# day_4_lecture

January 14, 2020

# 1 Day 4: Advanced Numpy and Plotting

## 1.1 More Numpy

More examples of useful numpy function. The docs to all function can be found under: https://docs.scipy.org/doc/numpy/

```
In [1]: import numpy as np
```

Sometimes, we want to combine arrays together. For that, we can use stacking.

```
In [2]: arrays = [np.random.randn(3, 4) for x in range(10)] # create 10 array with shape (3,4)

        np.stack(arrays, axis=0).shape # specify where the new axis will be created

Out[2]: (10, 3, 4)

In [3]: np.stack((arrays[0],arrays[1]))

Out[3]: array([[[-1.56254981,  0.63903137,  0.10649677, -0.60159657],
                [-1.51743291, -1.80964567,  1.8827107 , -0.41040602],
                [ 0.28098785,  0.44528155,  0.33501681,  0.46723945]],

               [[ 0.30020535, -0.60786912, -0.41944632, -0.16965588],
                [-0.90997145,  1.27928062, -0.54854871,  0.63767082],
                [-0.83580798, -1.34163416,  0.7109665 , -0.68488355]]])
```

We can also use `vstack` and `hstack` to stack arrays without creating a new axis. As always, we need to make sure the dimensions are correct.

```
In [4]: a = np.array([1, 2, 3])
        b = np.array([2, 3, 4])
        np.vstack((a,b))

Out[4]: array([[1, 2, 3],
               [2, 3, 4]])

In [5]: np.hstack((a,b))
```

```
Out[5]: array([1, 2, 3, 2, 3, 4])
```

If we want to repeat elements of an array mutiple times, we can use the `repeat` function.

```
In [6]: np.repeat(3, 4) # simple repeat, like with lists
```

```
Out[6]: array([3, 3, 3, 3])
```

```
In [7]: x = np.array([[1,2],[3,4]])
```

```
In [8]: np.repeat(x, 2) # if no axis is specified, the output is a flattened array
```

```
Out[8]: array([1, 1, 2, 2, 3, 3, 4, 4])
```

```
In [9]: y = np.repeat(x, 3, axis=0) # if axis is given, the repeat value is broadcasted
        y
```

```
Out[9]: array([[1, 2],
               [1, 2],
               [1, 2],
               [3, 4],
               [3, 4],
               [3, 4]])
```

```
In [10]: np.repeat(y, [3,2], axis=1) # we can also use a list of ints to specify the repeats for
```

```
Out[10]: array([[1, 1, 1, 2, 2],
                [1, 1, 1, 2, 2],
                [1, 1, 1, 2, 2],
                [3, 3, 3, 4, 4],
                [3, 3, 3, 4, 4],
                [3, 3, 3, 4, 4]])
```

Similar to repeating, we can also repeat an entire array with the `tile` function

```
In [11]: a = np.array([0, 1, 2])
         np.tile(a, 2)
```

```
Out[11]: array([0, 1, 2, 0, 1, 2])
```

```
In [12]: np.tile(a, (2, 2)) # we ca also multiple dimension in which the array should be tiled
```

```
Out[12]: array([[0, 1, 2, 0, 1, 2],
                [0, 1, 2, 0, 1, 2]])
```

```
In [13]: np.tile(a, (2, 2, 2)).shape
```

```
Out[13]: (2, 2, 6)
```

```
In [14]: # Create checkerboard with tiling

         mini_checker = np.array([[0,1],[1,0]])
         np.tile(mini_checker,(4,4))
```

```
Out[14]: array([[0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0, 1, 0, 1],
                [1, 0, 1, 0, 1, 0, 1, 0]])
```

To create a multi dimensional coordinate system fast, we can use the meshgrid

```
In [17]: x = np.arange(15)
         y = np.arange(10,20)
         xx, yy = np.meshgrid(x,y)
         xx.shape, yy.shape
```

```
Out[17]: ((10, 15), (10, 15))
```

```
In [18]: coord = np.stack((xx,yy), axis=2)
         coord
```

```
Out[18]: array([[[ 0, 10],
                 [ 1, 10],
                 [ 2, 10],
                 [ 3, 10],
                 [ 4, 10],
                 [ 5, 10],
                 [ 6, 10],
                 [ 7, 10],
                 [ 8, 10],
                 [ 9, 10],
                 [10, 10],
                 [11, 10],
                 [12, 10],
                 [13, 10],
                 [14, 10]],

                [[ 0, 11],
                 [ 1, 11],
                 [ 2, 11],
                 [ 3, 11],
                 [ 4, 11],
                 [ 5, 11],
                 [ 6, 11],
                 [ 7, 11],
                 [ 8, 11],
                 [ 9, 11],
                 [10, 11],
                 [11, 11],
 [12, 11],
 [13, 11],
 [14, 11]],

[[ 0, 12],
 [ 1, 12],
 [ 2, 12],
 [ 3, 12],
 [ 4, 12],
 [ 5, 12],
 [ 6, 12],
 [ 7, 12],
 [ 8, 12],
 [ 9, 12],
 [10, 12],
 [11, 12],
 [12, 12],
 [13, 12],
 [14, 12]],

[[ 0, 13],
 [ 1, 13],
 [ 2, 13],
 [ 3, 13],
 [ 4, 13],
 [ 5, 13],
 [ 6, 13],
 [ 7, 13],
 [ 8, 13],
 [ 9, 13],
 [10, 13],
 [11, 13],
 [12, 13],
 [13, 13],
 [14, 13]],

[[ 0, 14],
 [ 1, 14],
 [ 2, 14],
 [ 3, 14],
 [ 4, 14],
 [ 5, 14],
 [ 6, 14],
 [ 7, 14],
 [ 8, 14],
 [ 9, 14],
 [10, 14],
 [11, 14],
 [12, 14],
 [13, 14],
 [14, 14]],

[[ 0, 15],
 [ 1, 15],
 [ 2, 15],
 [ 3, 15],
 [ 4, 15],
 [ 5, 15],
 [ 6, 15],
 [ 7, 15],
 [ 8, 15],
 [ 9, 15],
 [10, 15],
 [11, 15],
 [12, 15],
 [13, 15],
 [14, 15]],

[[ 0, 16],
 [ 1, 16],
 [ 2, 16],
 [ 3, 16],
 [ 4, 16],
 [ 5, 16],
 [ 6, 16],
 [ 7, 16],
 [ 8, 16],
 [ 9, 16],
 [10, 16],
 [11, 16],
 [12, 16],
 [13, 16],
 [14, 16]],

[[ 0, 17],
 [ 1, 17],
 [ 2, 17],
 [ 3, 17],
 [ 4, 17],
 [ 5, 17],
 [ 6, 17],
 [ 7, 17],
 [ 8, 17],
 [ 9, 17],
 [10, 17],
 [11, 17],
        [12, 17],
        [13, 17],
        [14, 17]],

       [[ 0, 18],
        [ 1, 18],
        [ 2, 18],
        [ 3, 18],
        [ 4, 18],
        [ 5, 18],
        [ 6, 18],
        [ 7, 18],
        [ 8, 18],
        [ 9, 18],
        [10, 18],
        [11, 18],
        [12, 18],
        [13, 18],
        [14, 18]],

       [[ 0, 19],
        [ 1, 19],
        [ 2, 19],
        [ 3, 19],
        [ 4, 19],
        [ 5, 19],
        [ 6, 19],
        [ 7, 19],
        [ 8, 19],
        [ 9, 19],
        [10, 19],
        [11, 19],
        [12, 19],
        [13, 19],
        [14, 19]]])
```

```
In [19]: # Random choice ->
```

