Rectangle-numbering.py

1) Importing important Libraries to solve this problem:

import cv2 → for image processing
import numpy → faster array/list processing
import math → for mathematical computation like 'sqrt'.

2) def stackImages(scale, imgArray)

input parameters

- scale → to scale & customize the size i.e. width and height of window.
- imgArray → It accepts list of '3 x K' shape (array)

where K = 1, 2, ...

- output → The images in the list are displayed in the window (i.e. single window).

3) Reading Image

- img = cv2.imread('shapedetector.jpg')

→ openCV works with BGR values. So, cv2.imread() will returns list of arrays with BGR values of each pixel of the image 'shapedetector.jpg'.

- imgContour = img.copy()

→ Original image in BGR format is duplicated and making it unchanged. We will change 'imgContour'.

4) imgBlur = cv2.GaussianBlur(img, (7, 7), 1)
imgGray = cv2.cvtColor(imgBlur, cv2.COLOR_BGR2GRAY)
imgCanny = cv2.Canny(imgGray, 20, 23)

→ For detecting shapes in an image, we need to use Canny() method of cv2 which detects the edges in an image.

→ Image input to Canny() must be blurred first & converted into GRAY in order to have smooth result with least number of noisy points. Play will different formats and kinds of images.

→ Also, values '20', '23' are threshold values in Hysteresis Thresholding. You can play with these values for smoothing your output images.

5) lineContours = getContours(imgCanny, imgContour, "line")

def getContours(img, imgContour, contourType)

→ This is the main function that does all processing & computations for achieving our main objective.
i.e. To Provide/Assign Numbers TO Rectangles on the basis of LENGTH of the LINE within it.

input parameters :

- img → takes canny Images with edges detected.
- imgContour → To make changes to the visualization of original image. i.e. Bounding boxes.
- contourTypes → Accepts "line" & "rectangle" values. Dont try with other shapes except line & rectangles. It doesn't work.

outputs :

→ Returns list of dictionary of line & rectangle contours.

→ Dictionary consists of center, size → (width, height), angle, box → coordinates of vertex of line or rectangle.

Remember, line is also a rectangle with small width.

Detailed Description of Function

'get Contours( )

→ cv2.findContours(img, cv2.RETR_EXTERNAL or cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPL

↳ findContours() method finds the contours & returns coordinates of contours with hierarchy.

Input parameters :

i) img → of which contours must be detecte
& returned its values.
It is our cannyImage.

ii) RETR_TREE
RETR_EXTERNAL
} Among various method of retrieval of contours, these are two of them.

- RETR_TREE → It retrieves all contours and creates a full family hierarchy list (i.e. parent, child contours.)
- RETR_EXTERNAL → It retrieves only the extreme outer contours.

So, what I am doing that,
As lines are within rectangles. So, using 'RETR_EXTERNAL', it could not retrieve line contours. So, I am using RETR_TRE for retrieving lines inside each rectangl

iii) CHAIN_APPROX_SIMPLE :
↳ It removes all redundant points & compresses the contour, thereby saving memory. It will return endpoints of contours.
As our contours are made up of strai ght lines, endpoints are enough for creati them.

→ Finding center, size & angle of contours.

- for i in range(0, len(contours))
  - peri = cv2.arcLength(contours[i], True)

arcLength() computes the perimeter of contour.

'True' means it considers the contour to be closed.

→ This 'peri' parameter will be used to calculate epsilon value for

  - approx = cv2.approxPolyDP(contours[i], 0.02 * peri, True)

We are using parameter 'peri' to approximate what type of shape of contour it is.

In order to that, we will use approximation of poly method & we will input our contour & provide resolution (i.e. '0.02')

These 'approx' array will have certain amount of points and based on these, we can determine shape of polynomials.

But these lines of codes are commented as I am dealing with rectangle only (line is also a rectangle with very small width)

Just for knowledge purpose, I am explaining about it.

cv2.contourArea(contours[i])
↳ returns area of contour.

cv2.minAreaRect(contour[i])
↳ coordinates of bounding box with covering minimum area.
↳ also returns angle.

