PRACTICE SESSION 4

PLOTTING AND VISUALIZATIONS

Plotting and Visualization

> Making informative visualizations (sometimes called *plots*) is one of the most important tasks in data analysis.

> It may be a part of the exploratory process

– Examples: to identify outliers or needed data transformations, or as a way of generating ideas for models

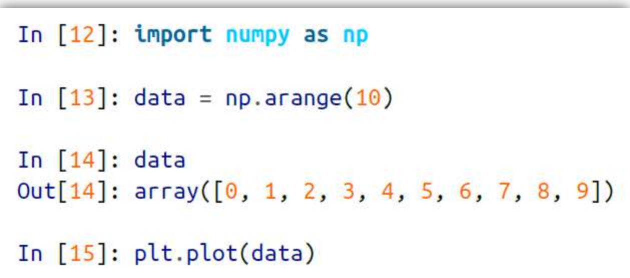
> matplotlib is a desktop plotting package designed for creating (mostly twodimensional) publication-quality plots.

– supports various GUI backends on all operating systems

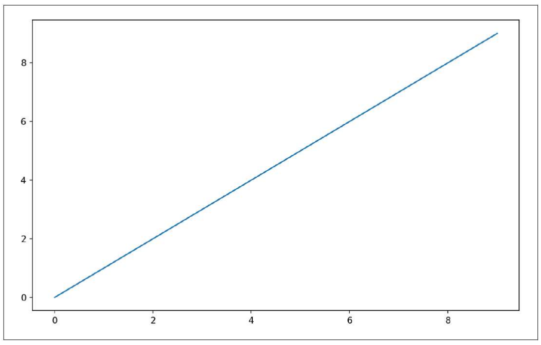
– can export visualizations to all of the common vector and raster graphics formats (PDF, SVG, JPG, PNG, BMP, GIF, etc.).

A Brief matplotlib API Primer

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



```
In [12]: import numpy as np

In [13]: data = np.arange(10)

In [14]: data
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: plt.plot(data)
```

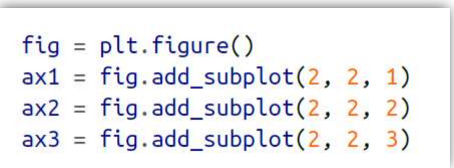


Figures and Subplots

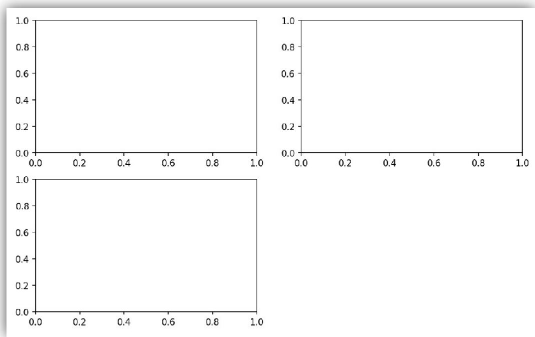
> Plots in matplotlib reside within a **`Figure`** object

> You can create a new figure with **`plt.figure`**

> You can't make a plot with a blank figure. You have to create one or more subplots using **`add_subplot`**:

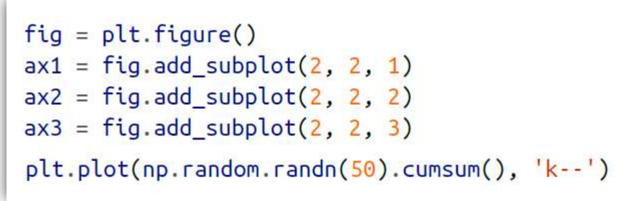


```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

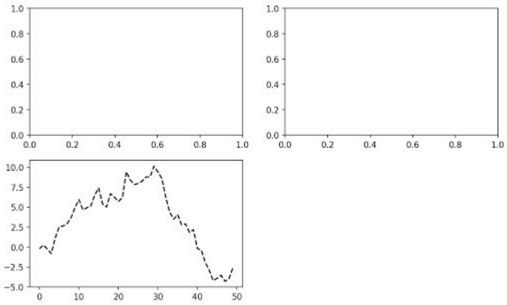


Figures and Subplots

> When you issue a plotting command matplotlib draws on the last figure and subplot used

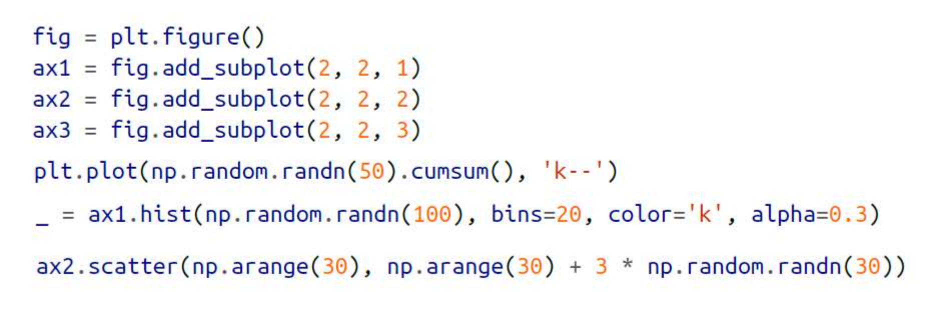


```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
plt.plot(np.random.randn(50).cumsum(), 'k--')
```



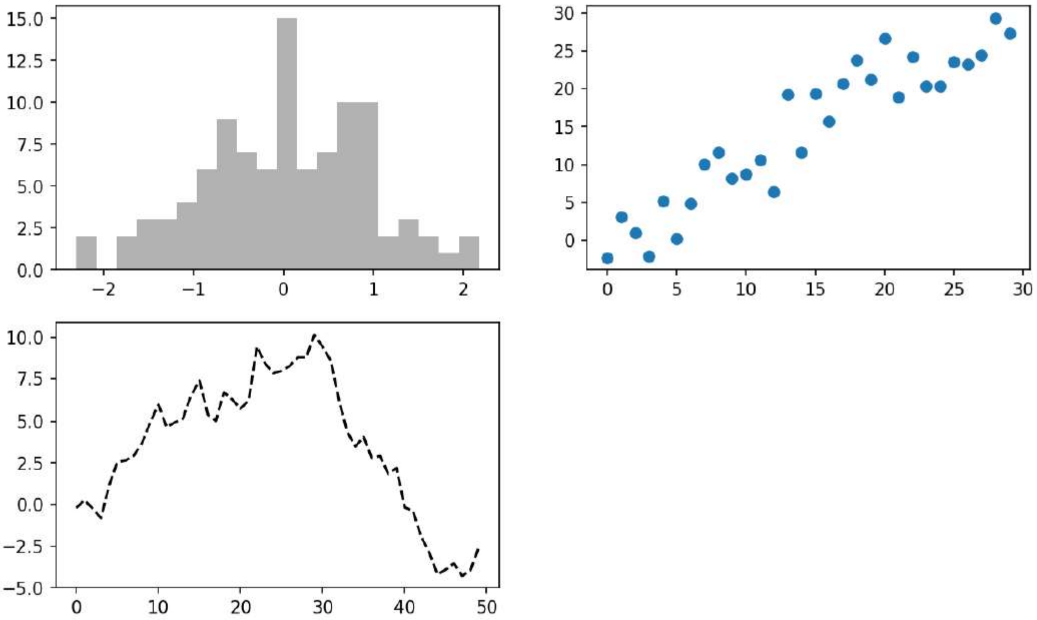
The 'k--' is a style option instructing matplotlib to plot a black dashed line.

Figures and Subplots



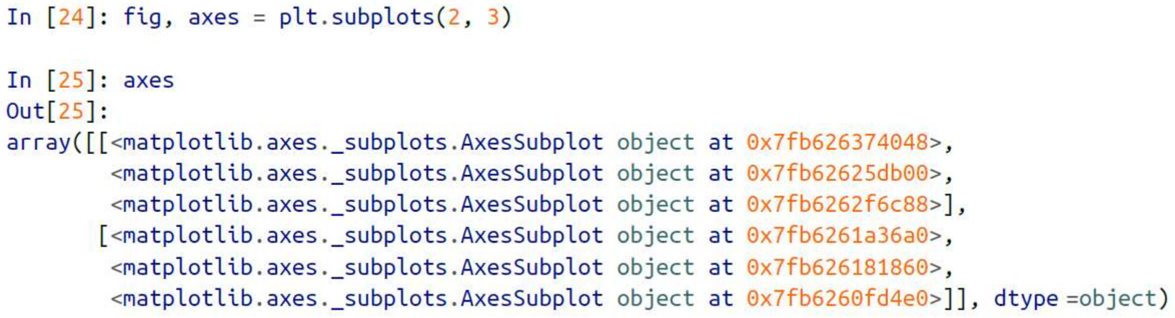
```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
plt.plot(np.random.randn(50).cumsum(), 'k--')
_ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3)
ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
```

Figures and Subplots



Figures and Subplots

> Creating a figure with a grid of subplots is a very common task, so matplotlib includes a convenience method, **`plt.subplots`**, that creates a new figure and returns a NumPy array containing the created subplot objects:



```
In [24]: fig, axes = plt.subplots(2, 3)

