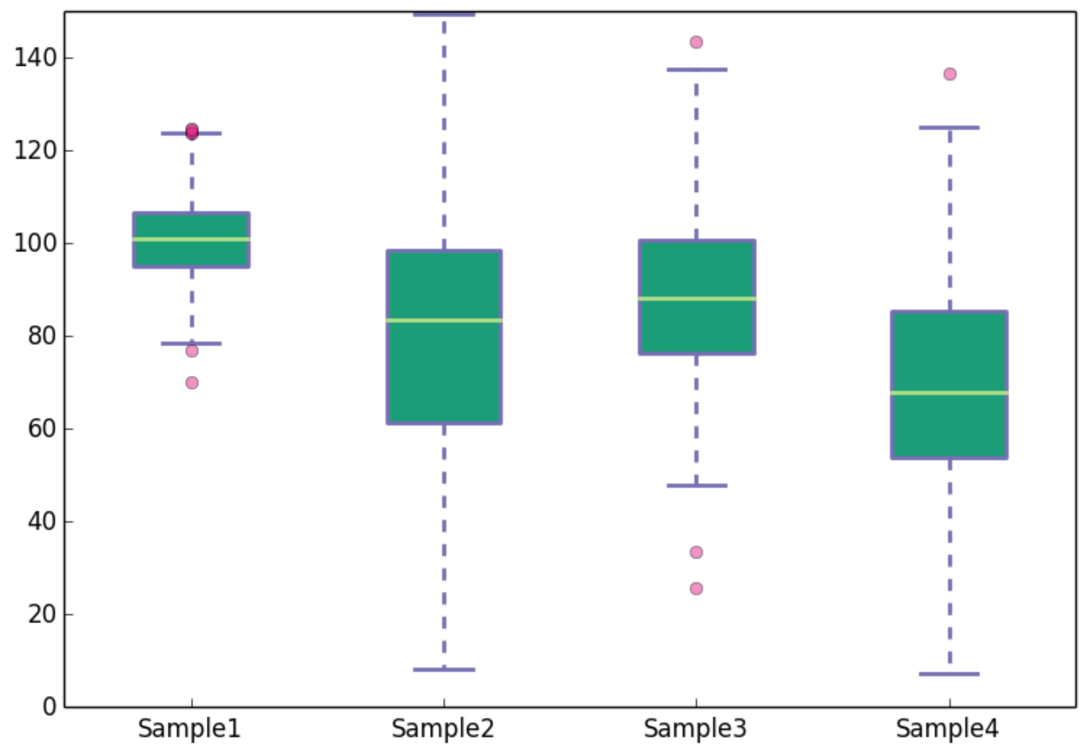
Within the plt.pie, we specify the “slices,” which are the relevant sizes for each part. Then, we specify the color list for the corresponding slices. Next, we can optionally specify the “Start angle” for the graph. This lets you start the line where you want. In our case, we chose a 90 degree angle for the pie chart, which means the first division will be a verticle line. Next, we can optionally add a shadow to the plot for a bit of character, and then we can even use “explode” to pull out a slice a bit.

We have four total slices, so, with explode, if we didn’t want to pull out any slices at all, we would do 0,0,0,0. If we wanted to pull out the first slice a bit, we would do 0.1,0,0,0.

Finally, we do autopct to optionally overlay the percentages on to the graph itself.

## Creating boxplots with Matplotlib

In this part, we will create some box-and-whisker plots (henceforth, referred to simply as boxplots) using Matplotlib. At the end of the sectionwe will have a boxplot which looks like the following:



**Import the libraries and specify the type of the output file.**

The first step is to import the python libraries that we will use.

```
## numpy is used for creating fake data
import numpy as np
import matplotlib as mpl

## agg backend is used to create plot as a .png file
mpl.use('agg')

import matplotlib.pyplot as plt
```

## Convert the data to an appropriate format

The second step is to ensure that your data is in an appropriate format. We need to provide a collection of values for each box in the boxplot. A collection can be expressed as a python list, tuple, or as a numpy array. Once you have the different collections, one for each box, you combine all these collections together in a list, tuple or a numpy array. That is, the data for the boxplot is in the form of a list of lists, or list of arrays, or a tuple of arrays etc.