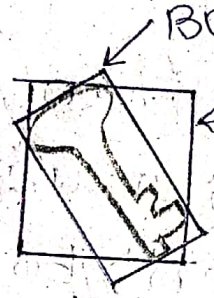
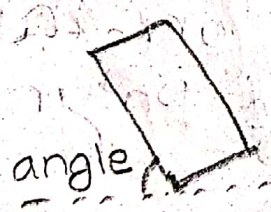
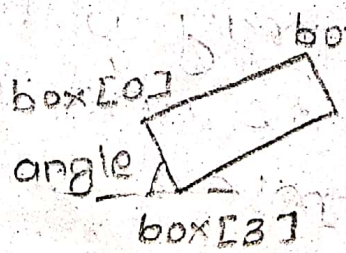


fig. minAreaRect()

rect = cv2.minAreaRect(contour[i])
center = rect[0]
size = rect[1]
angle = ~~angle~~ rect[2]
} coordinates of Bounding Box.
len(rect) = 3





Through my analysis,
angle = angle made by box[3] & box[0] points joining line with horizontal line through point box[3].

What is box ?

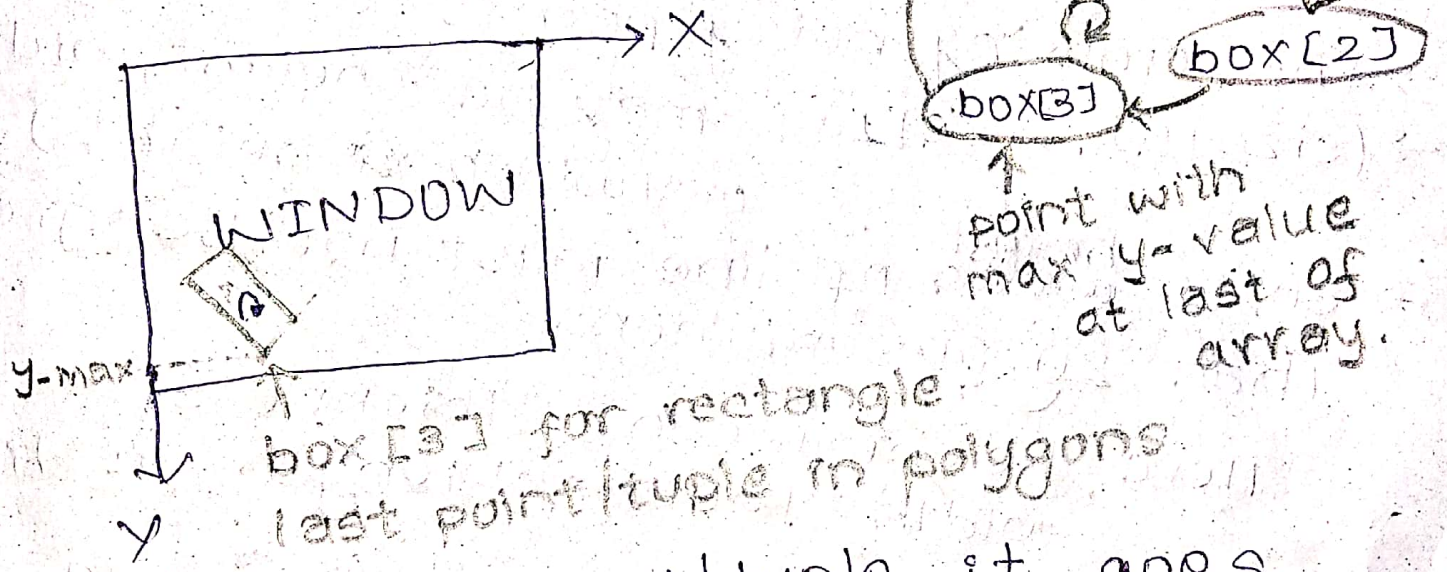
box = cv2.boxPoints(rect)

↓ computes vertices of rectangle.

