Lecture 9

CSS 507 Data Collection, Wrangling, Analysis and Visualization

By: PhD, Andrey Bogdanchikov
Edited by: MSc, Elnura Nabigazinova

Content

Introduction

Visualization of data is one of the important parts of Data Science. You should be able to communicate your findings to all around you in most simple and understandable way.

The most famous library in python is matplotlib, so we will start with this library and then mention some others.

import matplotlib.pyplot as plt

Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line.

Importing library

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Setting Styles

We will use the plt.style directive to choose appropriate aesthetic styles for our figures. Here we will set the classic style, which ensures that the plots we create use the classic Matplotlib style:

```
plt.style.use('classic')
```

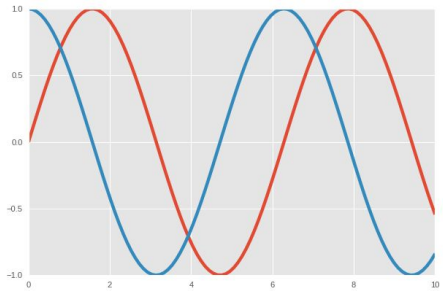
How to display your plots

A visualization you can't see won't be of much use, but just how you view your Matplotlib plots depends on the context.

The best use of Matplotlib differs depending on how you are using it; roughly, the three applicable contexts are using Matplotlib:

- in a script,
- in an IPython terminal,
- or in an IPython notebook.

Plotting from a script




```
# ------- file: myplot.py ------
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

If you are using Matplotlib from within a script, the function plt.show() is your friend.

plt.show() starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display your figure or figures.

One thing to be aware of: the plt.show() command should be used only once per Python session, and is most often seen at the very end of the script.

Plotting from an IPython shell

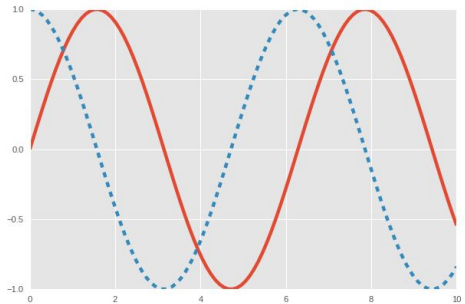
IPython is built to work well with Matplotlib if you specify Matplotlib mode. To enable this mode, you can use the %matplotlib magic command after starting ipython:

- In [1]: %matplotlib
  Using matplotlib backend: TkAgg

  In [2]: import matplotlib.pyplot as plt

At this point, any plt plot command will cause a figure window to open, and further commands can be run to update the plot.

Plotting from an IPython notebook



Plotting interactively within an IPython notebook can be done with the %matplotlib command, and works in a similar way to the IPython shell. In the IPython notebook, you also have the option of embedding graphics directly in the notebook, with two possible options:

- **%matplotlib notebook** will lead to interactive plots embedded within the notebook
- **%matplotlib inline** will lead to static images of your plot embedded in the notebook

Saving Figures to File

One nice feature of Matplotlib is the ability to save figures in a wide variety of formats. Saving a figure can be done using the savefig() command. For example, to save the previous figure as a PNG file, you can run this:

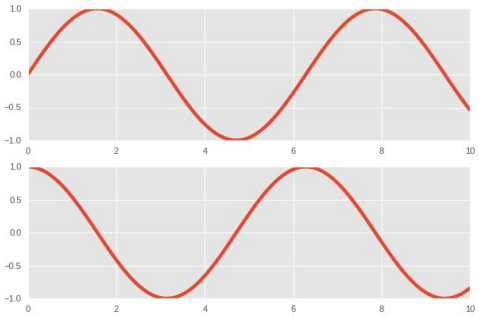
```
fig.savefig('my_figure.png')
```

Two Interfaces for the Price of One

A potentially confusing feature of Matplotlib is its dual interfaces: a convenient MATLAB-style state-based interface, and a more powerful object-oriented interface. We'll quickly highlight the differences between the two here.

- MATLAB-style Interface
- Object-oriented interface

MATLAB-style Interface

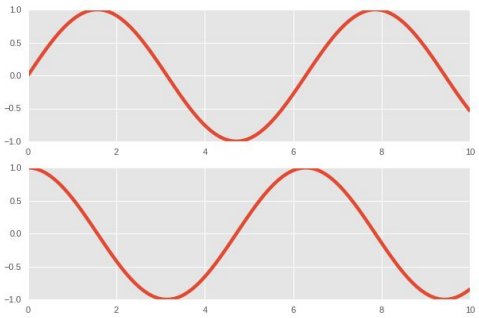


- Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (plt) interface. For example, the following code will probably look quite familiar to MATLAB users:

- plt.figure() # create a plot figure
  # create the first of two panels and set current axis
  plt.subplot(2, 1, 1) # (rows, columns, panel number)
  plt.plot(x, np.sin(x))

  # create the second panel and set current axis
  plt.subplot(2, 1, 2)
  plt.plot(x, np.cos(x));

Object-oriented interface



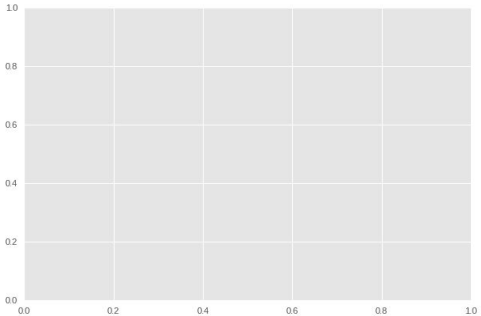
The object-oriented interface is available for these more complicated situations, and for when you want more control over your figure. Rather than depending on some notion of an "active" figure or axes, in the object-oriented interface the plotting functions are methods of explicit Figure and Axes objects. To re-create the previous plot using this style of plotting, you might do the following:

```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)

# Call plot() method on the appropriate object
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x));
```

Plots

Simple plot



Perhaps the simplest of all plots is the visualization of a single function y=f(x). Here we will take a first look at creating a simple plot of this type.
For all Matplotlib plots, we start by creating a figure and an axes. In their simplest form, a figure and axes can be created as follows:

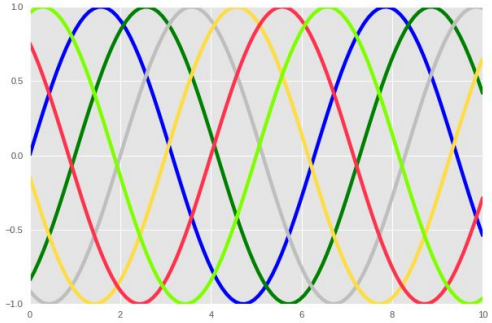
```
fig = plt.figure()
ax = plt.axes()
```

A plot

n Matplotlib, the figure (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels. The axes (an instance of the class plt.Axes) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization.

```
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```

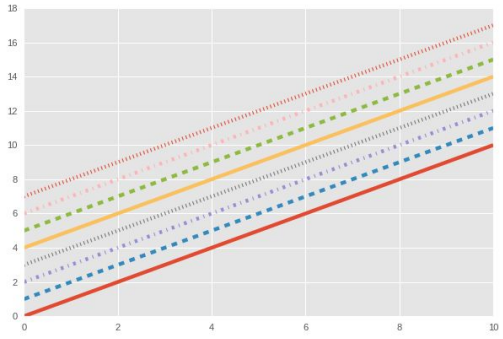
Adjusting the Plot: Line Colors and Styles



The first adjustment you might wish to make to a plot is to control the line colors and styles. The plt.plot() function takes additional arguments that can be used to specify these.

```
plt.plot(x, np.sin(x - 0), color='blue')      # specify color by name
plt.plot(x, np.sin(x - 1), color='g')         # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')      # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')   # Hex code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 to 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names supported
```

