforward — take an object and compute the "box" that it fits into.

Rotated Bounding Boxes

While simple bounding boxes are great, they do not take into the consideration the *rotation* of the shape in an image. To account for rotation, we'll need to extend our approach:

contour_properties_1.py Python

```
72 # loop over the contours
73 for c in cnts:
74     # fit a rotated bounding box to the contour and draw a rotated bounding box
75     box = cv2.minAreaRect(c)
76     box = np.int0(cv2.cv.BoxPoints(box) if imutils.is_cv2() else cv2.boxPoints(box))
77     cv2.drawContours(clone, [box], -1, (0, 255, 0), 2)
78
79 # show the output image
80 cv2.imshow("Rotated Bounding Boxes", clone)
81 cv2.waitKey(0)
82 clone = image.copy()
```

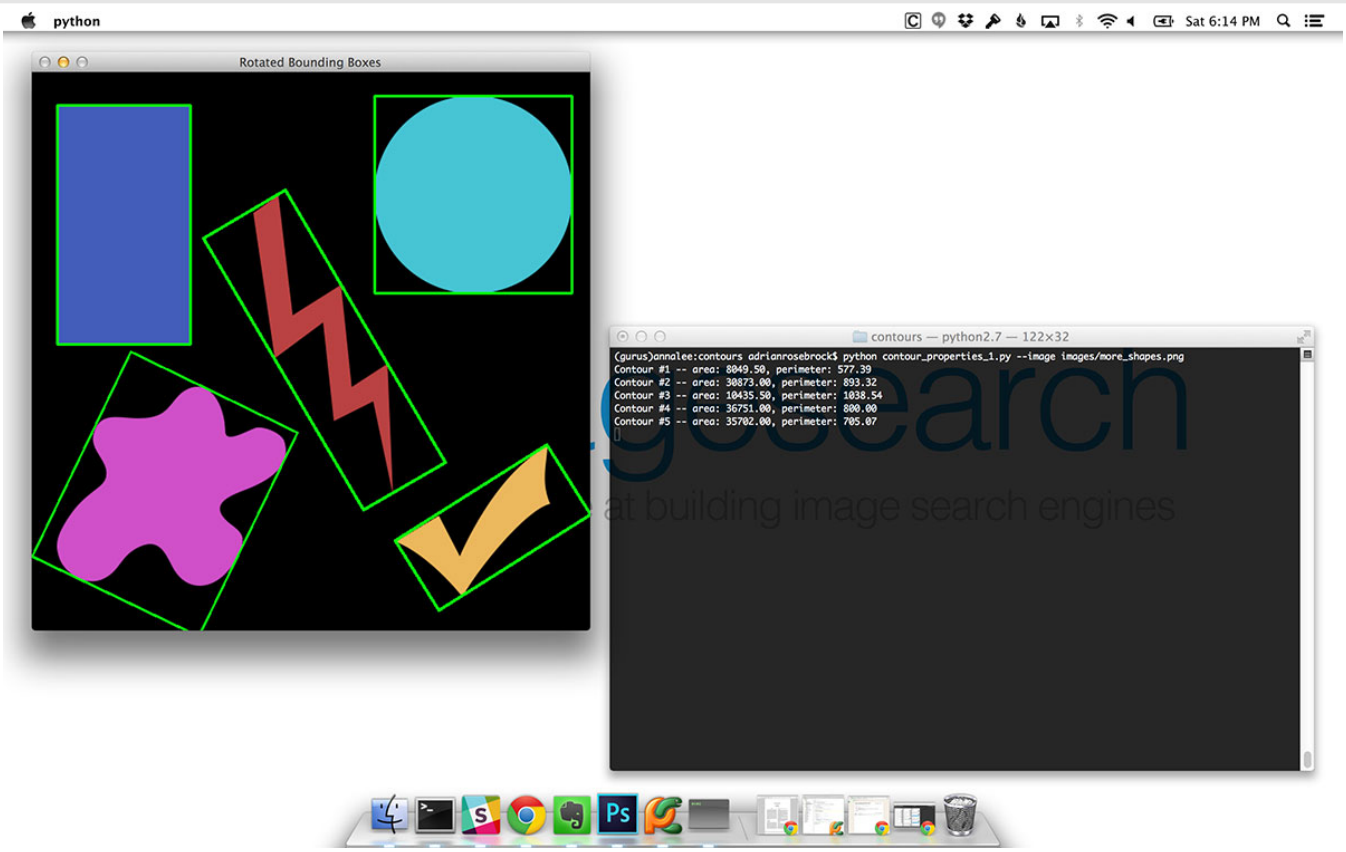
Computing the rotated bounding box requires two OpenCV functions: `cv2.minAreaRect` and `cv2.cv.BoxPoints`.

