

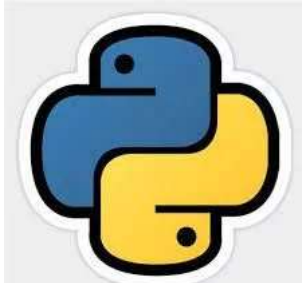
```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

#reading the image files
img1 = cv2.imread('/content/drive/MyDrive/AI_Video_Monitoring/test-img-1.jpg.webp')
img2 = cv2.imread('/content/drive/MyDrive/AI_Video_Monitoring/test-img-2.jpg.webp')

cv2_imshow(img1)
```



⌂ B I <> 🔗 🖼️ 📄 📋 📌 ⋮ 🌀 😊 ☰

***Arithmetic Operators**

Arithmetic Operators

Addition

Note: In OpenCV addition, simply load the image files and pass the Numpy N-d arrays returned after loading the images to othe cv2.add() method as args. It is a saturated operation that means if the resultant pixel value is greater than 255 after the addition of the pixel values of the input (loaded) images then it is saturated to 255 so that any pixel value cannot exceed 255. This is called ****saturation**.

```
#now applying openCv addition on images
out_img = cv2.add(img1, img2)
```

```
#displaying the image after addition of 2 images
cv2_imshow(out_img)
```



Subtraction

Image subtraction is used both as an intermediate step in complicated image processing techniques and also as an important operation on its own. One most common use of image subtraction is to subtract background variations in illumination from a scene so that the objects in foreground can be analyzed more easily and clearly.

```
# Applying OpenCV subtraction on images
out_img = cv2.subtract(img1, img2)

#displaying the image after subtraction of 2 images
cv2.imshow(out_img)
```



Multiplication

Like other arithmetic operations on images, image multiplication can also be implemented in forms. The first form of image multiplication takes two input images and produces an output image in which the pixel values are the product of the corresponding pixel values of the input images.

And the second form takes a single input image and produces output in which each pixel value is the product of the corresponding pixel values of the input image and a specified constant (scaling factor). This second form of image multiplication is more widely used and is generally called scaling.

There are several uses of image scaling but in general a scaling factor greater than unity, the scaling will brighten the image, and a scaling factor less than unity will darken the image.

```
# Reading image file
img = cv2.imread('/content/drive/MyDrive/AI_Video_Monitoring/img3.jpg')

# Applying NumPy scalar multiplication on image
out_img = img * 1.5

# Saving the output image
cv2.imshow(out_img)
```



```
# Reading image file
img = cv2.imread('/content/drive/MyDrive/AI_Video_Monitoring/img3.jpg')

# Applying OpenCV scalar multiplication on image
out_img = cv2.multiply(img, 1.5)

# Saving the output image
cv2.imshow(out_img)
```



Division

The image division operation normally takes two images as input and produces a third image whose pixel values are the pixel values of the first image divided by the corresponding pixel values of the second image.

It can also be used with a single input image, in which case every pixel value of the image is divided by a specified constant.

```
# Reading image file
img = cv2.imread('/content/drive/MyDrive/AI_Video_Monitoring/img3.jpg')
```

```
# Applying OpenCV scalar division on image
out_img = cv2.divide(img, 2)
```

```
# Saving the output image
cv2.imshow(out_img)
```



```
# Reading image file
img = cv2.imread('/content/drive/MyDrive/AI_Video_Monitoring/img3.jpg')
```

```
# Applying NumPy scalar division on image
out_img = img / 2
```

```
# Saving the output image
cv2.imshow(out_img)
```

