A

APEC

UNIT V

DATA VISUALIZATION

Importing Matplotlib – Line plots – Scatter plots – visualizing errors – density and contour plots – Histograms – legends – colors – subplots – text and annotation – customization – three dimensional plotting - Geographic Data with Basemap - Visualization with Seaborn.

Simple Line Plots

The simplest of all plots is the visualization of a single function y = f x . Here we will take a first look at creating asimple plot of this type.

The *figure* (an instance of the class plt.Figure) can be thought of as a single container that contains all the objectsrepresenting 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 willeventually contain the plot elements that make up our visualization.

Line Colors and Styles

- The first adjustment you might wish to make to a plot is to control the line colors and styles.
- To adjust the color, you can use the color keyword, which accepts a string argument representing virtuallyany imaginable color. The color can be specified in a variety of ways
- If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines

Different forms of color representation.

specify color by name - color='blue'
short color code (rgbcmyk) - color='g'
Grayscale between 0 and 1 - color='0.75'
Hex code (RRGGBB from 00 to FF) - color='#FFDD44'
RGB tuple, values 0 and 1 -
color=(1.0,0.2,0.3)all HTML color names supported -
color='chartreuse'

- We can adjust the line style using the linestyle keyword.

Different line styles

```
linestyle='solid'
linestyle='dashed'
linestyle='dashdo
t'
linestyle='dotted'
```

Short assignment

```
linestyle='-' # solid
linestyle='--' #
dashed linestyle='-.'
# dashdotlinestyle=':'
# dotted
```

- linestyle and color codes can be combined into a single nonkeyword argument to the plt.plot() function

```
plt.plot(x, x + 0, '-g') # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot
blackplt.plot(x, x + 3, ':r'); # dotted
red
```

A

Axes Limits

- The most basic way to adjust axis limits is to use the plt.xlim() and plt.ylim() methodsExample

```
plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```

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

```
plt.axis([-1, 11, -1.5, 1.5]);
```

- Aspect ratio equal is used to represent one unit in x is equal to one unit in y. plt.axis('equal')

Labeling Plots

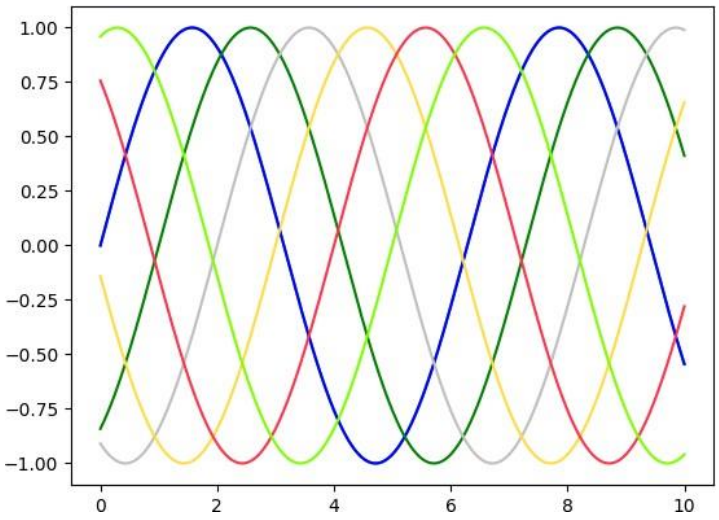
The labeling of plots includes titles, axis labels, and simple legends.Title - plt.title()
Label - plt.xlabel()
plt.ylabel()
Legend - plt.legend()

Example programsLine color

```
import matplotlib.pyplot as
pltimport numpy as np
fig =
plt.figure()ax =
plt.axes()
x = np.linspace(0, 10,
1000)ax.plot(x, np.sin(x));
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 and 1
plt.plot(x, np.sin(x - 5), color='chartreuse');# all HTML color names
supported
```

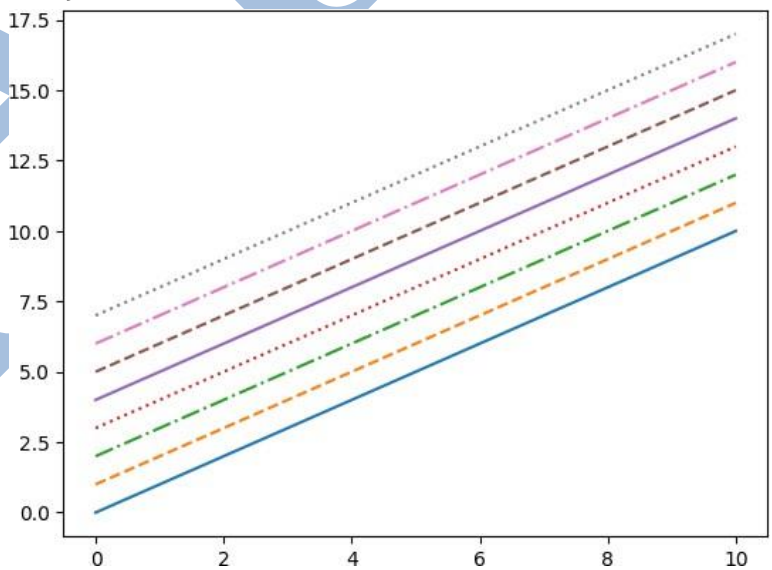
APEC

A



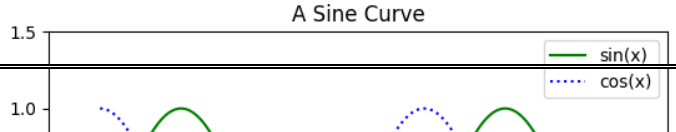
Line style

```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
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
```



Axis limit with label and legend

```
import matplotlib.pyplot as
```



```
pltimport numpy as np
fig =
plt.figure()ax =
plt.axes()
x = np.linspace(0, 10, 1000)
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b',
label='cos(x)')plt.title("A Sine
Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
plt.legend();
```

Simple Scatter Plots

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

Syntax

```
plt.plot(x, y, 'type of symbol ', color);
```

Example

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

- 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 '-' and '--' to control the line style, the marker style has its own set of short string codes.

Example

- Various symbols used to specify ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']

- Short hand assignment of line, symbol and color also allowed.

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

- Additional arguments in plt.plot()
  We can specify some other parameters related with scatter plot which makes it more attractive. They arecolor, marker size, linewidth, marker face color, marker edge color, marker edge width, etc

Example

```
plt.plot(x, y, '-p',
color='gray',
markersize=15,
linewidth=4,
markerfacecolor='white',
markeredgecolor='gray',
markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

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.
- Notice that the color argument is automatically mapped to a color scale (shown here by the colorbar() command), and the size argument is given in pixels.
- Cmap – color map used in scatter plot gives different color combinations.

```
Perceptually Uniform Sequential
['viridis', 'plasma', 'inferno', 'magma']
Sequential
['Greys', 'Purples', 'Blues', 'Greens', 'Oranges', 'Reds', 'YlOrBr', 'YlOrRd',
'OrRd', 'PuRd', 'RdPu', 'BuPu', 'GnBu', 'PuBu', 'YlGnBu', 'PuBuGn', 'BuGn',
'YlGn']
Sequential (2)
['binary', 'gist_yarg', 'gist_gray', 'gray', 'bone', 'pink', 'spring', 'summer',
'autumn', 'winter', 'cool', 'Wistia', 'hot', 'afmhot', 'gist_heat', 'copper']
```

APEC

A

APEC

Diverging

```
['PiYG', 'PRGn', 'BrBG', 'PuOr', 'RdGy', 'RdBu', 'RdYlBu', 'RdYlGn', 'Spectral',
'coolwarm', 'bwr', 'seismic']
```

Qualitative

```
['Pastel1', 'Pastel2', 'Paired', 'Accent', 'Dark2', 'Set1', 'Set2', 'Set3',
'tab10', 'tab20', 'tab20b', 'tab20c']
```

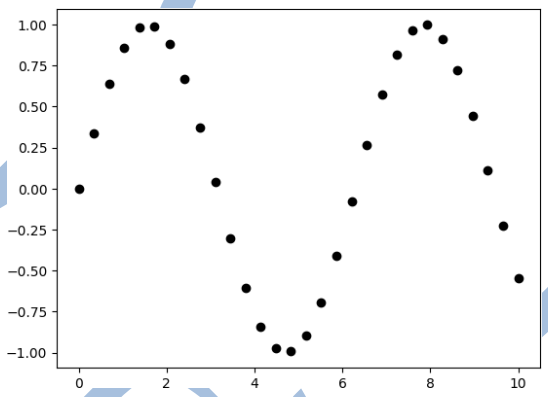
Miscellaneous

```
['flag', 'prism', 'ocean', 'gist_earth', 'terrain', 'gist_stern', 'gnuplot',
'gnuplot2', 'CMRmap', 'cubehelix', 'brg', 'hsv', 'gist_rainbow', 'rainbow',
'jet', 'nipy_spectral', 'gist_ncar']
```

Example programs.

