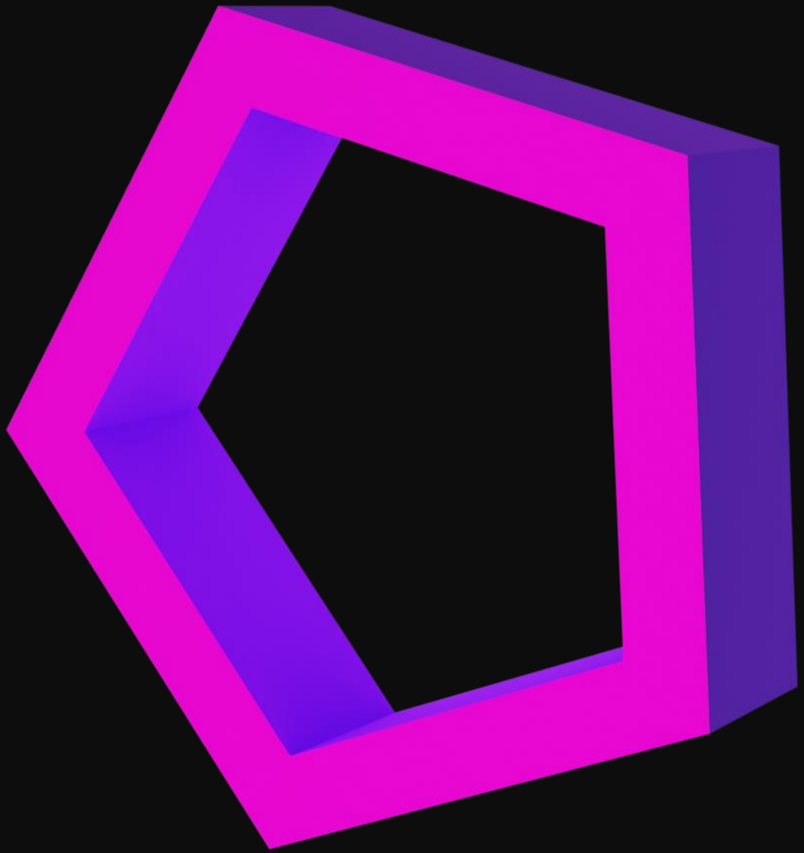
Project goals

. Edge detection

. Polygon representation



Different Phases

Phase 0: Installing libraries in virtual environment

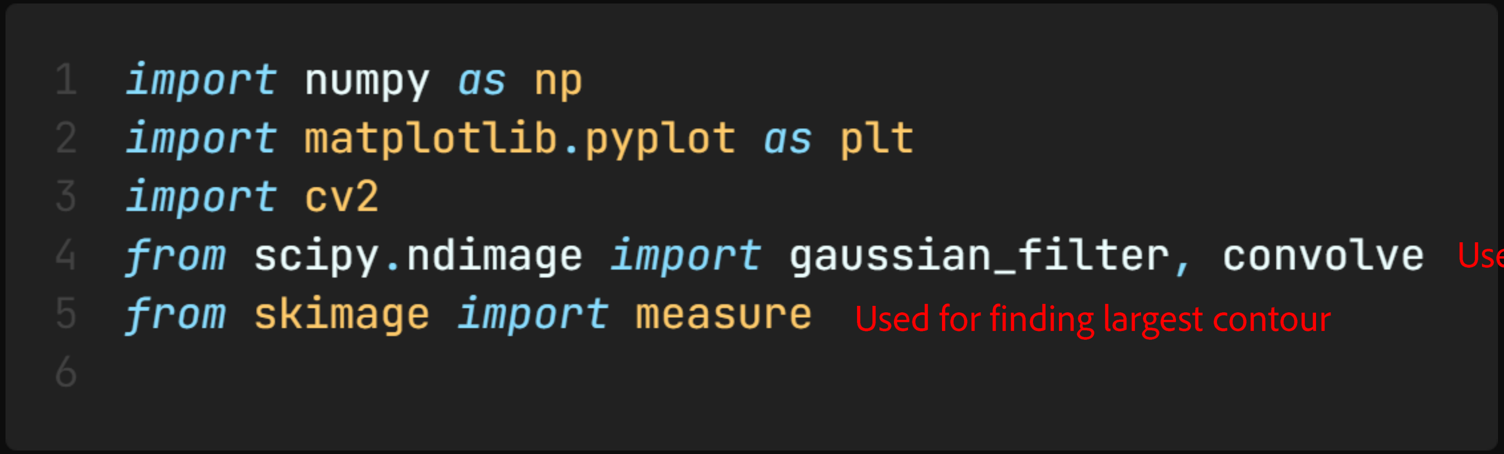
Phase 1: Opening an Image and Converting It to Grayscale