Line styles



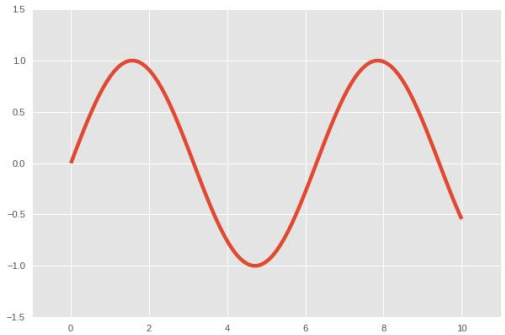
If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines.

Similarly, the line style can be adjusted using the linestyle keyword:

```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');

# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':');  # dotted
```

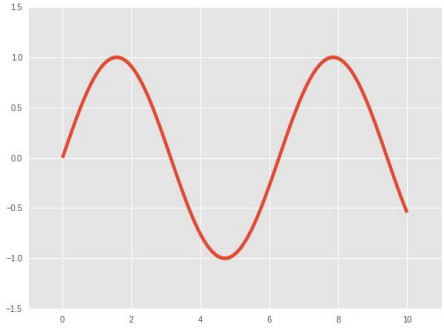
Adjusting the Plot: Axes Limits

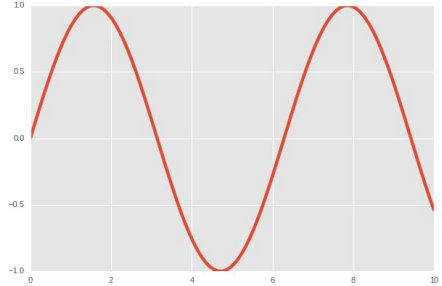


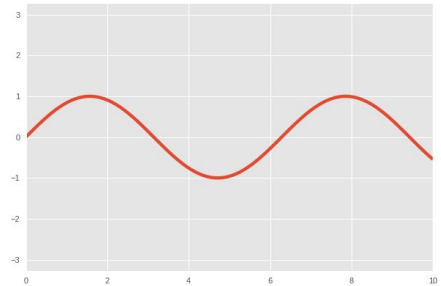
Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes it's nice to have finer control. The most basic way to adjust axis limits is to use the plt.xlim() and plt.ylim() methods:

```
plt.plot(x, np.sin(x))

plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```







Axis not Axes

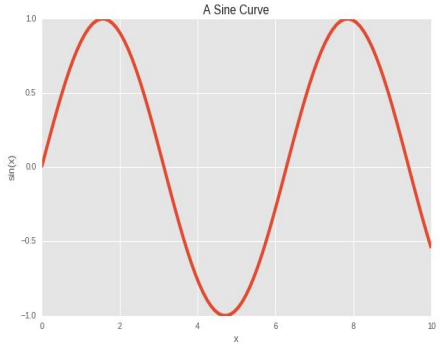
A useful related method is plt.axis() (note here the potential confusion between axes with an e, and axis with an i).

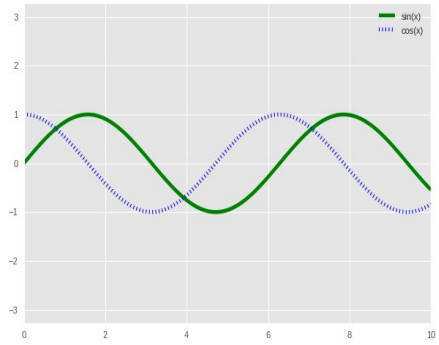
The plt.axis() method allows you to set the x and y limits with a single call, by passing a list which specifies [xmin, xmax, ymin, ymax]:

```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```

```
plt.plot(x, np.sin(x))
plt.axis('tight');
```

```
plt.plot(x, np.sin(x))
plt.axis('equal');
```





Labeling Plots

As the last piece of this section, we'll briefly look at the labeling of plots: titles, axis labels, and simple legends.