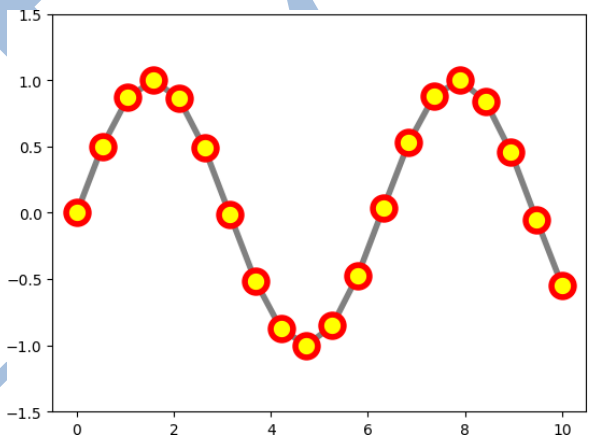
Simple scatter plot.

```
import numpy as np
import matplotlib.pyplot as
pltx = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o', color='black');
```



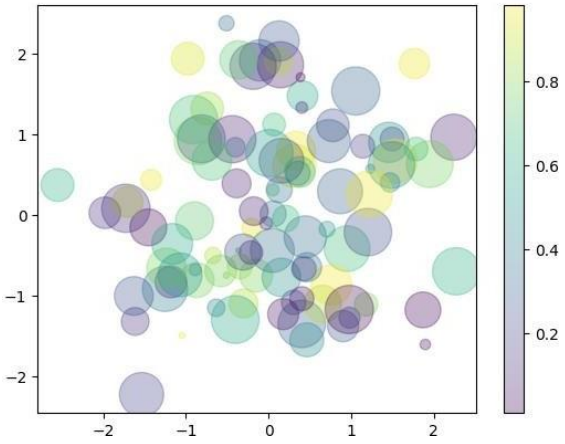
Scatter plot with edge color, face color, size, and width of marker. (Scatter plot with line)

```
import numpy as np
import matplotlib.pyplot as
pltx = np.linspace(0, 10, 20)
y = np.sin(x)
plt.plot(x, y, '-o',
color='gray',
markersize=15,
linewidth=4,
markerfacecolor='yellow',
markeredgecolor='red',
markeredgewidth=4)
plt.ylim(-1.5, 1.5);
```



Scatter plot with random colors, size and transparency

```
import numpy as np
import matplotlib.pyplot as plt
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,
map='viridis')plt.colorbar()
```



A

APEC

Visualizing Errors

For any scientific measurement, accurate accounting for errors is nearly as important, if not more important, than accurate reporting of the number itself. For example, imagine that I am using some astrophysical observations to estimate the Hubble Constant, the local measurement of the expansion rate of the Universe. In visualization of data and results, showing these errors effectively can make a plot convey much more completeinformation.

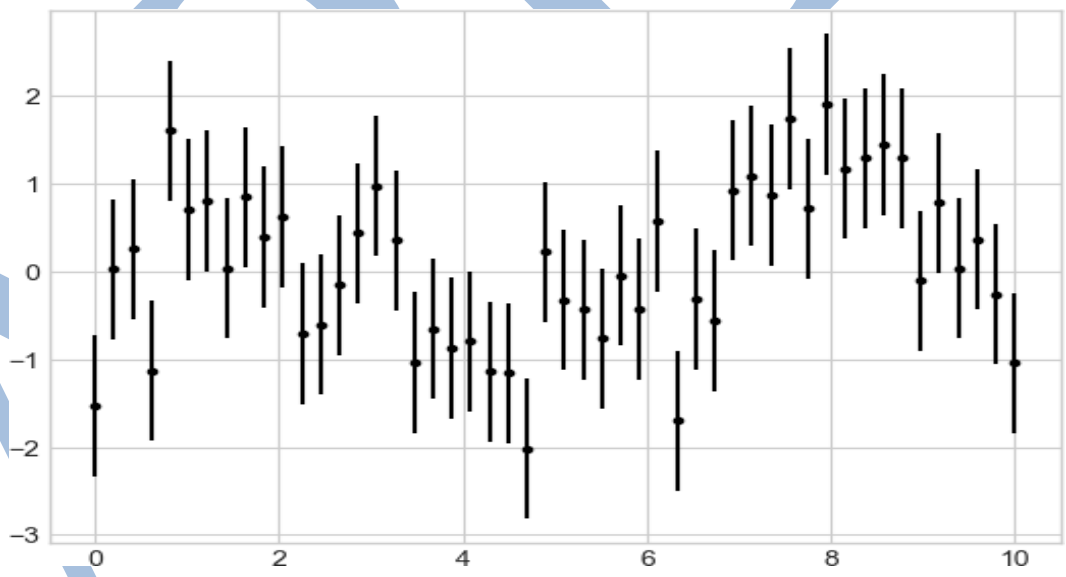
Types of errors

- Basic Errorbars
- Continuous Errors

Basic Errorbars

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

```
import matplotlib.pyplot as
plt plt.style.use('seaborn-
whitegrid')import numpy as
np
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');
```

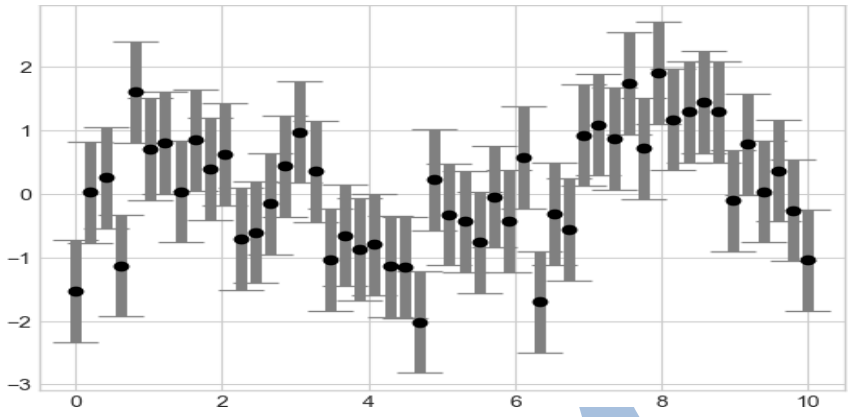


- Here the fmt is a format code controlling the appearance of lines and points, and has the same syntax as theshorthand used in plt.plot()
- In addition to these basic options, the errorbar function has many options to fine tune the outputs. Usingthese additional options you can easily customize the aesthetics of your errorbar plot.

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

A

APEC



Continuous Errors

- In some situations it is desirable to show errorbars on continuous quantities. Though Matplotlib does not have a built-in convenience routine for this type of application, it's relatively easy to combine primitives like plt.plot and plt.fill_between for a useful result.
- Here we'll perform a simple Gaussian process regression (GPR), using the Scikit-Learn API. This is a method of fitting a very flexible nonparametric function to data with a continuous measure of the uncertainty.

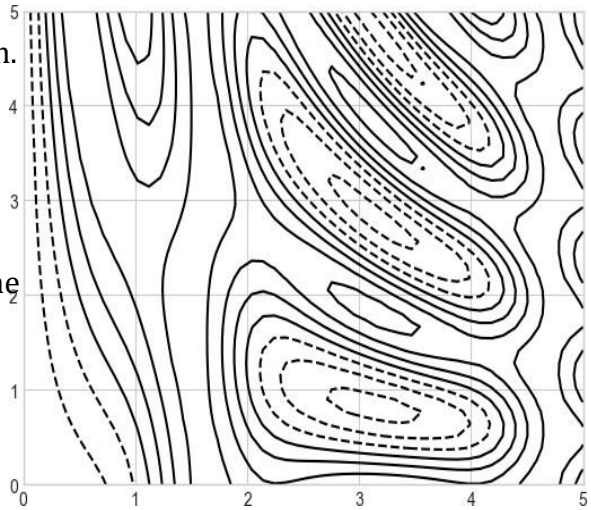
Density and Contour Plots

To display three-dimensional data in two dimensions using contours or color-coded regions.There are three Matplotlib functions that can be helpful for this task:

- plt.contour for contour plots,
- plt.contourf for filled contour plots, and
- plt.imshow for showing images.

Visualizing a Three-Dimensional Function

A contour plot can be created with the plt.contour function.



I

ttakes three arguments:

- a grid of x values,
- a grid of y values, and
- a grid of z values.

z

The x and y values represent positions on the plot, and the values will be represented by the contour levels.

The way to prepare such data is to use the np.meshgrid function, which builds two-dimensional grids from one-dimensional arrays:

Example

```
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');
```

APEC

- Notice that by default when a single color is used, negative values are represented by dashed lines, andpositive values by solid lines.
- Alternatively, you can color-code the lines by specifying a colormap with the cmap argument.
- We'll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range.