In [25]: axes
Out[25]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fb626374048>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb62625db00>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6262f6c88>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fb6261a36a0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb626181860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6260fd4e0>]], dtype =object)
```

`pyplot.subplots` options

Argument	Description
`nrows`	Number of rows of subplots
`ncols`	Number of columns of subplots
`sharex`	All subplots should use the same x-axis ticks (adjusting the **`xlim`** will affect all subplots)
`sharey`	All subplots should use the same y-axis ticks (adjusting the **`ylim`** will affect all subplots)
`subplot_kw`	Dict of keywords passed to **`add_subplot`** call used to create each subplot
`**fig_kw`	Additional keywords to subplots are used when creating the figure, such as **`plt.subplots(2, 2, figsize=(8, 6))`**

Adjusting the spacing around subplots

> By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots.

> This spacing is all specified relative to the height and width of the plot

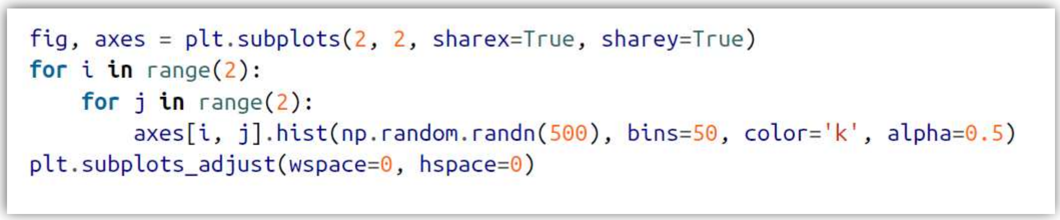
> If you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself.

> You can change the spacing using the **`subplots_adjust`** method on **`Figure`** objects

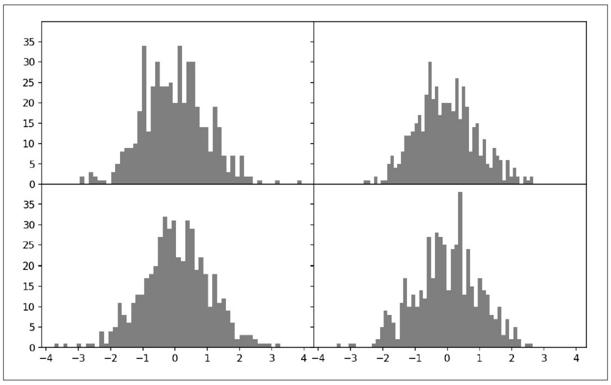
```
subplots_adjust(left=None, bottom=None, right=None,
top=None, wspace=None, hspace=None)
```

> wspace and hspace controls the percent of the figure width and figure height, respectively, to use as spacing between subplots.

Adjusting the spacing around subplots



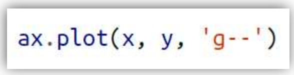
```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(np.random.randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```



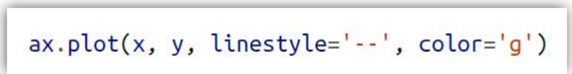
Colors, Markers, and Line Styles

> Matplotlib’s main **plot** function accepts arrays of x and y coordinates and optionally a string abbreviation indicating color and line style.

> For example, to plot x versus y with green dashes, you would execute:

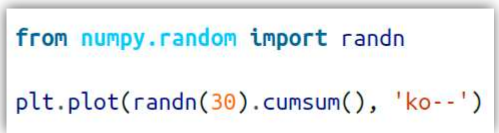


```
ax.plot(x, y, 'g--')
```

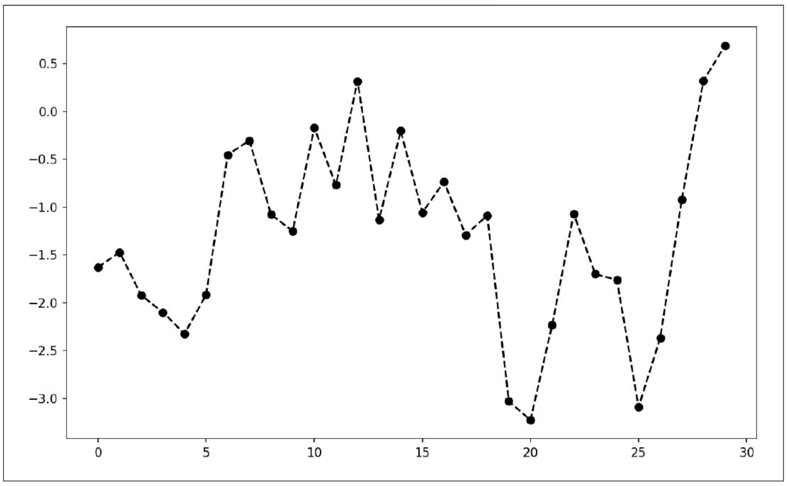


```
ax.plot(x, y, linestyle='--', color='g')
```

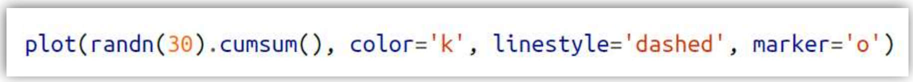
Colors, Markers, and Line Styles



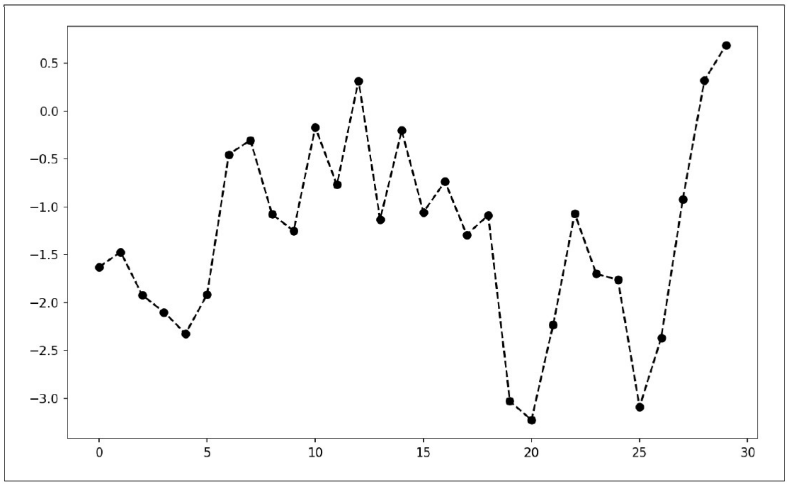
```
from numpy.random import randn

plt.plot(randn(30).cumsum(), 'ko--')
```



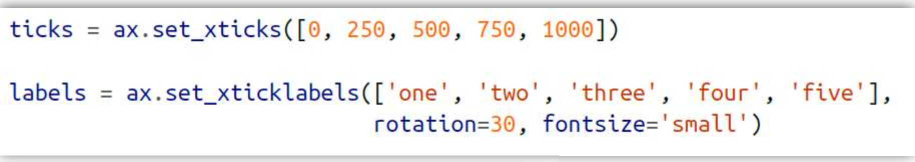
Colors, Markers, and Line Styles



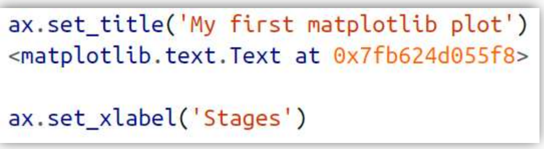
```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```



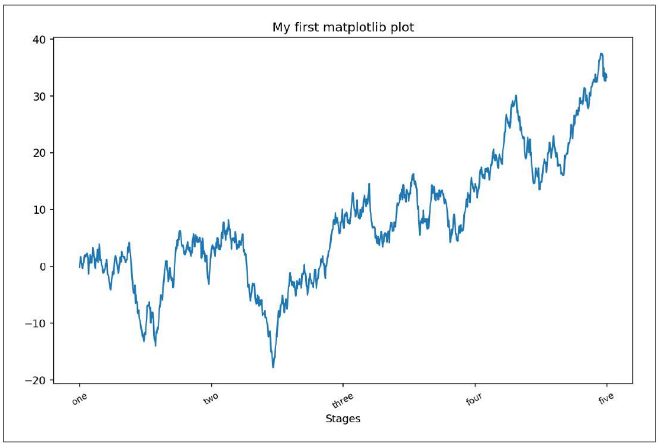
Ticks, Labels, and Legends

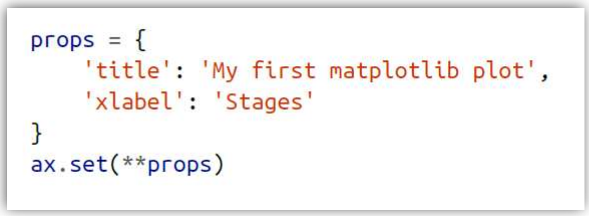


```
ticks = ax.set_xticks([0, 250, 500, 750, 1000])

labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
                            rotation=30, fontsize='small')
```