As it is rectangle, it consists of 4 points

box[0] box[1]
box[3] box[2]

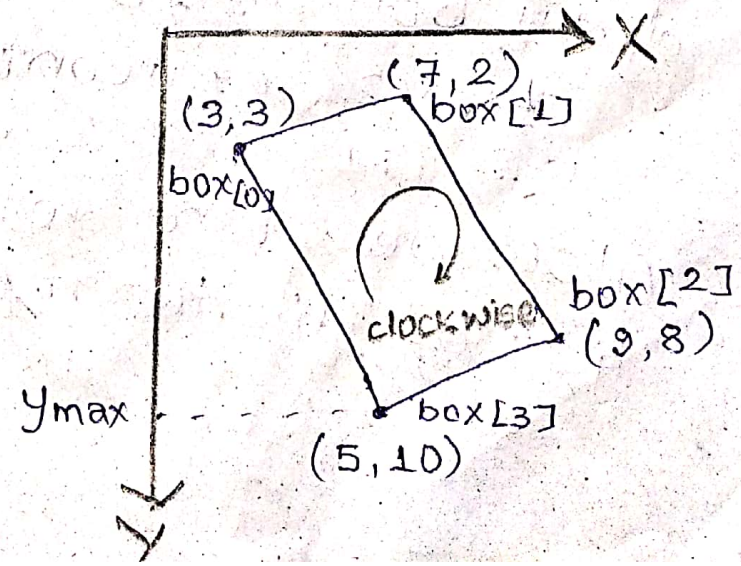
box[3] → ~~bottom-left~~ point with maximum y-value.