Titles and axis labels are the simplest such labels—there are methods that can be used to quickly set them:

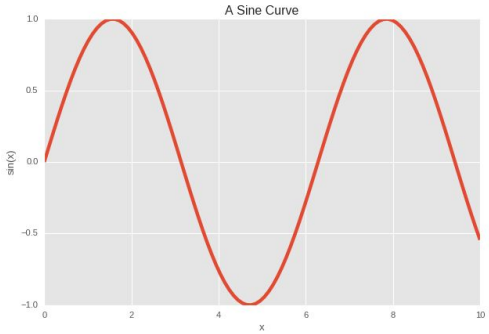
When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type. Again, Matplotlib has a built-in way of quickly creating such a legend.

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
#---------------------------------

plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.axis('equal')

plt.legend();
```

Aside: Matplotlib Gotchas



While most plt functions translate directly to ax methods (such as plt.plot() → ax.plot(), plt.legend() → ax.legend(), etc.), this is not the case for all commands. In particular, functions to set limits, labels, and titles are slightly modified. For transitioning between MATLAB-style functions and object-oriented methods, make the following changes:

- plt.xlabel() → ax.set_xlabel()
- plt.ylabel() → ax.set_ylabel()
- plt.xlim() → ax.set_xlim()
- plt.ylim() → ax.set_ylim()
- plt.title() → ax.set_title()

it is often more convenient to use the ax.set() method to set all these properties at once:

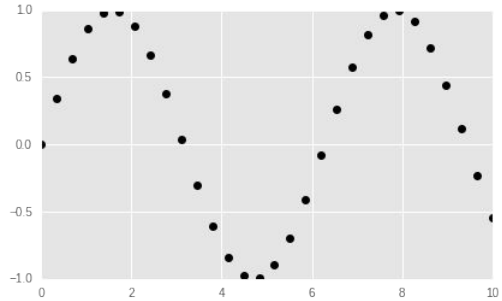
```
ax = plt.axes()
ax.plot(x, np.sin(x))
ax.set(xlim=(0, 10), ylim=(-2, 2),
       xlabel='x', ylabel='sin(x)',
       title='A Simple Plot');
```

Simple Scatter Plots

Scatter plot

Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape.

Scatter Plots with plt.plot

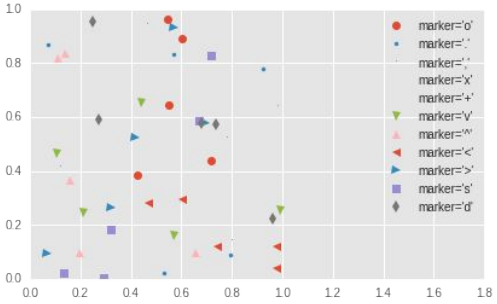


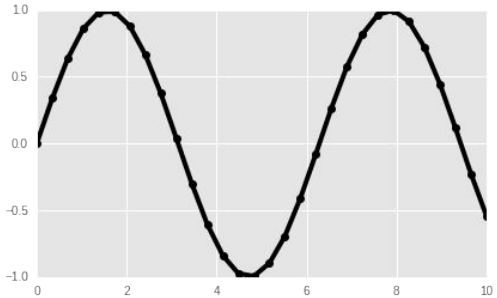
In the previous section we looked at plt.plot/ax.plot to produce line plots. It turns out that this same function can produce scatter plots as well:

```
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, 'o', color='black');
```

Common scatters





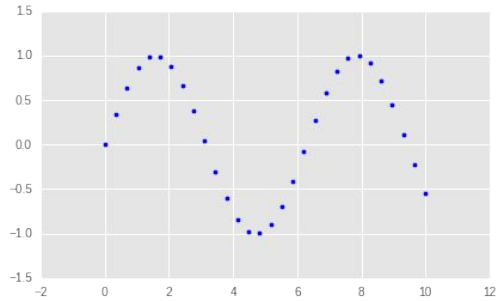
The third argument in the function call is a character that represents the type of symbol used for the plotting. Just as you can specify options such as '-', '--' to control the line style, the marker style has its own set of short string codes. The full list of available symbols can be seen in the documentation of plt.plot, or in Matplotlib's online documentation. Most of the possibilities are fairly intuitive, and we'll show a number of the more common ones here:

```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's',
'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,
             label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

For even more possibilities, these character codes can be used together with line and color codes to plot points along with a line connecting them:

```
plt.plot(x, y, '-ok');
```

Scatter Plots with plt.scatter



A second, more powerful method of creating scatter plots is the plt.scatter function, which can be used very similarly to the plt.plot function

```
plt.scatter(x, y, marker='o');
```

The primary difference of plt.scatter from plt.plot is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

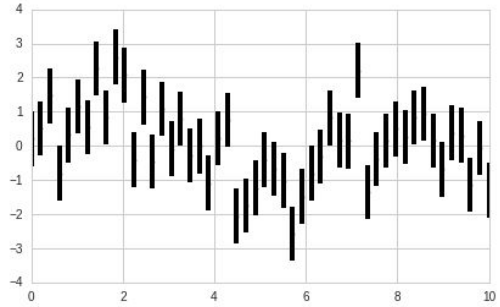
Color scatter

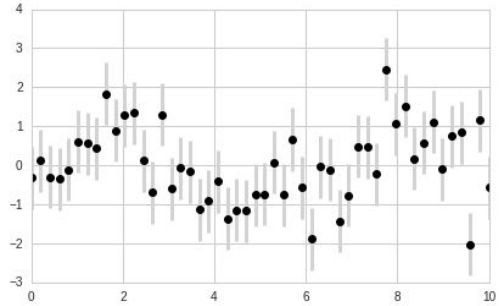

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes,
alpha=0.3,
        cmap='viridis')
plt.colorbar();  # show color scale
```

Let's show this by creating a random scatter plot with points of many colors and sizes. In order to better see the overlapping results, we'll also use the alpha keyword to adjust the transparency level:

Basic Errorbars





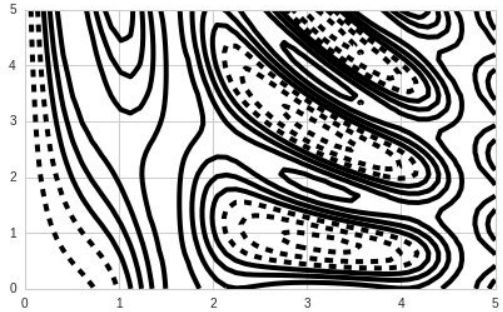
A basic errorbar can be created with a single Matplotlib function call:

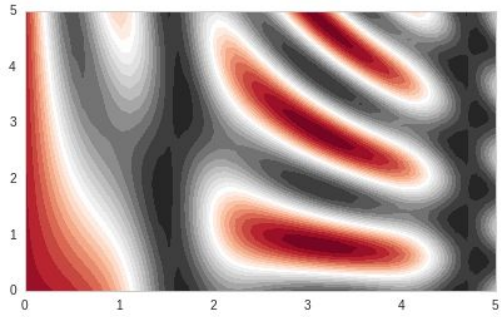
```
x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)

plt.errorbar(x, y, yerr=dy, fmt='.k');

#-----------------------------------------
plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
        ecolor='lightgray', elinewidth=3, capsize=0);
```

Visualizing a Three-Dimensional Function



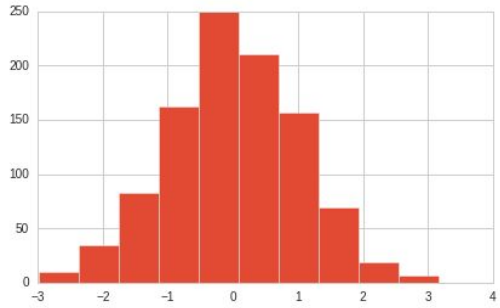


We'll start by demonstrating a contour plot using a function *z=f*(*x*,*y*), using the following particular choice for *f.* A contour plot can be created with the plt.contour function. It takes three arguments: a grid of *x* values, a grid of *y* values, and a grid of *z* values.