```
ax.set_title('My first matplotlib plot')
<matplotlib.text.Text at 0x7fb624d055f8>

ax.set_xlabel('Stages')
```





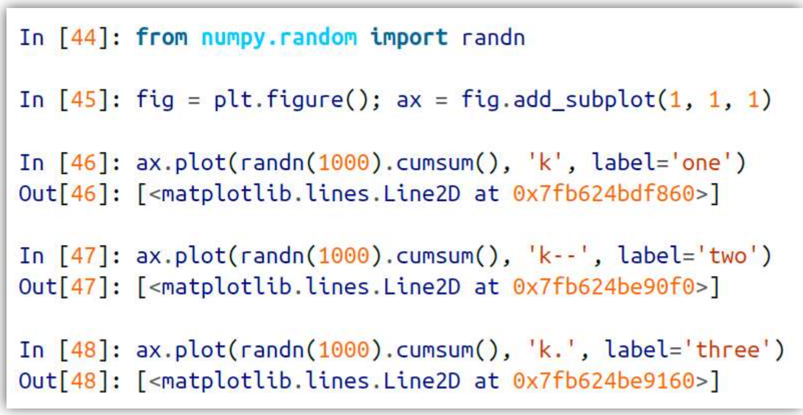
```
props = {
    'title': 'My first matplotlib plot',
    'xlabel': 'Stages'
}
ax.set(**props)
```

Adding legends

> Legends are another critical element for identifying plot elements.

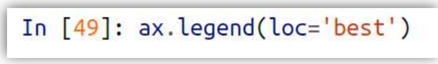
> There are a couple of ways to add one.

> The easiest is to pass the label argument when adding each piece of the plot:

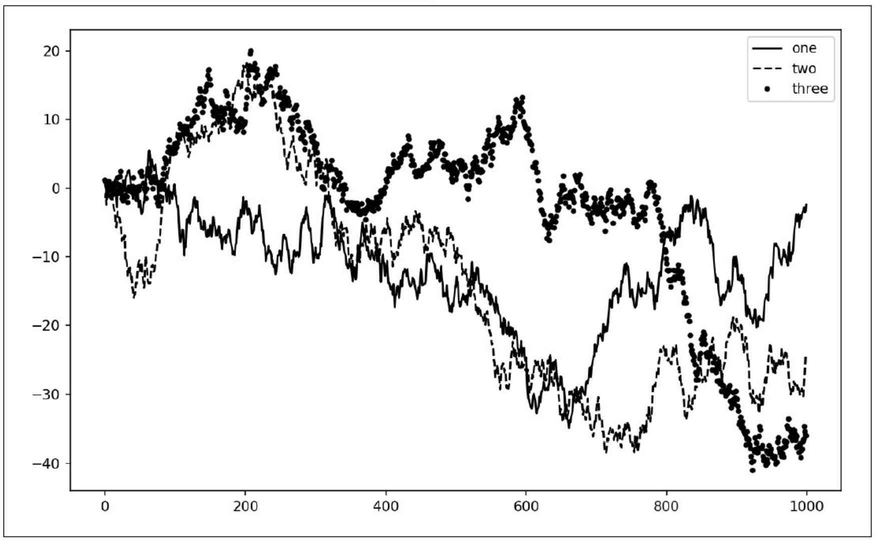


```
In [44]: from numpy.random import randn

In [45]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)

In [46]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[46]: [<matplotlib.lines.Line2D at 0x7fb624bdf860>]

In [47]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[47]: [<matplotlib.lines.Line2D at 0x7fb624be90f0>]

In [48]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[48]: [<matplotlib.lines.Line2D at 0x7fb624be9160>]
```

Adding legends



```
In [49]: ax.legend(loc='best')
```

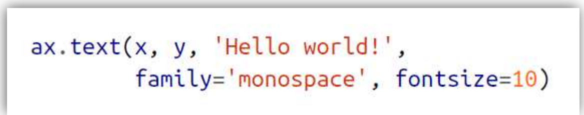


Annotations and Drawing on a Subplot

> In addition to the standard plot types, you may wish to draw your own plot annotations, which could consist of text, arrows, or other shapes.

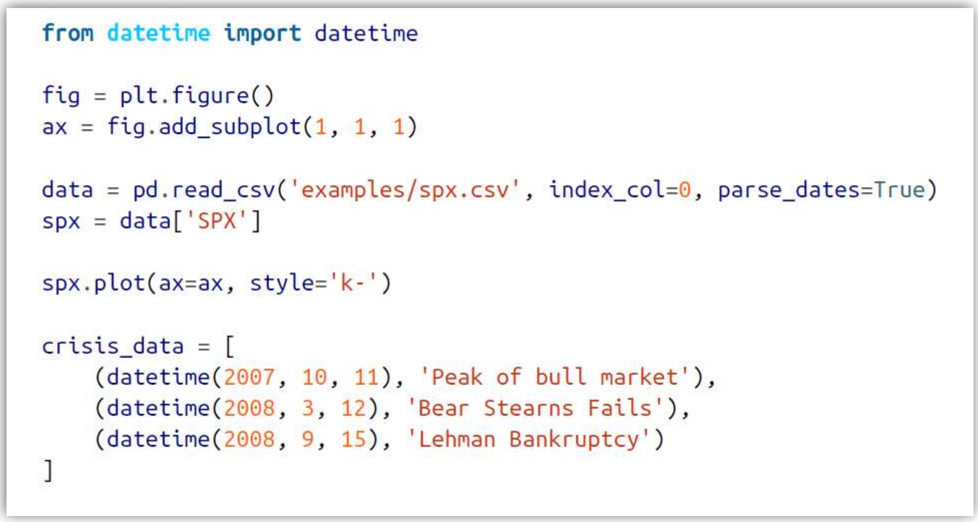
> You can add annotations and text using the **`text`**, **`arrow`**, and **`annotate`** functions.

> **`text`** draws text at given coordinates **`(x, y)`** on the plot with optional custom styling:



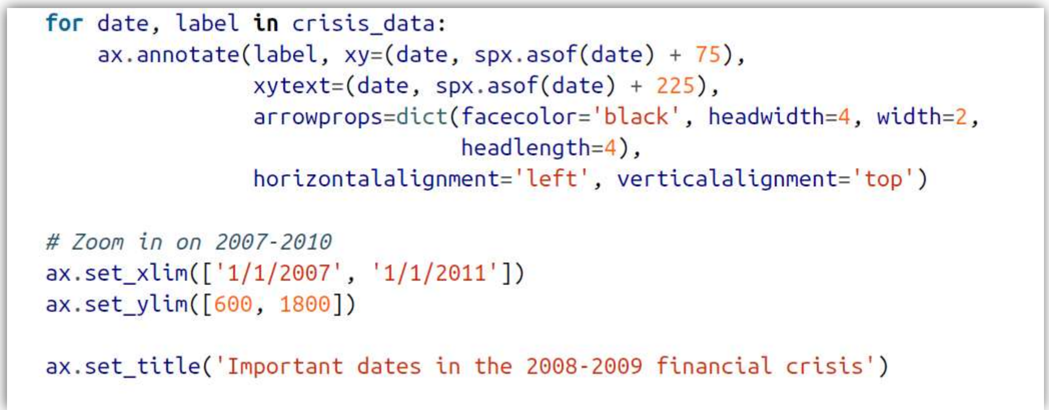
```
ax.text(x, y, 'Hello world!',
        family='monospace', fontsize=10)
```

Annotations and Drawing on a Subplot



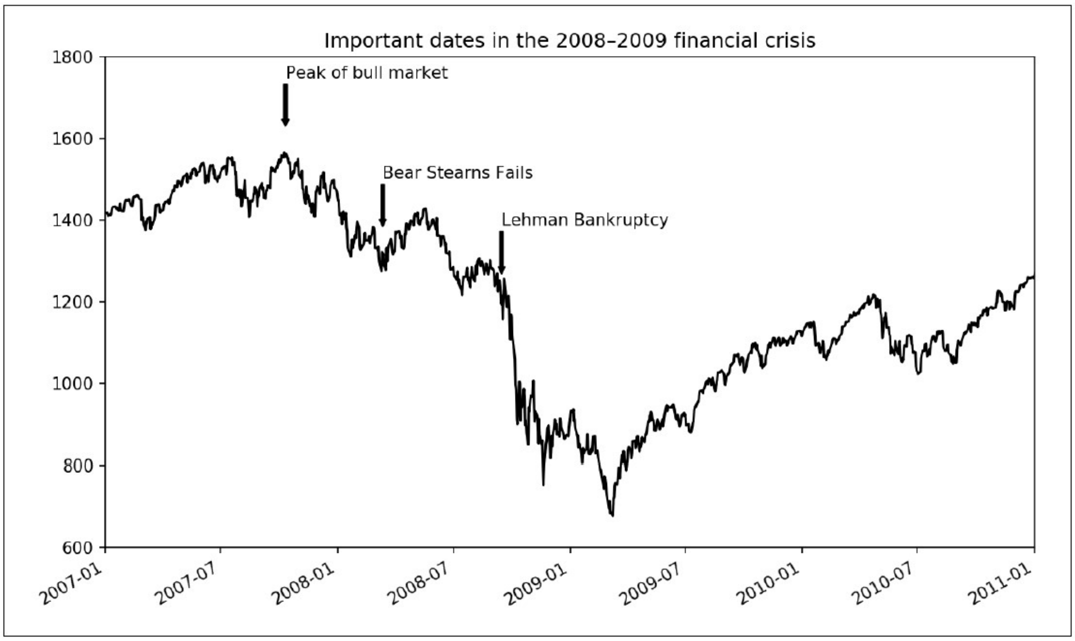
```
from datetime import datetime