The `cv2.minAreaRect` (**Line 75**) function takes our contour and returns a tuple with 3 values. The first value of the tuple is the starting *(x, y)*-coordinates of the rotated bounding box. The second value is the width and height of the bounding box. And the final value is our θ, or angle of rotation of the shape.

However, since we want to draw a *rotated* bounding box rather than a *standard* bounding box we won't be able to leverage the `cv2.rectangle` function. Instead, we'll pass the output of `cv2.minAreaRect` to the `cv2.boxPoints` (**Line 76**) function (`cv2.cv.BoxPoints` for OpenCV 2.4) which converts the *(x, y)*-coordinates, width and height, and angle of rotation into a set of coordinates points.

In essence, the `cv2.minAreaRect` function just gives us another **contour**, which we then draw on our image on **Line 77**.

Below you can see the output of computing the rotated bounding boxes:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/contours_props1_rotated_bounding_boxes.jpg)

FIGURE 4: COMPUTING *ROTATED* BOUNDING BOXES RATHER THAN *STANDARD* ONES.

In general, you'll want to use *standard* bounding boxes when you want to crop a shape from an image (**Module 1.4.5**) (https://gurus.pyimagesearch.com/topic/cropping/). And you'll want to use *rotated* bounding boxes when you are utilizing masks to extract regions from an image.

Minimum Enclosing Circles

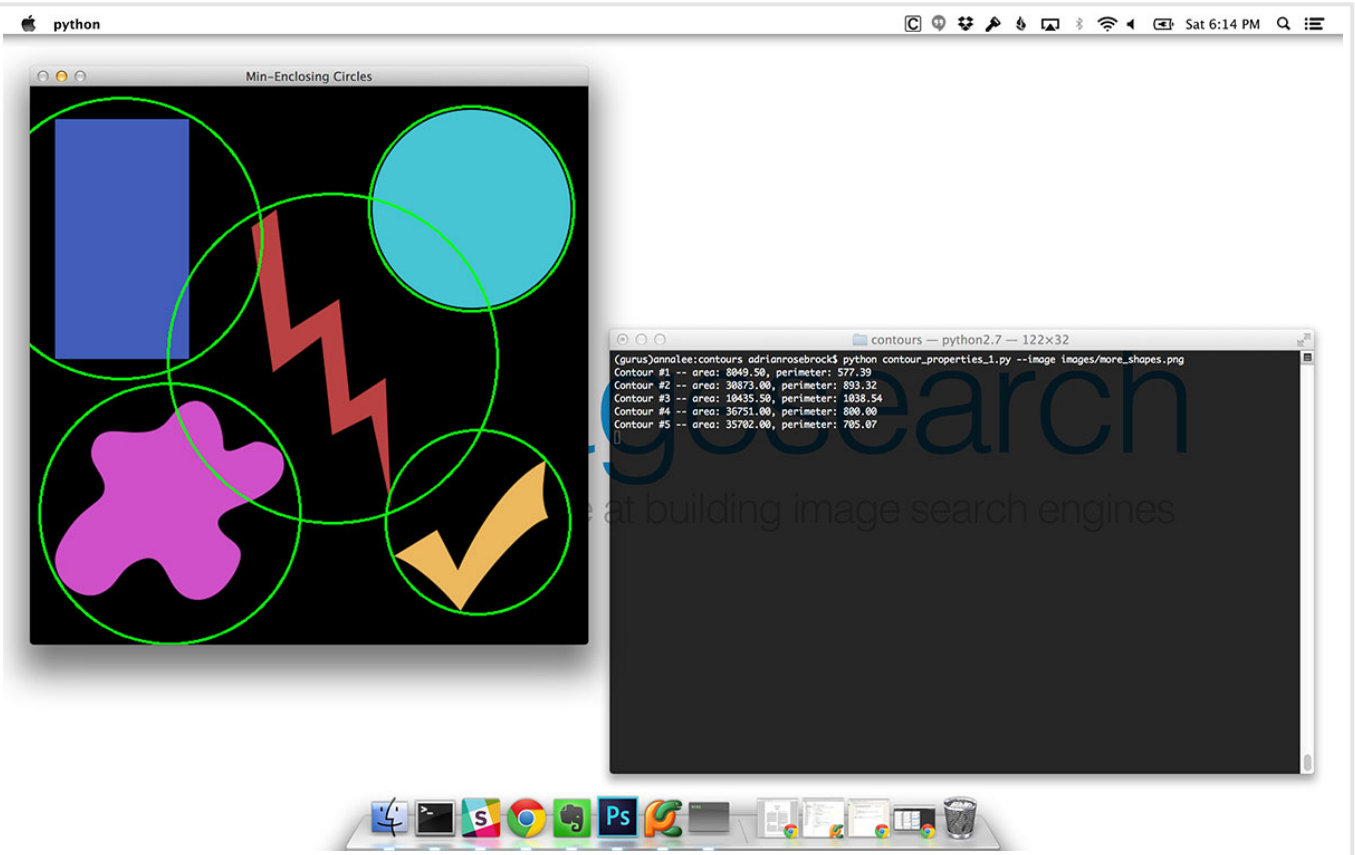
Just as we can fit a rectangle to a contour, we can also fit a circle:

contour_properties_1.py — Python

```
84 # loop over the contours
85 for c in cnts:
86     # fit a minimum enclosing circle to the contour
87     ((x, y), radius) = cv2.minEnclosingCircle(c)
88     cv2.circle(clone, (int(x), int(y)), int(radius), (0, 255, 0), 2)
89
90 # show the output image
91 cv2.imshow("Min-Enclosing Circles", clone)
92 cv2.waitKey(0)
93 clone = image.copy()
```

The most important function to take note of here is `cv2.minEnclosingCircle`, which takes our contour and returns the *(x, y)*-coordinates of the **center** of circle along with the **radius** of the circle.

Now that we have the center and the radius, it's fairly simple to utilize the `cv2.circle` function to draw the minimum enclosing circle of the contour:



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

FIGURE 5: THE MINIMUM ENCLOSING CIRCLE FOR EACH SHAPE IN THE IMAGE.

Notice that even though the lightning bolt shape has a very small area, it has a large perimeter — thus it has a large minimum enclosing circle to fit the entire shape into the circle region.

Fitting an Ellipse

Fitting an ellipse to a contour is much like fitting a rotated rectangle to a contour.

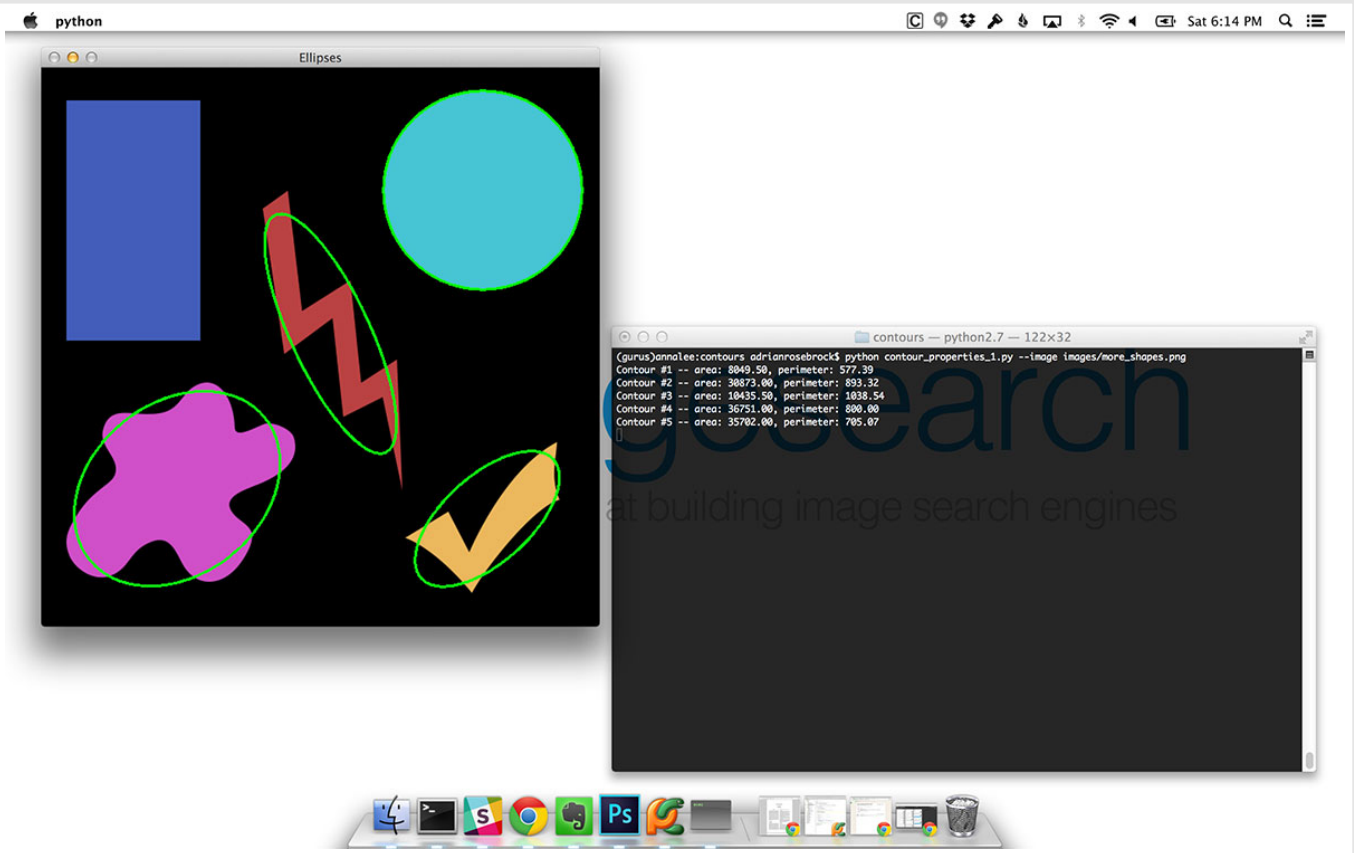
Under the hood, OpenCV is computing the rotated rectangle of the contour. And then it's taking the rotated rectangle and computing an ellipse to fit in the rotated region:

contour_properties_1.py — Python

```
95  # loop over the contours
96  for c in cnts:
97      # to to fit an ellipse, our contour must have at least 5 points
98      if len(c) >= 5:
99          # fit an ellipse to the contour
100         ellipse = cv2.fitEllipse(c)
101         cv2.ellipse(clone, ellipse, (0, 255, 0), 2)
102
103 # show the output image
104 cv2.imshow("Ellipses", clone)
105 cv2.waitKey(0)
```

It's important to take note of **Line 98**. A contour must have ***at least*** 5 points for an ellipse to be computed — if a contour has less than 5 points, then an ellipse cannot be fit to the rotated rectangle region.

The actual ellipse is fit to the shape on **Line 100** using the `cv2.fitEllipse` function, which we then pass on the `cv2.ellipse` (**Line 101**) to draw the enclosing region:



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

FIGURE 6: FITTING THE MINIMUM ENCLOSING ELLIPSE TO OUR SHAPES.

Notice that since a rectangle only has 4 points, we cannot fit an ellipse to it — but for all other shapes, we have > 5 points so this is not a concern.

This rounds out our simple contour properties. By far, you'll be using the standard bounding box the most in your own computer vision applications. There are also times where the rotated bounding box is used, but not nearly as much as the standard bounding box.

The perimeter/arc length is also used a lot, but normally only in the context of contour approximation (which we'll cover towards the end of this lesson).

Personally, I rarely use minimum enclosing circles and the minimum enclosing ellipse, simply because I do not have much of a use for them. You may find in your own applications that they can be useful, however, which is why I included it here — just in case.

In the upcoming lesson we'll be looking at some of the more advanced contour properties that we can compute. These contour properties are much more powerful than the ones we have looked at so far. Using these more advanced properties, we'll be able to identify X's and O's on a tic-tac-toe board, along with the different Tetris blocks — all by computing contour properties.

Summary

This lesson introduced us to the basic properties of contours.

We learned how to compute the *centroid/center of mass* of a contour, which is simply its center *(x, y)*-coordinate. We also learned how to compute the actual *area* of the contour, where the area is defined as the number of *white* pixels that appear inside the contour. From there we also explored the *perimeter* of the contour.

We then explored various approaches to detect and extract the region surrounding an object. These methods included: *bounding box*, *rotated bounding box*, *minimum enclosing circle*, and *fitting an ellipse*.

While these methods may seem arbitrary, by the next lesson you'll see how these properties build on each other and allow us to perform tasks such as *distinguishing between shapes* and *recognize various Tetris blocks*. All of a sudden these "arbitrary" contour properties will seem quite powerful!

Feedback

Downloads:

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

Quizzes		Status
1	Simple Contour Properties Quiz (https://gurus.pyimagesearch.com/quizzes/simple-contour-properties-quiz/)	

← Previous Topic (https://gurus.pyimagesearch.com/topic/finding-and-drawing-contours/) Next Topic → (https://gurus.pyimagesearch.com/topic/advanced-contour-properties/)

Upgrade to the *Instant Access Membership* to get **immediate access** to **every lesson** inside the PyImageSearch Gurus course for a one-time, upfront payment:

- ***100%, entirely self-paced***
- Finish the course in *less than 6 months*
- Focus on the lessons that *interest you the most*
- **Access the *entire* course** as soon as you upgrade

This upgrade offer will expire in **30 days, 18 hours**, so don't miss out — be sure to upgrade now.

Upgrade Your Membership! (https://gurus.pyimagesearch.com/register/pyimagesearch-gurus-instant-access-membership-fcff7b5e/)

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)