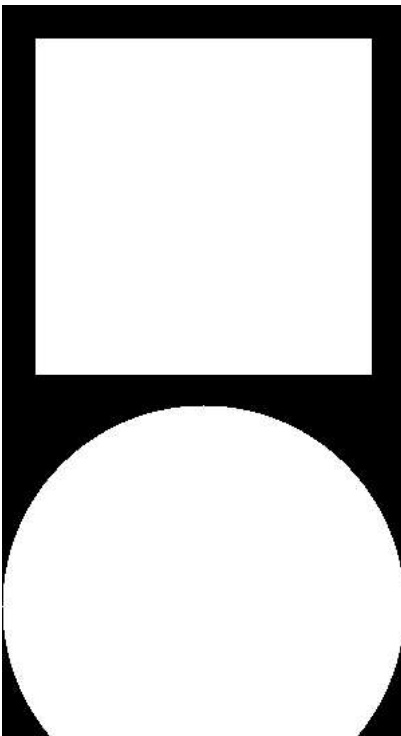


▼ BitWise Operators

for bitwise operators, we need 2 variables or images, let's create a bitwise circle and a bitwise square through which we can use the bitwise operations. Note: bitwise operations require the images to be black and white

```
# creating a square of zeros using a variable
rectangle = np.zeros((300, 300), dtype="uint8")
cv2.rectangle(rectangle, (25, 25), (275, 275), 255, -1)
cv2.imshow(rectangle)
```

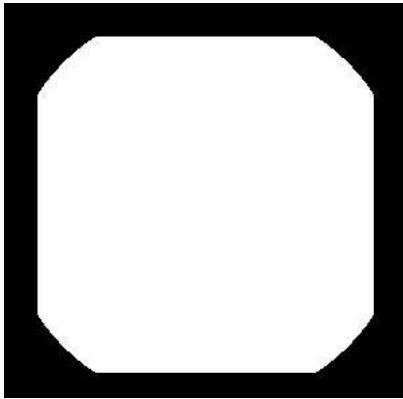
```
# creating a circle of zeros using a variable
circle = np.zeros((300, 300), dtype="uint8")
cv2.circle(circle, (150, 150), 150, 255, -1)
cv2.imshow(circle)
```



AND operation: Bitwise addition refers to the addition of two different images, and decide which is to be displayed using an AND operation on each pixel of the images.

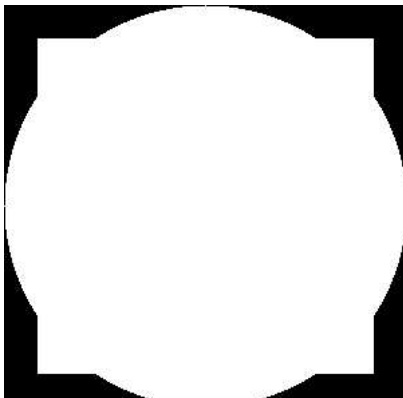
```
# the bitwise_and function executes the AND operation
# on both the images
```

```
bitwiseAnd = cv2.bitwise_and(rectangle, circle)
cv2.imshow( bitwiseAnd)
cv2.waitKey(0)
```



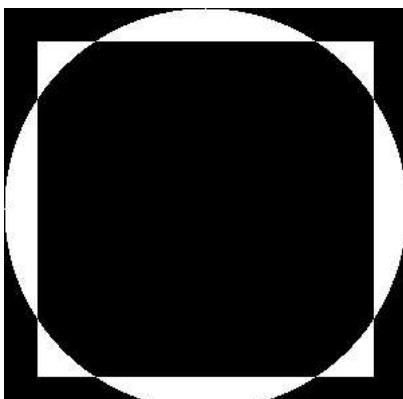
OR operation: Bitwise OR provides us with a product of the two images with an OR operation performed on each pixel of the images.

```
# the bitwise_or function executes the OR operation
# on both the images
bitwiseOr = cv2.bitwise_or(rectangle, circle)
cv2.imshow(bitwiseOr)
cv2.waitKey(0)
```



XOR operation: Another operation that is provided by the cv2 module is the XOR operation, which we can use through the bitwise_xor function.

