Image Processing Lab

All of the programming assignments are to be done in Python using additional libraries specified in the assignments. There are many libraries available, some of which we will be using, and you are welcome to use them with one exception: if the library or a function within it performs the specific function you are asked to code, you may not use that other than perhaps as a reference to compare against. All of the code you submit must be your own. You are welcome to turn in a completed jupyter notebook.

Pre-lab Notes

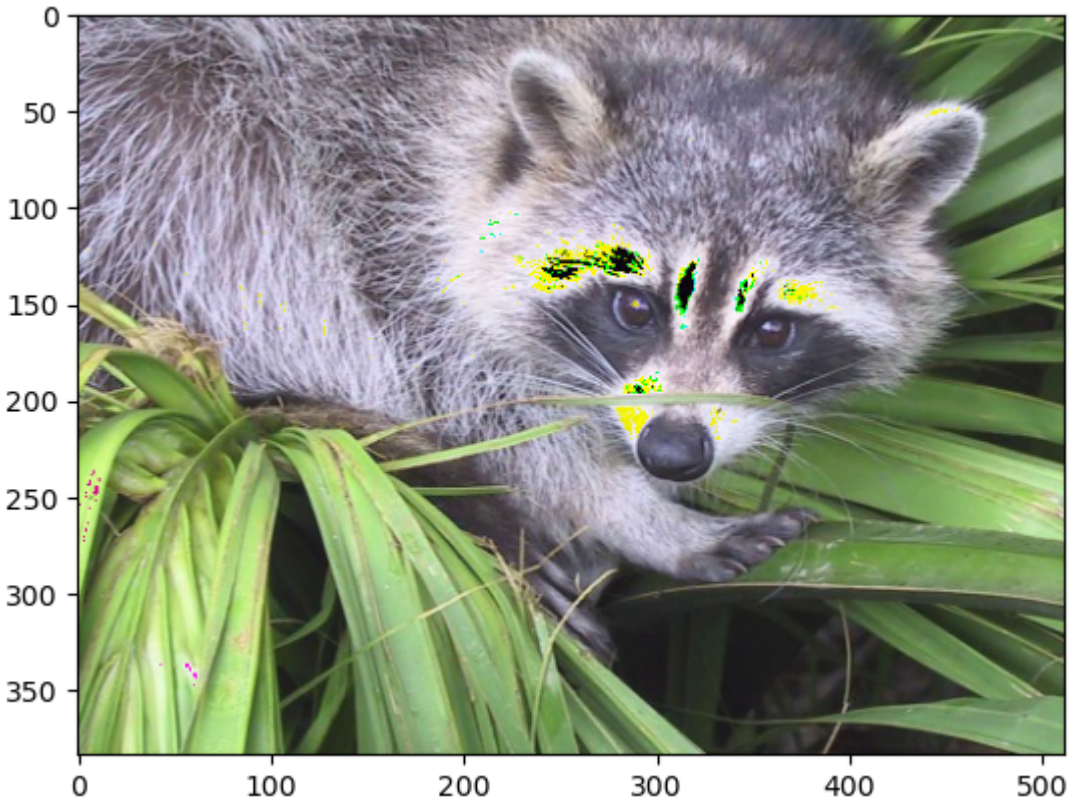
Since it is our first time working with color images, we need to be careful of a couple of things. Most RGB color images store each RGB value as an 8-bit number. This is fine for displaying images as shown below:

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np

racoon = plt.imread('racoon.jpg')
plt.imshow(racoon); plt.show()
```

This is fine for displaying, but becomes a problem once we try to do any image manipulations. We have to beware of overflow. For example, lets say we just want to add 20 to all of the RGB

values.

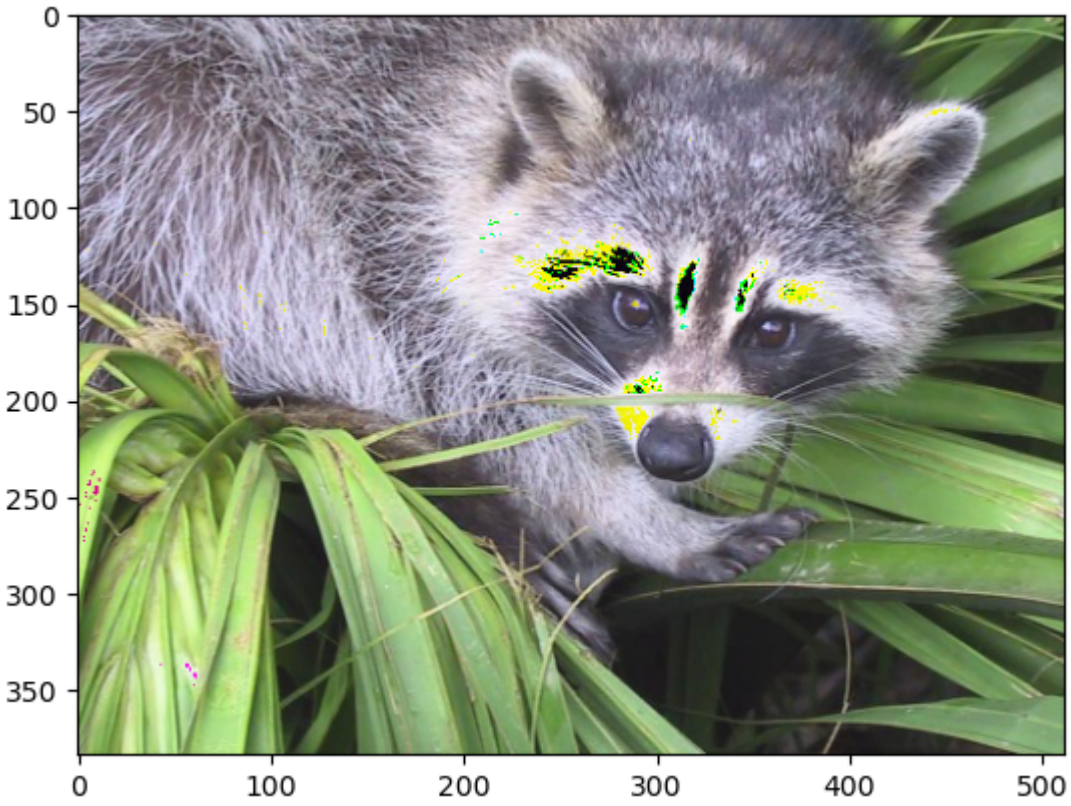
In [2]:
```
plt.imshow(racoon + 20)
plt.show()
```



Notice what happens near the white areas of the image. We only had 8-bits to respresent each RGB value (0 to 255). White areas will have values near 255. So when we add 20, the colors go crazy because values have overflowed (ex. 240 + 20 = 4).

You maybe tempted to try something like this:

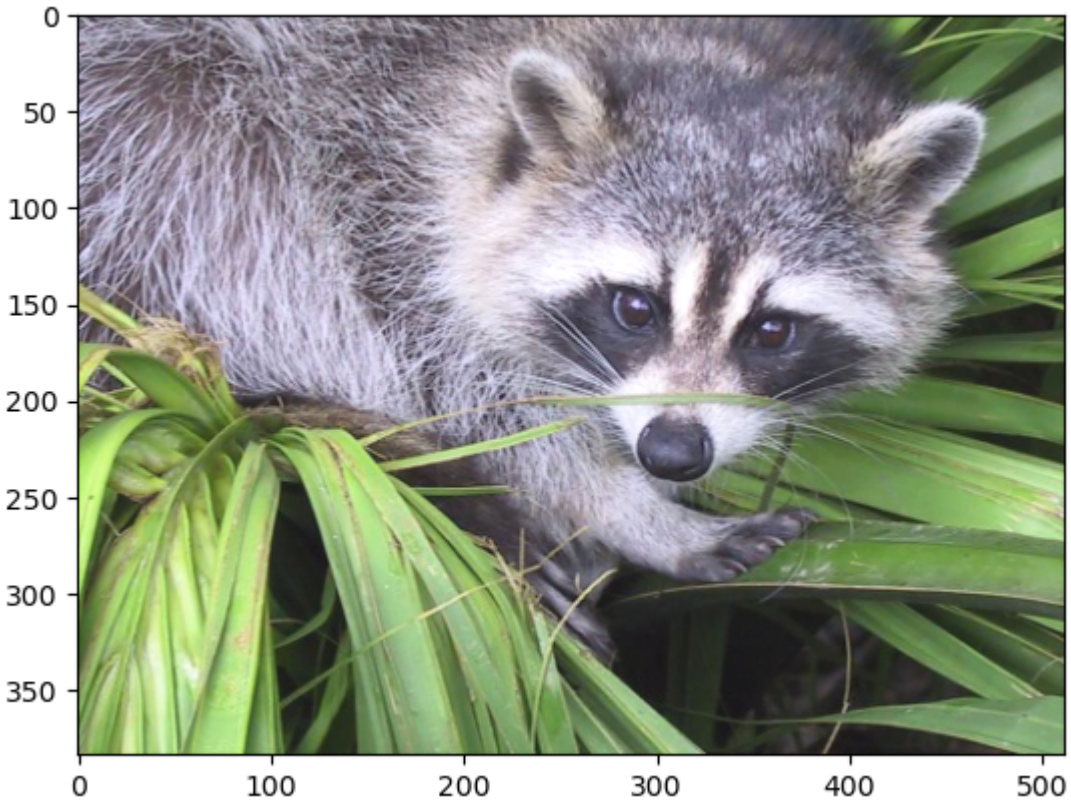
In [3]:
```
plt.imshow(np.clip(racoon + 20,0,255))
plt.show()
```



Notice that this still doesn't work because the overflow occurs before the maximum check.

The way to beat this problem is to convert the image into a higher bit representation, do the manipulations, then convert it back down to the 8-bit representation.

In [4]:
```
racoon_32 = np.array(racoon, dtype=np.int32)
racoon_32 = np.clip(racoon_32 + 20, 0, 255)
racoon_8 = np.array(racoon_32, dtype=np.uint8)