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```

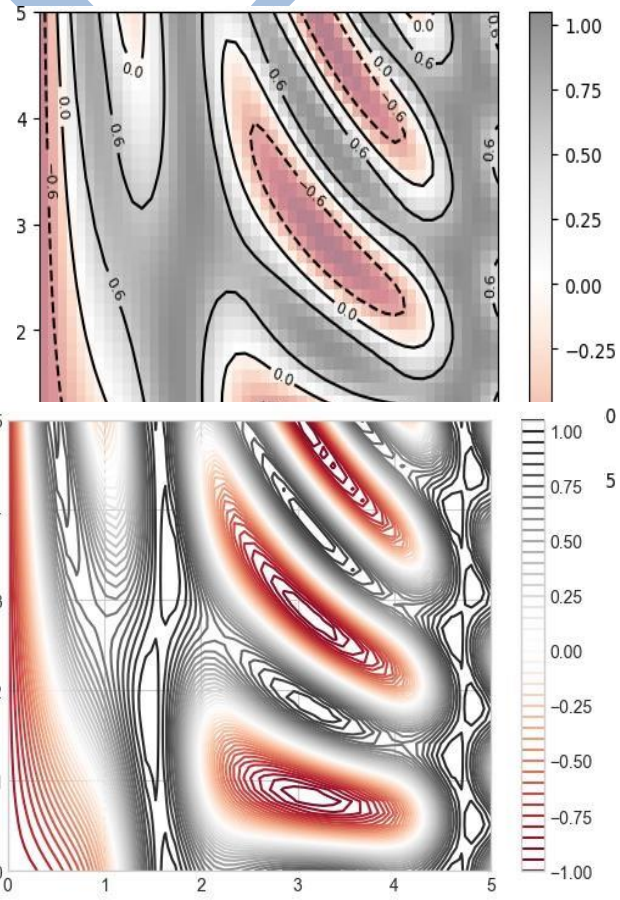
- One potential issue with this plot is that it is a bit "splotchy." That is, the color steps are discrete rather thancontinuous, which is not always what is desired.
- You could remedy this by setting the number of contours to a very high number, but this results in a ratherinefficient plot: Matplotlib must render a new polygon for each step in the level.
- A better way to handle this is to use the plt.imshow() function, which interprets a two-dimensional grid ofdata as an image.

There are a few potential gotchas with imshow().

- plt.imshow() doesn't accept an x and y grid, so you must manually specify the extent [xmin, xmax, ymin,ymax] of the image on the plot.
- plt.imshow() by default follows the standard image array definition where the origin is in the upper left, notin the lower left as in most contour plots. This must be changed when showing gridded data.
- plt.imshow() will automatically adjust the axis aspect ratio to match the input data; you can change this bysetting, for example, plt.axis(aspect='image') to make x and y units match.

Finally, it can sometimes be useful to combine contour plots and image plots. we'll use a partially transparent background image (with transparency set via the alpha parameter) and over-plot contours with labels on the contours themselves (using the plt.clabel() function):

```
contours = plt.contour(X, Y, Z, 3,
colors='black')plt.clabel(contours,
inline=True, fontsize=8) plt.imshow(Z,
extent=[0, 5, 0, 5], origin='lower',
cmap='RdGy', alpha=0.5)
plt.colorbar();
```



Example Program

```
import numpy as np
import matplotlib.pyplot as
pltdef 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.imshow(Z, extent=[0, 10, 0,
10],origin='lower', cmap='RdGy')
plt.colorbar()
```

A

APEC

Histograms

- Histogram is the simple plot to represent the large data set. A histogram is a graph showing frequencydistributions. It is a graph showing the number of observations within each given interval.

Parameters

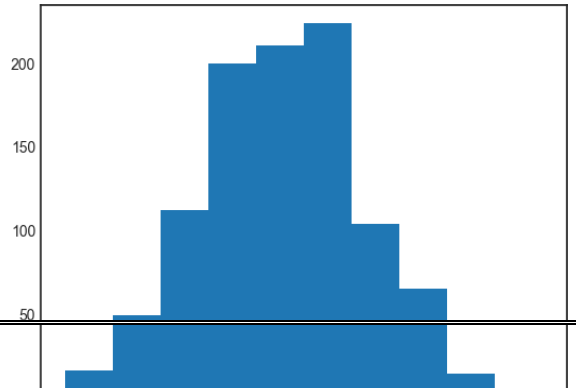
- plt.hist( ) is used to plot histogram. The hist() function will use an array of numbers to create a histogram,the array is sent into the function as an argument.

- bins - A histogram displays numerical data by grouping data into "bins" of equal width. Each bin is plotted as a bar whose height corresponds to how many data points are in that bin. Bins are also sometimes called "intervals", "classes", or "buckets".
- normed - Histogram normalization is a technique to distribute the frequencies of the histogram over a wider range than the current range.
- x - (n,) array or sequence of (n,) arrays Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.
- histtype - {'bar', 'barstacked', 'step', 'stepfilled'}, optionalThe type of histogram to draw.
    - 'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by side.
    - 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
    - 'step' generates a lineplot that is by default unfilled.
    - 'stepfilled' generates a lineplot that is by default filled.Default is 'bar'
- align - {'left', 'mid', 'right'}, optional
  Controls how the histogram is plotted.
    - 'left': bars are centered on the left bin edges.
    - 'mid': bars are centered between the bin edges.
    - 'right': bars are centered on the right bin edges.Default is 'mid'
- orientation - {'horizontal', 'vertical'}, optional
  If 'horizontal', barh will be used for bar-type histograms and the bottom kwarg will be the left edges.
- color - color or array_like of colors or None, optional
  Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

  Default is None
- label - str or None, optional. Default is None

Other parameter

- **kwargs - Patch properties, it allows us to pass avariable number of keyword arguments to a python function. ** denotes this type of function.



A

Example

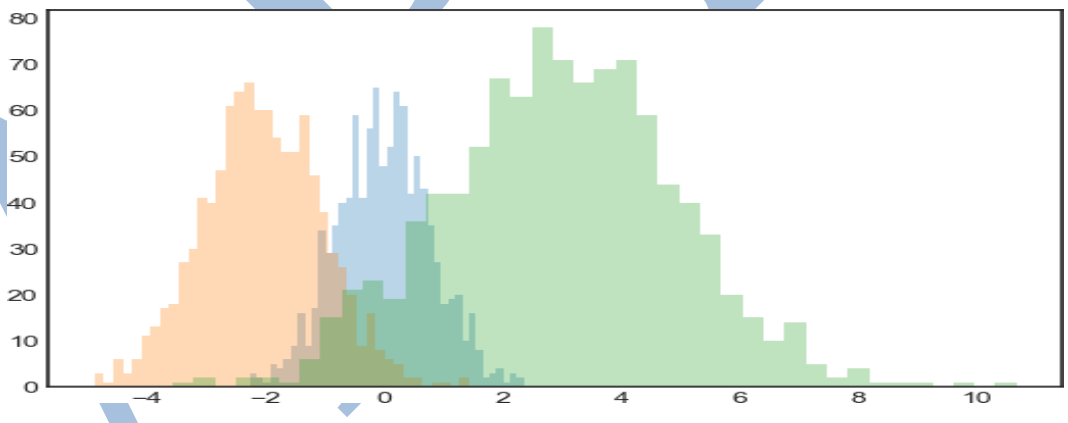
```
import numpy as np
import matplotlib.pyplot as
pltplt.style.use('seaborn-
white') data =
np.random.randn(1000)
plt.hist(data);
```

The hist() function has many options to tune both the calculation and the display; here's an example of a morecustomized histogram.

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

The plt.hist docstring has more information on other customization options available. I find this combination of histtype='stepfilled' along with some transparency alpha to 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,
bins=40)plt.hist(x1, **kwargs)
plt.hist(x2,
**kwargs)
```




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

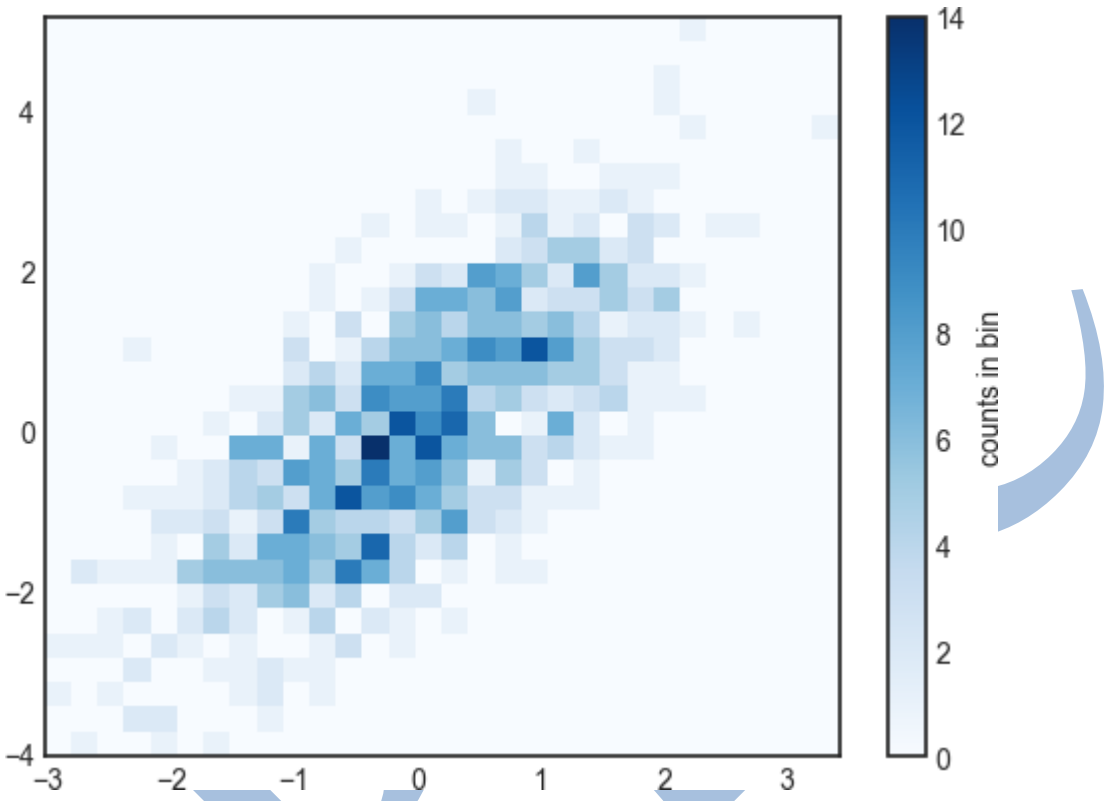
Two-Dimensional Histograms and Binnings

- We can create histograms in two dimensions by dividing points among two dimensional bins.
- We would define x and y values. Here for example We'll start by defining some data—an x and y arraydrawn from a multivariate Gaussian distribution:
- Simple way to plot a two-dimensional histogram is to use Matplotlib's plt.hist2d() function

Example

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 1000).T
plt.hist2d(x, y, bins=30, cmap='Blues')
```

```
cb = plt.colorbar()
cb.set_label('counts in bin
```

Legends

Plot legends give meaning to a visualization, assigning labels to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib.