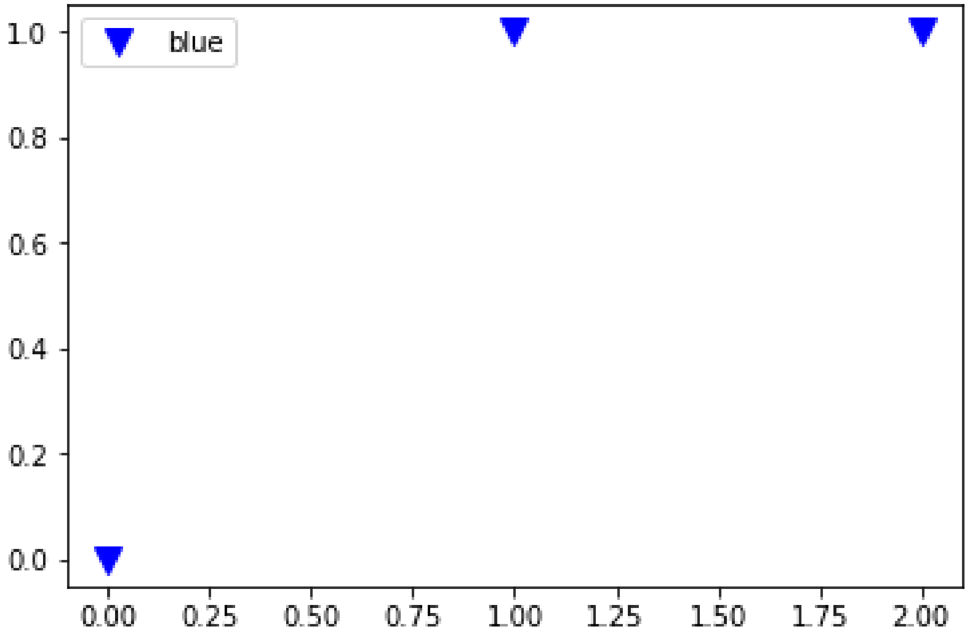
# 2 Plotting

## 2.1 Scatter and Line Plots

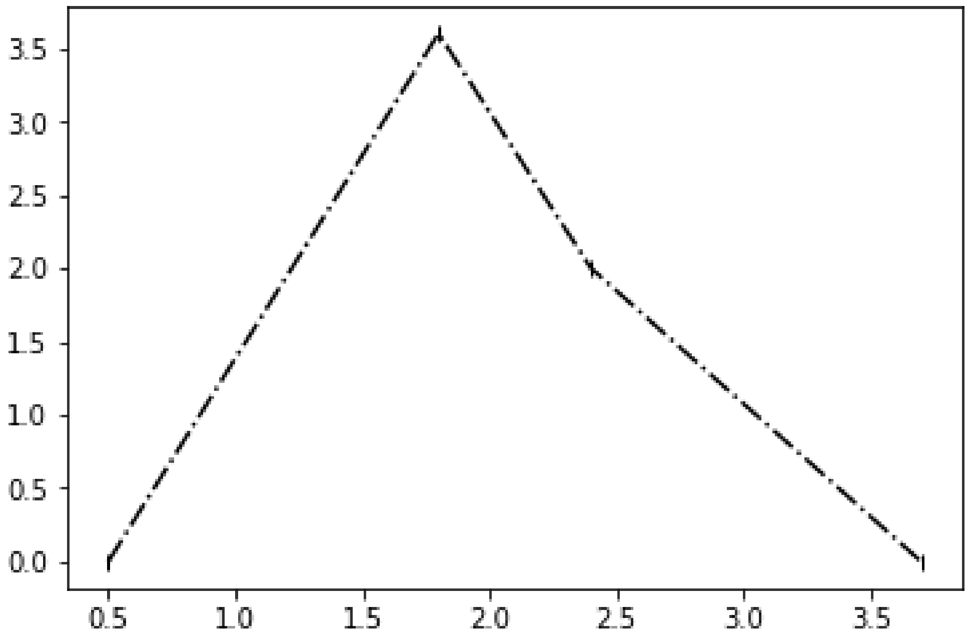
With `plt.plot` we can plot scatter plots and line plots, depending on the input. Useful to show functions(lines) and points (scatter) in datasets. It is the most basic matplotlib function and easy to use! The documentation can be found here: https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.plot.html (there is a lot to adjust to make cool plots!)

```
In [20]: import matplotlib.pyplot as plt
         import matplotlib.image as mpimg
         %matplotlib inline
```

```
In [21]: plt.plot([0,1,2], [0,1,1], marker="v", linestyle="", color="b", markersize=10, label="b
         plt.legend()
         plt.show()
```

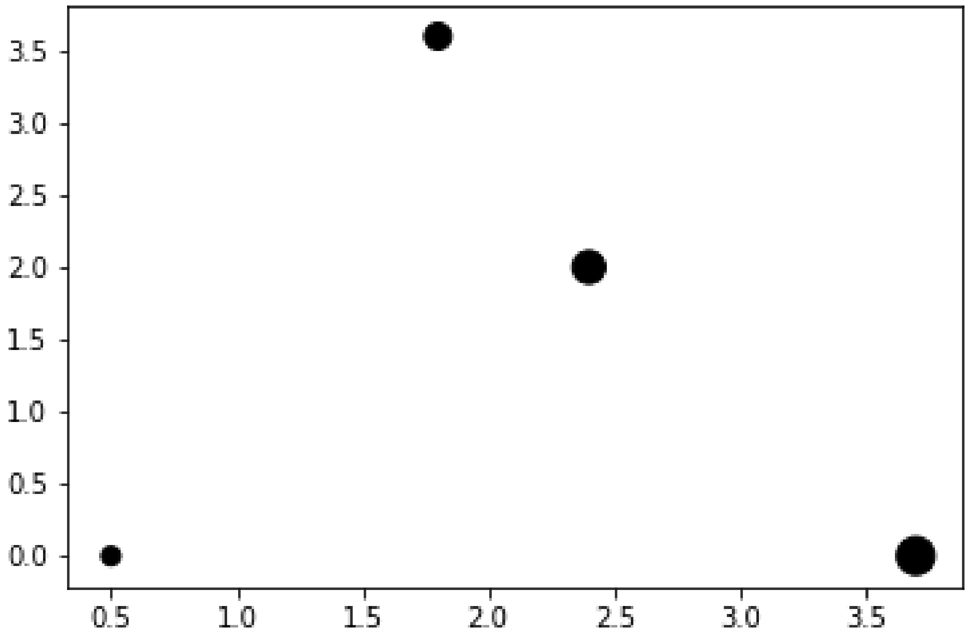


```
In [22]: plt.plot([0.5,1.8,2.4,3.7], [0,3.6,2,0], marker="|", linestyle="-.", color="k")
         plt.show()
```



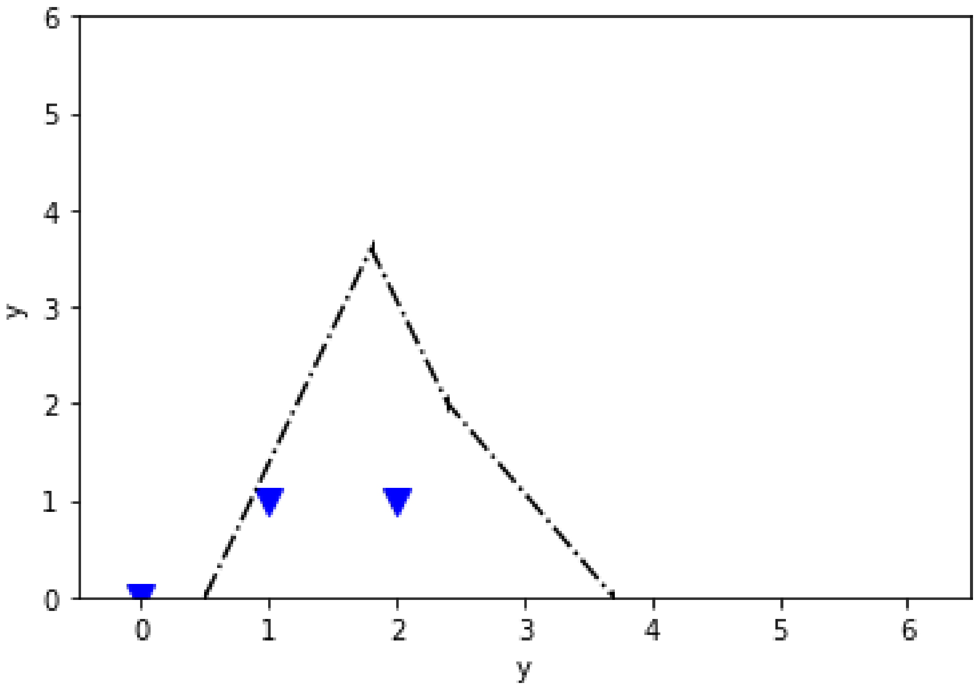
Scatterplots are almost identical to line plots, but have more advanced options. For example, we can change points style(such as color,size etc.) depending on other values.

```
In [23]: plt.scatter([0.5,1.8,2.4,3.7], [0,3.6,2,0], marker="o", color="k", s=np.array([1,2,3,4]
         plt.show()
```