Phase 2: Applying Sobel Edge Detection (comes after phase 3)

Phase 3: Using a Gaussian Filter to Reduce Noise

Phase 4: Finding largest Polygon from Detected Edges using skimage

Step 1 : Libraries

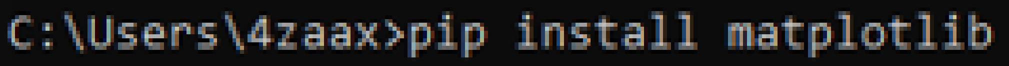


```python
import numpy as np
import matplotlib.pyplot as plt
import cv2
from scipy.ndimage import gaussian_filter, convolve
from skimage import measure
```

Used for applying noise reduction

Used for finding largest contour

```
Method used for installing libraries :
(for both global installation or virtual)
```



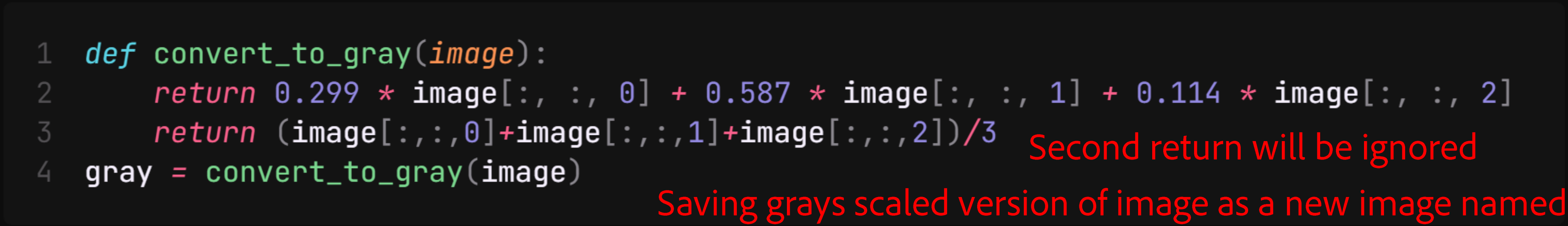
```
C:\Users\4zaax>pip install matplotlib
```

Step 2 : Load Image

```python
# Load the image
image_pth = 'tst.png'
image = plt.imread(image_pth)
```

Reading the image with imread function , we can replace image_path with user Input later after ensuring code performing well.

Step 3 : Convert to grayscale



```
def convert_to_gray(image):
    return 0.299 * image[:, :, 0] + 0.587 * image[:, :, 1] + 0.114 * image[:, :, 2]
    return (image[:,:,0]+image[:,:,1]+image[:,:,2])/3
gray = convert_to_gray(image)
```

Second return will be ignored

Saving grays scaled version of image as a new image named “gray”

```
image[:, :, 0]
```

Red channel

```
image[:, :, 1]
```

Green channel

```
image[:, :, 2]
```

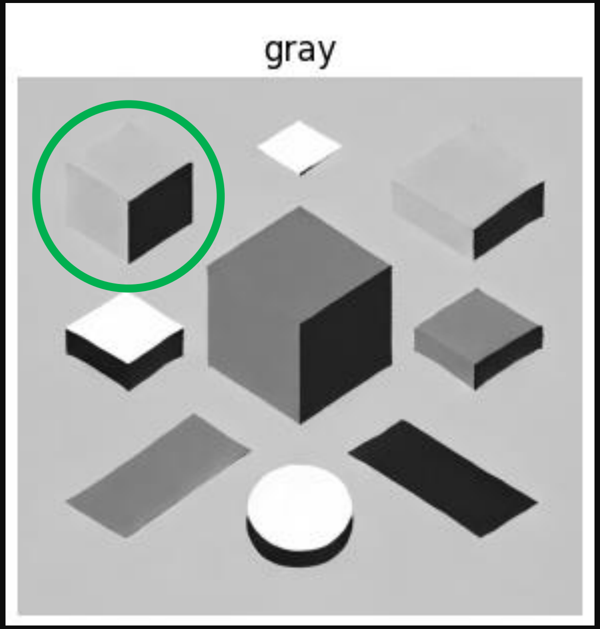
Blue channel

```
return 0.299 * image[:, :, 0] + 0.587 * image[:, :, 1] + 0.114 * image[:, :, 2]
```