Let us create the data for the boxplots. We use numpy.random.normal() to create the fake data. It takes three arguments, mean and standard deviation of the normal distribution, and the number of values desired.

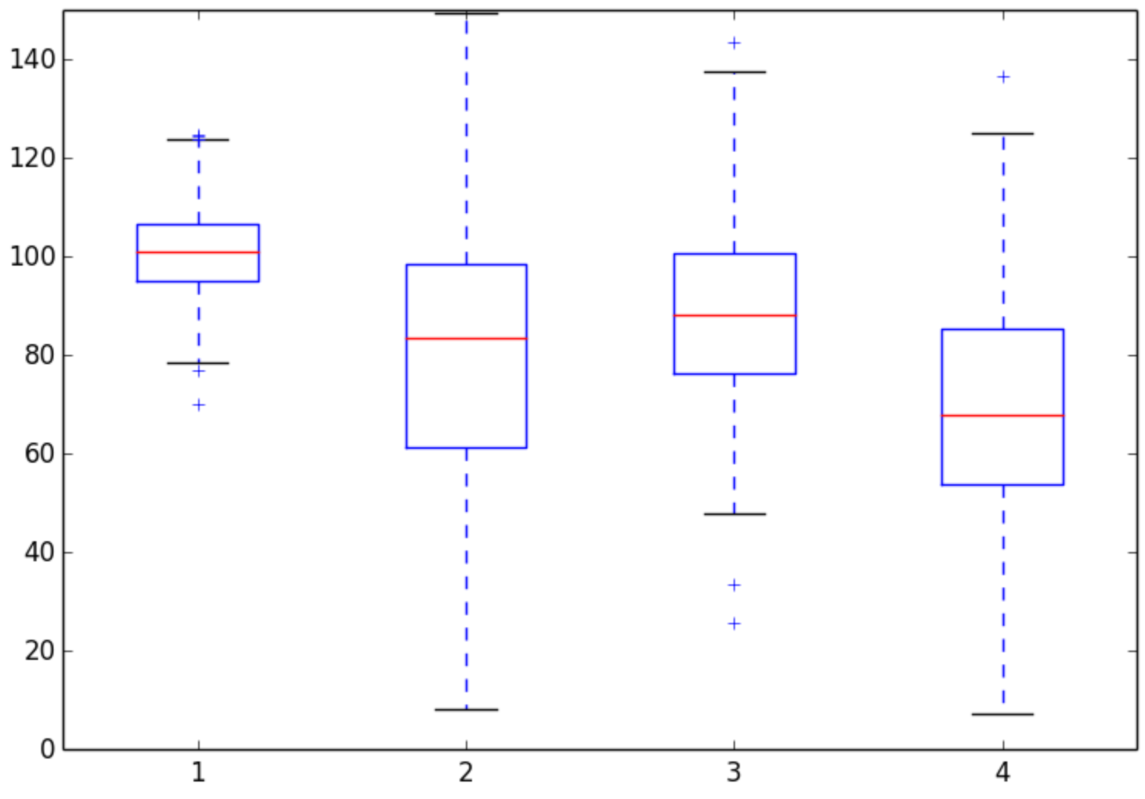
```
## Create data
np.random.seed(10)
collectn_1 = np.random.normal(100, 10, 200)
collectn_2 = np.random.normal(80, 30, 200)
collectn_3 = np.random.normal(90, 20, 200)
collectn_4 = np.random.normal(70, 25, 200)

## combine these different collections into a list
data_to_plot = [collectn_1, collectn_2, collectn_3, collectn_4]
```

## Create the boxplot

The list of arrays that we created above is the only required input for creating the boxplot. Using data_to_plot we can create the boxplot with the following code:

```
# Create a figure instance

fig = plt.figure(1, figsize=(9, 6))


# Create an axes instance

ax = fig.add_subplot(111)


# Create the boxplot

bp = ax.boxplot(data_to_plot)


# Show the figure

fig.show()
```

This gives:



That was easy. If you are satisfied with the default choices for the axes limits, labels and the look of the plot there is nothing more to do. Most likely that is not the case and you want to customize these features. Matplotlib provides endless customization possibilities.