```
def f(x, y):
        return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
#plt.contour(X, Y, Z, colors='black');
plt.contourf(X, Y, Z, 20, cmap="RdGy");
```

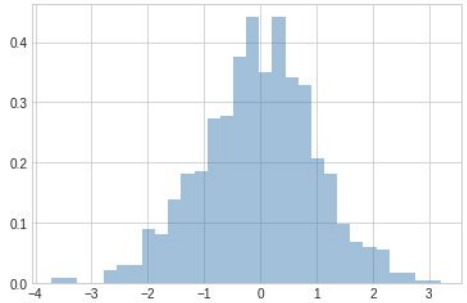
Histograms

Histograms, Binnings, and Density



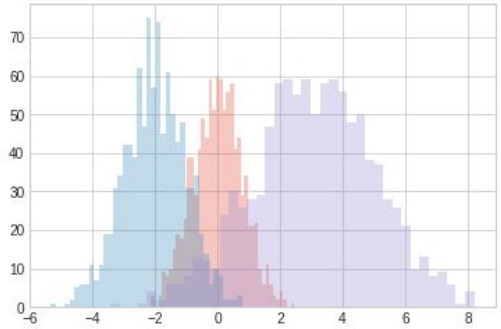
A simple histogram can be a great first step in understanding a dataset. Earlier, we saw a preview of Matplotlib's histogram function, which creates a basic histogram in one line, once the normal boiler-plate imports are done:

```
data = np.random.randn(1000)
plt.hist(data);
```




```
plt.hist(data, bins=30, normed=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='none');
```

Multiple histograms



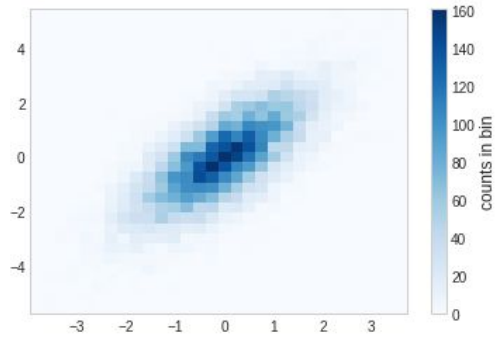
The plt.hist docstring has more information on other customization options available. This combination of histtype='stepfilled' along with some transparency alpha can be very useful when comparing histograms of several distributions:

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

kwargs = dict(histtype='stepfilled',
alpha=0.3, normed=True, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```

Two-Dimensional Histograms and Binnings



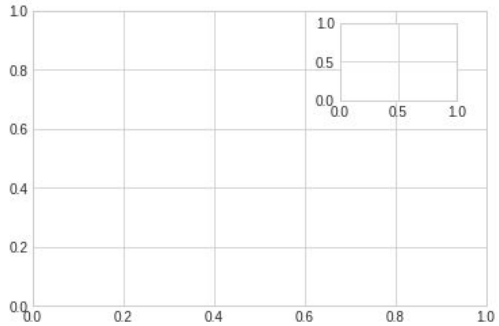
One straightforward way to plot a two-dimensional histogram is to use Matplotlib's plt.hist2d function:

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```

Multiple Subplots

Sometimes it is helpful to compare different views of data side by side. To this end, Matplotlib has the concept of subplots: groups of smaller axes that can exist together within a single figure. These subplots might be insets, grids of plots, or other more complicated layouts. In this section we'll explore four routines for creating subplots in Matplotlib.

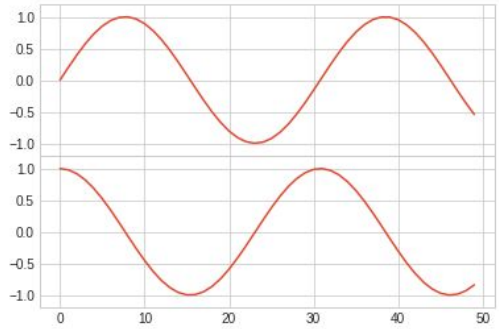
plt.axes: Subplots by Hand



The most basic method of creating an axes is to use the plt.axes function. As we've seen previously, by default this creates a standard axes object that fills the entire figure. plt.axes also takes an optional argument that is a list of four numbers in the figure coordinate system. These numbers represent [left, bottom, width, height] in the figure coordinate system, which ranges from 0 at the bottom left of the figure to 1 at the top right of the figure.