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

data = pd.read_csv('examples/spx.csv', index_col=0, parse_dates=True)
spx = data['SPX']

spx.plot(ax=ax, style='k-')

crisis_data = [
    (datetime(2007, 10, 11), 'Peak of bull market'),
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')
]
```

Annotations and Drawing on a Subplot



```
for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor='black', headwidth=4, width=2,
                                headlength=4),
                horizontalalignment='left', verticalalignment='top')

# Zoom in on 2007-2010
ax.set_xlim(['1/1/2007', '1/1/2011'])
ax.set_ylim([600, 1800])

ax.set_title('Important dates in the 2008-2009 financial crisis')
```

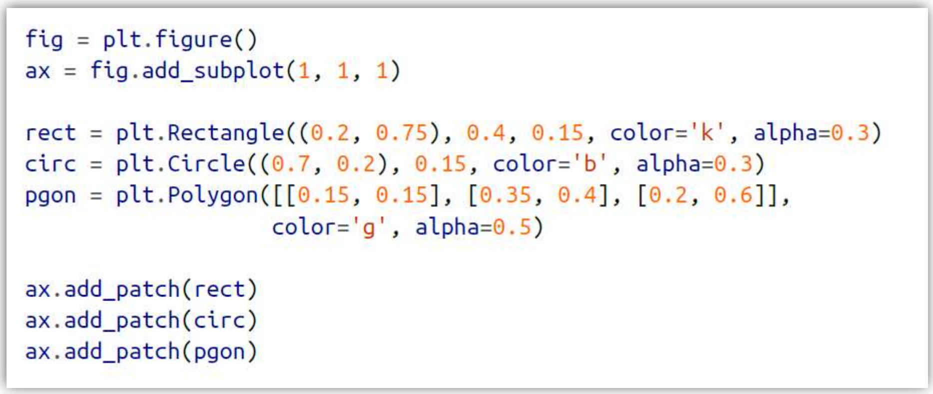
Annotations and Drawing on a Subplot



Annotations and Drawing on a Subplot

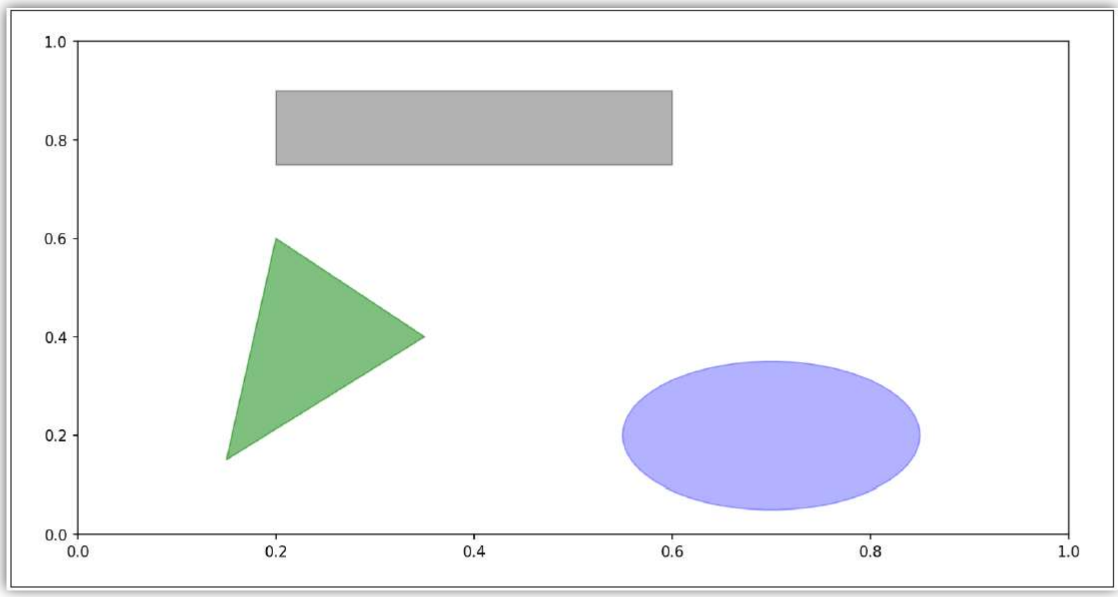
> matplotlib has objects that represent many common shapes, called *patches*.

> Some of these, like **`Rectangle`** and **`Circle`**, are found in **`matplotlib.pyplot`**, but the full set is located in **`matplotlib.patches`**.



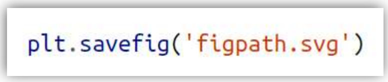
```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                   color='g', alpha=0.5)

ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

Annotations and Drawing on a Subplot



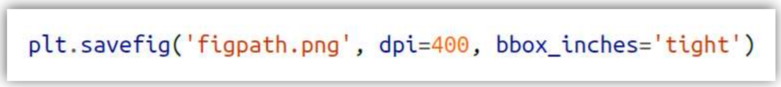
Saving Plots to File

- > You can save the active figure to file using **`plt.savefig`**.
- > This method is equivalent to the figure object’s savefig instance method.
- > For example, to save an SVG version of a figure, you need only type:



```
plt.savefig('figpath.svg')
```

- > *The file type is inferred from the file extension*
- > To get the same plot as a PNG with minimal whitespace around the plot and at 400 DPI, you would do:



```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

`Figure.savefig` options

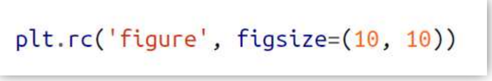
Argument	Description
`fname`	String containing a filepath or a Python file-like object. The figure format is inferred from the file extension (e.g., `.pdf` for PDF or .**png** for PNG)
`dpi`	The figure resolution in dots per inch; defaults to 100 out of the box but can be configured
`facecolor`, `edgecolor`	The color of the figure background outside of the subplots; `'w'` (white), by default
`format`	The explicit file format to use (`'png'`, `'pdf'`, `'svg'`, `'ps'`, `'eps'`, ...)
`bbox_inches`	The portion of the figure to save; if 'tight' is passed, will attempt to trim the empty space around the figure

matplotlib Configuration

> matplotlib comes configured with color schemes and defaults that are geared primarily toward preparing figures for publication

> Fortunately, nearly all of the default behavior can be customized via an extensive set of global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on

> One way to modify the configuration programmatically from Python is to use the **`rc`** method



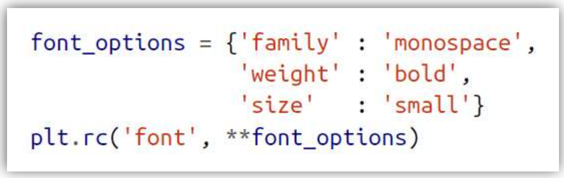
```
plt.rc('figure', figsize=(10, 10))
```

> The first argument to rc is the component you wish to customize, such as 'figure', 'axes', 'xtick', 'ytick', 'grid', 'legend', or many others.

> After that can follow a sequence of the new parameters.

matplotlib Configuration

> An easy way to write down the options in your program is as a dict:



```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size'   : 'small'}
plt.rc('font', **font_options)
```

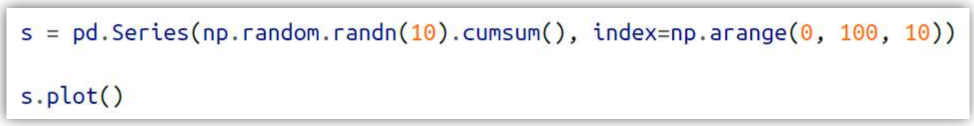
PLOTTING WITH PANDAS AND SEABORN

Plotting with pandas and seaborn

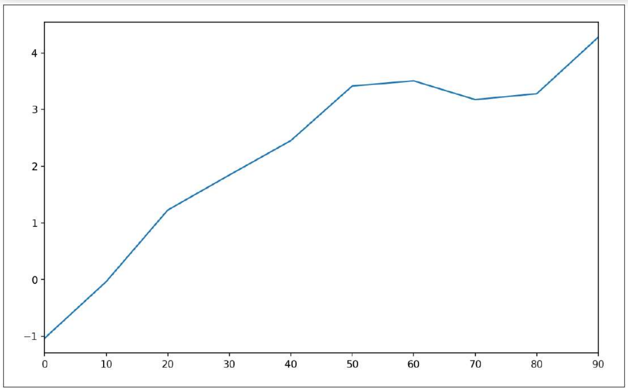
- > matplotlib can be a fairly low-level tool
- > In pandas we may have multiple columns of data, along with row and column labels.
- > pandas itself has built-in methods that simplify creating visualizations from Data-Frame and Series objects
- > Another library is **seaborn**, a statistical graphics library created by Michael Waskom.
- > Seaborn simplifies creating many common visualization types.
- > Importing seaborn modifies the default matplotlib color schemes and plot styles to improve readability and aesthetics
- > Even if you do not use the seaborn API, you may prefer to import seaborn as a simple way to improve the visual aesthetics of general matplotlib plots.

Line Plots

- > Series and DataFrame each have a plot attribute for making some basic plot types.

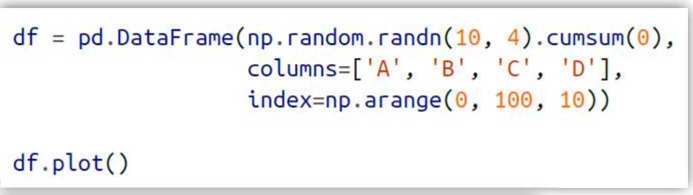


