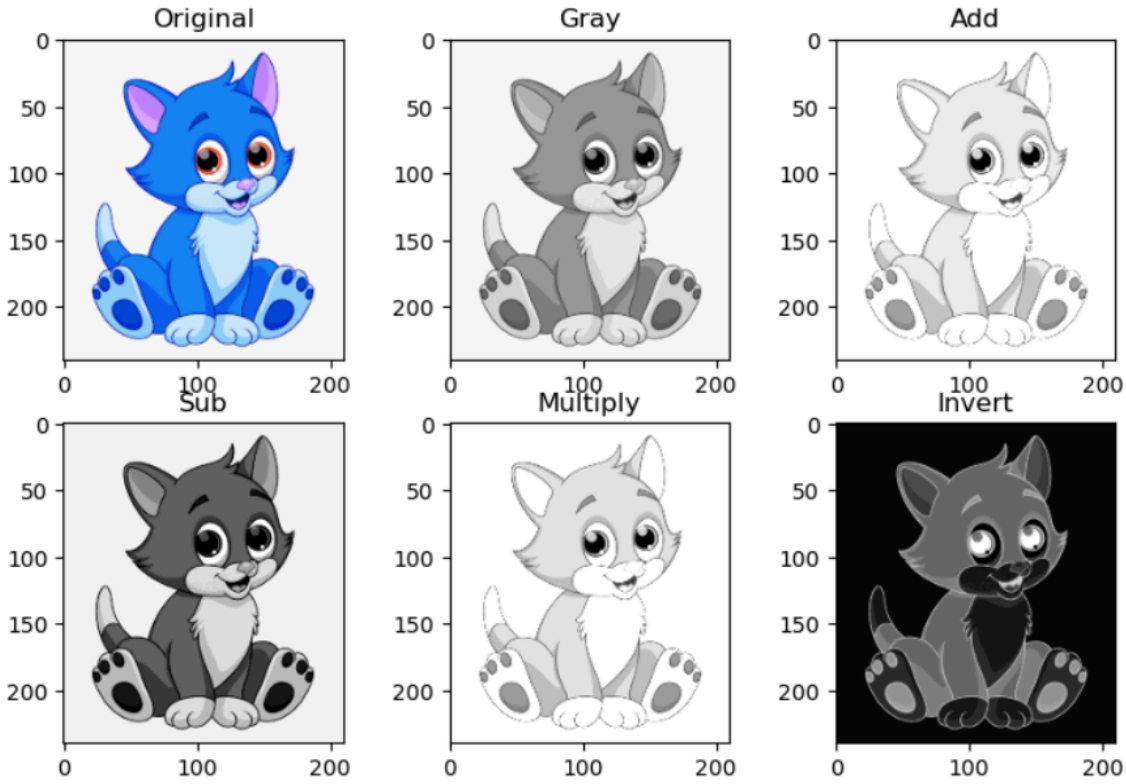
LAB 02

Name: Bushra Shahbaz

Roll no: BSDSF21M020

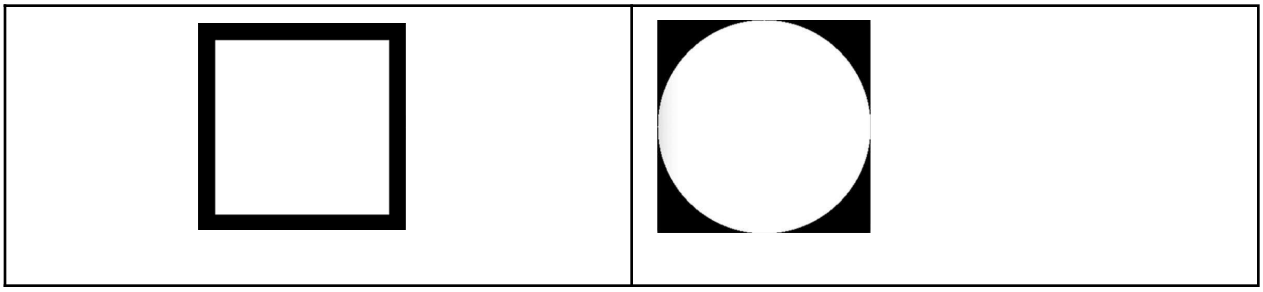
Practice Tasks : **Evaluation in next Lecture**