Plot legends give meaning to a visualization, assigning labels to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib

```
plt.plot(x, np.sin(x), '-b', label='Sine')
plt.plot(x, np.cos(x), '--r',
label='Cosine')plt.legend();
```

Customizing Plot Legends

Location and turn off the frame - We can specify the location and turn off the frame. By the parameter loc and framon.

```
ax.legend(loc='upper left',
frameon=False)fig
```

A

Number of columns - We can use the ncol command to specify the number of columns in the legend.

```
ax.legend(frameon=False, loc='lower center',
ncol=2)fig
```

Rouded box, shadow and frame transparency

APEC

We can use a rounded box (fancybox) or add shadow, change the transparency (alpha value) of the frame, or change the padding around the text.

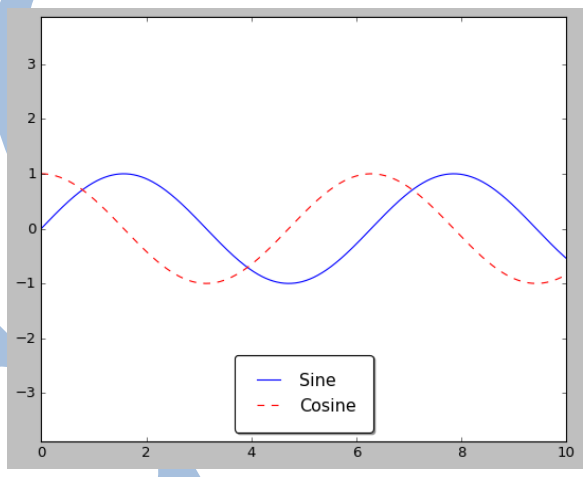
```
ax.legend(fancybox=True, framealpha=1, shadow=True,
borderpad=1)fig
```

Choosing Elements for the Legend

- The legend includes all labeled elements by default. We can change which elements and labels appear in thelegend by using the objects returned by plot commands.
- The plt.plot() command is able to create multiple lines at once, and returns a list of created line instances.
  Passing any of these to plt.legend() will tell it which to identify, along with the labels we'd like to specify

```
y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2,
0.5))lines = plt.plot(x, y)
plt.legend(lines[:2],['first','second']);

# Applying label
individually. plt.plot(x, y[:,
0], label='first')
plt.plot(x, y[:, 1], label='second')
plt.plot(x, y[:, 2:])
plt.legend(framealpha=1,
frameon=True);
```



Multiple legends

It is only possible to create a single legend for the entire plot. If you try to create a second legend using plt.legend() or ax.legend(), it will simply override the first one. We can work around this by creating a
new legend artist from scratch, and then using the lower-level ax.add_artist() method to manually add the second artist to the plot

Example

```
import matplotlib.pyplot as
pltplt.style.use('classic')
import numpy as np
x = np.linspace(0, 10, 1000)
ax.legend(loc='lower center', frameon=True,
shadow=True,borderpad=1,fancybox=True)fig
```

Color Bars

In Matplotlib, a color bar is a separate axes that can provide a key for the meaning of colors in a plot. Forcontinuous labels based on the color of points, lines, or regions, a labeled color bar can be a great tool.
The simplest colorbar can be created with the plt.colorbar() function.

A

APEC

Customizing
ColorbarsChoosing
color map.

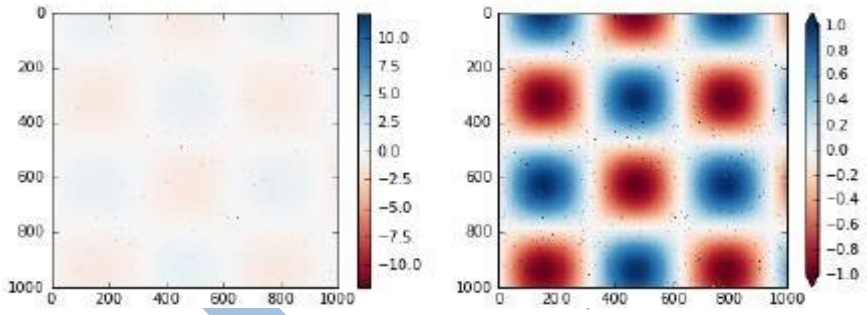
We can specify the colormap using the cmap argument to the plotting function that is creating the visualization.Broadly, we can know three different categories of colormaps:

- **Sequential colormaps** - These consist of one continuous sequence of colors (e.g., binary or viridis).
- **Divergent colormaps** - These usually contain two distinct colors, which show positive and negativedeviations from a mean (e.g., RdBu or PuOr).
- **Qualitative colormaps** - These mix colors with no particular sequence (e.g., rainbow or jet).

Color limits and extensions

- Matplotlib allows for a large range of colorbar customization. The colorbar itself is simply an instance ofplt.Axes, so all of the axes and tick formatting tricks we've learned are applicable.
- We can narrow the color limits and indicate the out-of-bounds values with a triangular arrow at the top andbottom by setting the extend property.

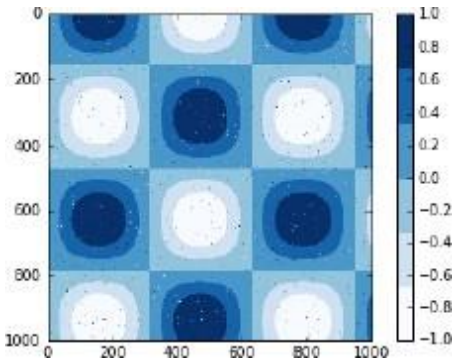
```
plt.subplot(1, 2, 2)
plt.imshow(I,
cmap='RdBu')
plt.colorbar(extend='both
')plt.clim(-1, 1);
```



Discrete colorbars

Colormaps are by default continuous, but sometimes you'd like to represent discrete values. The easiest way to do this is to use the plt.cm.get_cmap() function, and pass the name of a suitable colormap along with the number of desired bins.

```
plt.imshow(I, cmap=plt.cm.get_cmap('Blues',
6))plt.colorbar()
plt.clim(-1, 1);
```



Subplots

- 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.
- We'll explore four routines for creating subplots in Matplotlib.
  - plt.axes: Subplots by Hand
  - plt.subplot: Simple Grids of Subplots
  - plt.subplots: The Whole Grid in One Go
  - plt.GridSpec: More Complicated Arrangements

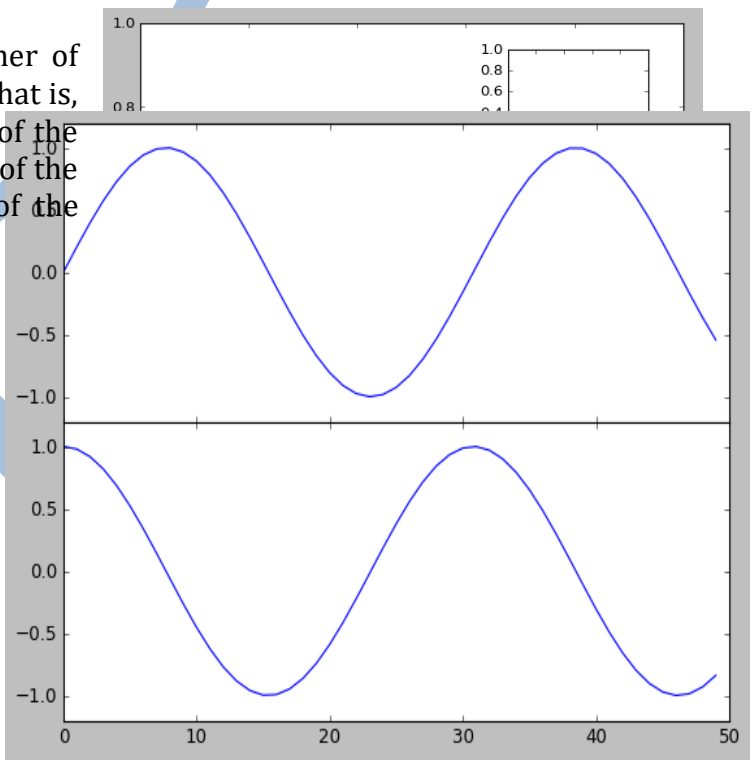
A

APEC

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, bydefault 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 [bottom, left, width,height] in the figure coordinate system, which ranges from 0 atthe bottom left of the figure to 1 at the top right of the figure.