```
s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))

s.plot()
```

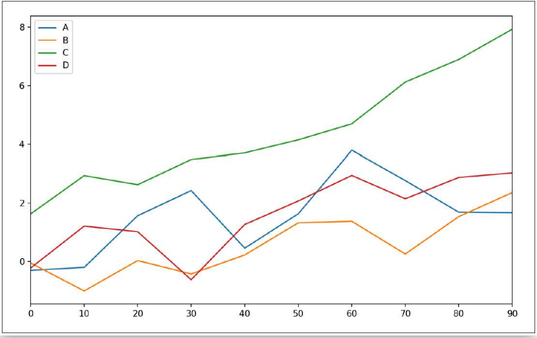


Line Plots

> DataFrame’s **plot** method plots each of its columns as a different line on the same subplot, creating a legend automatically



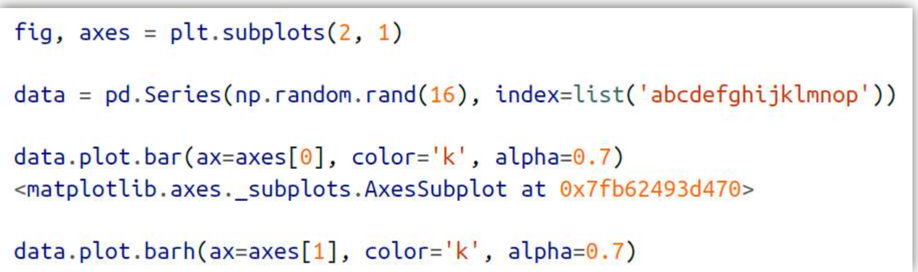
```
df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
                  columns=['A', 'B', 'C', 'D'],
                  index=np.arange(0, 100, 10))

df.plot()
```



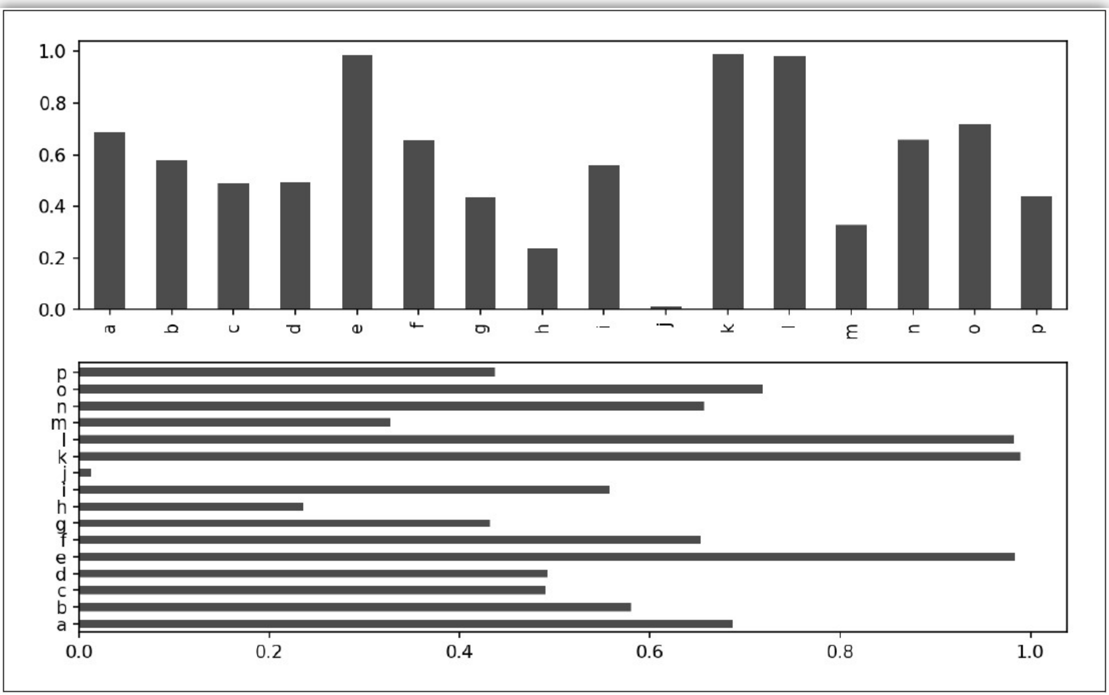
Bar Plots

> The **plot.bar()** and **plot.barh()** make vertical and horizontal bar plots, respectively.

> The Series or DataFrame index will be used as the x (bar) or y (barh) ticks

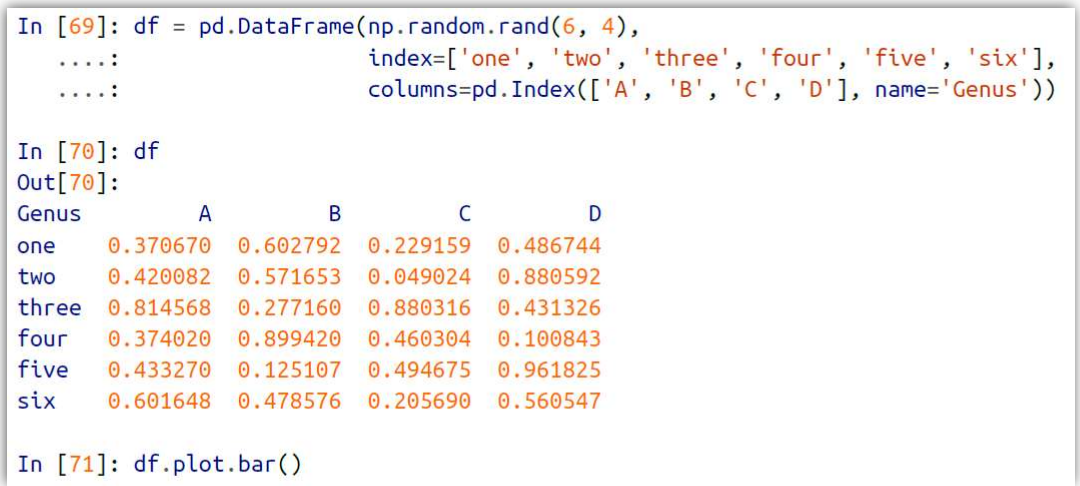


```
fig, axes = plt.subplots(2, 1)

data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))

data.plot.bar(ax=axes[0], color='k', alpha=0.7)
<matplotlib.axes._subplots.AxesSubplot at 0x7fb62493d470>

data.plot.barh(ax=axes[1], color='k', alpha=0.7)
```



Bar Plots

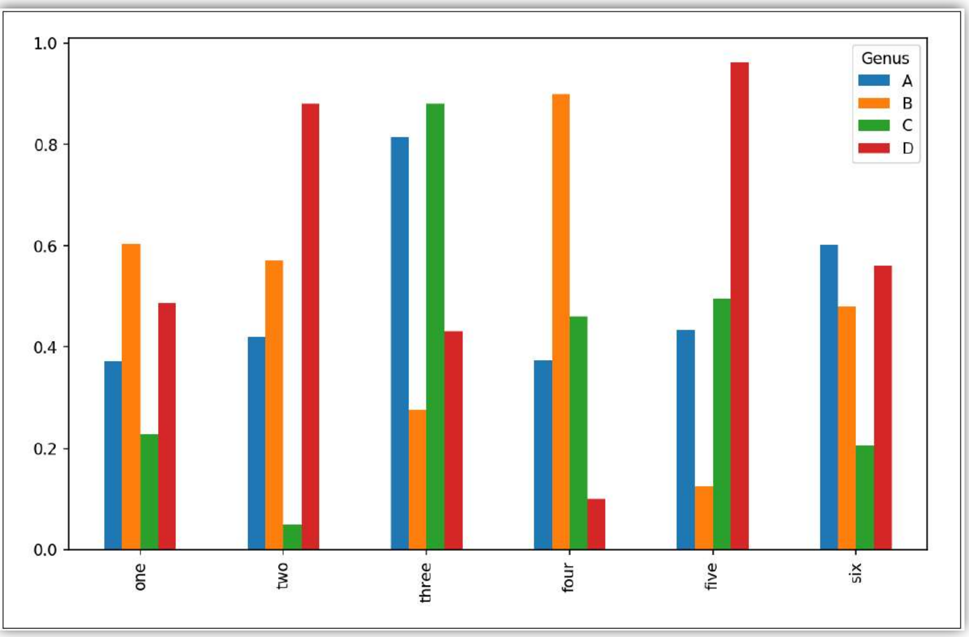
> With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value.