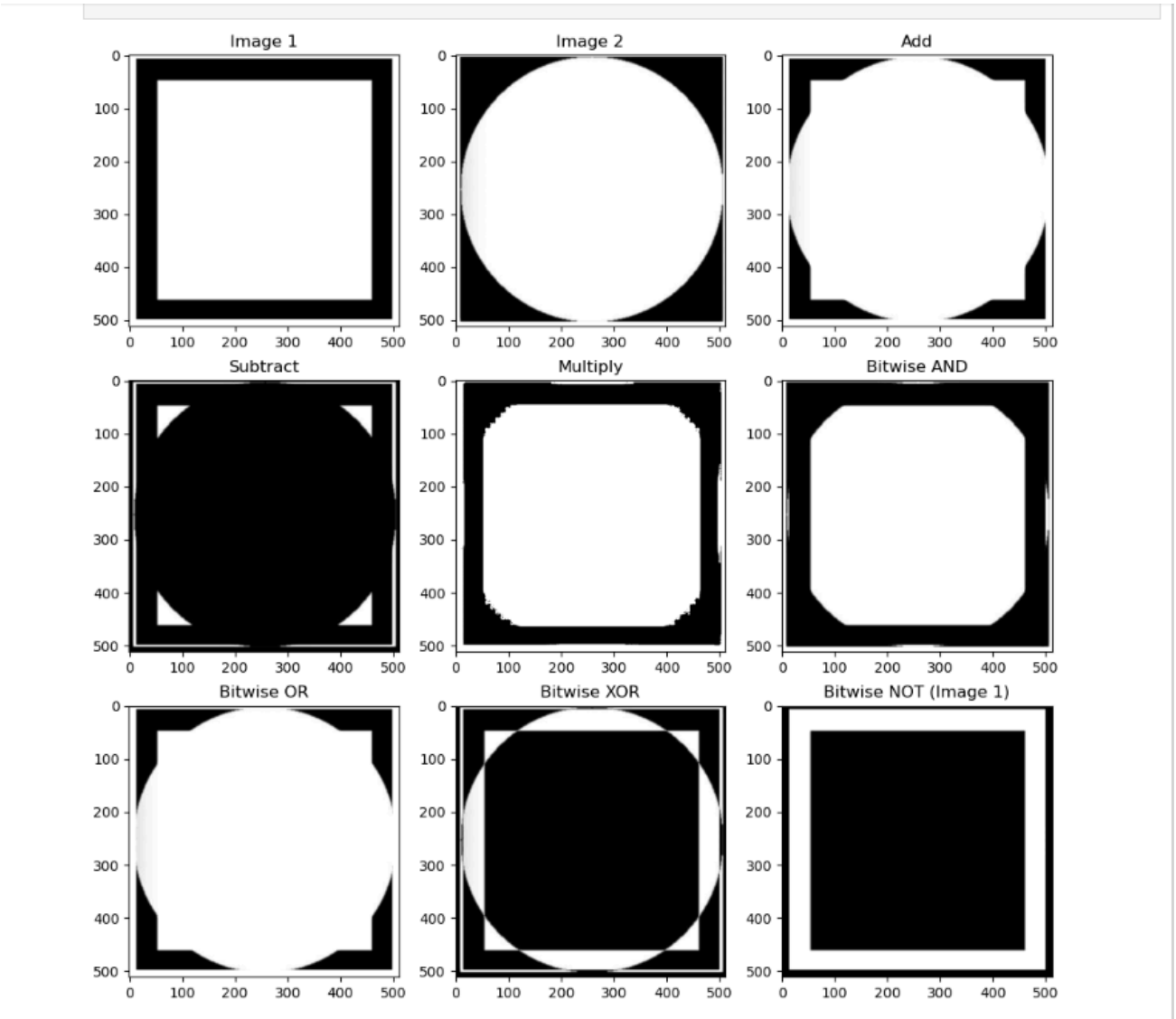
Task#1: Execute lab1.py and describe your observations.



The original image is shown in color, as it was first loaded. After converting it to grayscale, we see the same image but in shades of gray, which simplifies it by focusing on light and dark areas without any color. When we add 90 to the grayscale image, the whole picture becomes brighter, making light areas stand out even more. In contrast, subtracting 90 makes the image darker, lowering the brightness. Multiplying by 1.5 increases the contrast, making the bright areas much brighter and the dark areas even darker. Finally, inverting the image flips the colors, turning light areas dark and dark areas light, similar to a photographic negative. Each operation changes how the image looks by adjusting brightness or contrast.

Task#2: Use the given below images as your input images and perform all arithmetic and logical operations. Show your code and also state your observations for each.





Observations:

1. **Image 1 & Image 2**: These are the two input grayscale images. We converted them from color to grayscale for simplicity in operations.
2. **Add**: The result of adding the two images blends them together, making areas of overlap brighter. This highlights areas where both images have light features.

3. **Subtract**: Subtracting one image from the other emphasizes the differences between them. Areas where the images are similar turn dark, and the differences appear as lighter areas.
4. **Multiply**: Multiplying the images increases the contrast where the pixel values in both images are high. Dark areas remain dark, while overlapping bright areas become even brighter.
5. **Bitwise AND**: This operation keeps only the areas where both images have high pixel values (bright areas). It effectively highlights areas of overlap between the two images.
6. **Bitwise OR**: This combines the bright areas from both images, making any pixel that is bright in either image remain bright in the result.
7. **Bitwise XOR**: The XOR operation shows the differences between the two images, highlighting areas where one image is bright while the other is dark.
8. **Bitwise NOT (Image 1)**: This operation inverts the grayscale values of Image 1, turning light areas dark and dark areas light.

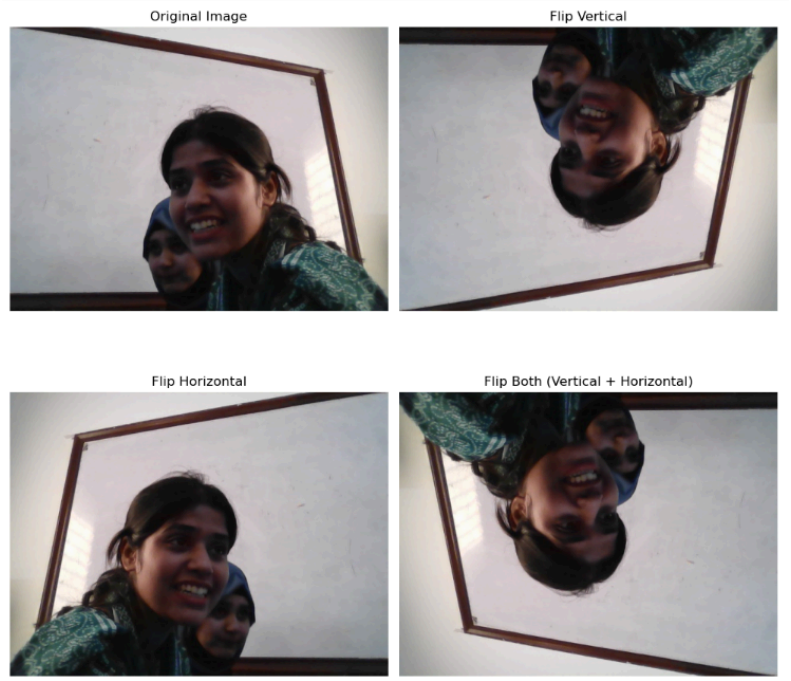
Each operation alters the images in a unique way by combining or manipulating the pixel values. Arithmetic operations tend to blend or emphasize brightness, while logical operations focus on similarities or differences between the images.

Task#3: Capture your image through webcam and save it. You may use code given in CaptureVideoImage.py.

My code opens the webcam, captures video, and displays it in real-time. It keeps showing the video until I press the 'q' key. When I press 'q', the program saves the current frame as "DIP.jpg", stops the camera, and closes the display window.

Task#4: Perform all the types of flipping on the captured image. Show the flipped images.

In the original image, we see the captured image as it was initially recorded without any modifications. When flipped vertically, the image appears upside down, reversing the top and bottom portions while keeping the left and right sides intact. In the horizontally flipped version, the left and right sides of the image are swapped, giving a mirror-like effect. Finally, in the image flipped both vertically and horizontally, the entire image is rotated, flipping both axes, so it appears completely reversed, as if rotated 180 degrees. Each flip operation changes the orientation, helping visualize how images can be manipulated along different axes.



Task#5: Perform all rotations on the captured image. Show the rotated images.