**Change x-axis labels and remove tick marks from the top and right axes.**

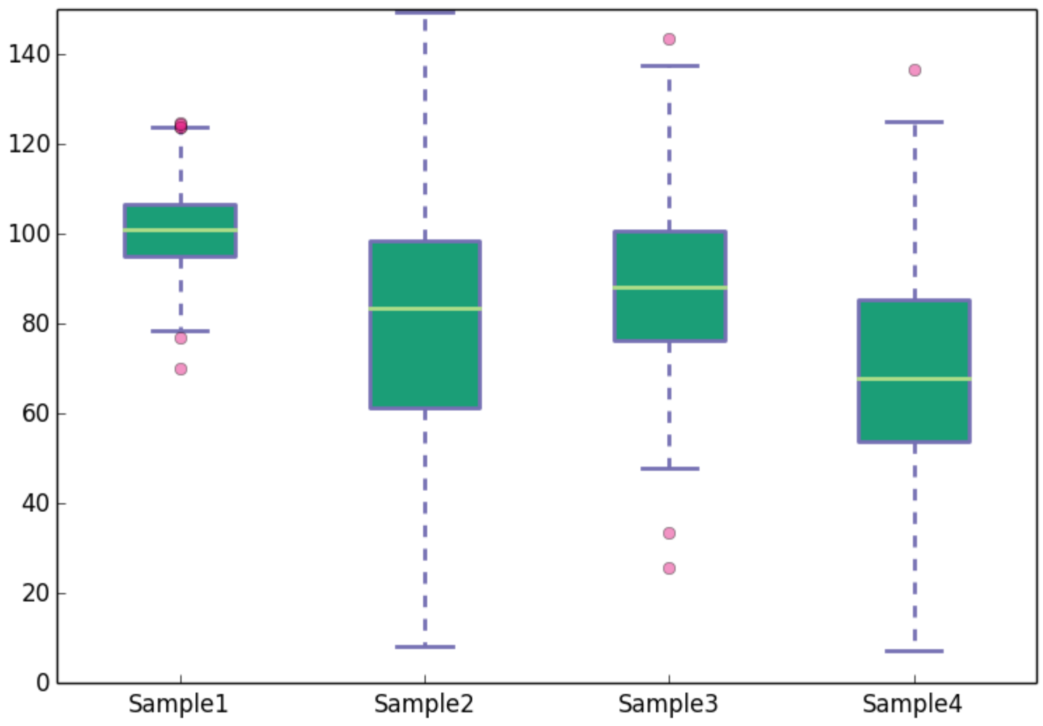
This is easy. To do this pass a list of custom labels to ax.set_xticklabels() function. The list should have the same length as the number of boxes in the boxplot. For example, suppose instead of the default x-axis labels that we see in the plots above, we want labels ‘Sample1’, ‘Sample2’, ‘Sample3’ and ‘Sample4’. We can modify the labels using the following line (add it before the line where the figure is saved):

```
## Custom x-axis labels

ax.set_xticklabels(['Sample1', 'Sample2', 'Sample3', 'Sample4'])
```

To remove the tick-marks from top and right spines, or rather to keep ticks only on the bottom and left axes, add the following lines to the code.

```
## Remove top axes and right axes ticks

ax.get_xaxis().tick_bottom()

ax.get_yaxis().tick_left()
```

With the custom x-axis labels and removal of top and right axes ticks, the boxplot now looks like the following:



If you are curious to learn more about creating boxplots with matplotlib, you may find the following links helpful.