```
In [69]: df = pd.DataFrame(np.random.rand(6, 4),
   ....:                   index=['one', 'two', 'three', 'four', 'five', 'six'],
   ....:                   columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))

In [70]: df
Out[70]:
Genus          A         B         C         D
one     0.370670  0.602792  0.229159  0.486744
two     0.420082  0.571653  0.049024  0.880592
three   0.814568  0.277160  0.880316  0.431326
four    0.374020  0.899420  0.460304  0.100843
five    0.433270  0.125107  0.494675  0.961825
six     0.601648  0.478576  0.205690  0.560547

In [71]: df.plot.bar()
```

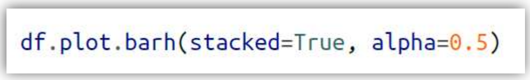
Bar Plots

DataFrame bar plot



Bar Plots

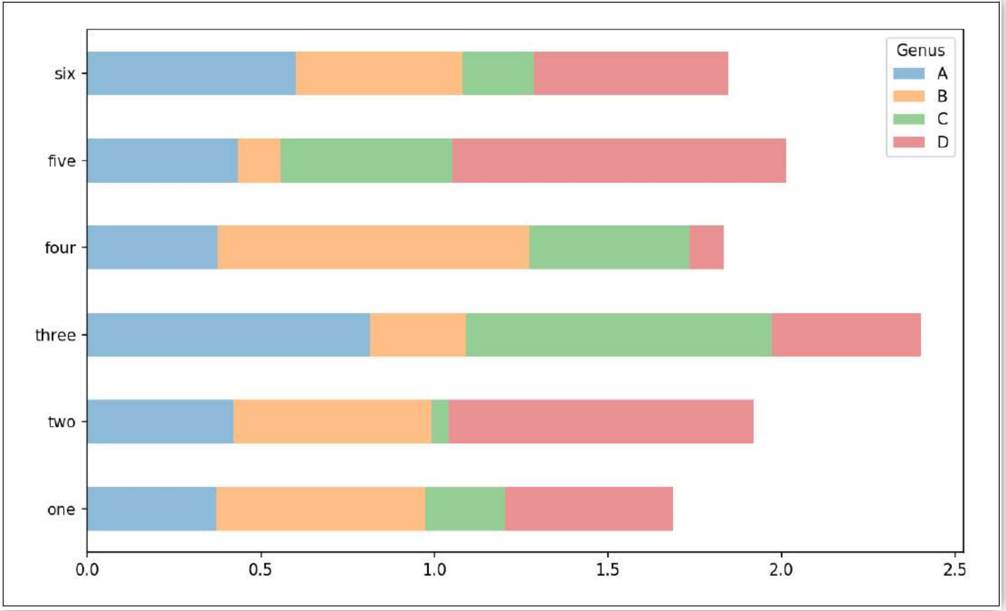
> We create stacked bar plots from a DataFrame by passing **`stacked=True`**, resulting in the value in each row being stacked together



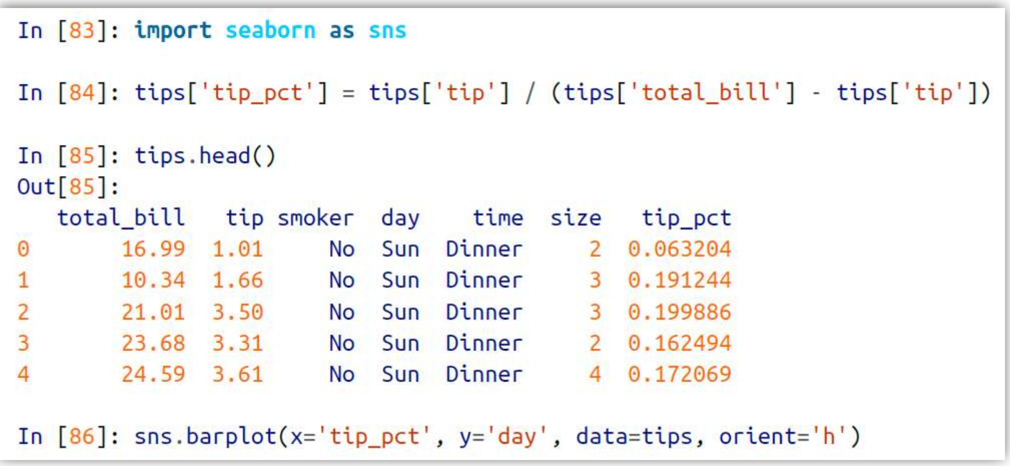
```
df.plot.barh(stacked=True, alpha=0.5)
```

Bar Plots

DataFrame stacked bar plot

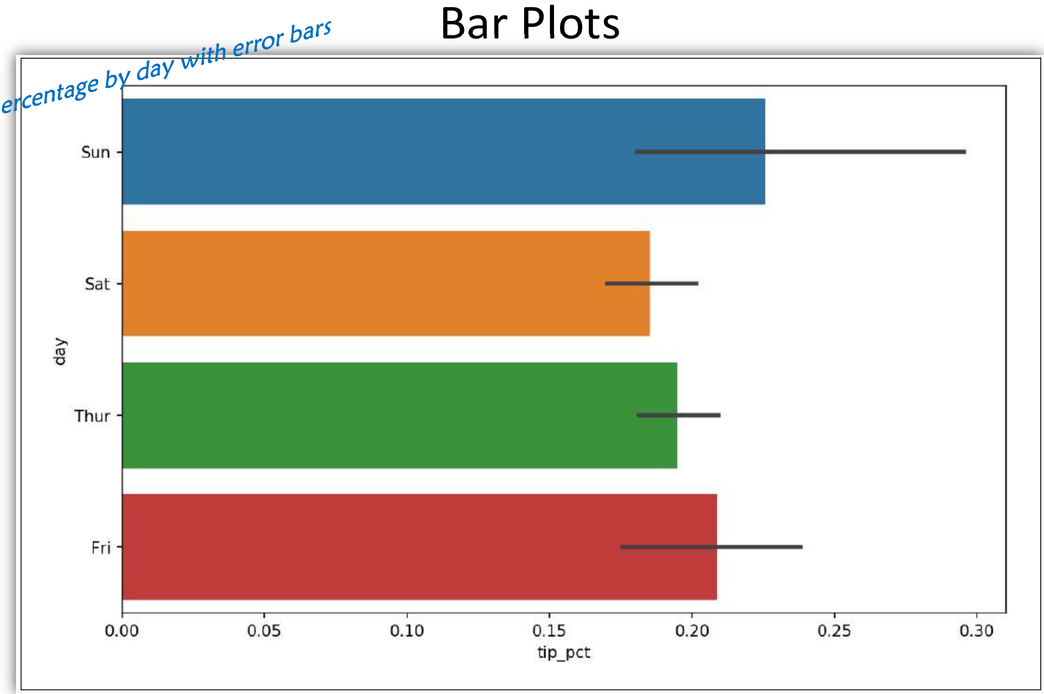


Bar Plots



```
In [83]: import seaborn as sns

In [84]: tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])

In [85]: tips.head()
Out[85]:
   total_bill   tip smoker  day    time  size   tip_pct
0       16.99  1.01     No  Sun  Dinner     2  0.063204
1       10.34  1.66     No  Sun  Dinner     3  0.191244
2       21.01  3.50     No  Sun  Dinner     3  0.199886
3       23.68  3.31     No  Sun  Dinner     2  0.162494
4       24.59  3.61     No  Sun  Dinner     4  0.172069

In [86]: sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

Bar Plots

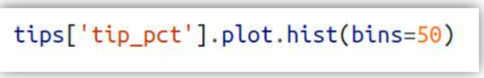


Tipping percentage by day with error bars

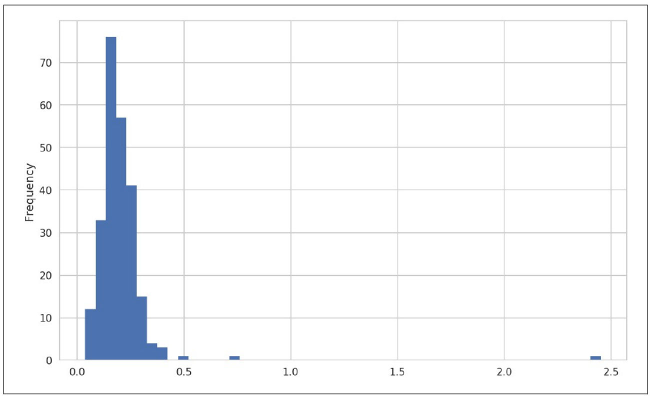
Histograms and Density Plots

> A histogram is a kind of bar plot that gives a discretized display of value frequency

> The data points are split into discrete, evenly spaced bins, and the number of data points in each bin is plotted

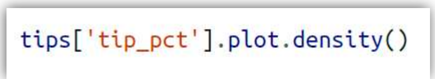


```
tips['tip_pct'].plot.hist(bins=50)
```