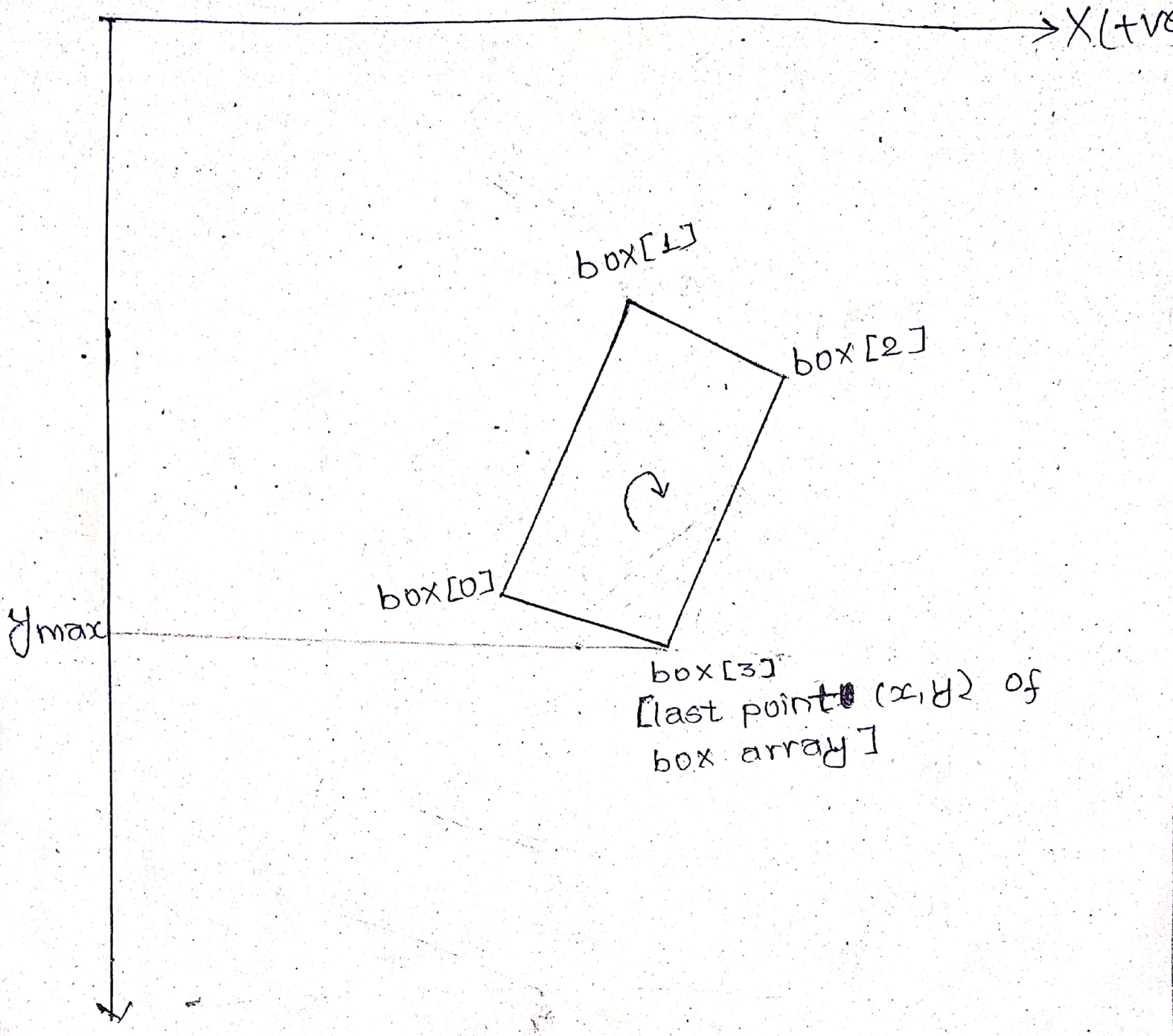


Then from last point/tuple, it goes on clockwise direction.

$$\begin{bmatrix} (3,3) \\ (7,2) \\ (9,8) \\ (5,10) \end{bmatrix}$$

box[0]
box[1]
box[2]
box[3]





→ point (x, y), with maximum y-value among all , will be at last of our 'box' array.

→ box[0] is the first joined point in clockwise direction joined by point box[3]. Then box[1] is first point in clockwise direction joined by point box[0]

After all these things, now you are ready for logics.

```
If contourType == "line":
        if area >= 5:
            if size[0]<20 or size[1] < 20:
                :
```

Using RETR_TREE for "line" type contour will return all contours with rectangles & noise points. But, we need to extract values of "line" contour only.

→ For removing noisy points with area < 5

→ (size[0], size[1]) may be (height, width) or (width, height)

→ So, if height of line must not be greater than 20. (assumption)

Here, height means "thickness"
width means "length" of line.

If it passes all "if" conditions, we draw bounding box as:

↳ cv2.drawContours(img contour, [box], 0, (0, 0, 255), 2)

Also, we store details of that line as list of dictionary in lineContours list.

else:
If it is rectangle,
we use
contour, hierarchy = cv2.getContours(img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

As there are rectangles in outer contour, so, it retrieves all rectangles.
We just return list of dictionary with key "center" as rectangleContours list.
For numbering purpose, we just require center to put text in output image.

Again,
if it is line:
"lineContours" consists of 2 same/similar line because

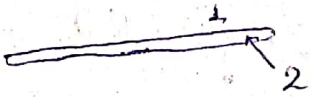


fig. line

For line, RETR_TREE will consider 1 contour as outer part of line & another contour as inner part of line. We take only one line which is inner part of line consisting of minimum area.
Finally, lineContour will have unique line information only.

6) for lineContour in lineContours:

→ This block of code will go through each line contour & compute 'length' & 'thickness/width' of line and append to array sorted-index as list of dictionary.

7) Assigning Numbers to Rectangles

→ index is maintained in dictionary in order to know the rectangle in which it is lying inside.

Initially, lineContours & rectangleContours will have same position or corresponding i.e.

item '0' of lineContours will be inside rectangle i.e. item '0' of rectangle Contours.

So, we store index value in sorted-index list.

⇒ "R-{i+1}" text is written below of each rectangle which position is computed using rectangleContours['center' that we returned from getContours() function.

8) Putting text within window images and showing using

cv2.imshow()

Also,

cv2.namedWindow() is used to display images in window as normal size of screen of laptop we are using.