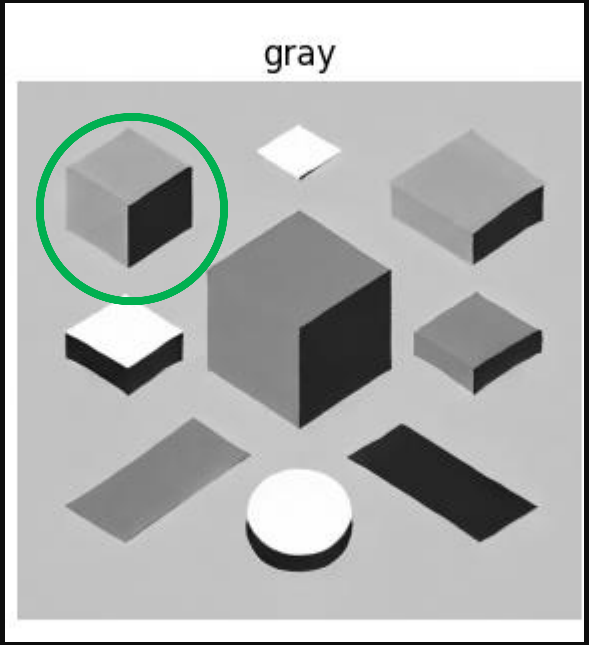
Turning 3 channels into one channel to apply grayscale filter (for better result we could also use gamma)

simple avg vs weighted avg to mix channels

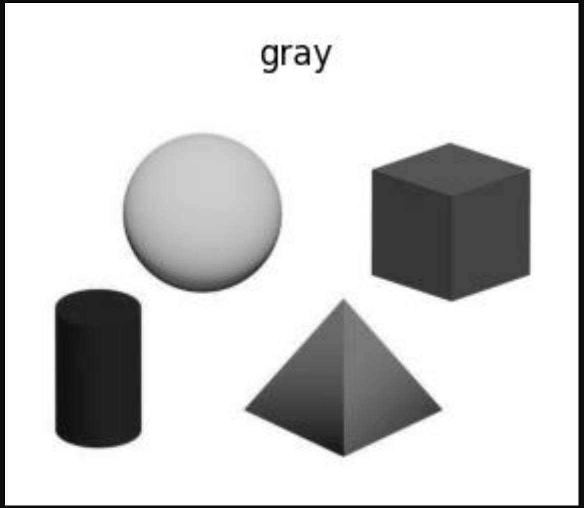
Reason : Human Perception of Brightness
The human eye is more sensitive to green light than to red or blue light.