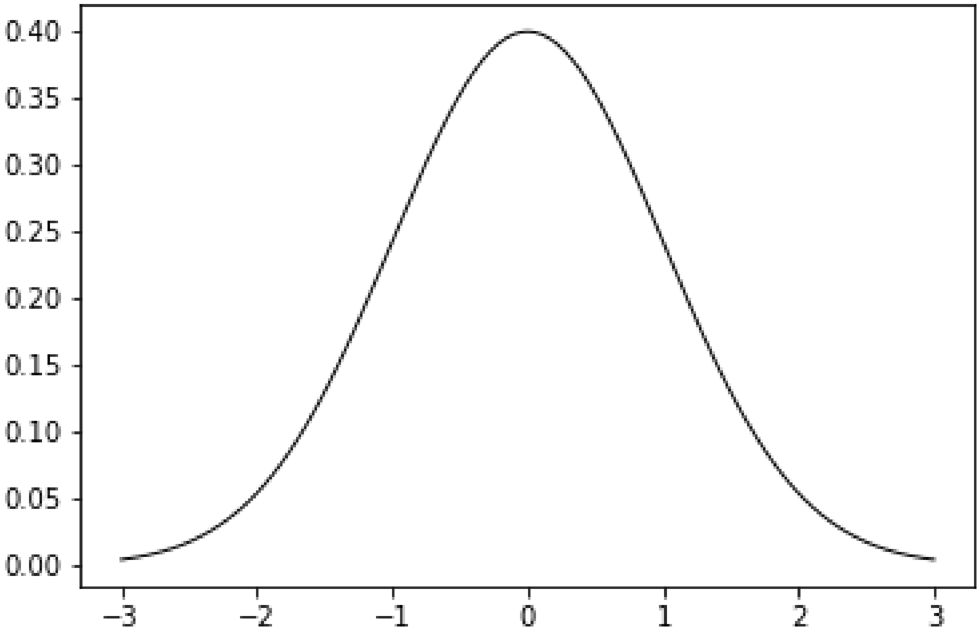
### 2.2 We can also add multiple plots into one figure

```
In [24]: plt.plot([0,1,2], [0,1,1], marker="v", linestyle="", color="b", markersize=10, label="b
         plt.plot([0.5,1.8,2.4,3.7], [0,3.6,2,0], marker="|", linestyle="-.", color="k")

         plt.ylabel("y")
         plt.xlabel("y")
         plt.xlim(-0.5, 6.5)
         plt.ylim(0,6)

         plt.show()
```



If we want to display functons, we can use `linspace` from numpy to create an array of values between start and end value. Different to `arange`, `linspace` creates a set number of values between start and end, while `arange` goes a certain step size.

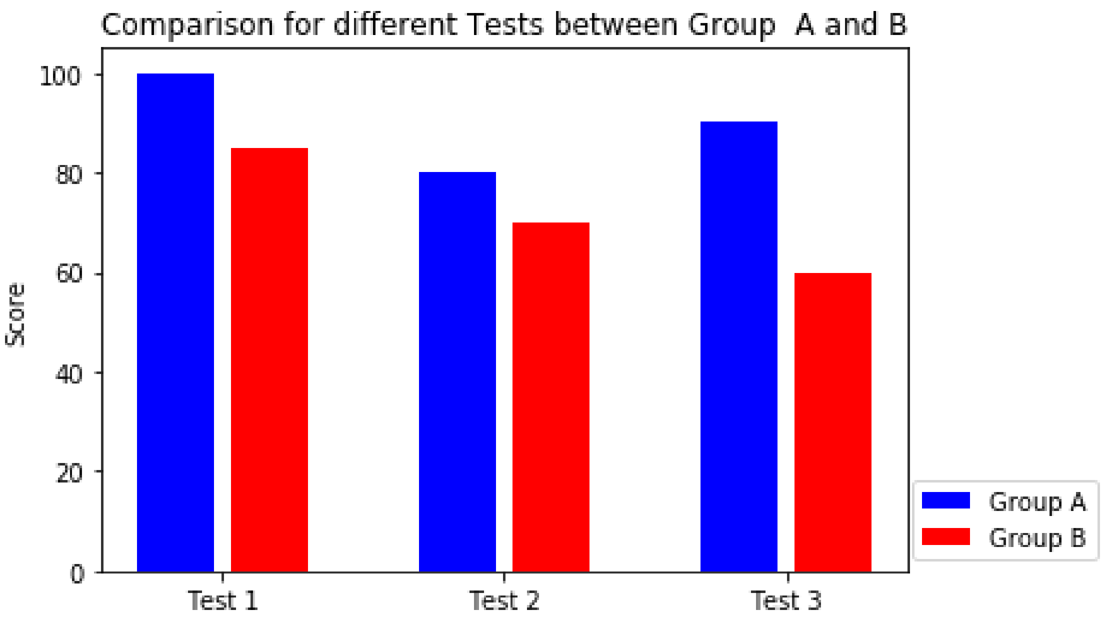
```
In [25]: # display a gaus curve
         mu = 0
         variance = 1
         sigma = np.sqrt(variance)
         x = np.linspace(mu - 3*sigma, mu + 3*sigma, 1000) # create 1000 values
         plt.plot(x, 1/(sigma * np.sqrt(2 * np.pi)) *
                  np.exp( - (x - mu)**2 / (2 * sigma**2) ),'k', linewidth=1)
         plt.show()
```



## 3 Bar Plots

Bar plots are good to display and compare values for multiple
https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.bar.html

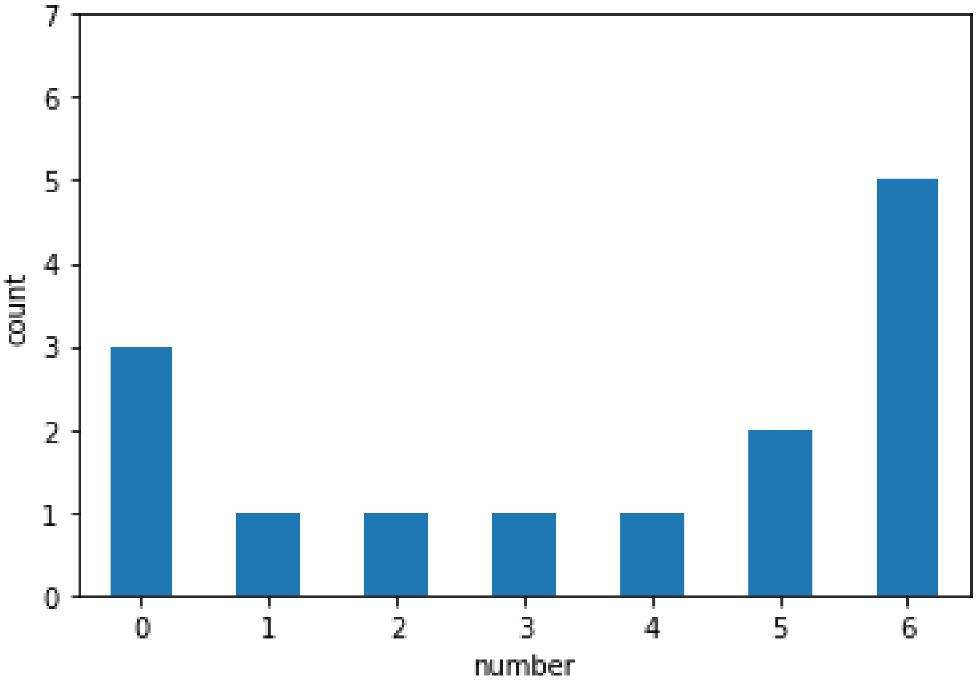
```
In [26]: values_group_A = [100, 80, 90]
         values_group_B = [85, 70, 60]

         position_X_A = [1, 4, 7]
         position_X_B = [2, 5, 8]

         plt.bar(position_X_A, values_group_A, color='b', label='Group A')
         plt.bar(position_X_B, values_group_B, color='r', label='Group B')

         plt.legend(loc='lower right', bbox_to_anchor=(1.25, 0))
         plt.xticks([1.5, 4.5, 7.5], ['Test 1', 'Test 2', 'Test 3'])
         plt.ylabel('Score')
         plt.title('Comparison for different Tests between Group  A and B')

Out[26]: Text(0.5, 1.0, 'Comparison for different Tests between Group  A and B')
```