```
ax1 = plt.axes()  # standard axes
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```

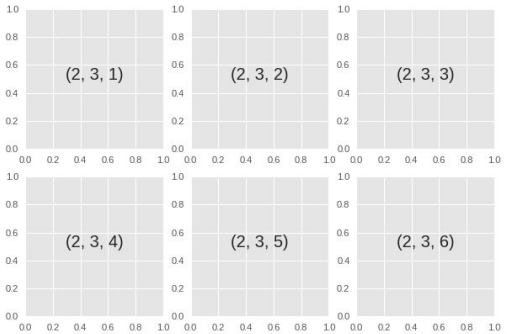
OO equivalent



The equivalent of this command within the object-oriented interface is fig.add_axes(). Let's use this to create two vertically stacked axes:

```
fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],
                   xticklabels=[], ylim=(-1.2, 1.2))
ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4],
                   ylim=(-1.2, 1.2))
x = np.linspace(0, 10)
ax1.plot(np.sin(x))
ax2.plot(np.cos(x));
```

plt.subplot: Simple Grids of Subplots

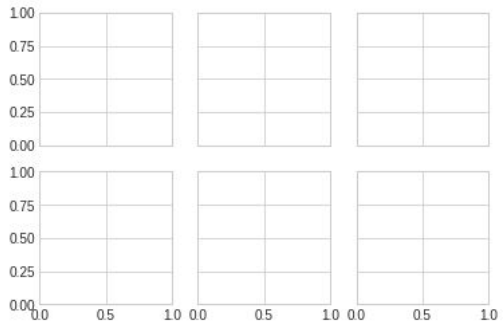


Aligned columns or rows of subplots are a common-enough need that Matplotlib has several convenience routines that make them easy to create. The lowest level of these is plt.subplot(), which creates a single subplot within a grid.

As you can see, this command takes three integer arguments—the number of rows, the number of columns, and the index of the plot to be created in this scheme, which runs from the upper left to the bottom right:

```
for i in range(1, 7):
   plt.subplot(2, 3, i)
   plt.text(0.5, 0.5, str((2, 3, i)),
         fontsize=18, ha='center')
```

plt.subplots: The Whole Grid in One Go



The approach just described can become quite tedious when creating a large grid of subplots, especially if you'd like to hide the x- and y-axis labels on the inner plots. For this purpose, plt.subplots() is the easier tool to use (note the s at the end of subplots). Rather than creating a single subplot, this function creates a full grid of subplots in a single line, returning them in a NumPy array.

fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')

Conclusion

Check following links for info:

- Matplotlib docs - https://matplotlib.org/users/index.html
- Matplotlib chapter - https://jakevdp.github.io/PythonDataScienceHandbook/04.00-introduction-to-matplotlib.html

Assignment 7

Use sdu_registration.xls file for this assignment.

Show following information in ONE good looking plot.

1. Find number of students in each course and group courses into 5 categories by their number of students. Plot a histogram with number of courses in each category.
2. Display bar plot of all instructors and their student counts sorted by counts.
3. Show a scatter plot of CIPHER and TEACHER with size of circle by student count.
4. Display horizontal bar plot with number of instructors who give at least one lesson on each educational level.

Thank You