1. Official matplotlib documentation on boxplots
2. Boxplot example on matplotlib website

## 3D Scatter Plot with Matplotlib

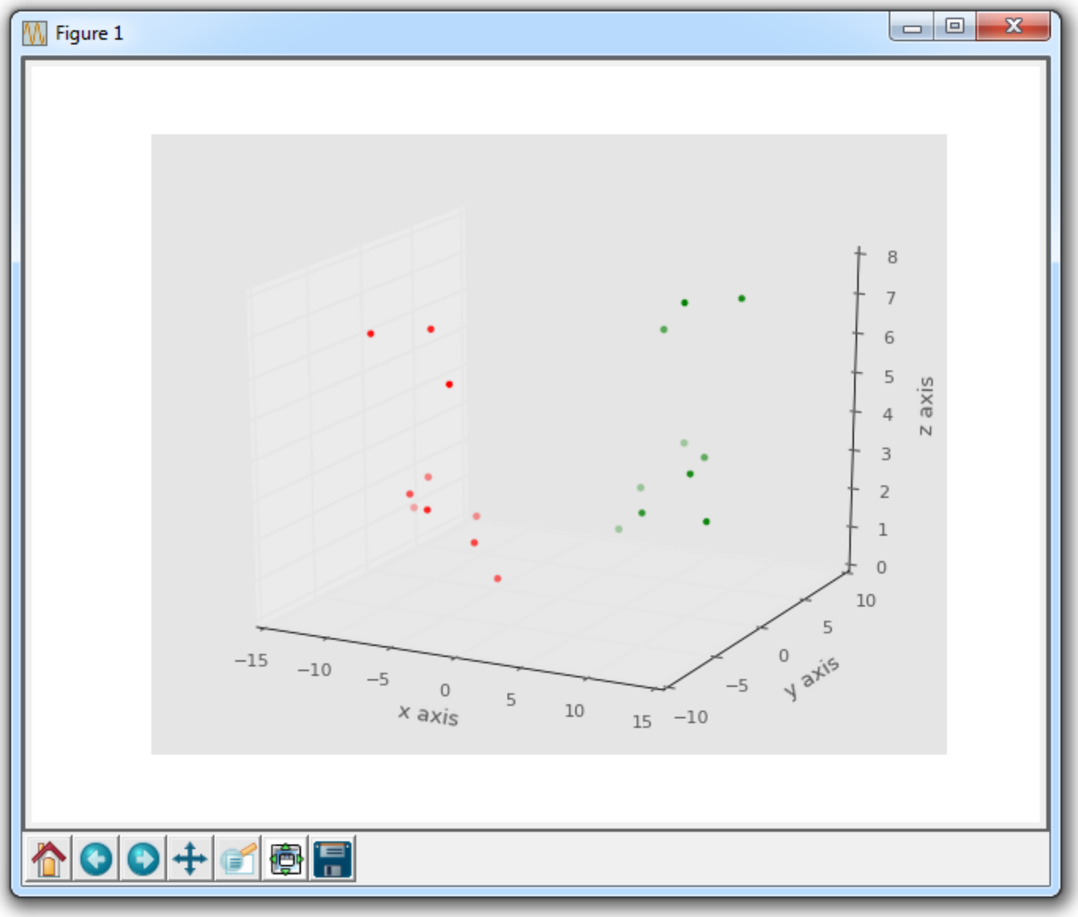
Graphing a 3D scatter plot is very similar to the typical scatter plot as well as the 3D wire_frame.

A quick example:

```
from mpl_toolkits.mplot3d import axes3d

import matplotlib.pyplot as plt

from matplotlib import style


style.use('ggplot')
```

```
fig = plt.figure()

ax1 = fig.add_subplot(111, projection='3d')


x = [1,2,3,4,5,6,7,8,9,10]

y = [5,6,7,8,2,5,6,3,7,2]

z = [1,2,6,3,2,7,3,3,7,2]


x2 = [-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]

y2 = [-5,-6,-7,-8,-2,-5,-6,-3,-7,-2]

z2 = [1,2,6,3,2,7,3,3,7,2]


ax1.scatter(x, y, z, c='g', marker='o')

ax1.scatter(x2, y2, z2, c ='r', marker='o')


ax1.set_xlabel('x axis')

ax1.set_ylabel('y axis')

ax1.set_zlabel('z axis')


plt.show()
```

The result here:



Note here that you can change the size and marker with these plots, just like you can with a typical scatter plot.