### 3.1 Histograms

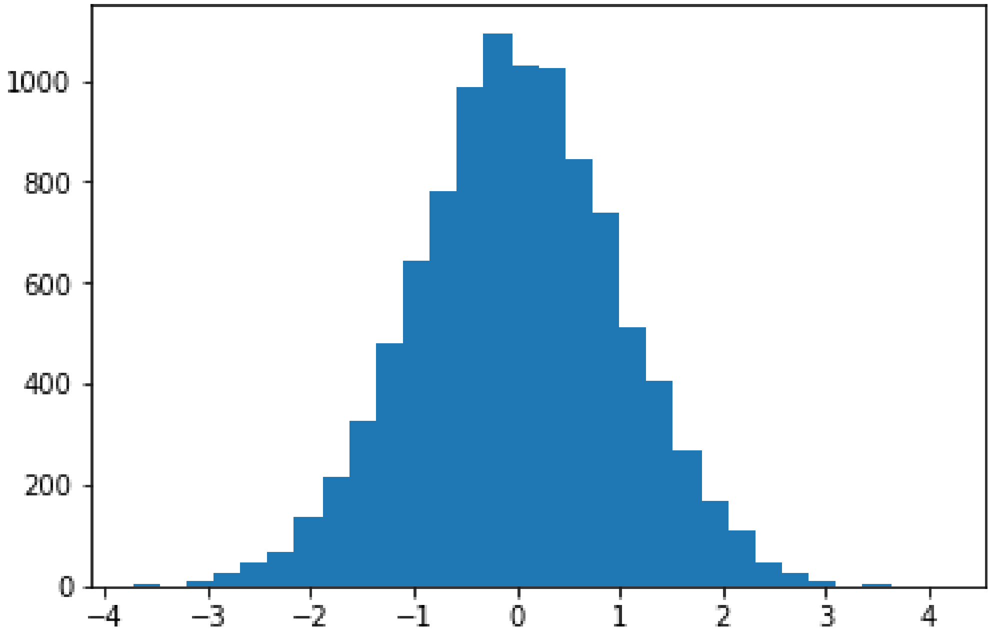
Histograms are useful to display the repetition and distrubution of elements in a dataset. They are very similar to plot bars in appearance, but
The documentation can be found under: https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.hist.html

```
In [27]: import numpy as np

         a = np.array([0,0,0,1,2,3,4,5,5,6,6,6,6,6])
         plt.hist(a, bins=np.arange(8)-0.5, rwidth=0.5) # arange 8, because we have 7 bin-> 8 ed
         plt.ylabel("count")
         plt.xlabel("number")
         plt.xlim(-0.5, 6.5)
         plt.ylim(0,7)

Out[27]: (0, 7)
```



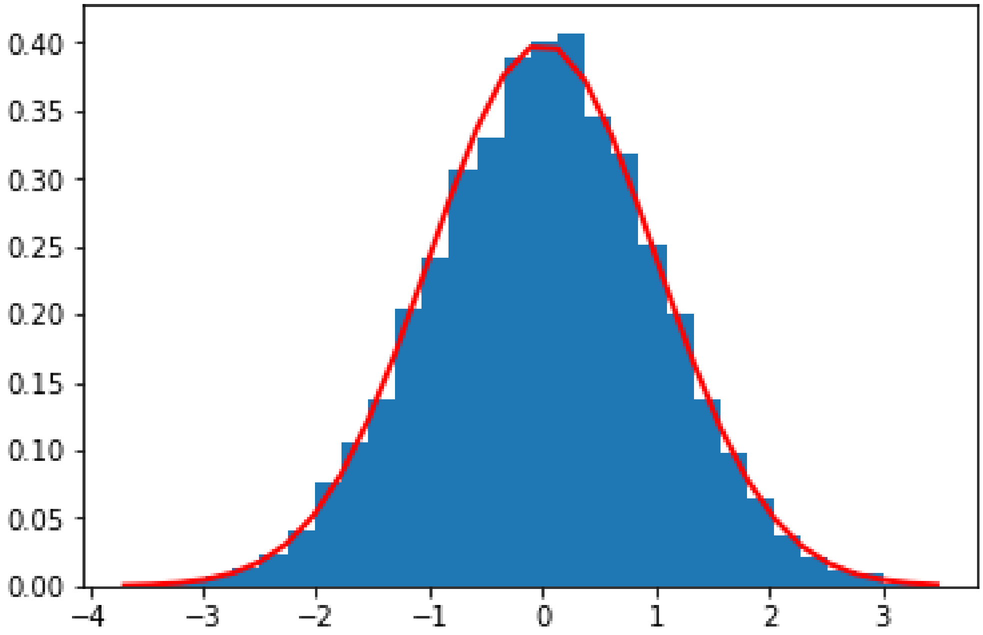
We can also combine histograms with line plots to show trends.

```
In [30]: mu = 0
         sigma = 1
         bins=30
         gauss = np.random.normal(mu, sigma, size=10000)
         count, bins, ignored = plt.hist(gauss, bins=bins)
         plt.show()
```



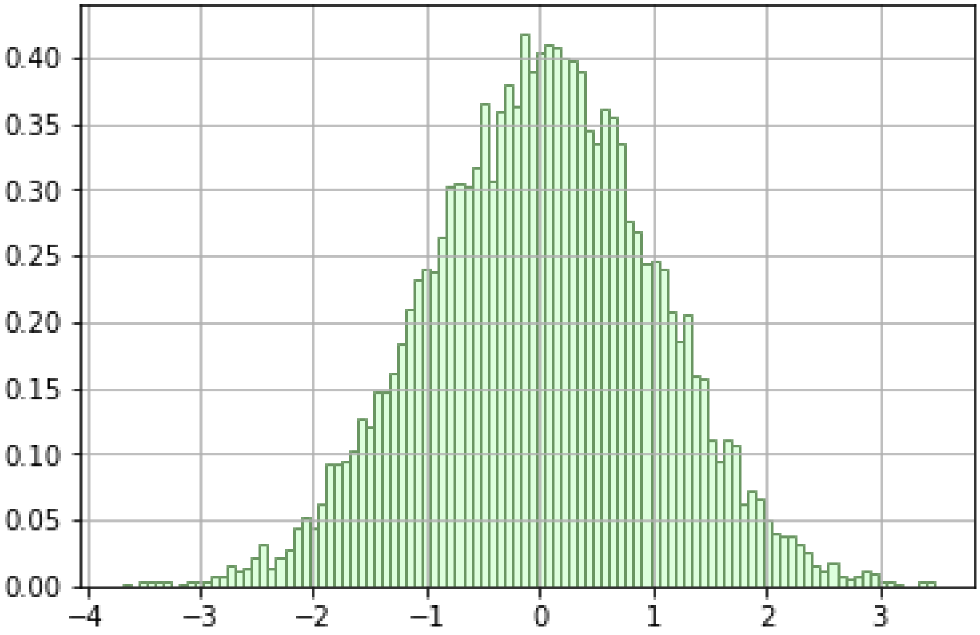
The plot above is not normalized. To get a probability distrubution, we need to set the parameter `density` to `True`.

```
In [33]: mu = 0
         sigma = 1
         bin_n=30
         gauss = np.random.normal(size=10000)
         counts, bins, patches = plt.hist(gauss, bins=bin_n, density=True)
         plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
                        np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
                        linewidth=2, color='r')
         plt.show()
```