plt.imshow(racoon_8)
plt.show()
```



For convenience, we will leave all data in int32 representation. Then, we will simply define a function that converts the image back to 8-bit representation before plotting.

In [5]:
```
def plotImage(image, title=""):
    im = np.array(image, dtype=np.uint8)
    plt.imshow(im, vmin = 0, vmax = 255)
    plt.title(title)
    plt.show()

racoon = plt.imread('racoon.jpg')
racoon = np.array(racoon, dtype=np.int32)
plotImage(racoon)
```

You are welcome to use the function above for plotting your own color images.

In this lab, you will also need to be able to convert between RGB values and HSB values. We have provided function that allow you to easily go back and forth while staying in the 0-255 representation for images. You are welcome to use the functions below in this lab.

In [6]:
```
def toHSB(image):
    from matplotlib import colors
    temp = 255*colors.rgb_to_hsv(image/255.0)
    return temp.astype(np.int32)

def toRGB(image):
    from matplotlib import colors
    temp = 255*colors.hsv_to_rgb(image/255.0)
    return temp.astype(np.int32)
```

Now that you have that understanding, you are ready to start the lab.

Implement each of the following functions. Use the provided test cases to test your functions.

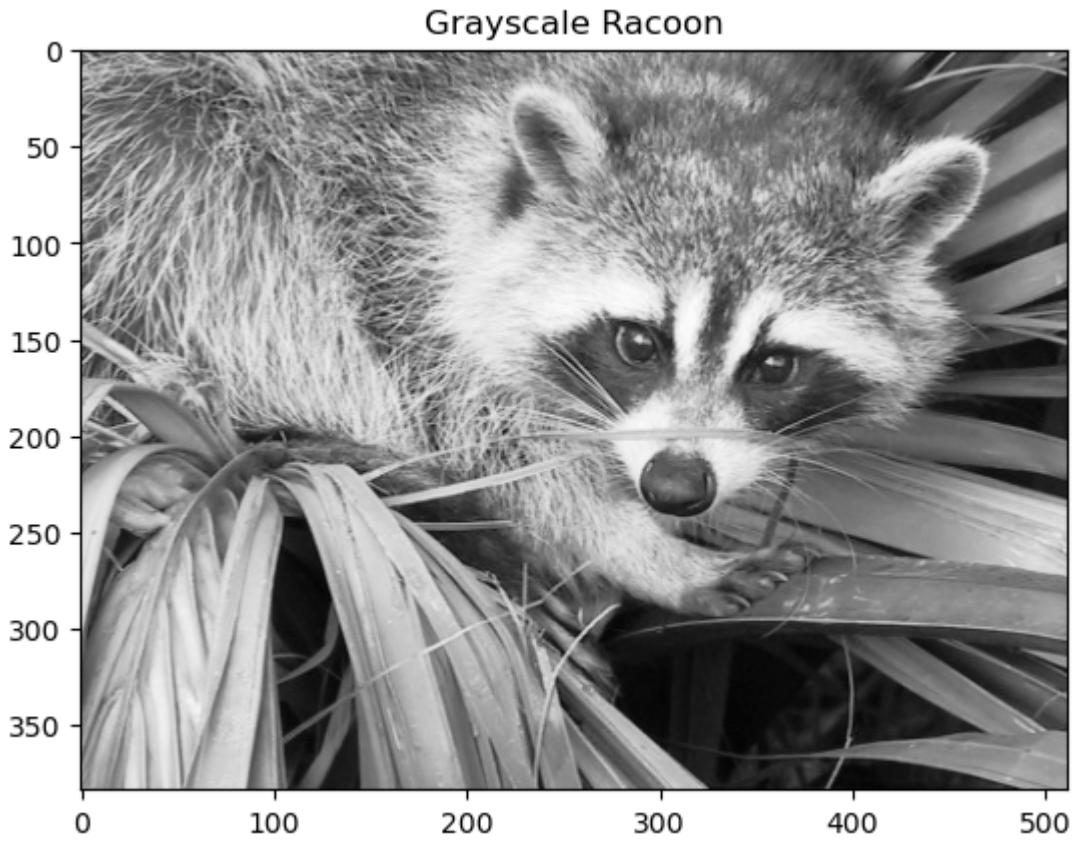
Function 1: Convert to grayscale

Takes in a color image and returns a grayscale image using the following formula:

Gray = 0.299 Red + 0.587 Green + 0.114 Blue

In [7]:
```
def toGrayScale(image):
    return 0.299 * image[:, :, 0] + 0.587 * image[:, :, 1] + 0.114 * image[:, :, 2]
```

In [8]:
```