For example,
we might create an inset axes at the top-right corner of another axes by setting the x and y position to 0.65 (that is, starting at 65% of the width and 65% of the height of the figure) and the xand y extents to 0.2 (that is, the size of the axes is 20% of the width and 20% of the height of the figure).



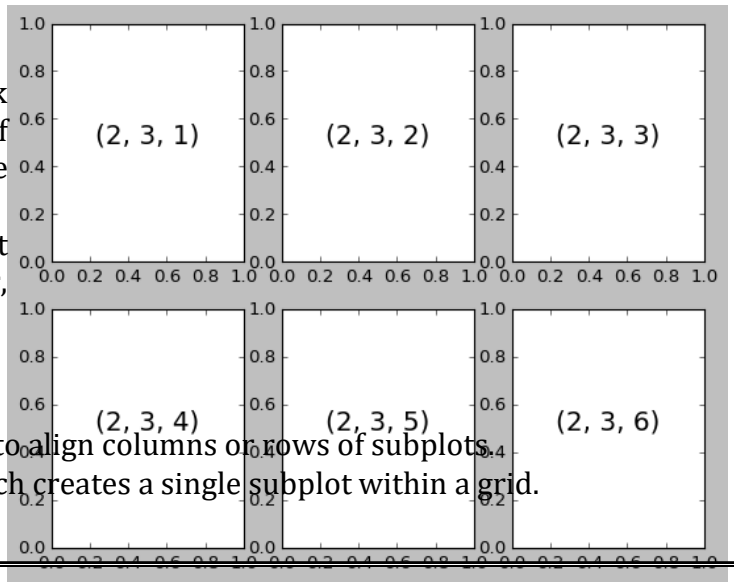
```
import matplotlib.pyplot as
pltimport numpy as np
ax1 = plt.axes() # standard axes
ax2 = plt.axes([0.65, 0.65, 0.2,
0.2])
```

Vertical sub plot

The equivalent of plt.axes() command within the object-oriented interface is ig.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));
```

- We now have two axes (the top with no tick labels) that are just touching: the bottom of the upper panel (at position 0.5) matches the top of the lower panel (at position 0.1+ 0.4).
- If the axis value is changed in second plot both the plots are separated with each other, exampleax2 = fig.add_axes([0.1, 0.01, 0.8, 0.4



plt.subplot: Simple Grids of Subplots

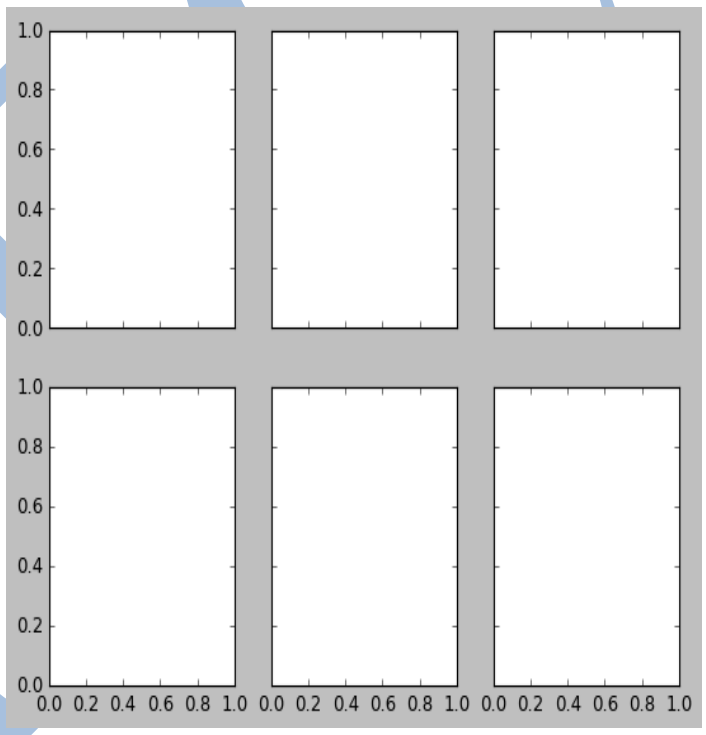
- Matplotlib has several convenience routines to align columns or rows of subplots.
- The lowest level of these is plt.subplot(), which creates a single subplot within a grid.

- 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 you're 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.
- The arguments are the number of rows and number of columns, along with optional keywords sharex and sharey, which allow you to specify the relationships between different axes.
- Here we'll create a 2×3 grid of subplots, where all axes in the same row share their y- axis scale, and all axes in the same column share their x-axis scale




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

Note that by specifying sharex and sharey, we've automatically removed inner labels on the grid to make the plot cleaner.

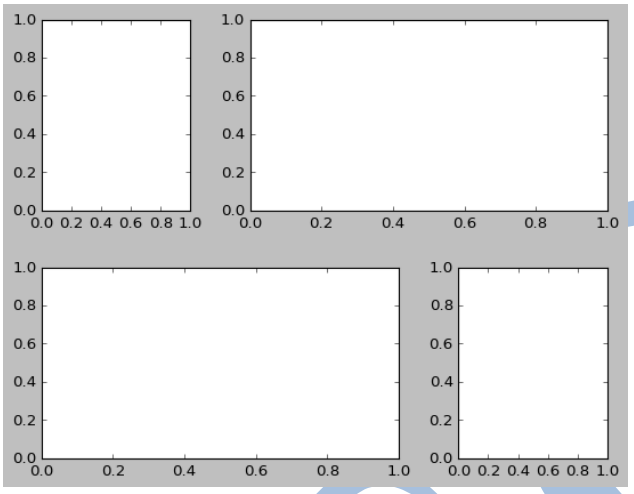
plt.GridSpec: More Complicated Arrangements

To go beyond a regular grid to subplots that span multiple rows and columns, plt.GridSpec() is the best tool. The plt.GridSpec() object does not create a plot by itself; it is simply a convenient interface that is recognizedby the plt.subplot() command.

For example, a gridspec for a grid of two rows and three columns with some specified width and height spacelooks like this:

```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
From this we can specify subplot locations and
extentsplt.subplot(grid[0, 0])
plt.subplot(grid[0, 1:])
```

```
plt.subplot(grid[1, :2])
plt.subplot(grid[1, 2]);
```

Text and Annotation

- The most basic types of annotations we will use are axes labels and titles, here we will see some more visualization and annotation information's.

- Text annotation can be done manually with the plt.text/ax.text command, which will place text at a particular x/y value.
- The ax.text method takes an x position, a y position, a string, and then optional keywords specifying the color, size, style, alignment, and other properties of the text. Here we used ha='right' and ha='center', where ha is short for horizontal alignment.

Transforms and Text Position

- We anchored our text annotations to data locations. Sometimes it's preferable to anchor the text to a position on the axes or figure, independent of the data. In Matplotlib, we do this by modifying the transform.
- Any graphics display framework needs some scheme for translating between coordinate systems.
- Mathematically, such coordinate transformations are relatively straightforward, and Matplotlib has a well- developed set of tools that it uses internally to perform them (the tools can be explored in the matplotlib.transforms submodule).
- There are three predefined transforms that can be useful in this situation.

    - ax.transData - Transform associated with data coordinates
    - ax.transAxes - Transform associated with the axes (in units of axes dimensions)
    - fig.transFigure - Transform associated with the figure (in units of figure dimensions)

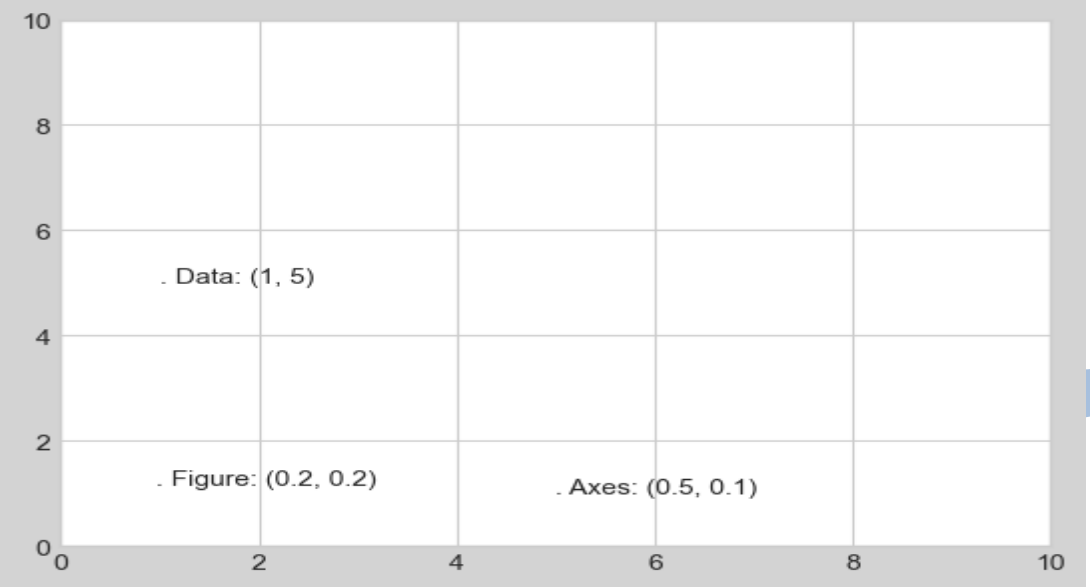
Example

```
import matplotlib.pyplot as plt
import matplotlib as mpl
plt.style.use('seaborn-
whitegrid')import numpy as np
import pandas as pd
fig, ax =
plt.subplots(facecolor='lightgray')
```

```
ax.axis([0, 10, 0, 10])
```



```
# transform=ax.transData is the default, but we'll specify it anyway
ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure);
```



Note that by default, the text is aligned above and to the left of the specified coordinates; here the "." at the beginning of each string will approximately mark the given coordinate location.

The transData coordinates give the usual data coordinates associated with the x- and y-axis labels. The transAxes coordinates give the location from the bottom-left corner of the axes (here the white box) as a fraction of the axes size.

The transfigure coordinates are similar, but specify the position from the bottom left of the figure (here the gray box) as a fraction of the figure size.
Notice now that if we change the axes limits, it is only the transData coordinates that will be affected, while the others remain stationary.

Arrows and Annotation

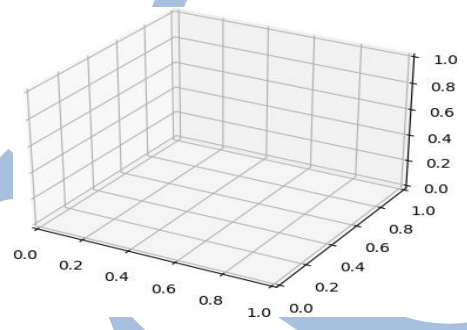
- Along with tick marks and text, another useful annotation mark is the simple arrow.
- Drawing arrows in Matplotlib is not much harder because there is a plt.arrow() function available.
- The arrows it creates are SVG (scalable vector graphics)objects that will be subject to the varying aspectratio of your plots, and the result is rarely what the user intended.
- The arrow style is controlled through the arrowprops dictionary, which has numerous options available.

A

Three-Dimensional Plotting in Matplotlib

We enable three-dimensional plots by importing the mplot3d toolkit, included with the main Matplotlib installation.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import
mplot3dfig = plt.figure()
ax = plt.axes(projection='3d')
```

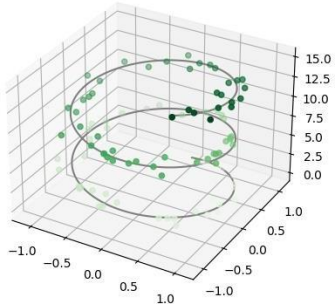


With this 3D axes enabled, we can now plot a varietyof three-dimensional plot types.

Three-Dimensional Points and Lines

The most basic three-dimensional plot is a line or scatter plot created from sets of (x, y, z) triples.
In analogy with the more common two-dimensional plots discussed earlier, we can create these using the ax.plot3D
and ax.scatter3D functions

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import
mplot3dax =
plt.axes(projection='3d')
# Data for a three-dimensional
linezline = np.linspace(0, 15,
1000) xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline,
'gray')
# Data for three-dimensional scattered
pointszdata = 15 *
np.random.random(100)
```




```
xdata = np.sin(zdata) + 0.1 *
np.random.randn(100)ydata = np.cos(zdata) + 0.1 *
np.random.randn(100)
```

```
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
plt.show()
```

Notice that by default, the scatter points have their transparency adjusted to give a sense of depth on the page

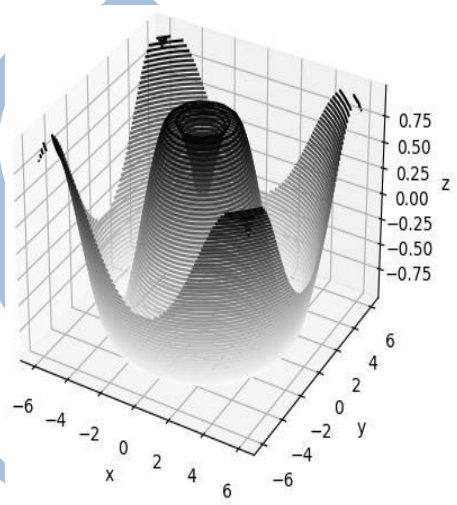
A

Three-Dimensional Contour Plots

APEC

- mplot3d contains tools to create three-dimensional relief plots using the same inputs.
- Like two-dimensional ax.contour plots, ax.contour3D requires all the input data to be in the form of two-dimensional regular grids, with the Z data evaluated at each point.
- Here we'll show a three-dimensional contour diagram of a three dimensional sinusoidal function

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import
mplot3ddef f(x, y):
   return np.sin(np.sqrt(x ** 2 + y **
2))x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6,
30)X, Y =
np.meshgrid(x, y) Z =
f(X, Y)
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50,
cmap='binary')ax.set_xlabel('x')
ax.set_ylabel('y'
)
ax.set_zlabel('z'
)plt.show()
```



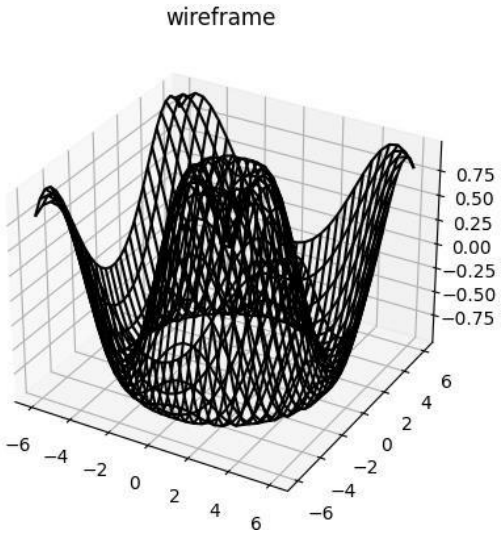
Sometimes the default viewing angle is not optimal, in which case we can use the view_init method to set theelevation and azimuthal angles.

```
ax.view_init(60,
35)fig
```

Wire frames and Surface Plots

- Two other types of three-dimensional plots that work on gridded data are wireframes and surface plots.
- These take a grid of values and project it onto the specified threedimensional surface, and can make theresulting three-dimensional forms quite easy to visualize.

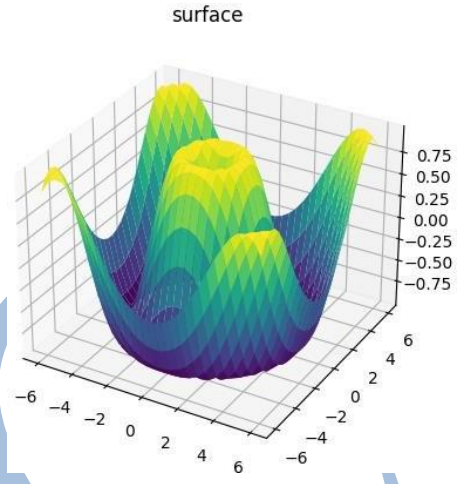
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import
mplot3dfig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z,
color='black')
ax.set_title('wireframe');
plt.show()
```



- A surface plot is like a wireframe plot, but each faceof the wireframe is a filled polygon.

- Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized

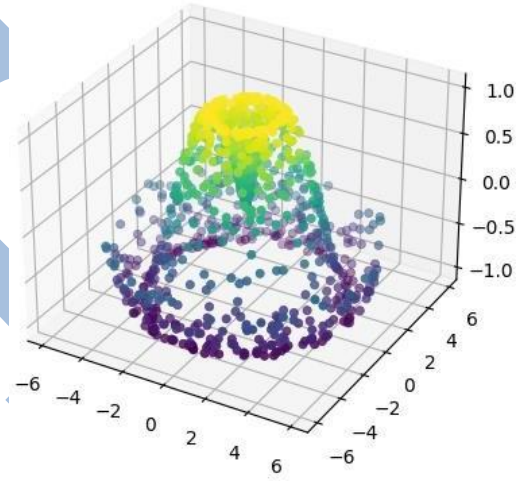
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import
mplot3dax =
plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1,
cstride=1,cmap='viridis',
edgecolor='none') ax.set_title('surface')
plt.show()
```



Surface Triangulations

- For some applications, the evenly sampled grids required by the preceding routines are overly restrictive and inconvenient.
- In these situations, the triangulation-based plots can be very useful.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import
mplot3d
theta = 2 * np.pi *
np.random.random(1000)r = 6 *
np.random.random(1000)
x = np.ravel(r *
np.sin(theta)) y = np.ravel(r
* np.cos(theta))z = f(x, y)
ax = plt.axes(projection='3d')
ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5)
```



Geographic Data with Basemap

- One common type of visualization in data science is thatof geographic data.
- Matplotlib's main tool for this type of visualization is the Basemap toolkit, which is one of several Matplotlib toolkits that live under the mpl_toolkits namespace.
- Basemap is a useful tool for Python users to have in their virtual toolbelts
- Installation of Basemap. Once you have the Basemap toolkit installed and imported, geographic plots alsorequire the PIL package in Python 2, or the pillow package in Python 3.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import
Basemapplt.figure(figsize=(8, 8))
m = Basemap(projection='ortho',
resolution=None,lat_0=50, lon_0=-100)
m.bluemarble(scale=0.5);
```

- Matplotlib axes that understands spherical coordinates andallows us to easily over-plot data on the map

- We'll use an etopo image (which shows topographical features both on land and under the ocean) as themap background

Program to display particular area of the map with latitude andlongitude lines



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import
Basemapfrom itertools import chain
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc',
resolution=None,width=8E6, height=8E6,
lat_0=45, lon_0=-100,)
m.etopo(scale=0.5,
alpha=0.5)def draw_map(m,
scale=0.2):
# draw a shaded-relief image
  m.shadedrelief(scale=scale)
# lats and longs are returned as a dictionary
lats = m.drawparallels(np.linspace(-90, 90,
13))
lons = m.drawmeridians(np.linspace(-180, 180,
13))# keys contain the plt.Line2D instances
lat_lines = chain(*(tup[1][0] for tup in lats.items()))
lon_lines = chain(*(tup[1][0] for tup in
lons.items()))all_lines = chain(lat_lines, lon_lines)
# cycle through these lines and set the desired
stylefor line in all_lines:
  line.set(linestyle='-', alpha=0.3, color='r')
```

Map Projections

The Basemap package implements several dozen such projections, all referenced by a short format code. Here we'llbriefly demonstrate some of the more common ones.

- Cylindrical projections
- Pseudo-cylindrical projections
- Perspective projections
- Conic projections

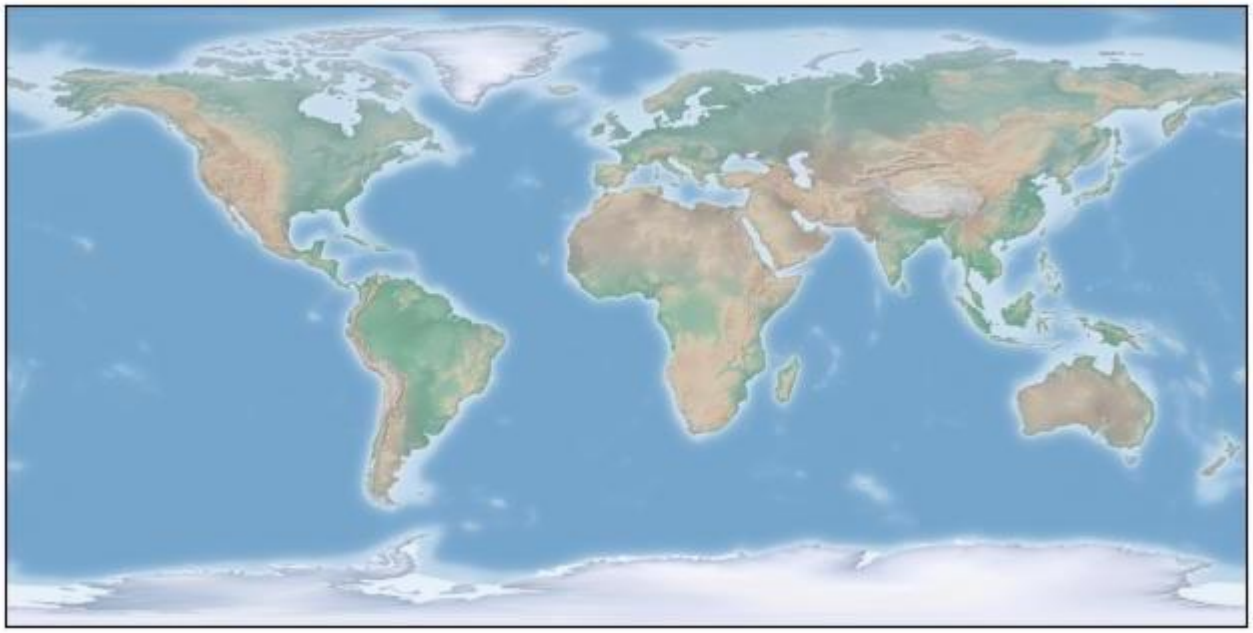
Cylindrical projection

- The simplest of map projections are cylindrical projections, in which lines of constant latitude and longitudeare mapped to horizontal and vertical lines, respectively.
- This type of mapping represents equatorial regions quite well, but results in extreme distortions near thepoles.
- The spacing of latitude lines varies between different cylindrical projections, leading to different conservation properties, and different distortion near the poles.
- Other cylindrical projections are the Mercator (projection='merc') and the cylindrical equal-area (projection='cea') projections.
- The additional arguments to Basemap for this view specify the latitude (lat) and longitude (lon) of thelower-left corner (llcrnr) and upper-right corner (urcrnr) for the desired map, in units of degrees.

A

APEC

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='cyl',
resolution=None,llcrnrlat=-90, urcrnrlat=90,
llcrnrlon=-180,
urcrnrlon=180, )
draw_map(m)
```



Pseudo-cylindrical projections

- Pseudo-cylindrical projections relax the requirement that meridians (lines of constant longitude) remainvertical; this can give better properties near the poles of the projection.
- The Mollweide projection (projection='moll') is one common example of this, in which all meridians areelliptical arcs
- It is constructed so as to
- preserve area across the map: though there are distortions near the poles, the area of small patches reflects the true area.
- Other pseudo-cylindrical projections are the sinusoidal (projection='sinu') and Robinson (projection='robin') projections.
- The extra arguments to Basemap here refer to the central latitude (lat_0) and longitude (lon_0) for the desired map.



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import
Basemapfig = plt.figure(figsize=(8, 6),
edgecolor='w')
m = Basemap(projection='moll',
resolution=None,lat_0=0, lon_0=0)
draw_map(m)
```

A

APEC

Perspective projections

- Perspective projections are constructed using a particular choice of perspective point, similar to if you photographed the Earth from a particular point in space (a point which, for some projections, technically lieswithin the Earth!)J.
- One common example is the orthographic projection (projection='ortho'), which shows one side of the globeas seen from a viewer at a very long distance.
- Thus, it can show only half the globe at a time.
- Other perspective-based projections include the gnomonic projection (projection='gnom') and stereographic projection (projection='stere').
- These are often the most useful for showing small portions of the map.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import
Basemapfig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho',
resolution=None,lat_0=50, lon_0=0)
draw_map(m);
```



Conic projections

- A conic projection projects the map onto a single cone, which is then unrolled.
- This can lead to very good local properties, but regions far from the focus point of the cone may becomevery distorted.
- One example of this is the Lambert conformal conic projection (projection='lcc').
- It projects the map onto a cone arranged in such a way that two standard parallels (specified in Basemap by lat_1 and lat_2) have well-represented distances, with scale decreasing between them and increasing outsideof them.
- Other useful conic projections are the equidistant conic (projection='eqdc') and the Albers equal-area (projection='aea') projection

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import
Basemapfig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None,
lon_0=0, lat_0=50, lat_1=45, lat_2=55, width=1.6E7,
height=1.2E7)draw_map(m)
```



APEC

A

Drawing a Map Background

The Basemap package contains a range of useful functions for drawing borders of physical features like continents,oceans, lakes, and rivers, as well as political boundaries such as countries and US states and counties. The following are some of the available drawing functions that you may wish to explore using IPython's helpfeatures:

• Physical boundaries and bodies of water
drawcoastlines() - Draw continental coast
lines
drawlsmask() - Draw a mask between the land and sea, for use with projecting images on one or
the otherdrawmapboundary() - Draw the map boundary, including the fill color for oceans
drawrivers() - Draw rivers on the map
fillcontinents() - Fill the continents with a given color; optionally fill lakes with another color

• Political boundaries
drawcountries() - Draw country
boundaries drawstates() - Draw US state
boundaries drawcounties() - Draw US
county boundaries

• Map features
drawgreatcircle() - Draw a great circle between two
pointsdrawparallels() - Draw lines of constant latitude
drawmeridians() - Draw lines of constant longitude
drawmapscale() - Draw a linear scale on the map

• Whole-globe images
bluemarble() - Project NASA's blue marble image onto the
mapshadedrelief() - Project a shaded relief image onto the
map etopo() - Draw an etopo relief image onto the map
warpimage() - Project a user-provided image onto the map

Plotting Data on Maps

- The Basemap toolkit is the ability to over-plot a variety of data onto a map background.
- There are many map-specific functions available as methods of the Basemap

instance.Some of these map-specific methods are:
contour()/contourf() - Draw contour lines or filled
contoursimshow() - Draw an image
pcolor()/pcolormesh() - Draw a pseudocolor plot for irregular/regular
meshesplot() - Draw lines and/or markers
scatter() - Draw points with
markersquiver() - Draw vectors
barbs() - Draw wind barbs
drawgreatcircle() - Draw a great
circle

APEC

A

APEC

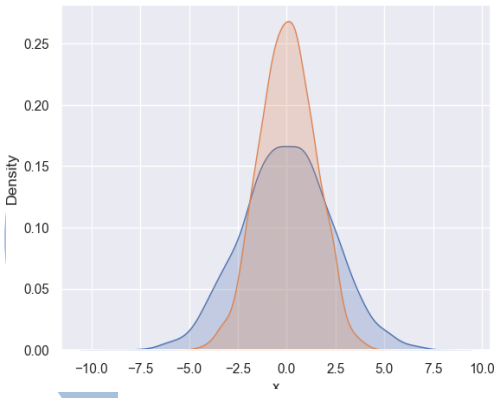
Visualization with Seaborn

The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful forstatistical data exploration, and even some statistical model fitting.

Histograms, KDE, and densities

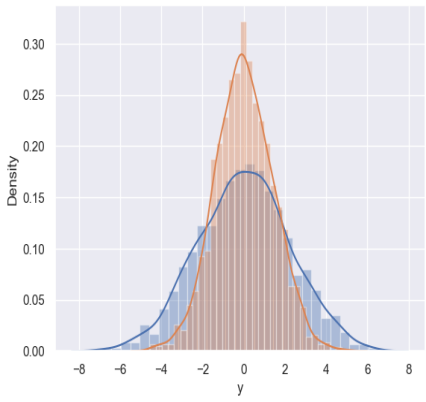
- In statistical data visualization, all you want is to plot histograms and joint distributions of variables. We have seen that this is relatively straightforward in Matplotlib
- Rather than a histogram, we can get a smooth estimate of the distribution using a kernel density estimation, which Seaborn does with sns.kdeplot

```
import pandas as
pd import seaborn
as sns
data = np.random.multivariate_normal([0, 0], [[5, 2], [2,
2]], size=2000)
data = pd.DataFrame(data, columns=['x', 'y'])
for col in 'xy':
        sns.kdeplot(data[col], shade=True)
```



- Histograms and KDE can be combined using distplot

```
sns.distplot(data['x'])
sns.distplot(data['y']);
```



- If we pass the full two-dimensional dataset to kdeplot, we will get atwo-dimensional visualization of the data.
- We can see the joint distribution and the marginal distributions together using sns.jointplot.

Pair plots

When you generalize joint plots to datasets of larger dimensions, you end up with pair plots. This is very useful forexploring correlations between multidimensional data, when you'd like to plot all pairs of values against each other.

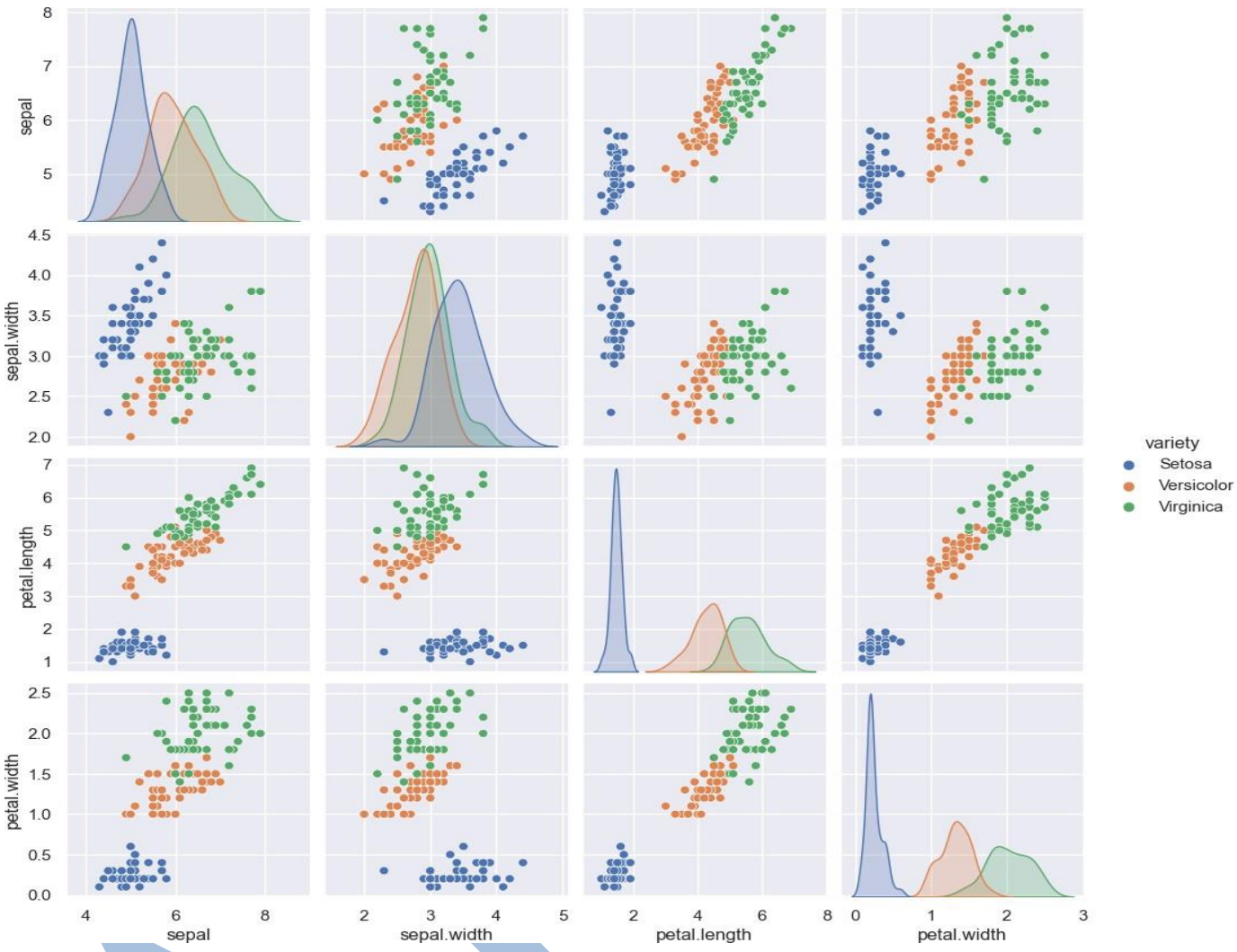
We'll demo this with the Iris dataset, which lists measurements of petals and sepals of three iris species:

```
import seaborn as sns
iris = sns.load_dataset("iris")
sns.pairplot(iris, hue='species',
size=2.5);
```

APEC

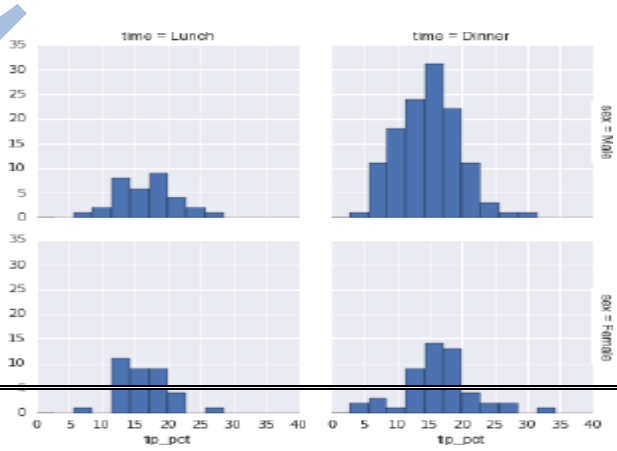
A

APEC



Faceted histograms

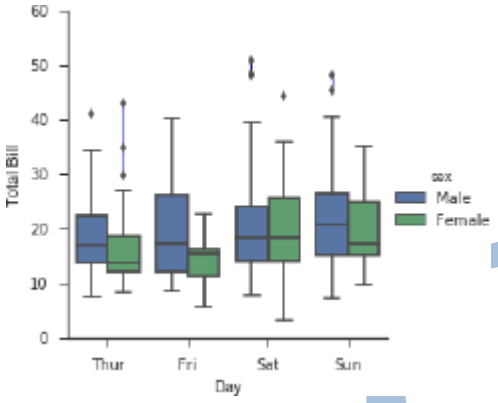
- Sometimes the best way to view data is via histograms of subsets. Seaborn's FacetGrid makes this extremely simple.
- We'll take a look at some data that shows the amount that restaurant staff receive in tips based on variousindicator data



A

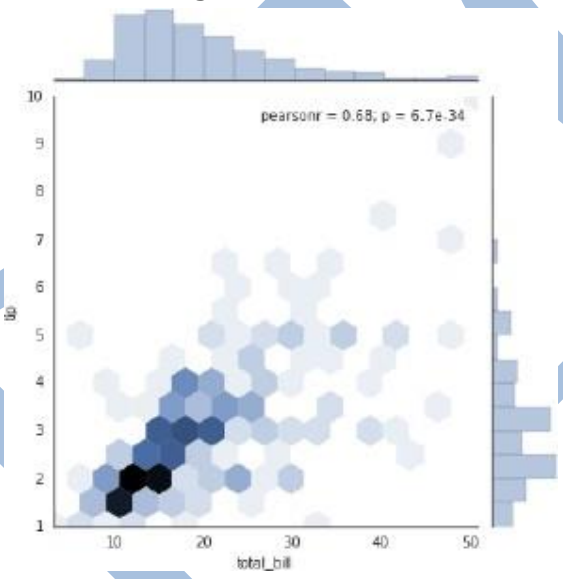
Factor plots

Factor plots can be useful for this kind of visualization as well. This allows you to view the distribution of aparameter within bins defined by any other parameter.



Joint distributions

Similar to the pair plot we saw earlier, we can use sns.jointplot to show the joint distribution between differentdatasets, along with the associated marginal distributions.



Bar plots

Time series can be plotted with sns.factorplot.