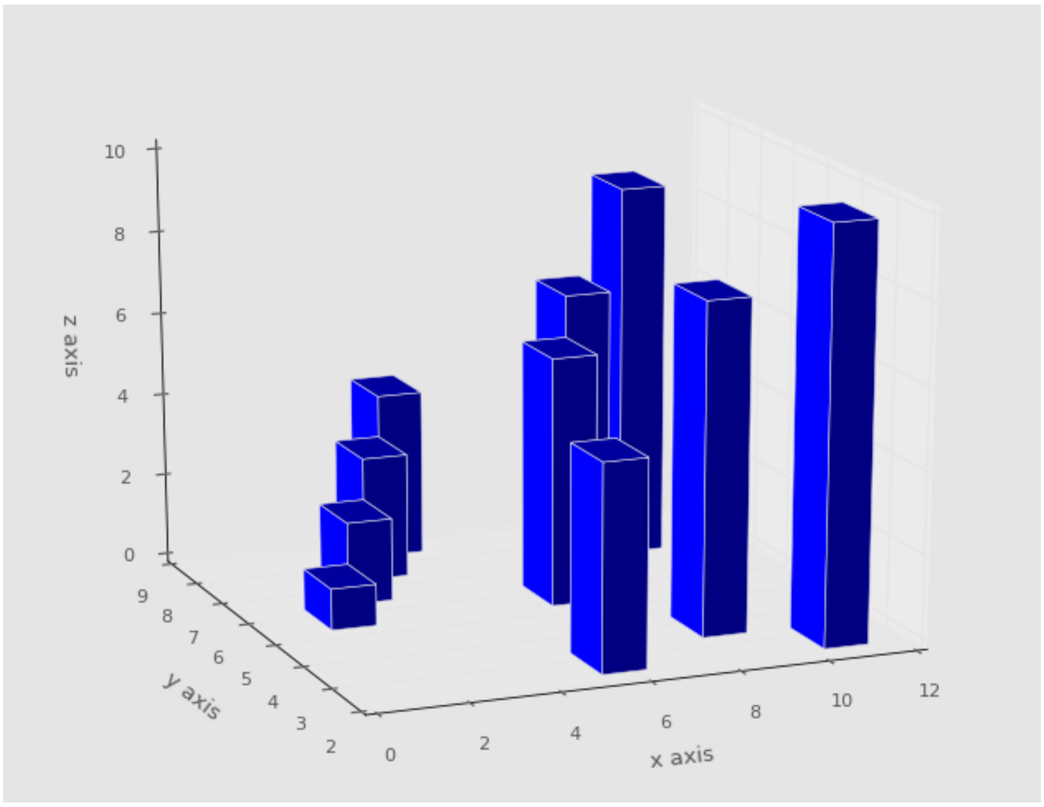
## 3D Bar Chart with Matplotlib

The 3D bar chart is quite unique, as it allows us to plot more than 3 dimensions. No, you cannot plot past the 3rd dimension, but you can plot more than 3 dimensions.