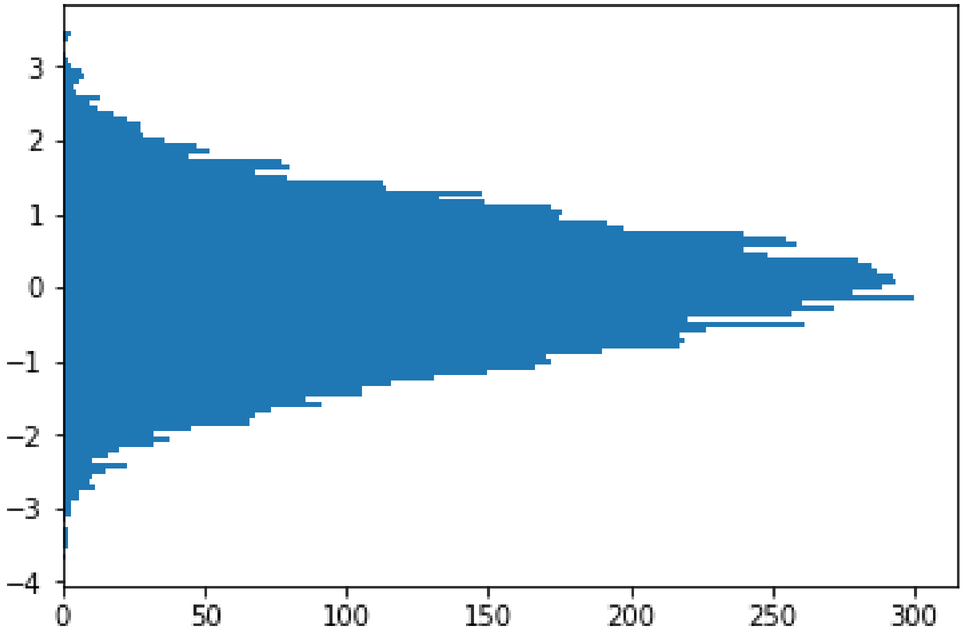
We of couse can also make the plot look more cool!

```
In [34]: plt.hist(gauss,
                  bins=100,
                  density=True,
                  stacked=True,
                  edgecolor="#6A9662",
                  color="#DDFFDD")
         plt.grid()
         plt.show()
```



Or vertical:

```
In [35]: plt.hist(gauss,
                  bins=100,
                  orientation="horizontal")
         plt.show()
```



```
In [ ]:
```

### 3.2 Boxplots

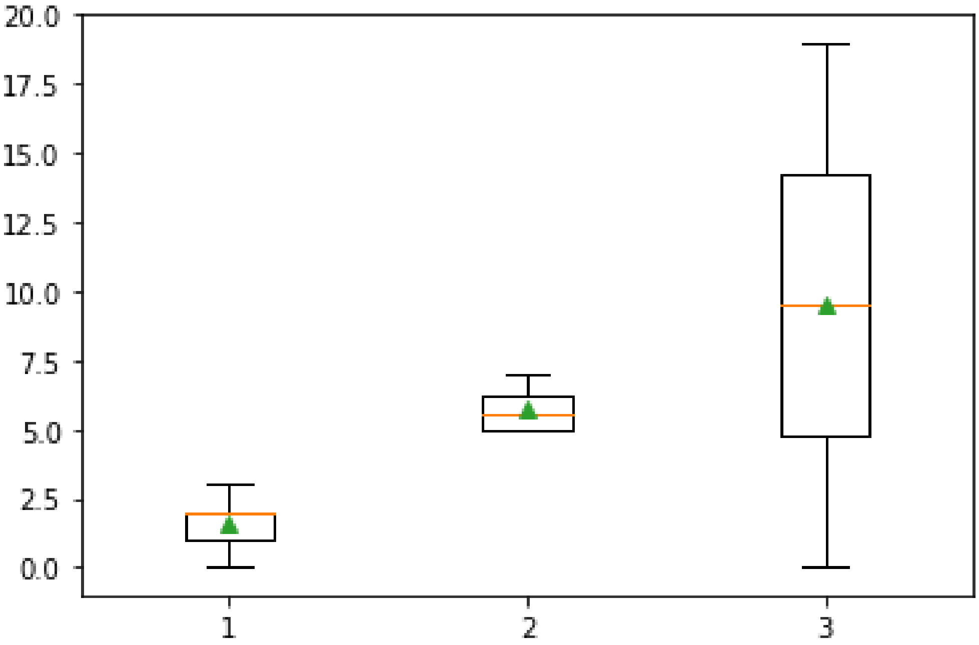
Boxplot are used to display the distribution of (numerical) data by combining key values of the distrubution in one graph. The five key values are: - Minimum - Maximum - Median - Upper quartile and quartil (upper and lower range of 50% of data) - Outliers

Boxplot are helpful to quickly see how data is distributeds.

More on box plots: https://en.wikipedia.org/wiki/Box_plot Documentation: https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.boxplot.html

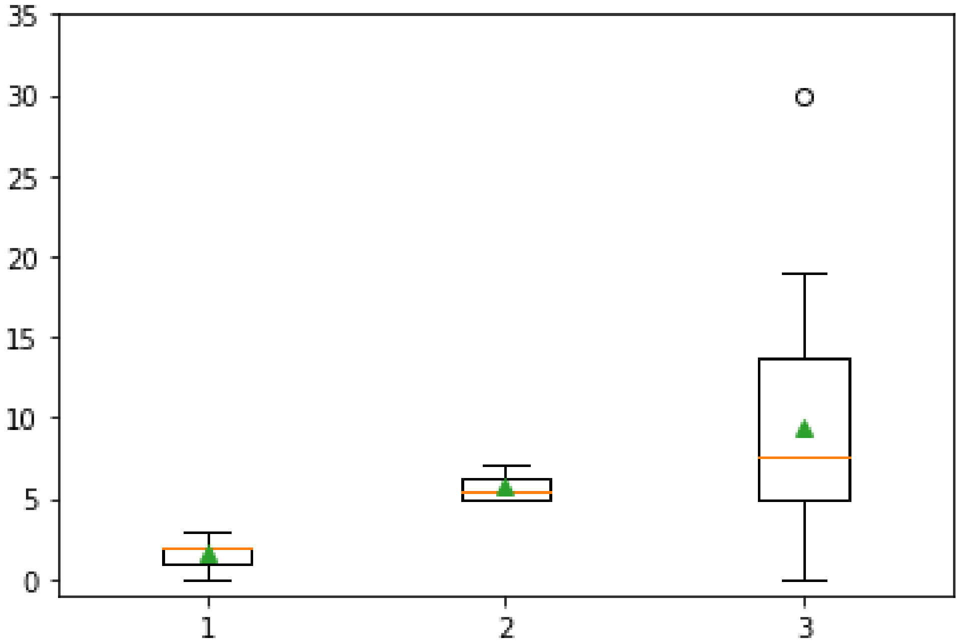
```
In [36]: values = plt.boxplot([[0,1,2,3,2], [5,6,7,5], range(20)], showmeans=True)
         plt.ylim(-1,20)
         print(values["means"][2].get_ydata())
         plt.show()
```

```
[9.5]
```



```
In [37]: #Show some outlier

         values = list(range(20))
         values.extend([5]*5)
         values.append(30)
         values = plt.boxplot([[0,1,2,3,2], [5,6,7,5], values], showmeans=True)
         plt.ylim(-1,35)
         print(values["means"][2].get_ydata())
         plt.show()
```

```
[9.42307692]
```