In the original image, we see the captured picture without any rotation. When the image is rotated by 90 degrees, it shifts clockwise, causing what was the top of the image to move to the right and the bottom to move to the left, effectively placing the image in a vertical position. The 180-degree rotation flips the image upside down, where the top moves to the bottom and vice versa, presenting a reversed version of the original. Finally, the 270-degree rotation shifts the image clockwise

again, making what was originally the top of the image move to the left side and the bottom to the right, placing the image horizontally but opposite of the 90-degree rotation. Each rotation alters the orientation of the image, making it appear in different directions depending on the degree of rotation.

Task#6:

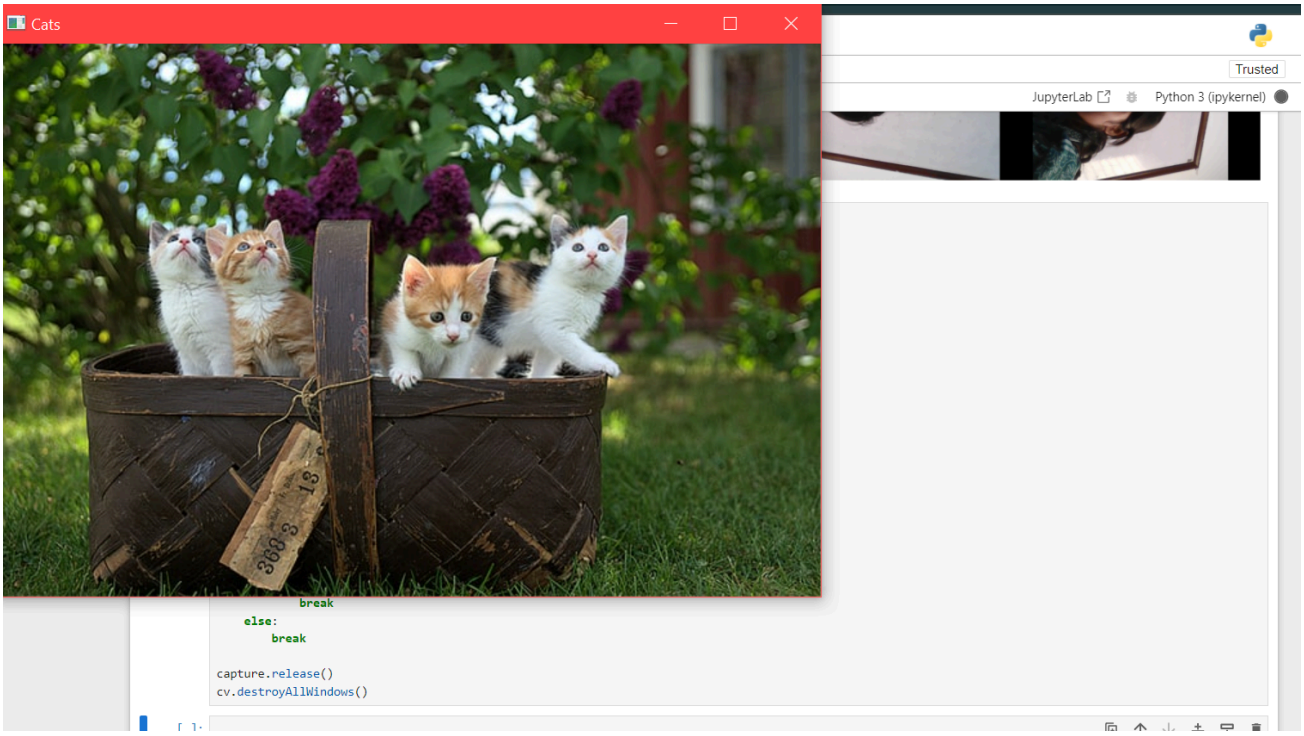
- **i)** Concatenate the original and flipped images into 1 image.
- **ii)** Similarly concatenate all the rotated images into 1 image.

The concatenated image of the original and flipped versions showcases how different flipping operations affect the orientation of the captured image. In this arrangement, the original image appears first, followed by the vertically flipped version, which presents the image upside down. Next, the horizontally flipped version reflects the image as if seen in a mirror, swapping left and right sides. Finally, the both-flipped version combines these effects, resulting in a complete inversion of the original image.

In the second concatenated image of the original and rotated versions, the original image appears first, followed by the 90-degree rotated version, which displays the image in a vertical orientation. The 180-degree rotation presents the image upside down, reversing the top and bottom. Lastly, the 270-degree rotation shows the image rotated back to a horizontal orientation, similar to the 90-degree rotation but positioned to the left.

Together, these concatenated images effectively demonstrate how flipping and rotation operations can manipulate the spatial arrangement of the image, providing a clear visual understanding of each transformation.

Task#7: Execute read.py and describe your observations.



he program begins by loading and displaying an image of cats using OpenCV. The image appears in a window titled "Cats." The program waits for a key press before proceeding. This allows the user to view the image without any automatic closure.

Next, the program moves on to read and display a video file titled `dog.mp4`. It uses a loop to continuously read frames from the video. As each frame is read, it displays the current frame in a window titled "Video." The video playback occurs at a rate of 20 milliseconds per frame, allowing for smooth playback.

If the user presses the 'd' key while the video is playing, the loop terminates, and the video playback stops. Additionally, if the video reaches its end or if a frame cannot be read (indicated by `isTrue` being `False`), the loop also breaks.

Finally, releases the video capture object and closes all OpenCV windows.

OpenCV help functions and their description is given on the next page.

Flip:

We can flip an image around either the x-axis, y-axis, or even both.

Basic Syntax is :

flipped = cv2.flip(image, value)

Value is 1 for horizontal flipping. Value is 0 for vertical flipping. Value is -1 for both axis.

Rotate:

cv2.rotate() method is used to rotate a 2D array in multiples of 90 degrees. The function cv::rotate rotates the array in three different ways.

1. Rotate by 90 degrees clockwise:
   cv2.rotate(image to be rotated, cv2.ROTATE_90_CLOCKWISE)
2. Rotate by 180 degrees clockwise: cv2.ROTATE_180
3. Rotate by 270 degrees clockwise : cv2.ROTATE_90_COUNTERCLOCKWISE

Concatenation of images:

To concatenate images vertically and horizontally with Python, cv2 library comes with two functions as:

1. **hconcat():** It is used as cv2.hconcat() to concatenate images horizontally. Here h means horizontal. cv2.hconcat() is used to combine images of same height horizontally.

2. **vconcat():** It is used as cv2.vconcat() to concatenate images vertically. Here v means vertical. cv2.vconcat() is used to combine images of same width vertically.

Arithmetic and Logical Operators- Bitwise AND, OR, NOR, XOR

Arithmetic Operations like Addition, Subtraction, and Bitwise Operations(AND, OR, NOT, XOR) can be applied to the input images

1. Addition
2. Subtraction

Bitwise operations are used in image manipulation and used for extracting essential parts in the image. In this article, Bitwise operations used are:

1. Bitwise AND
2. Bitwise OR
3. Bitwise XOR
4. Bitwise NOT

1. Addition

- Syntax: cv2.add(img1, img2)
  But adding the pixels is not an ideal situation. So, we use cv2.addweighted(). Remember, both images should be of equal size and depth.

- Syntax: cv2.addWeighted(img1, wt1, img2, wt2, gammaValue)

Parameters:

- img1: First Input Image array(Single-channel, 8-bit or floating-point)
- wt1: Weight of the first input image elements to be applied to the final image
- img2: Second Input Image array(Single-channel, 8-bit or floating-point)
- wt2: Weight of the second input image elements to be applied to the final image
- gammaValue: Measurement of light

2. Subtraction of Image:

Just like addition, we can subtract the pixel values in two images and merge them with the help of cv2.subtract(). The images should be of equal size and depth.

Syntax: cv2.subtract(image1, image2)

3. **AND:** A bitwise AND is true *if and only if* both pixels are greater than zero.
4. **OR:** A bitwise OR is true *if either* of the two pixels is greater than zero.
5. **XOR:** A bitwise XOR is true *if and only if* one of the two pixels is greater than zero, *but not both.*
6. **NOT:** A bitwise NOT inverts the <on= and <off= pixels in an image.

Syntax:

bitwiseAnd = cv2.bitwise_and(rectangle,

circle) bitwiseOr =

cv2.bitwise_or(rectangle, circle) bitwiseXor

= cv2.bitwise_xor(rectangle, circle)

bitwiseNot = cv2.bitwise_not(circle)