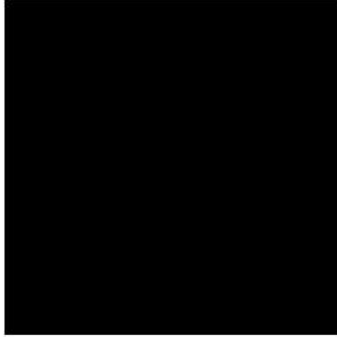
```
# the bitwise_xor function executes the XOR operation
# on both the images
bitwiseXor = cv2.bitwise_xor(rectangle, circle)
cv2.imshow(bitwiseXor)
cv2.waitKey(0)
```



NOT operation: the negation operation is performed using the `bitwise_not` function.

The NOT operation only requires a single image as we're not adding or subtracting anything here.


```
# the bitwise_not function executes the NOT operation
# on both the images
bitwiseNot = cv2.bitwise_not(rectangle, circle)
cv2.imshow('bitwiseNot', bitwiseNot)
cv2.waitKey(0)
```

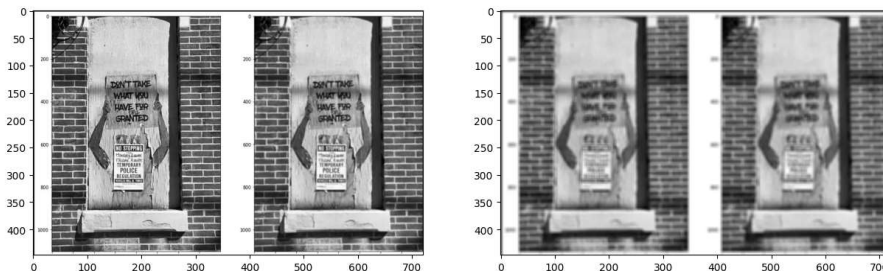


```
from matplotlib import pyplot as plt
import numpy as np
```

▼ Blurring of an image

```
# Plotting image and its blurred version
f,ax = plt.subplots(1,2,figsize=(15,20))
ax = ax.flatten()
im = cv2.imread('/content/drive/MyDrive/AI_Video_Monitoring/img4.webp',-1)
im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
im_blur = cv2.GaussianBlur(im,(7,7), 5, 5)
ax[0].imshow(im,cmap='gray')
ax[1].imshow(im_blur,cmap='gray')
```

 <matplotlib.image.AxesImage at 0x7f41a6e09120>



▼ Edge Detection

```
import cv2
```

```
# Read the original image
img = cv2.imread('/content/drive/MyDrive/AI_Video_Monitoring/img5.jpg')
```