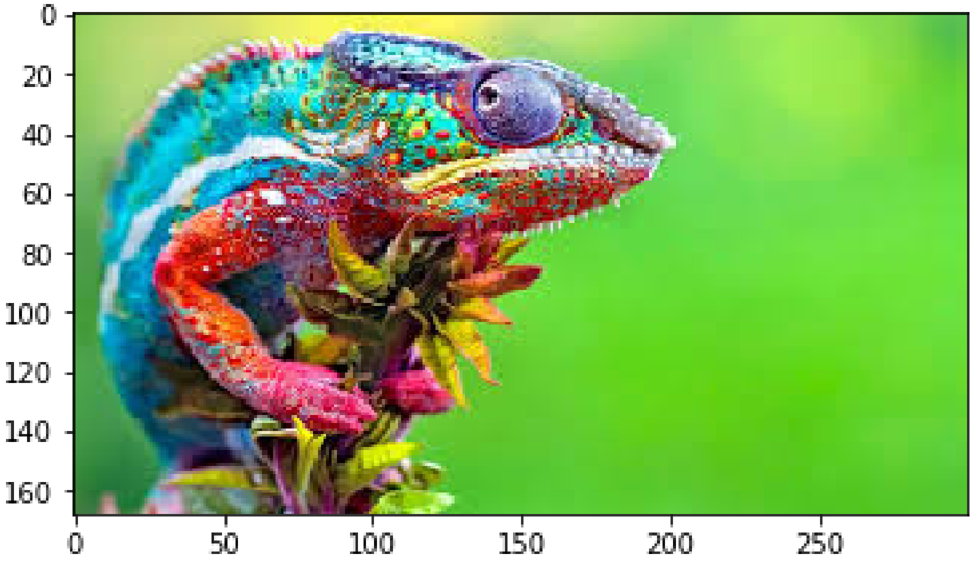
## 3.3 Image Plotting

We can also dislay images with the matplotlib library, and manipulate the images using known numpy mechannics. After all, an image is just an array of RGB-values for each pixel. Therefore we can load images into a multidimensional numpy array. To load a image, we can use the `imread` function from `matplotlib.image`.

```
In [38]: import matplotlib.pyplot as plt
         import matplotlib.image as mpimg
         %matplotlib inline

         img = mpimg.imread('chameleon.jpeg')
         print(img.shape)
         plt.imshow(img)
```

```
(168, 300, 3)
```

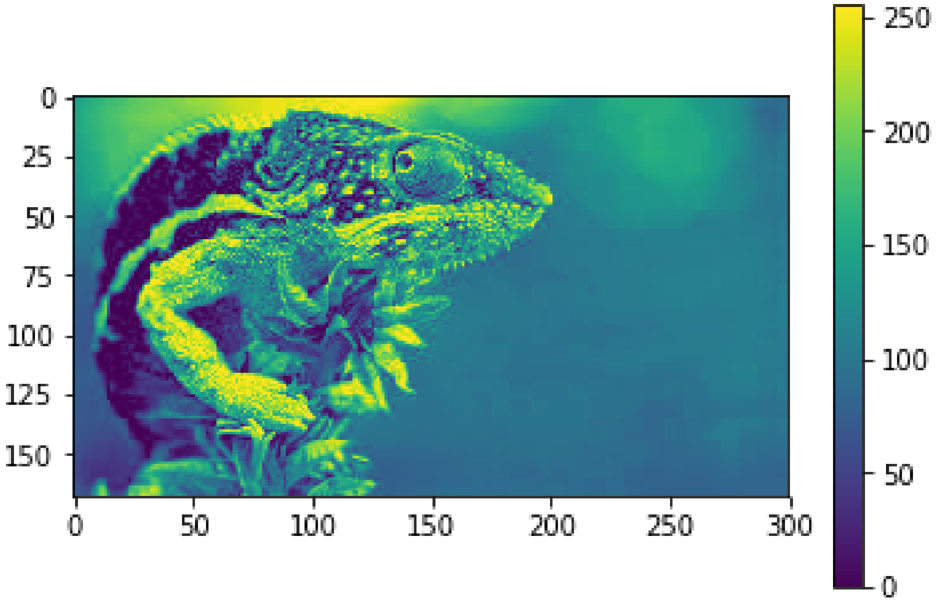
```
Out[38]: <matplotlib.image.AxesImage at 0x7f2728be7908>
```



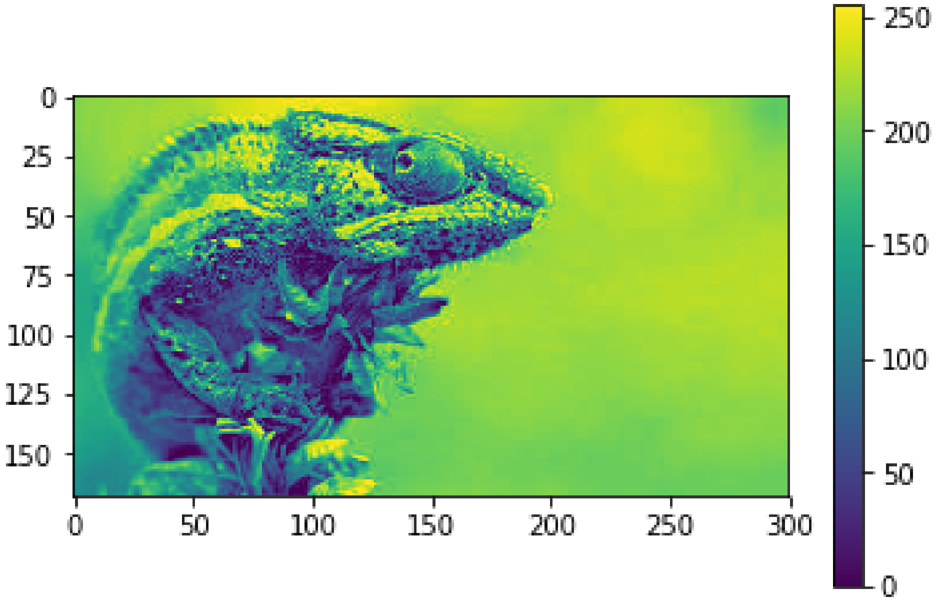
When we look at the shape, we see there are 3 dimensions, while the last dimension has the size 3. There are the RGB values contained.

```
In [39]: plt.imshow(img[:,:,0])
         plt.colorbar()
```

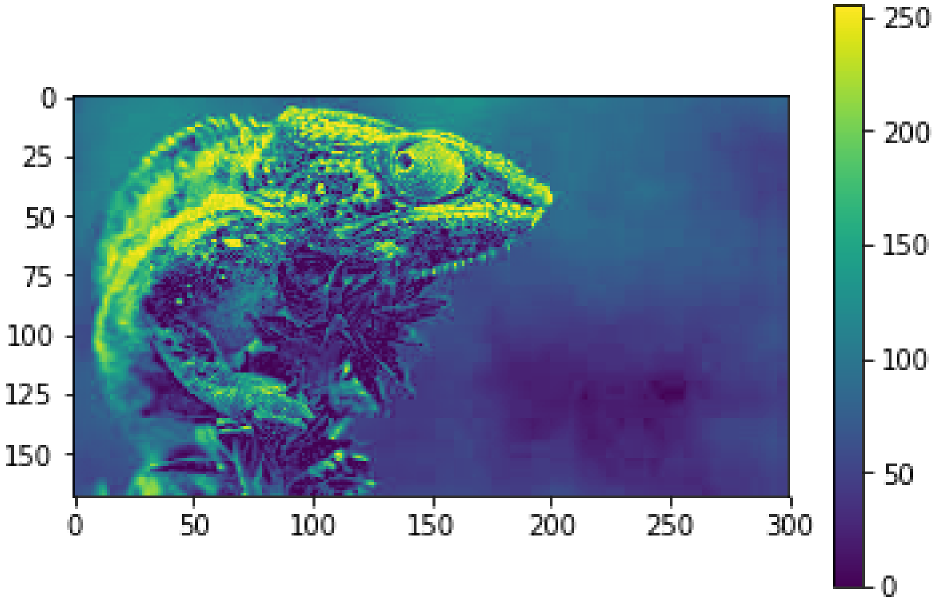
```
Out[39]: <matplotlib.colorbar.Colorbar at 0x7f2728bcc8d0>
```



```
In [40]: plt.imshow(img[:,:,1])
         plt.colorbar()

Out[40]: <matplotlib.colorbar.Colorbar at 0x7f27283389b0>
```