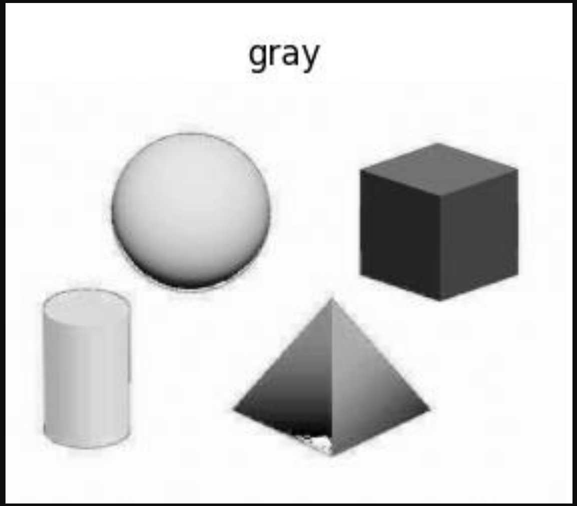
weighted



simple

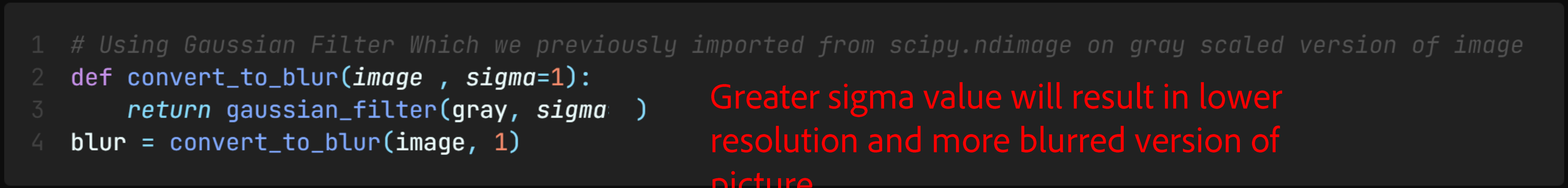


weighted



simple

Step 4 : Apply Blur



```python
# Using Gaussian Filter Which we previously imported from scipy.ndimage on gray scaled version of image
def convert_to_blur(image , sigma=1):
    return gaussian_filter(gray, sigma  )
blur = convert_to_blur(image, 1)
```

Greater sigma value will result in lower resolution and more blurred version of picture

This function receives an image as input and a sigma (if assigned) with the default value of 1 and apply the gaussian filter we've imported from scipy on the image

* We could also define a kernel for applying gaussian blur the way we are gonna be using in following parts for Sobel edge detection algorithm

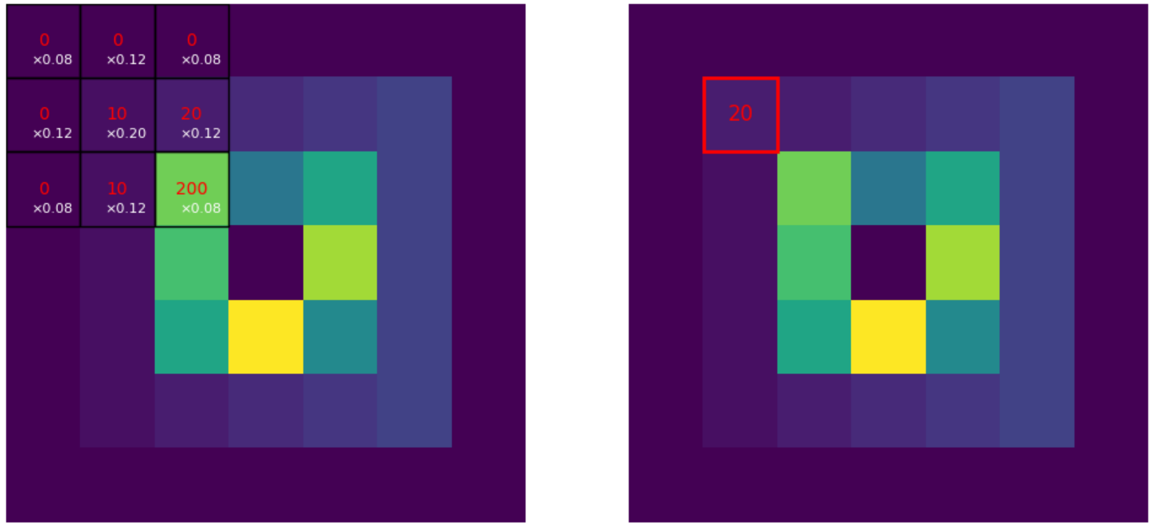
How does gaussian blur work ?