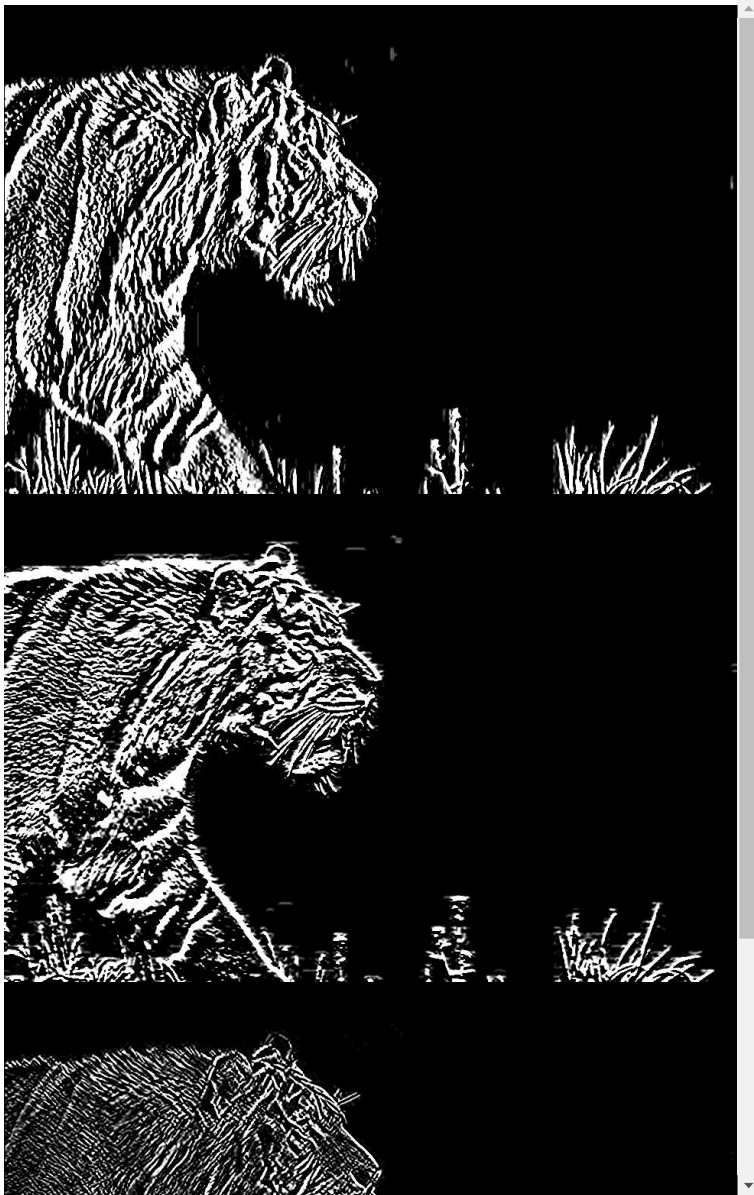
```
# Display original image
cv2.imshow(img)
cv2.waitKey(0)

# Convert to grayscale
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Blur the image for better edge detection
img_blur = cv2.GaussianBlur(img_gray, (3,3), 0)
```



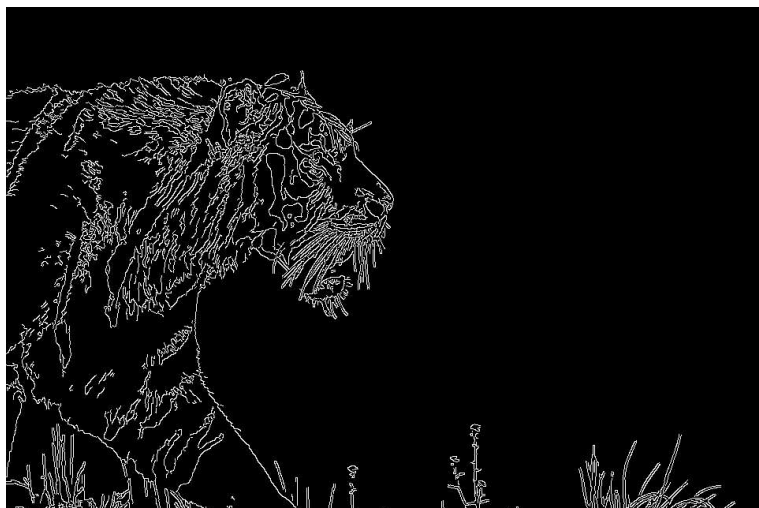
```
# Sobel Edge Detection
sobelx = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=1, dy=0, ksize=5) # Sobel Edge Detection on the X axis
sobely = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=0, dy=1, ksize=5) # Sobel Edge Detection on the Y axis
sobelxy = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=1, dy=1, ksize=5) # Combined X and Y Sobel Edge Detection

# Display Sobel Edge Detection Images
cv2.imshow(sobelx) #Sobel X
cv2.waitKey(0)
cv2.imshow(sobely) #Sobel Y
cv2.waitKey(0)
cv2.imshow(sobelxy) #Sobel X Y using Sobel() function
cv2.waitKey(0)
```




```
# Canny Edge Detection
edges = cv2.Canny(image=img_blur, threshold1=100, threshold2=200) # Canny Edge Detection
# Display Canny Edge Detection Image
cv2.imshow(edges) #Canny Edge Detection
cv2.waitKey(0)

cv2.destroyAllWindows()
```



✓ 0s completed at 06:33