# Test Case
gray_racoon = toGrayScale(racoon)
plt.imshow(gray_racoon,cmap="Greys_r",vmin=0,vmax=255)
plt.title("Grayscale Racoon")
plt.show()
```

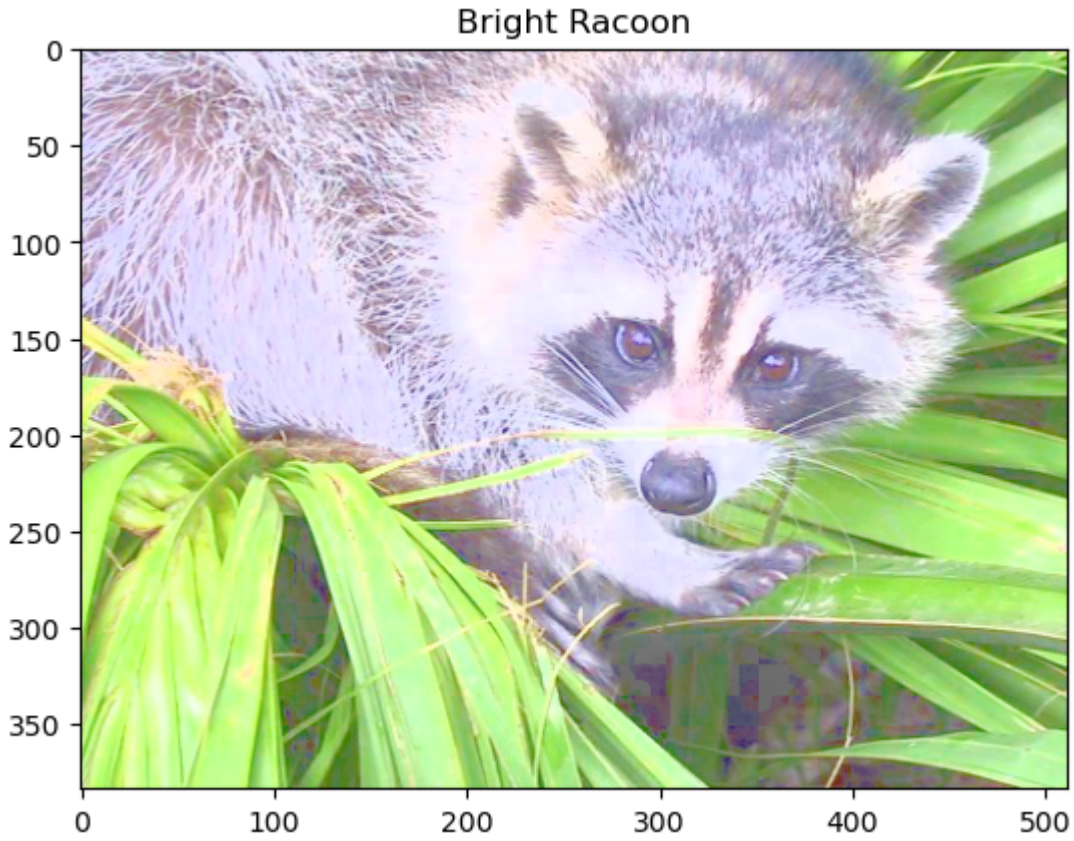


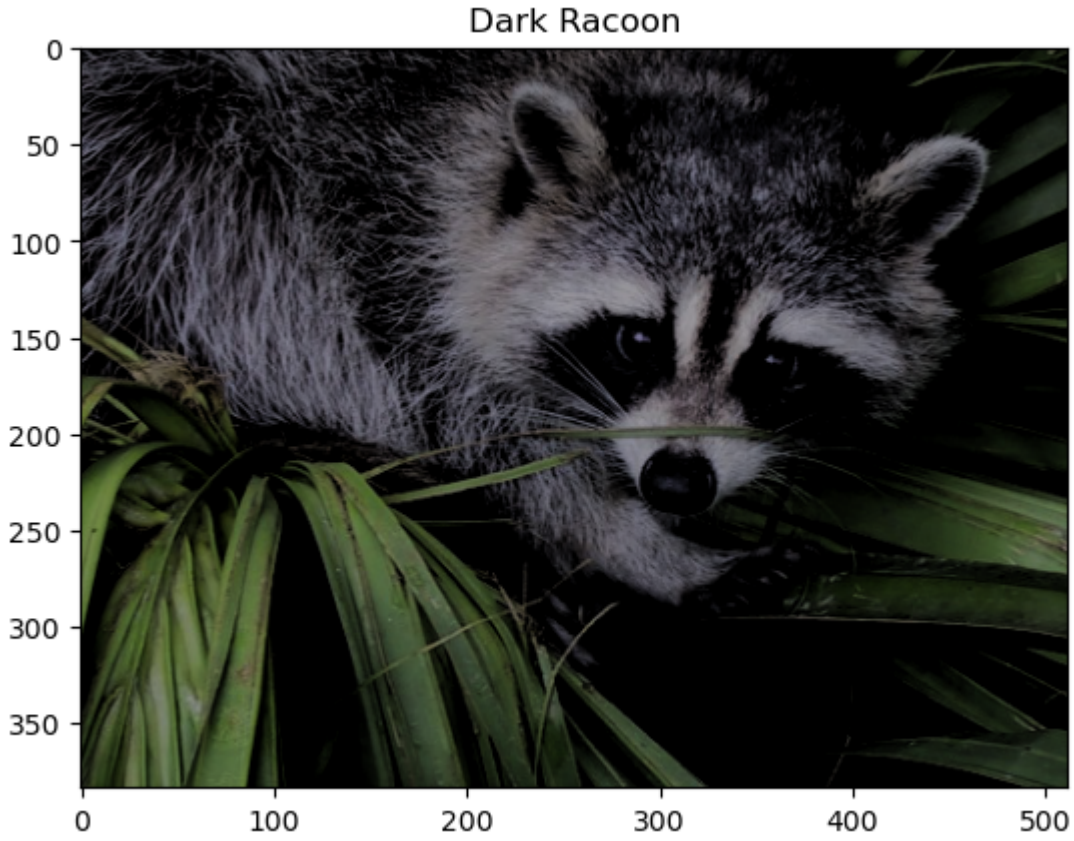
Function 2: Brightness Adjustment

Takes in a color image and returns the brightened version of that image according to a passed in parameter. Use a max image value of 255.

In [9]:
```
def brightAdjust(image,c):
    hsb_image = toHSB(image)
    hsb_image[:, :, 2] = np.clip(hsb_image[:, :, 2] + c, 0, 255)
    rgb_image = toRGB(hsb_image)
    rgb_image = np.clip(rgb_image, 0, 255).astype(np.uint8)
    return rgb_image
```

In [10]:
```
# Test Case
bright_racoon = brightAdjust(racoon,100)
plotImage(bright_racoon, "Bright Racoon")
dark_racoon = brightAdjust(racoon,-100)
plotImage(dark_racoon, "Dark Racoon")
```





Function 3: Contrast Adjustment

Takes in a color image and returns the contrasted version of that image according to a passed in parameter. Use a max image value of 255.

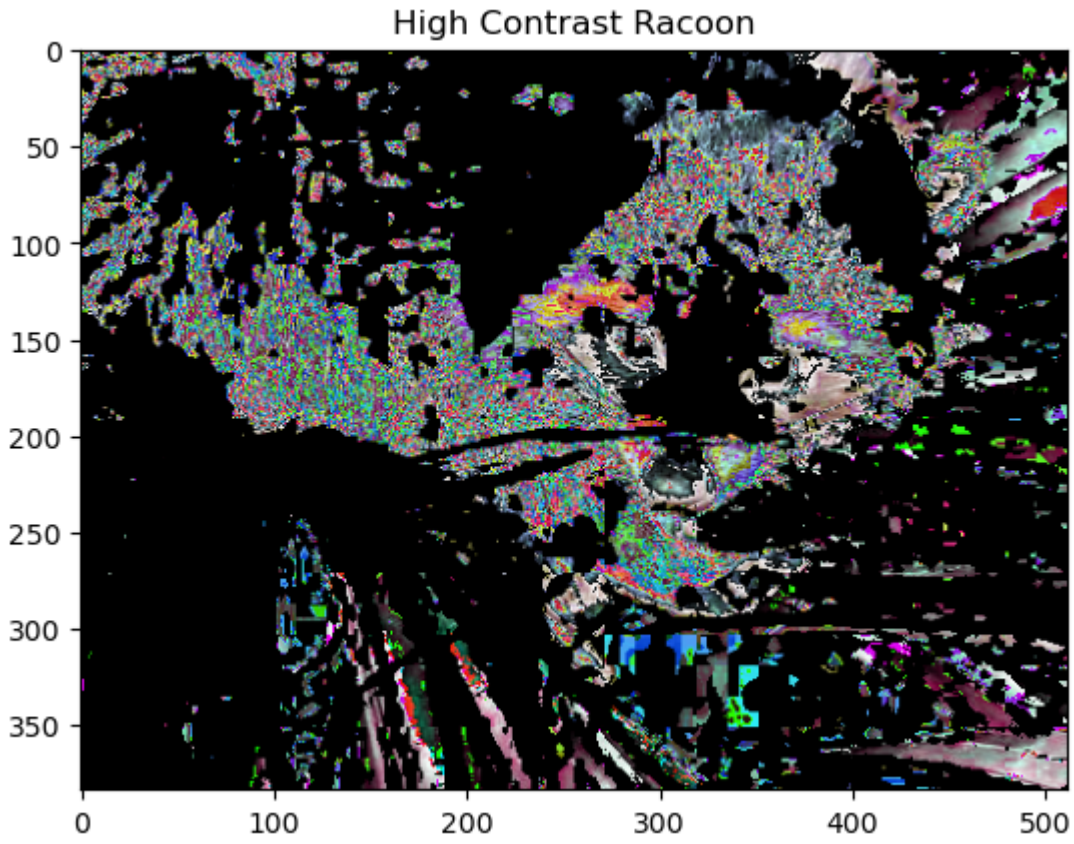
Also, rather than a straight linear operation, we will use a mapping similar to what Photoshop does. In particular, the contrast will be in the range [-100,100] where 0 denotes no change, -100 denotes complete loss of contrast, and 100 denotes maximum enhancement (8x multiplier). If c is the contrast parameter, then the level operation applied is:

$$s = \left(\frac{c+100}{100}\right)^4 (r - 128) + 128$$

Make sure you work in floating point, not integers. Integer division would not be very acurate.

In [11]:
```
def contrastAdjust(image, c):
    im = toHSB(image)