With bars, you have the starting point of the bar, the height of the bar, and the width of the bar. With a 3D bar, you also get another choice, which is depth of the bar. Most of the time, a bar chart starts with the bar flat on an axis, but you can add another dimension by releasing this constraint as well. We'll keep it rather simple, however:

```
from mpl_toolkits.mplot3d import axes3d

import matplotlib.pyplot as plt
import numpy as np

from matplotlib import style

style.use('ggplot')


fig = plt.figure()

ax1 = fig.add_subplot(111, projection='3d')


x3 = [1,2,3,4,5,6,7,8,9,10]

y3 = [5,6,7,8,2,5,6,3,7,2]

z3 = np.zeros(10)


dx = np.ones(10)

dy = np.ones(10)

dz = [1,2,3,4,5,6,7,8,9,10]


ax1.bar3d(x3, y3, z3, dx, dy, dz)


ax1.set_xlabel('x axis')

ax1.set_ylabel('y axis')

ax1.set_zlabel('z axis')


plt.show()
```

Note here that we have to define x,y,z… and then 3 more dimensions for depth. This gives us:

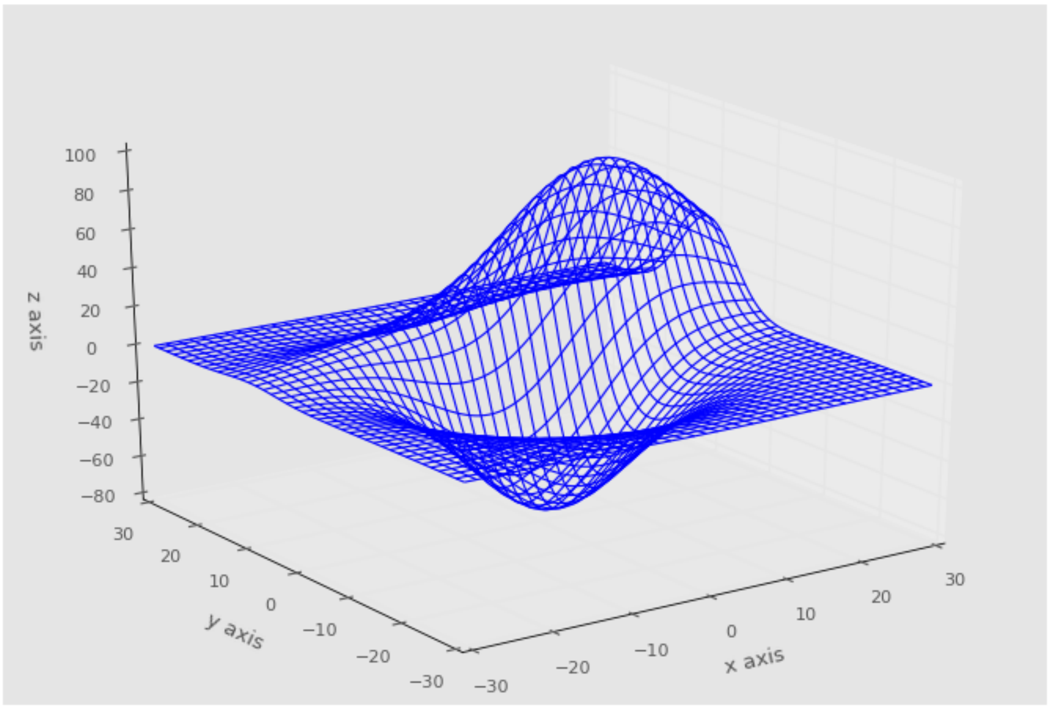


## Conclusion

We will wrap up the activity here, and show a slightly more complex 3D wireframe:

```
from mpl_toolkits.mplot3d import axes3d

import matplotlib.pyplot as plt

import numpy as np

from matplotlib import style

style.use('ggplot')


fig = plt.figure()

ax1 = fig.add_subplot(111, projection='3d')

x, y, z = axes3d.get_test_data()


print(axes3d.__file__)
```

```
ax1.plot_wireframe(x,y,z, rstride = 3, cstride = 3)


ax1.set_xlabel('x axis')

ax1.set_ylabel('y axis')

ax1.set_zlabel('z axis')


plt.show()
```



If you have followed along since the beginning, then you have learned most of what Matplotlib has to offer. You might not believe it, but there’s still a lot of other things Matplotlib can do! Moving forward, you can always head to Matplotlib.org, and check out the examples and the gallery section.