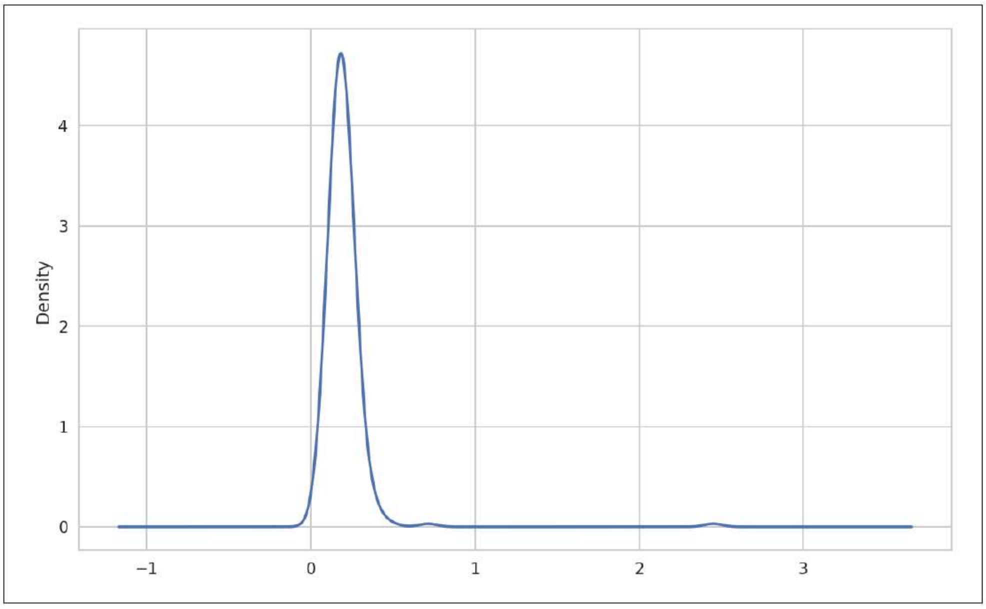
Histograms and Density Plots

- A related plot type is a *density plot*, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data
- The usual procedure is to approximate this distribution as a mixture of “kernels”
- Density plots are also known as kernel density estimate (KDE) plots



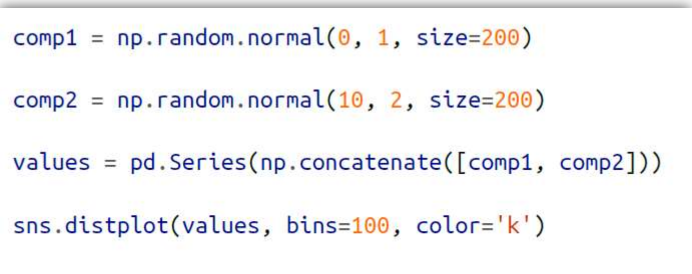
```
tips['tip_pct'].plot.density()
```

Histograms and Density Plots



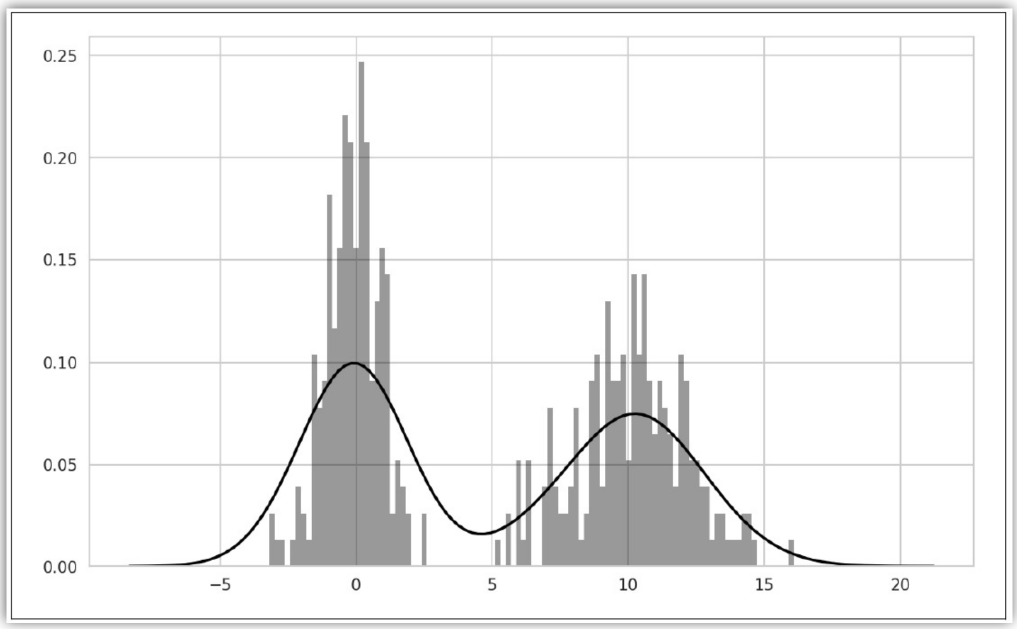
Histograms and Density Plots

- Seaborn makes histograms and density plots even easier through its **distplot** method, which can plot both a histogram and a continuous density estimate simultaneously
- Consider a bimodal distribution consisting of draws from two different standard normal distributions



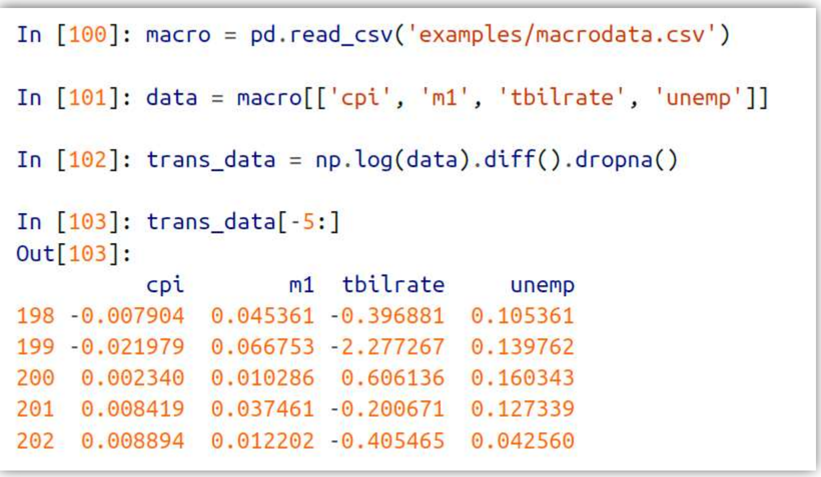
```
comp1 = np.random.normal(0, 1, size=200)

comp2 = np.random.normal(10, 2, size=200)

values = pd.Series(np.concatenate([comp1, comp2]))

sns.distplot(values, bins=100, color='k')
```

Histograms and Density Plots



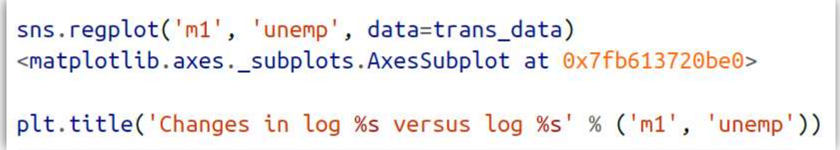
Scatter or Point Plots

> Point plots or scatter plots can be a useful way of examining the relationship between two one-dimensional data series.

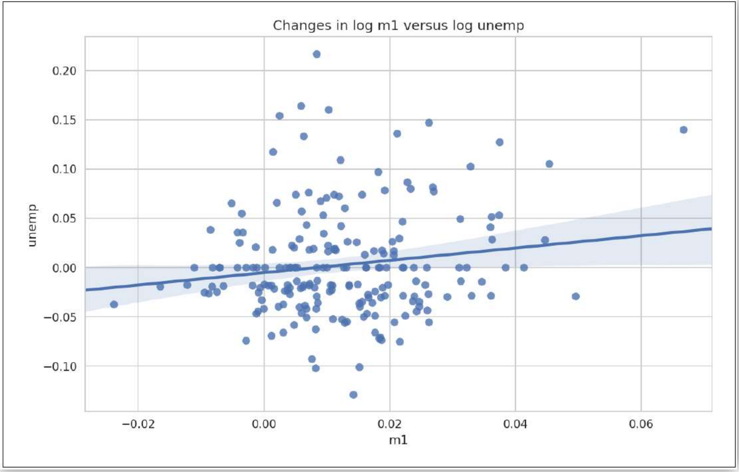


```
In [100]: macro = pd.read_csv('examples/macrodata.csv')

In [101]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]

In [102]: trans_data = np.log(data).diff().dropna()

In [103]: trans_data[-5:]
Out[103]:
          cpi        m1  tbilrate     unemp
198 -0.007904  0.045361 -0.396881  0.105361
199 -0.021979  0.066753 -2.277267  0.139762
200  0.002340  0.010286  0.606136  0.160343
201  0.008419  0.037461 -0.200671  0.127339
202  0.008894  0.012202 -0.405465  0.042560
```

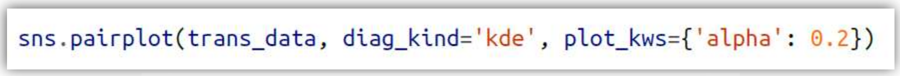
Scatter or Point Plots



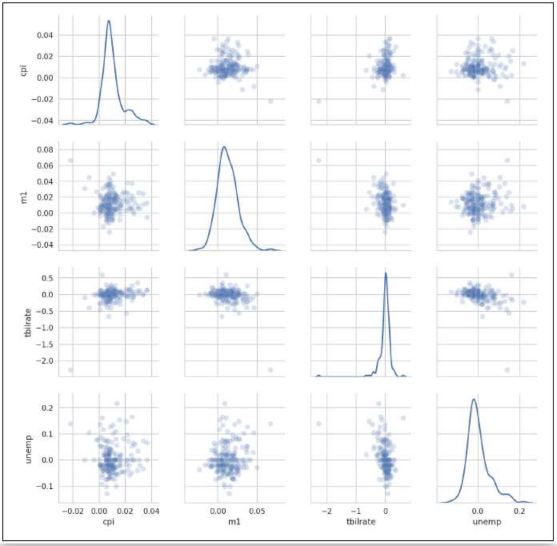
```
sns.regplot('m1', 'unemp', data=trans_data)
<matplotlib.axes._subplots.AxesSubplot at 0x7fb613720be0>

plt.title('Changes in log %s versus log %s' % ('m1', 'unemp'))
```



Scatter or Point Plots



```
sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha': 0.2})
```

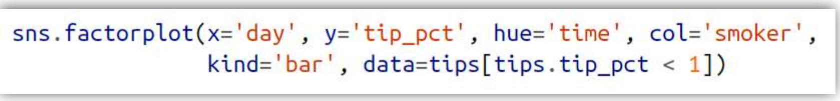


Facet Grids and Categorical Data

> What about datasets where we have additional grouping dimensions?