    im= np.clip(im,0,255)

    s= ((c + 100) / 100)**4 * (im - 128) + 128
    return toRGB(s)
```

In [13]:
```
contrast_racoon = contrastAdjust(racoon,30)
plotImage(contrast_racoon, "High Contrast Racoon")
```

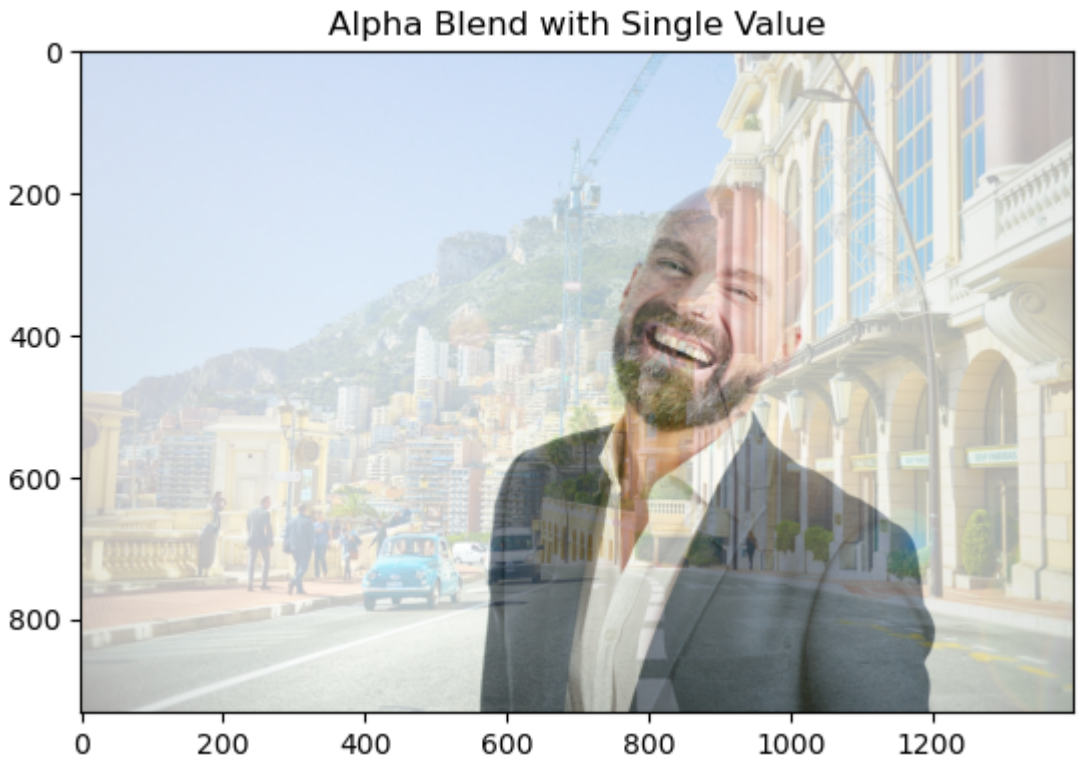


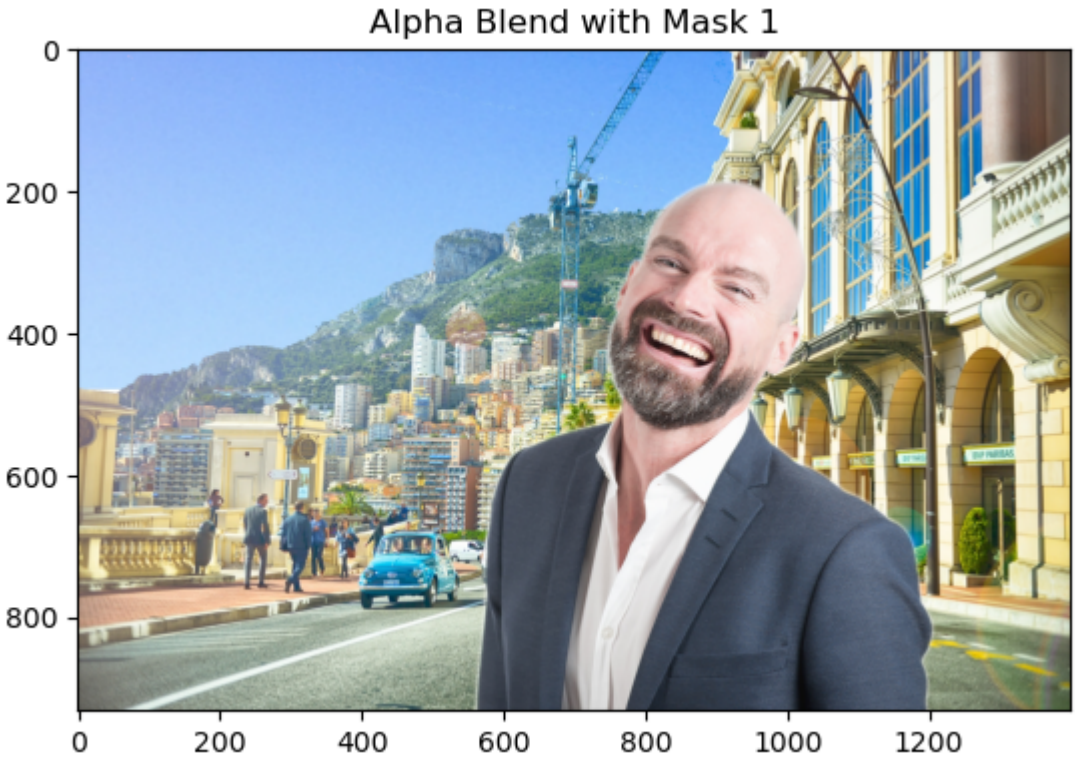
Function 4: Image Blending

Takes in 2 color images of the same size. Given an alpha value it returns the blended image according to the alpha value. Note that your alpha value can be a single number or a mask image of the same size. The alpha values will be between 0 and 1.

In [14]:
```python
def alphaBlend(image1, image2, alpha = .5):