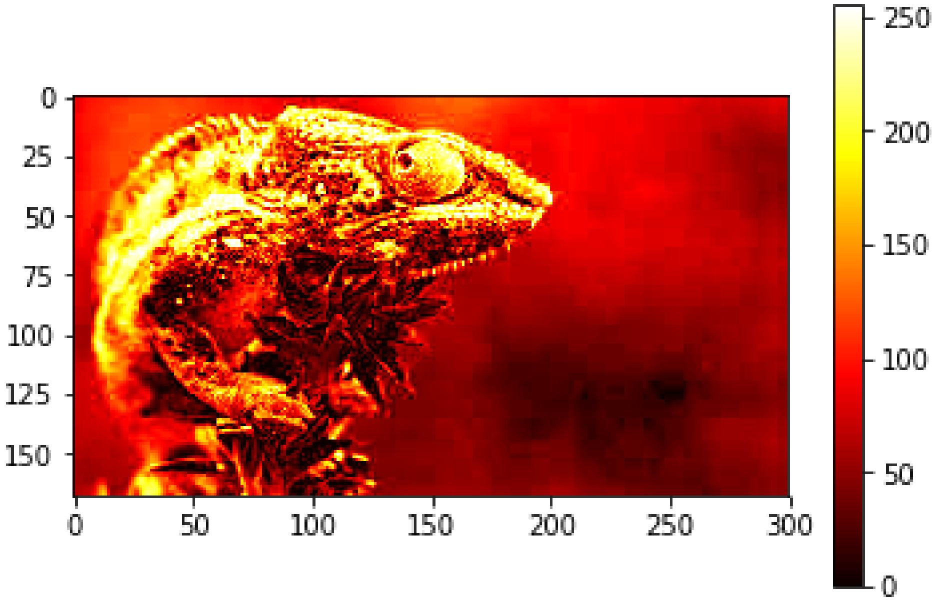
```
In [41]: plt.imshow(img[:,:,2])
         plt.colorbar()

Out[41]: <matplotlib.colorbar.Colorbar at 0x7f272831bcf8>
```



Now, with a luminosity (2D, no color) image, the default colormap (aka lookup table, LUT), is applied. The default is called viridis. There are plenty of others to choose from.

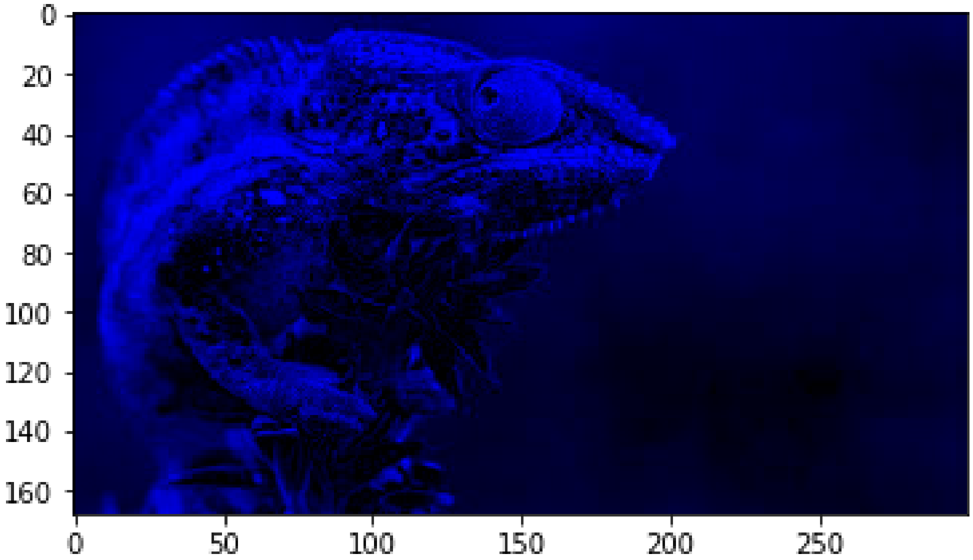
```
In [42]: plt.imshow(img[:,:,2], cmap="hot")
         plt.colorbar()
```

```
Out[42]: <matplotlib.colorbar.Colorbar at 0x7f2728f673c8>
```



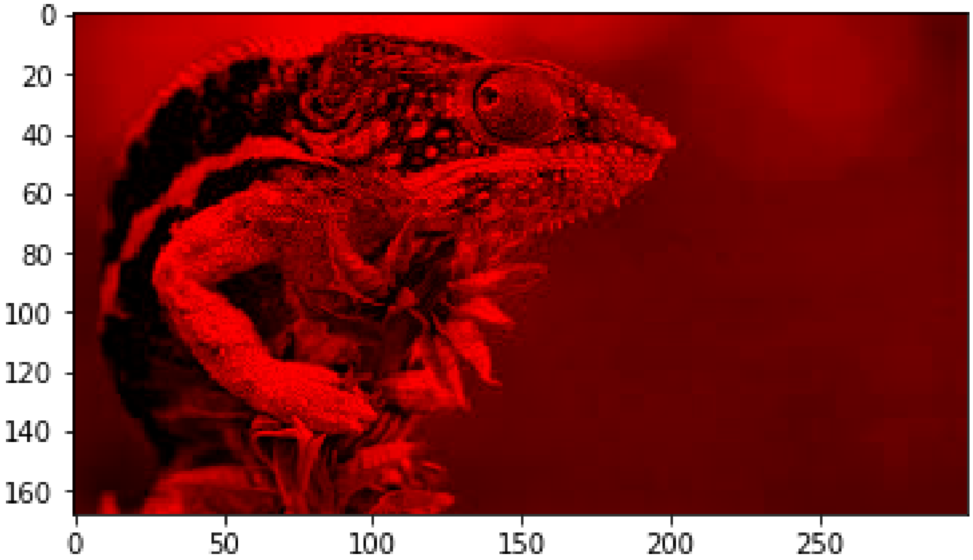
```
In [43]: img_blue = img.copy()
         img_blue[:,:,:2] = 0
         plt.imshow(img_blue)
```

```
Out[43]: <matplotlib.image.AxesImage at 0x7f2728e649b0>
```



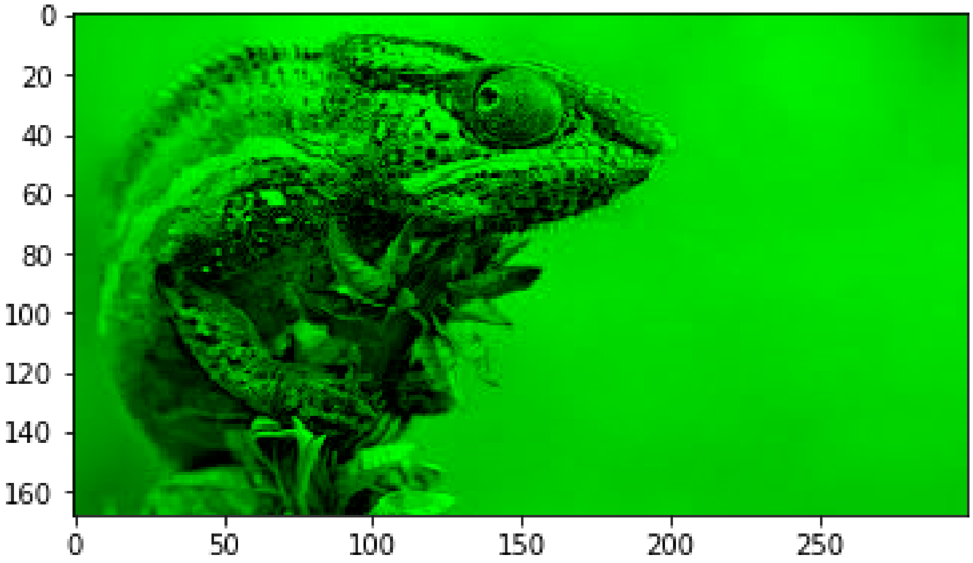
```
In [44]: img_red = img.copy()
         img_red[:,:,1:3] = 0
         plt.imshow(img_red)
```

```
Out[44]: <matplotlib.image.AxesImage at 0x7f2728f46e80>
```



```
In [45]: img_green = img.copy()
         img_green[:,:,::2] = 0
         plt.imshow(img_green)
```