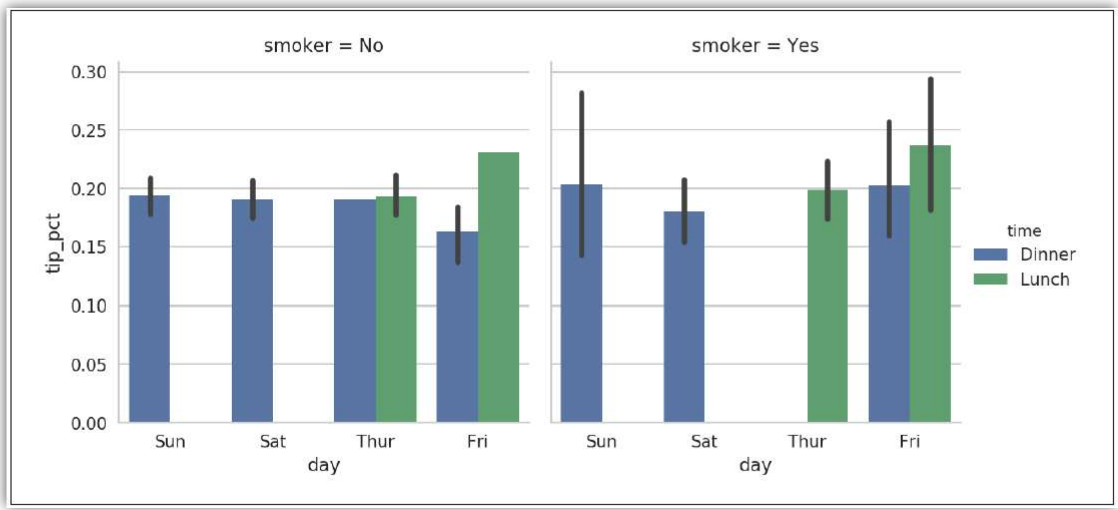
> One way to visualize data with many categorical variables is to use a ***facet grid***.

> Seaborn has a useful built-in function **factorplot** that simplifies making many kinds of faceted plots



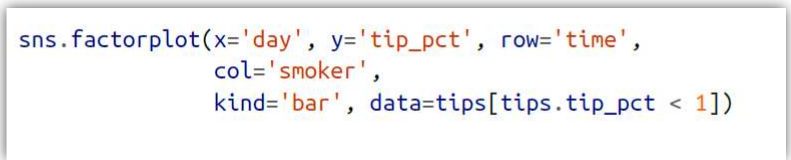
```
sns.factorplot(x='day', y='tip_pct', hue='time', col='smoker',
               kind='bar', data=tips[tips.tip_pct < 1])
```

Facet Grids and Categorical Data

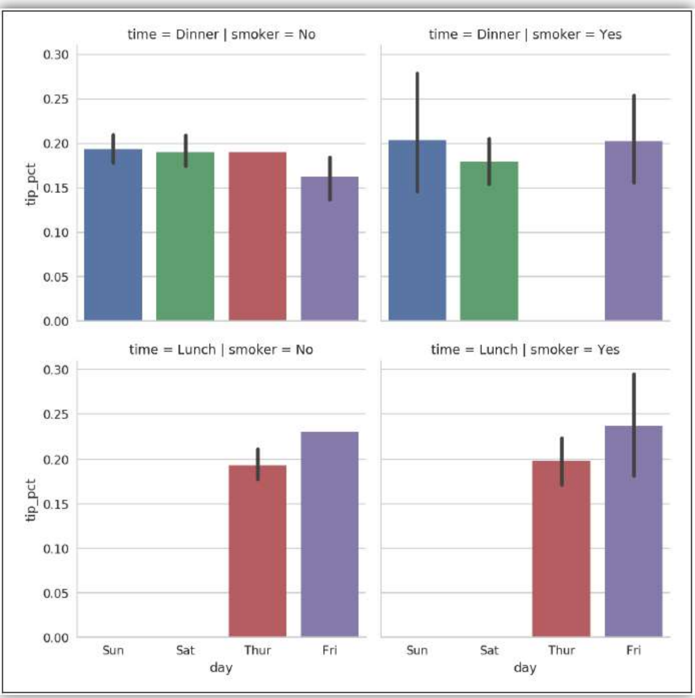


Facet Grids and Categorical Data

> Instead of grouping by **'time'** by different bar colors within a facet, we can also expand the facet grid by adding one row per **time** value



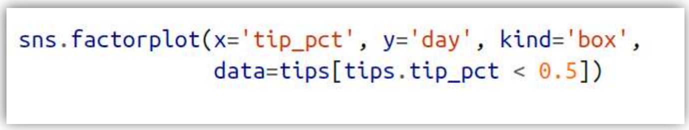
```
sns.factorplot(x='day', y='tip_pct', row='time',
               col='smoker',
               kind='bar', data=tips[tips.tip_pct < 1])
```



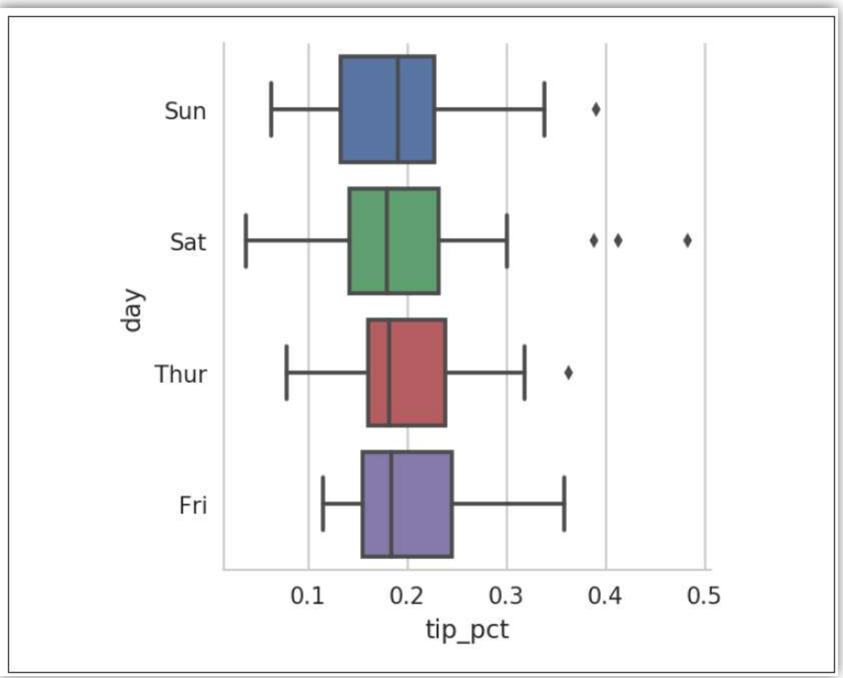
Facet Grids and Categorical Data

> **factorplot** supports other plot types that may be useful depending on what you are trying to display

> For example, box plots (which show the median, quartiles, and outliers) can be an effective visualization type



```
sns.factorplot(x='tip_pct', y='day', kind='box',
               data=tips[tips.tip_pct < 0.5])
```



`Series.plot` method arguments

Argument	Description
`label`	Label for plot legend
`ax`	matplotlib subplot object to plot on; if nothing passed, uses active matplotlib subplot
`style`	Style string, like 'ko--', to be passed to matplotlib
`alpha`	The plot fill opacity (from 0 to 1)
`kind`	Can be 'area', 'bar', 'barh', 'density', 'hist', 'kde', 'line', 'pie'
`logy`	Use logarithmic scaling on the y-axis
`use_index`	Use the object index for tick labels
`rot`	Rotation of tick labels (0 through 360)
`xticks`	Values to use for x-axis ticks
`yticks`	Values to use for y-axis ticks
`xlim`	x-axis limits (e.g., [0, 10])
`ylim`	y-axis limits
`grid`	Display axis grid (on by default)

DataFrame-specific plot arguments

Argument	Plot each DataFrame column in a separate subplot
`subplots`	Plot each DataFrame column in a separate subplot
`sharex`	If **`subplots=True`**, share the same x-axis, linking ticks and limits
`sharey`	If **`subplots=True`**, share the same y-axis
`figsize`	Size of figure to create as tuple
`title`	Plot title as string
`legend`	Add a subplot legend (**`True`** by default)
`sort_columns`	Plot columns in alphabetical order; by default uses existing column order