    if not isinstance(alpha, np.ndarray):
        alpha = alpha * np.ones_like(image1, dtype=np.float32)

    alpha = np.clip(alpha, 0.0, 1.0)

    blended_image = alpha * image1 + (1 - alpha) * image2

    return blended_image
```

In [15]:
```python
# Test Cases
man = plt.imread("man.jpg")
city = plt.imread("city.jpg")
blended = alphaBlend(man, city, .7)
plotImage(blended, "Alpha Blend with Single Value")

mask1 = plt.imread("alphamask1.jpg")/255.0
blended1 = alphaBlend(man, city, mask1)
plotImage(blended1, "Alpha Blend with Mask 1")

beach = plt.imread("beach.jpg")
boat = plt.imread("boat.jpg")
mask2 = plt.imread("alphamask2.jpg")/255.0
blended2 = alphaBlend(boat, beach, mask2)
plotImage(blended2, "Alpha Blend with Mask 2")
```





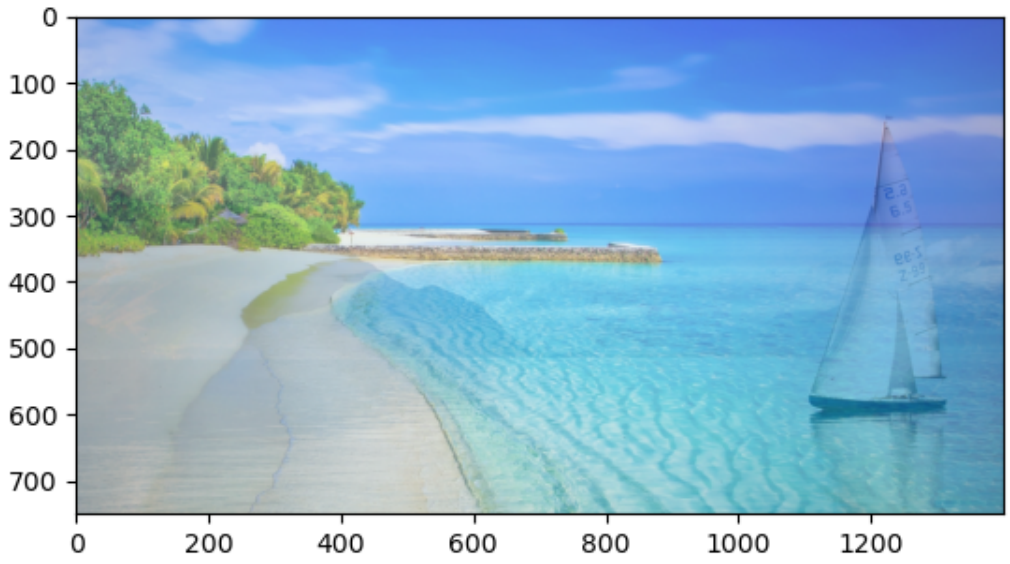
Function 5: Cross Dissolve

Takes in 2 color images of the same size. Returns an array of alpha blend of those two images, where the first picture is an alpha value of 1, the last picture is an alpha value of 0, and the middle pictures slowly decrease until reaching zero. Allow the user to specify the number of steps in the cross dissolve. You can then feed this array into our animation function to view the cross dissolve.

In [32]:
```
def crossDissolve(image1, image2, numsteps = 10):
    blended_images = []
```

```
    for step in range(numsteps + 1):
        alpha = step / numsteps
        blended_image = alpha * image1 + (1 - alpha) * image2
        blended_images.append(blended_image)

    return blended_images
```

In [33]:
```
#Test Case
import matplotlib.animation as animation
%matplotlib notebook

beach = plt.imread("beach.jpg")
boat = plt.imread("boat.jpg")
dis = crossDissolve(beach, boat)

fig = plt.figure()
ims = []
for im in dis:
    im = np.array(im, dtype=np.uint8)
    result = plt.imshow(im, vmin=0, vmax=255, animated=True)
    ims.append([result])

ani = animation.ArtistAnimation(fig, ims, interval=500, blit=True)
plt.show()
```



Because we are working in a notebook, this may not display properly. If necessary, plot the individual pictures to verify that the cross dissolve is working. Also, run the following line of code once you are done to return back to normal plotting functions.

In [19]:
```
%matplotlib inline
```

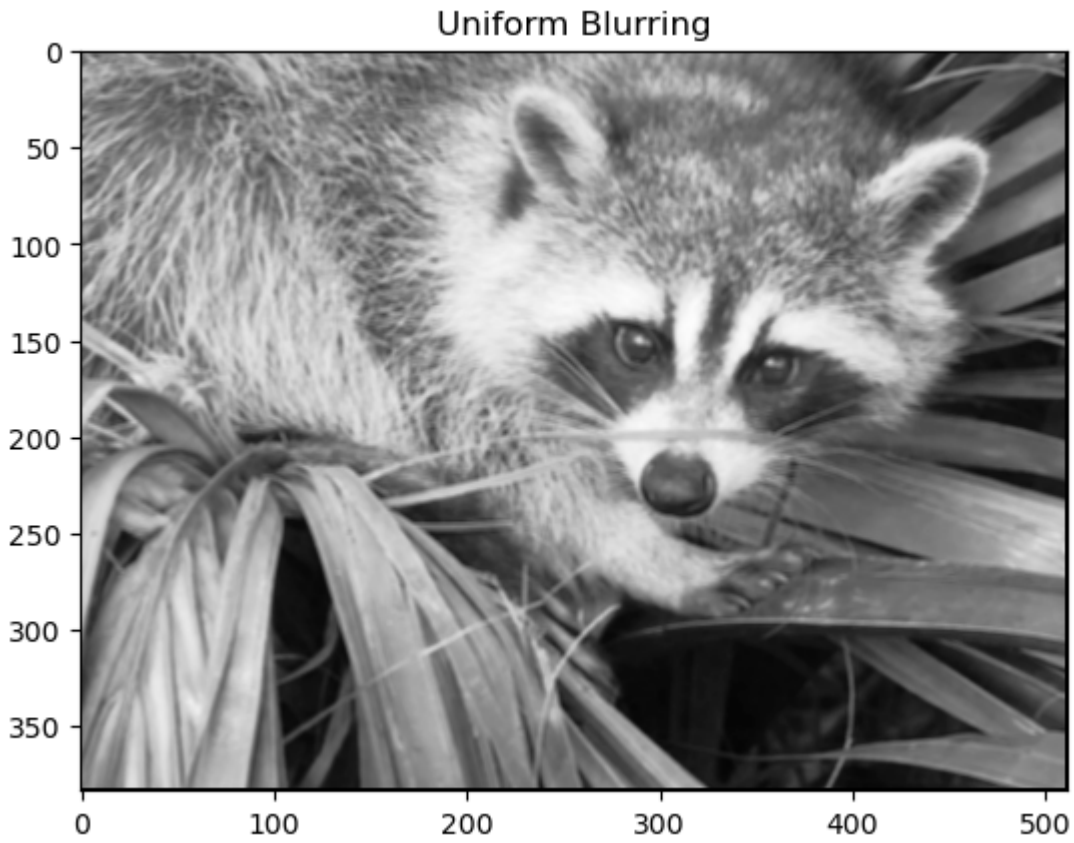
Function 6: Uniform Blurring

Takes in a grayscale image and returns a corresponding result that has been blurred (spatially filtered) using a uniform averaging. Allow the user to specify the size of the kernel (ex. size=3 would give a 3x3 kernel). You can ignore the edge pixels. (Hint: np.sum() may be useful)

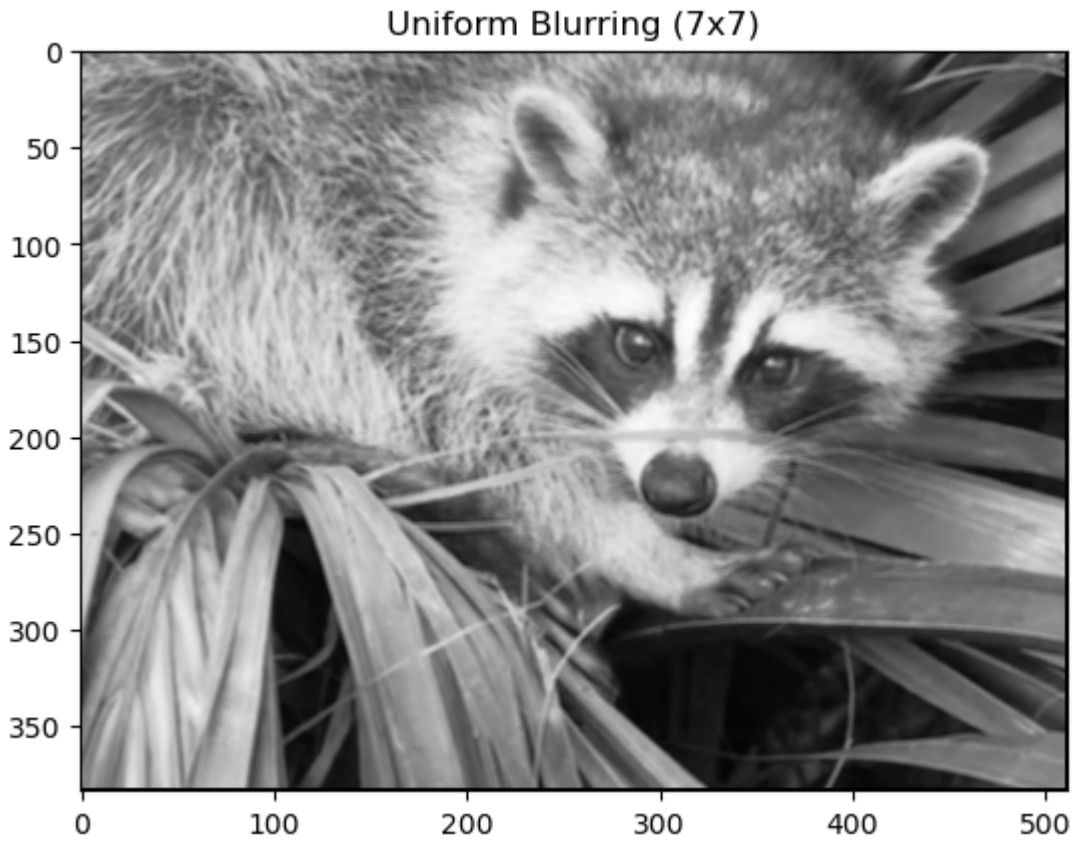
In [22]:
```
def blur(image,size=3):
    # Create a result buffer so that you don't affect the original image
    result = np.zeros(image.shape).astype(np.float32)