Gussian blur

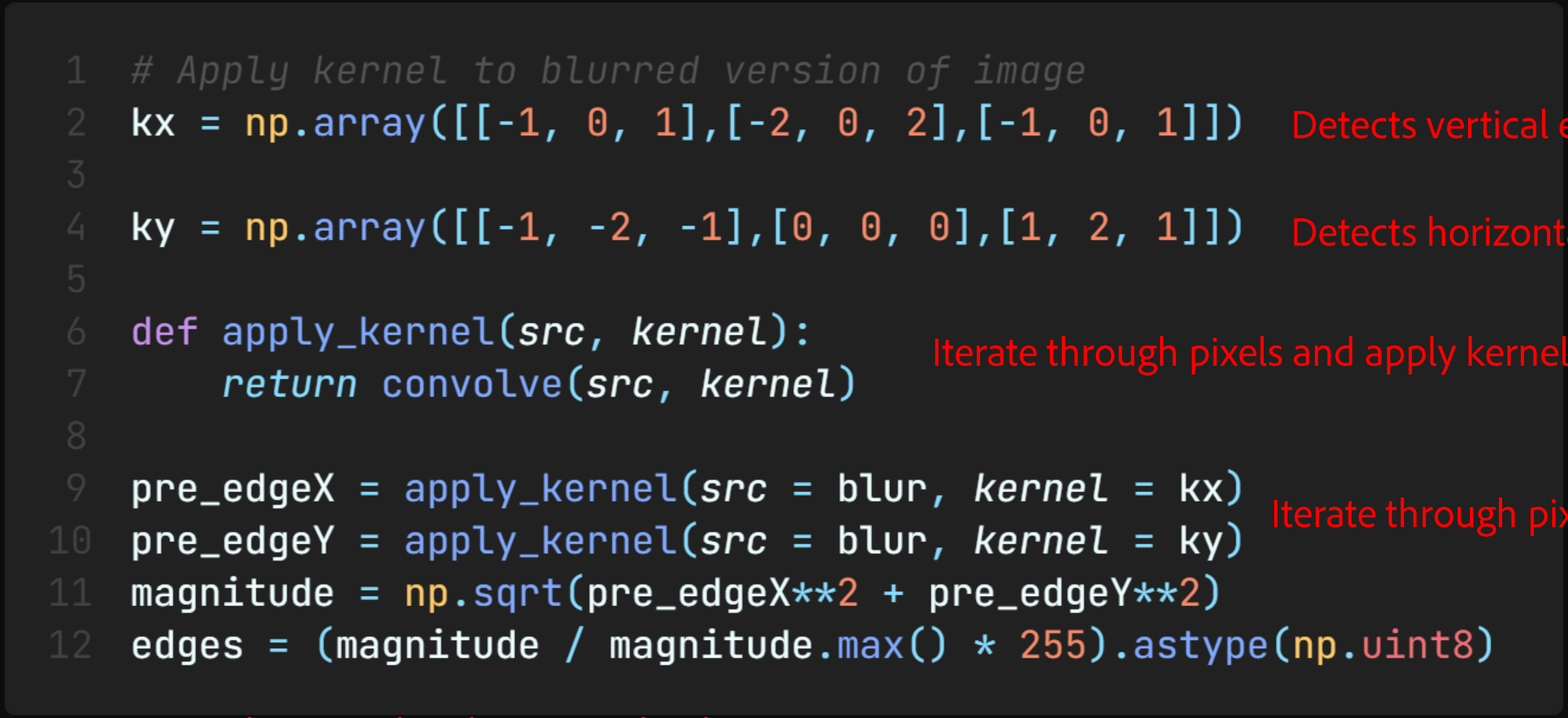
$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

Simple blur

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$



Step 5 : Sobel edge detection



```python
# Apply kernel to blurred version of image
kx = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])

ky = np.array([[-1, -2, -1],[0, 0, 0],[1, 2, 1]])

def apply_kernel(src, kernel):
    return convolve(src, kernel)

pre_edgeX = apply_kernel(src = blur, kernel = kx)
pre_edgeY = apply_kernel(src = blur, kernel = ky)
magnitude = np.sqrt(pre_edgeX**2 + pre_edgeY**2)
edges = (magnitude / magnitude.max() * 255).astype(np.uint8)
```

Detects vertical edge

Detects horizontal edge

Iterate through pixels and apply kernel on them

Iterate through pixels and apply kernel on them

Normalizing and making sure the dtype is uint8

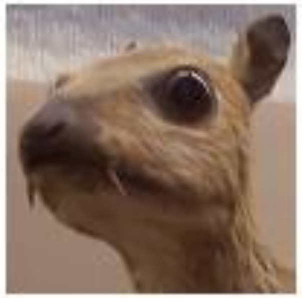
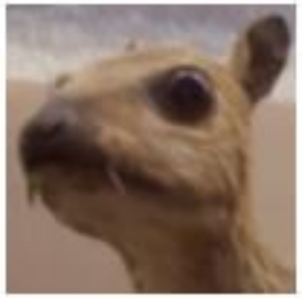
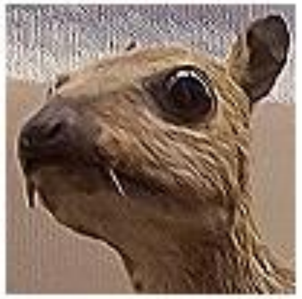
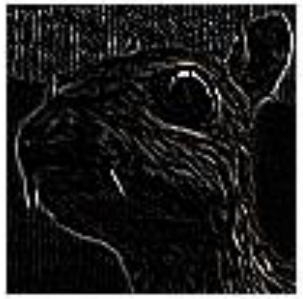
Since OpenCV uses Numpy to display images, you can simply create a convolution kernel using Numpy.