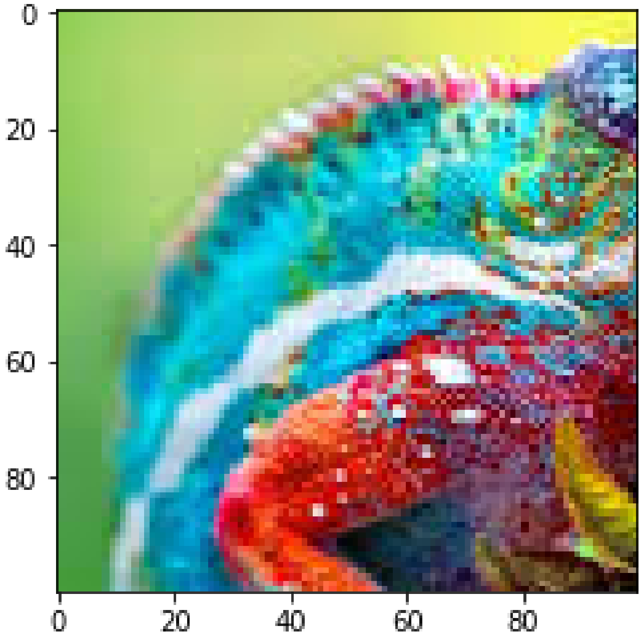
```
Out[45]: <matplotlib.image.AxesImage at 0x7f2728e70fd0>
```



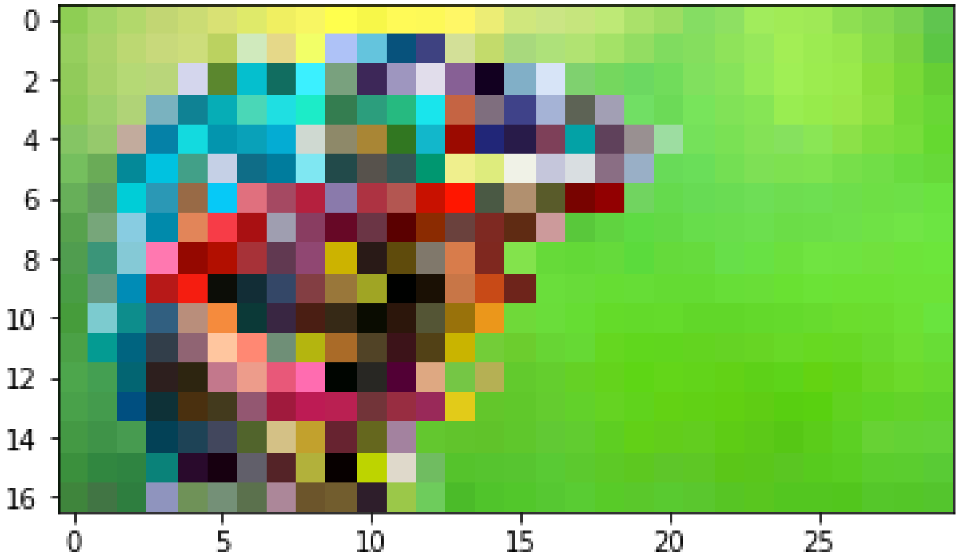
We can also slice the image into pieces using mumpy slicing.

```
In [46]: img_slice = img[:100,:100,:] # get the first 100*100 window from the window
         plt.imshow(img_slice)
```

```
Out[46]: <matplotlib.image.AxesImage at 0x7f2728d53438>
```



```
In [47]: img_pixelated = img[::10,::10,:] # select only every tenth pixel
         plt.imshow(img_pixelated)

Out[47]: <matplotlib.image.AxesImage at 0x7f2728e8f1d0>
```



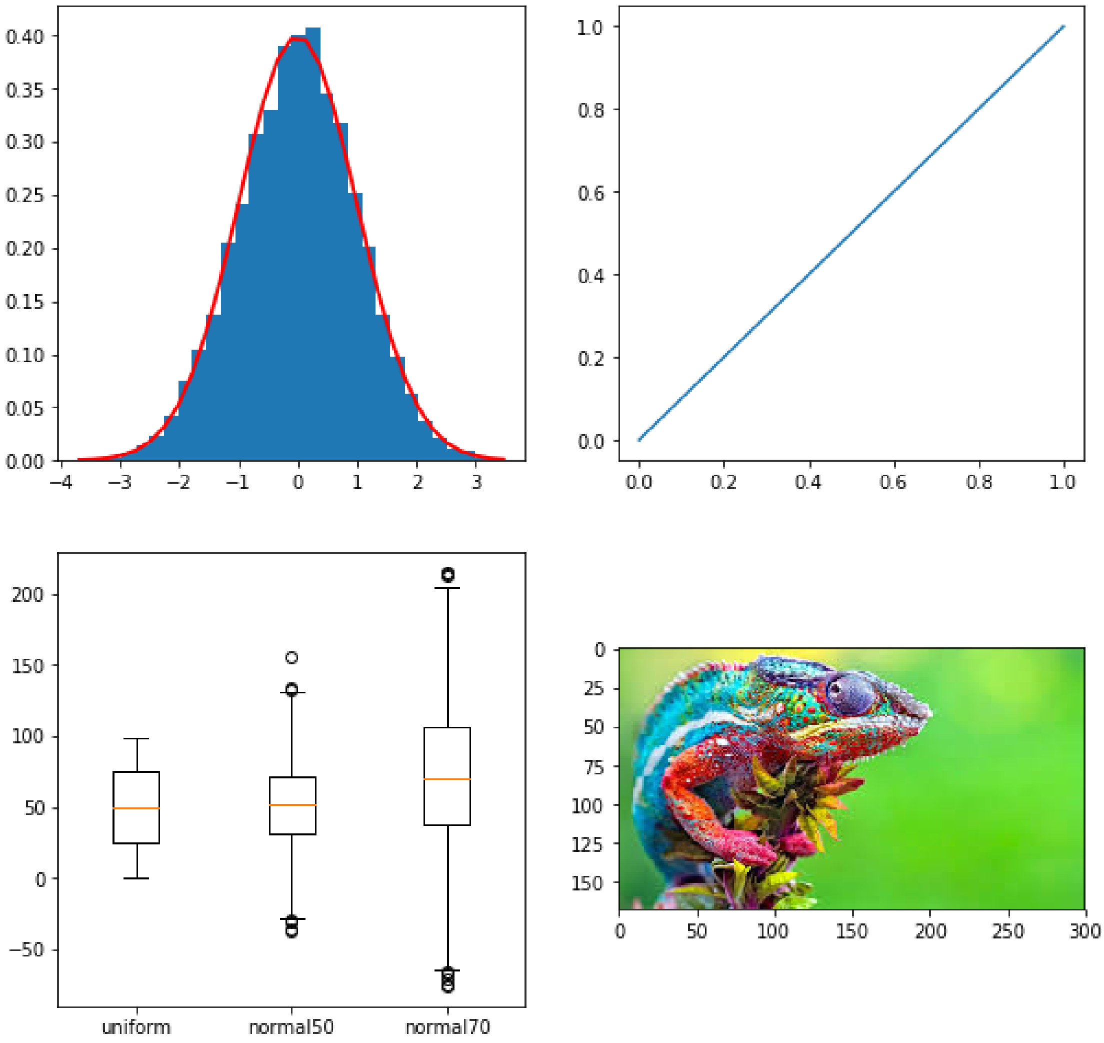
```
In [ ]:
```

### 3.4 Multiple subplots

We can also easily create plots containing multiple plots together. For that we can use the `subplot` function.

```
In [50]: arr1 = np.random.randint(0, 100, 1000)
         arr2 = np.random.normal(50, 30, 1000)
         arr3 = np.random.normal(70, 50, 1000)

         fig, axes = plt.subplots(2,2, figsize=(10,10))
         print(axes)
         axes[0][0].hist(gauss, bins=bins, density=True)

         axes[0][0].plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
                  np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
                  linewidth=2, color='r')
         axes[0][1].plot([0,1],[0,1])
         axes[1][0].boxplot([arr1,arr2,arr3], labels=['uniform','normal50', 'normal70'])
         axes[1][1].imshow(img)

         plt.show()
```

```
[[<matplotlib.axes._subplots.AxesSubplot object at 0x7f27281d8080>
  <matplotlib.axes._subplots.AxesSubplot object at 0x7f2728133160>]
 [<matplotlib.axes._subplots.AxesSubplot object at 0x7f27281506d8>
  <matplotlib.axes._subplots.AxesSubplot object at 0x7f27280e7c50>]]
```


0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4
−3
−2
−1
0
1
2
3
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
200
150
100
50
0
−50
uniform
normal50
normal70
0
25
50
75
100
125
150
0
50
100
150
200
250
300