    rows, cols = image.shape

    for i in range(1,rows-1):
        for j in range(1,cols-1):
            result[i,j] = np.sum(image[i-1:i+2,j-1:j+2])

    result = result/9

    print(result)

    return result.astype(np.uint8)
```

In [23]:
```
# Test Cases
gray_racoon = racoon[:,:,0]#toGrayScale(racoon)
blur_racoon = blur(gray_racoon)
plt.imshow(blur_racoon,cmap="Greys_r",vmin=0,vmax=255); plt.title("Uniform Blurring")
plt.show()

blur_racoon2 = blur(gray_racoon, 7)
plt.imshow(blur_racoon2,cmap="Greys_r",vmin=0,vmax=255); plt.title("Uniform Blurring (
plt.show()
```

```
[[  0.         0.         0.       ...   0.         0.         0.      ]
 [  0.       127.333336 138.66667  ... 113.       115.44444    0.      ]
 [  0.       112.77778  127.888885 ... 108.333336 113.55556    0.      ]
 ...
 [  0.       109.888885 113.111115 ... 117.666664 118.333336   0.      ]
 [  0.       109.333336 116.333336 ... 119.       119.44444    0.      ]
 [  0.         0.         0.       ...   0.         0.         0.      ]]
```




```
[[  0.         0.         0.       ...   0.         0.         0.      ]
 [  0.       127.333336 138.66667  ... 113.       115.44444    0.      ]
 [  0.       112.77778  127.888885 ... 108.333336 113.55556    0.      ]
 ...
 [  0.       109.888885 113.111115 ... 117.666664 118.333336   0.      ]
 [  0.       109.333336 116.333336 ... 119.       119.44444    0.      ]
 [  0.         0.         0.       ...   0.         0.         0.      ]]
```

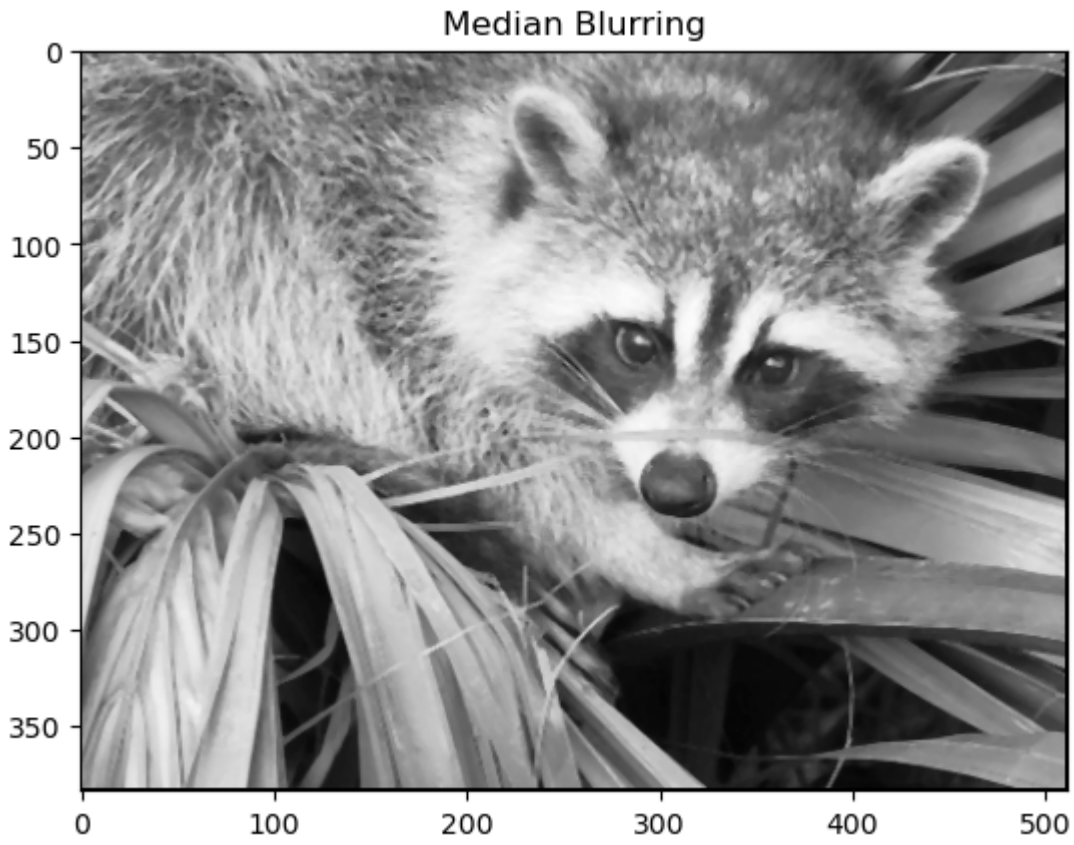


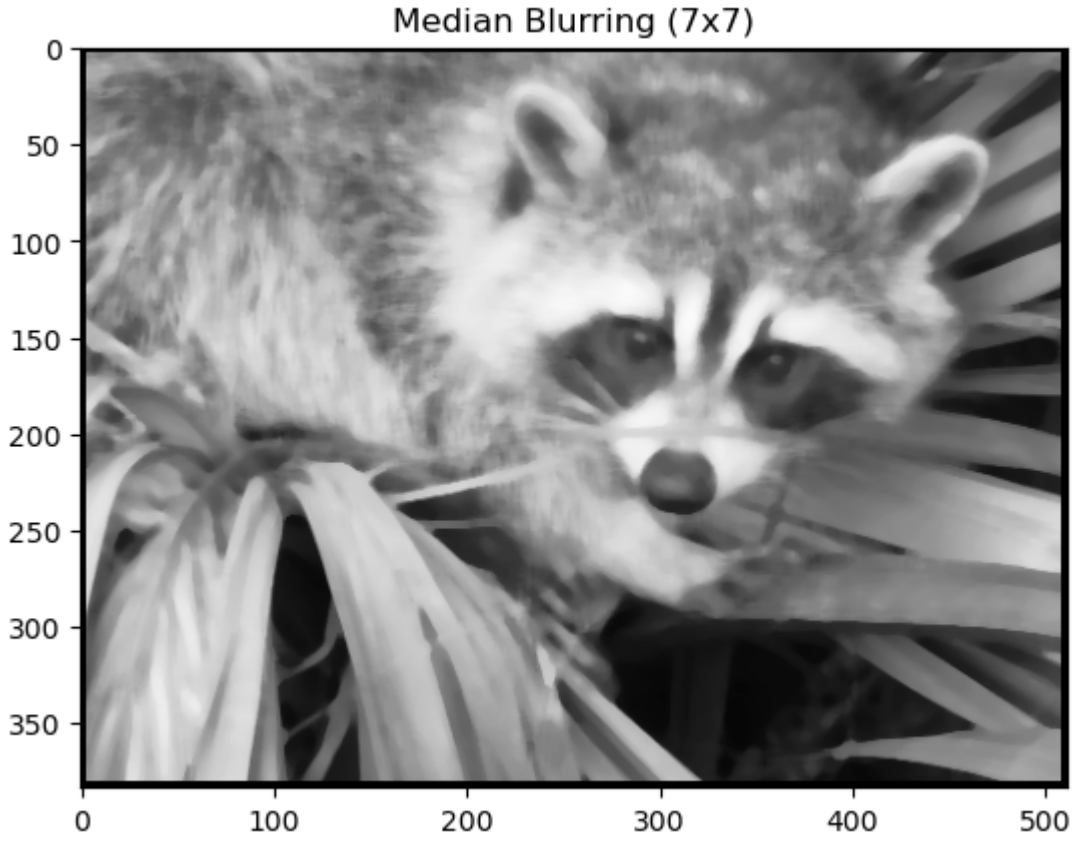
Function 7: Median Filter

Takes in a grayscale image and returns a corresponding result that has been median filtered. Allow the user to specify the size of the kernel (ex. size=3 would give a 3x3 kernel). You can ignore the edge pixels.

In [24]:
```
def medianFilter(image,size=3):
    if size % 2 == 0:
        raise ValueError("size must be an odd number")
    height, width = image.shape
    result = np.zeros_like(image)
    radius = size // 2
    for y in range(radius, height - radius):
        for x in range(radius, width - radius):
            neighborhood = image[y - radius:y + radius + 1, x - radius:x + radius + 1]
            median_value = np.median(neighborhood)
            result[y, x] = median_value
    return result
```

In [25]:
```
# Test Cases
gray_racoon = toGrayScale(racoon)
median_racoon = medianFilter(gray_racoon)
plt.imshow(median_racoon,cmap="Greys_r",vmin=0,vmax=255); plt.title("Median Blurring")
plt.show()

median_racoon2 = medianFilter(gray_racoon, 7)
plt.imshow(median_racoon2,cmap="Greys_r",vmin=0,vmax=255); plt.title("Median Blurring
plt.show()
```





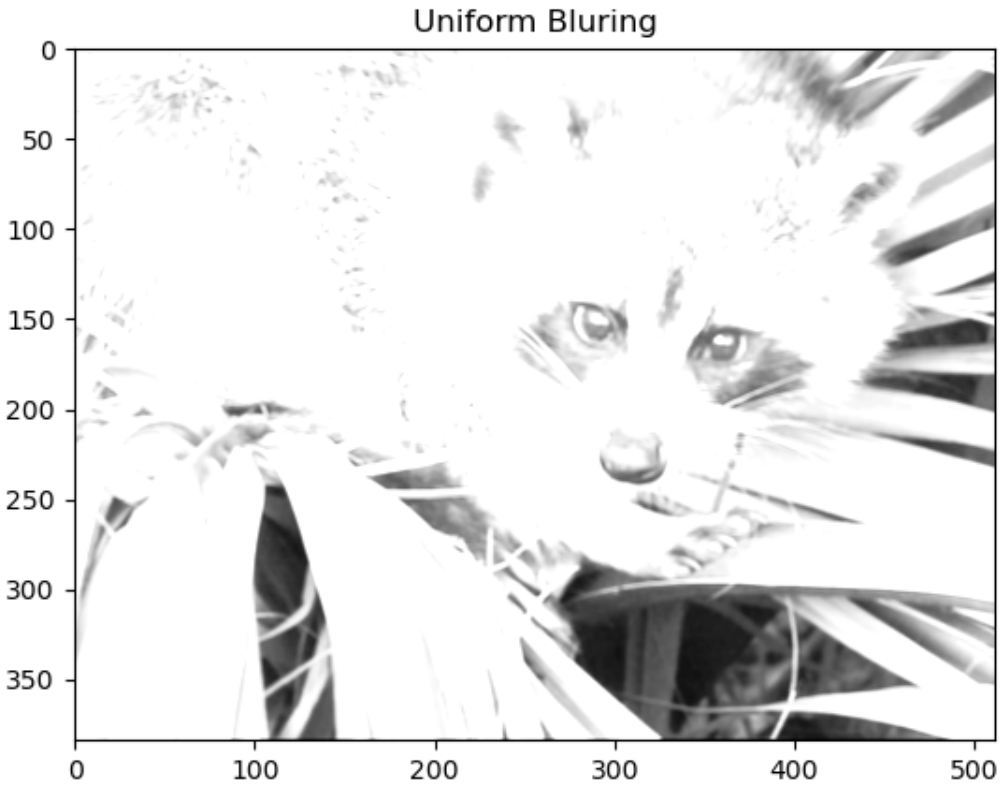
Function 8: General Convolution

Now that you have written a couple of different kernels, write a general convolution function that takes in an image and kernel (stored as a numpy matrix), and performs the appropriate convolution. You can assume the kernel is 3x3 if you would like, but it is not much harder to do a general size kernel as well.

In [43]:
```
def convolution(image,kernel):

# Get the dimensions of the input image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Calculate the padding to ensure the output has the same size as the input
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    # Create an output array with the same shape as the input image
    output = np.zeros_like(image)

    # Pad the input image with zeros
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width, pad_width)), m

    # Perform convolution
    for i in range(image_height):
        for j in range(image_width):
            # Extract the region of interest from the padded image
            roi = padded_image[i:i+kernel_height, j:j+kernel_width]