Second way

```python
# Apply kernel to blurred version of image
"""sobel kernel"""
sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])  # Sobel-x
sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])  # Sobel-y
appl_x = cv2.filter2D(src=blur , ddepth=-1 , kernel=sobel_x) # returns numpy array
appl_y = cv2.filter2D(src=blur , ddepth=-1 , kernel=sobel_y) # returns numpy array
sobel_complete = np.sqrt(appl_x**2 + appl_y**2)

sobel_complete = cv2.normalize(sobel_complete, None, 0, 255, cv2.NORM_MINMAX, dtype=cv2.CV_8U)
```

What are some other kernels for image processing ?

Original	*Gaussian Blur*	*Sharpen*	*Edge Detection*
$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
			

Step 6 : Applying threshold

```python
# Apply thresholding to make a binary photo
_, thresh = cv2.threshold(sobel_complete, 8, 255, cv2.THRESH_BINARY)
```

Thresholding with thresh binary method :

0 : mean value There are also other methods to find mean value without entering it manually like np.mean()

255 : max value

THRESH_BINARY: Pixels above the threshold become 255 (white); others become 0 (black).

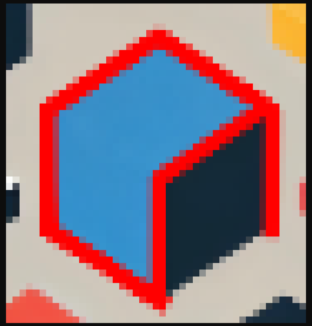
THRESH_OTSU: Automatically calculates the optimal threshold value using Otsu’s algorithm,

Step 7 : finding contour

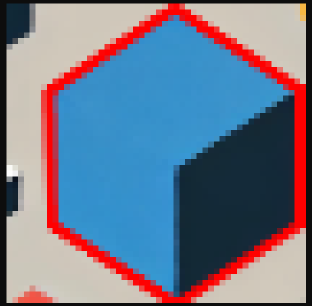
```python
contours = measure.find_contours(thresh, fully_connected="high") #finding contours with skimage.measure
#Returns a list of NumPy arrays. Each array is a contour represented as an (N, 2) array where each row gives the (row, column) coordinates of a contour point.
```

.find_contours : uses scikit-image lib to find contours in the binary_edged image

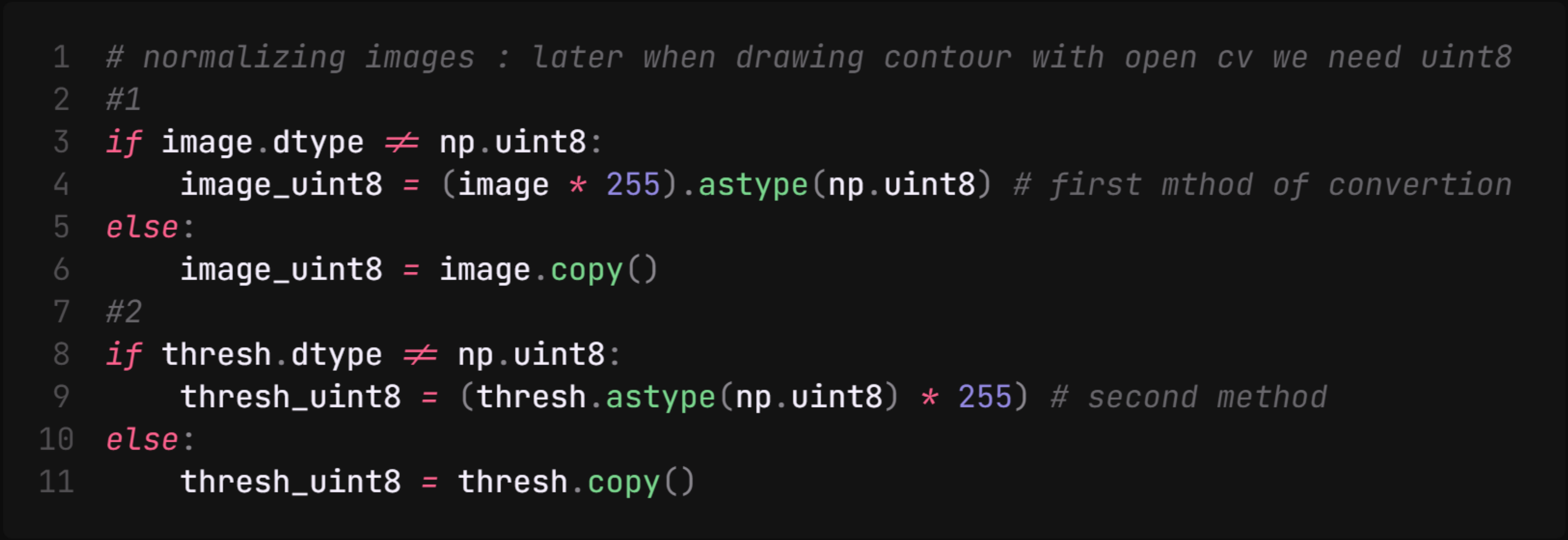
.fully_connected : helping to make more complete contours



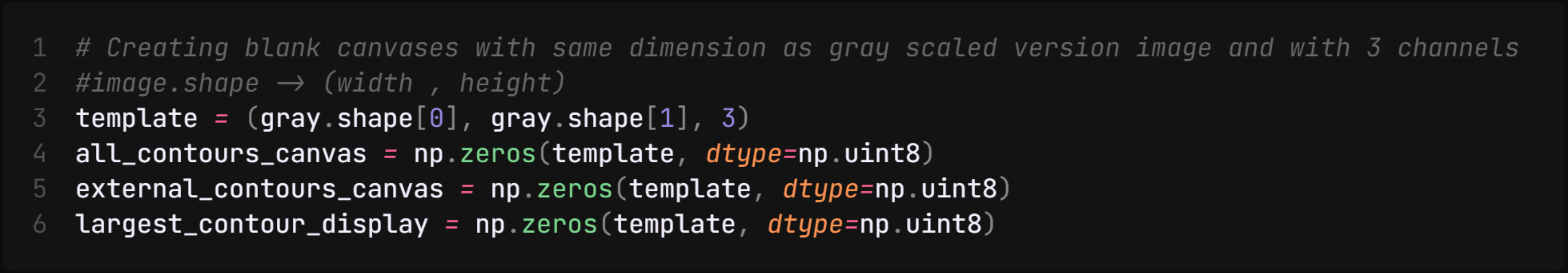
Before using fully_connected



After using fully_connected



```python
# normalizing images : later when drawing contour with open cv we need uint8
#1
if image.dtype != np.uint8:
    image_uint8 = (image * 255).astype(np.uint8) # first mthod of convertion
else:
    image_uint8 = image.copy()
#2
if thresh.dtype != np.uint8:
    thresh_uint8 = (thresh.astype(np.uint8) * 255) # second method
else:
    thresh_uint8 = thresh.copy()
```



```python
# Creating blank canvases with same dimension as gray scaled version image and with 3 channels
#image.shape -> (width , height)
template = (gray.shape[0], gray.shape[1], 3)
all_contours_canvas = np.zeros(template, dtype=np.uint8)
external_contours_canvas = np.zeros(template, dtype=np.uint8)
largest_contour_display = np.zeros(template, dtype=np.uint8)
```

```python
contours_approximated = []
```

```python
for contour in contours:
    approx = measure.approximate_polygon(contour, tolerance=2) #tolerance can be adaptive
    approx_cv = approx[:, [1, 0]].reshape(-1, 1, 2).astype(np.int32)
    contours_approximated.append(approx_cv)


largest_contour_cv = max(contours_approximated, key=cv2.contourArea)
ext_contours_appr, _ = cv2.findContours(thresh_uint8, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
image_with_largest = image_uint8.copy()

cv2.drawContours(all_contours_canvas, contours_approximated, -1, (255, 255, 255), 2)
cv2.drawContours(external_contours_canvas, ext_contours_appr, -1, (255, 255, 255), 2)
cv2.drawContours(largest_contour_display, [largest_contour_cv], -1, (0, 0, 255), 10)  #red outline in BGR
cv2.drawContours(image_with_largest, [largest_contour_cv], -1, (255 , 0, 0), 10)
```

```python
def bgr_to_rgb(img):
    return cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
image_with_largest = bgr_to_rgb(image_with_largest)
largest_contour_display = bgr_to_rgb(largest_contour_display)
image_with_largest = bgr_to_rgb(image_with_largest)
```

I appreciate your attention