            # Apply the kernel to the ROI and store the result in the output array
            output[i, j] = np.sum(roi * kernel)

    return output
```

To make sure your general convolution is working, compare the following test case with your original blur results.

In [44]:
```
# Test Cases
gray_racoon = toGrayScale(racoon)

blur_kernel = np.matrix([[1, 1, 1],
                         [1, 1, 1],
                         [1, 1, 1]])

blur_racoon2 = convolution(gray_racoon, blur_kernel)/9
plt.imshow(blur_racoon2,cmap="Greys_r",vmin=0,vmax=255); plt.title("Uniform Bluring")
plt.show()
```

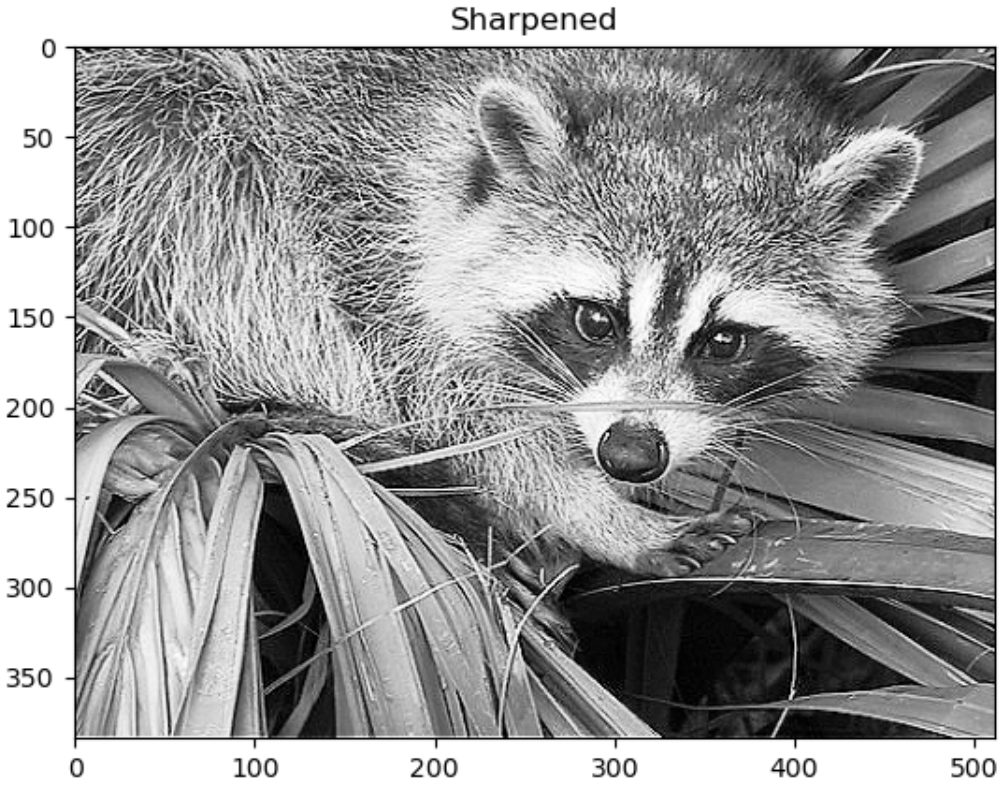


Function 9: Sharpening

Takes in a grayscale image and returns a corresponding result that has been sharpened using an unsharp masking kernel that has a 6 in the middle and -1s for the four-connected neighbors. You can use your general convolution function. You can ignore the edge pixels. **Don't forget to normalize your results.**

In [47]:
```
def sharpen(image):
    kernel = np.array([[0, -1, 0],
                       [-1, 5, -1],
                       [0, -1, 0]])

    sharpened_image = convolution(image, kernel)

    # Normalize the result
    sharpened_image = np.clip(sharpened_image, 0, 255).astype(np.uint8)

    return sharpened_image
```

In [48]:
```
# Test Cases
gray_racoon = toGrayScale(racoon)
sharpen_racoon = sharpen(gray_racoon)
plt.imshow(sharpen_racoon,cmap="Greys_r",vmin=0,vmax=255); plt.title("Sharpened")
plt.show()
```

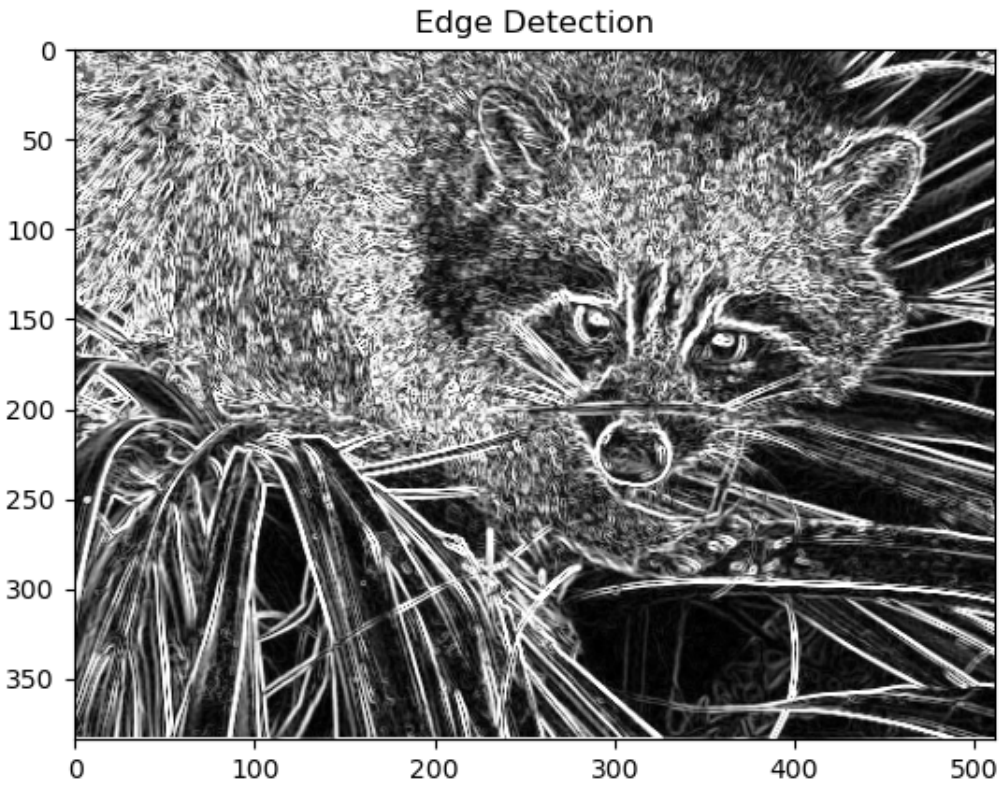


Function 10: Edge Detection

Takes in a grayscale image and returns a corresponding result that shows the gradient magnitude of the input. Use a Sobel kernel. You may afterward scale the result for visibilty if you wish when you demonstrate the function. You can use your general convolution function. You can ignore the edge pixels.

In [49]:
```
def edgeDetect(image):
    # Define the Sobel kernels for horizontal and vertical gradients
    sobel_x = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]])

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

    # Apply convolution with the Sobel kernels for horizontal and vertical gradients
    gradient_x = convolution(image, sobel_x)
    gradient_y = convolution(image, sobel_y)

    # Calculate the gradient magnitude
    gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)

    # Normalize the result
    gradient_magnitude = np.clip(gradient_magnitude, 0, 255).astype(np.uint8)

    return gradient_magnitude
```

In [50]:
```
# Test Cases
gray_racoon = toGrayScale(racoon)
edge_racoon = edgeDetect(gray_racoon)
plt.imshow(edge_racoon,cmap="Greys_r",vmin=0,vmax=255); plt.title("Edge Detection")
plt.show()
```



In [ ]: