Plotting may be a part of the exploratory process—for example, to help identify outliers or needed data transformations.

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

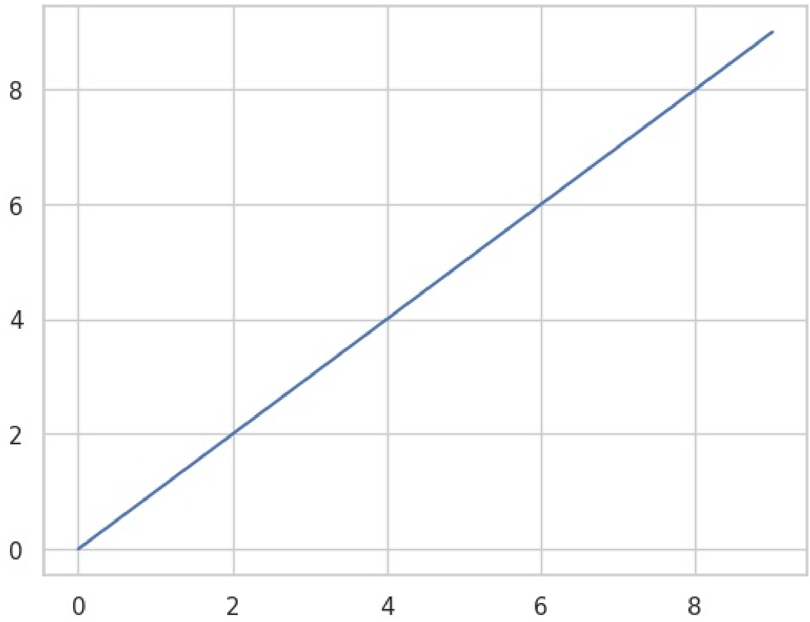
The project was started by John Hunter in 2002 to enable a MATLAB-like plotting interface in Python.

In [51]:

```python
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
data=np.arange(10)
plt.plot(data) #x=[0,1,2,...,9] and y=[0,1,2,...,9]
```

Out[51]:

```
[<matplotlib.lines.Line2D at 0x74da50437b20>]
```



Plots in matplotlib reside within a Figure object. You can create a new figure with `plt.figure`.

`plt.figure` has a number of options; notably, figsize will guarantee the figure has a certain size and aspect ratio if saved to disk.

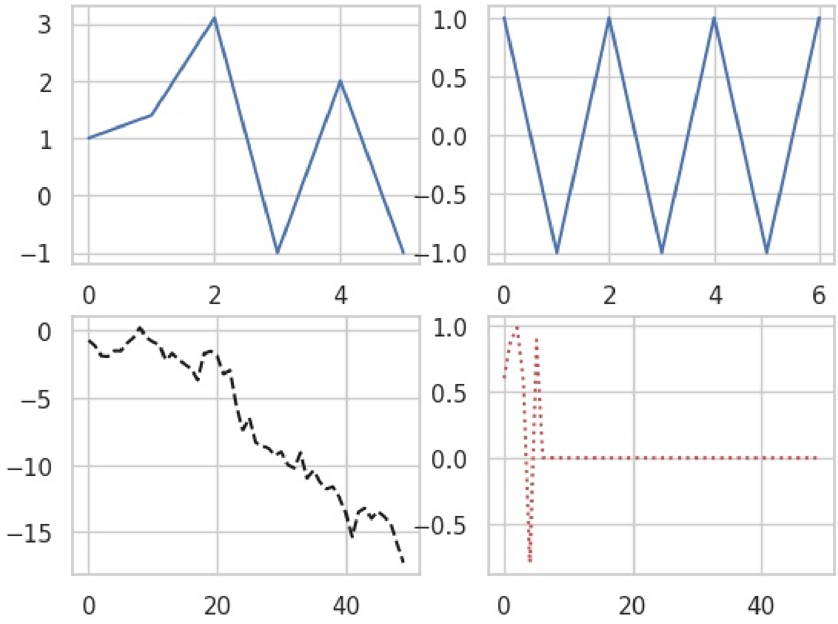
You have to create one or more subplots using add_subplot:

One nuance of using Jupyter notebooks is that plots are reset after each cell is evaluated, so for more complex plots you must put all of the plotting commands in a single notebook cell.

When you issue a plotting command like `plt.plot([1.5, 3.5, -2, 1.6])`, matplotlib draws on the last figure and subplot used (creating one if necessary), thus hiding the figure and subplot creation.

In [52]:

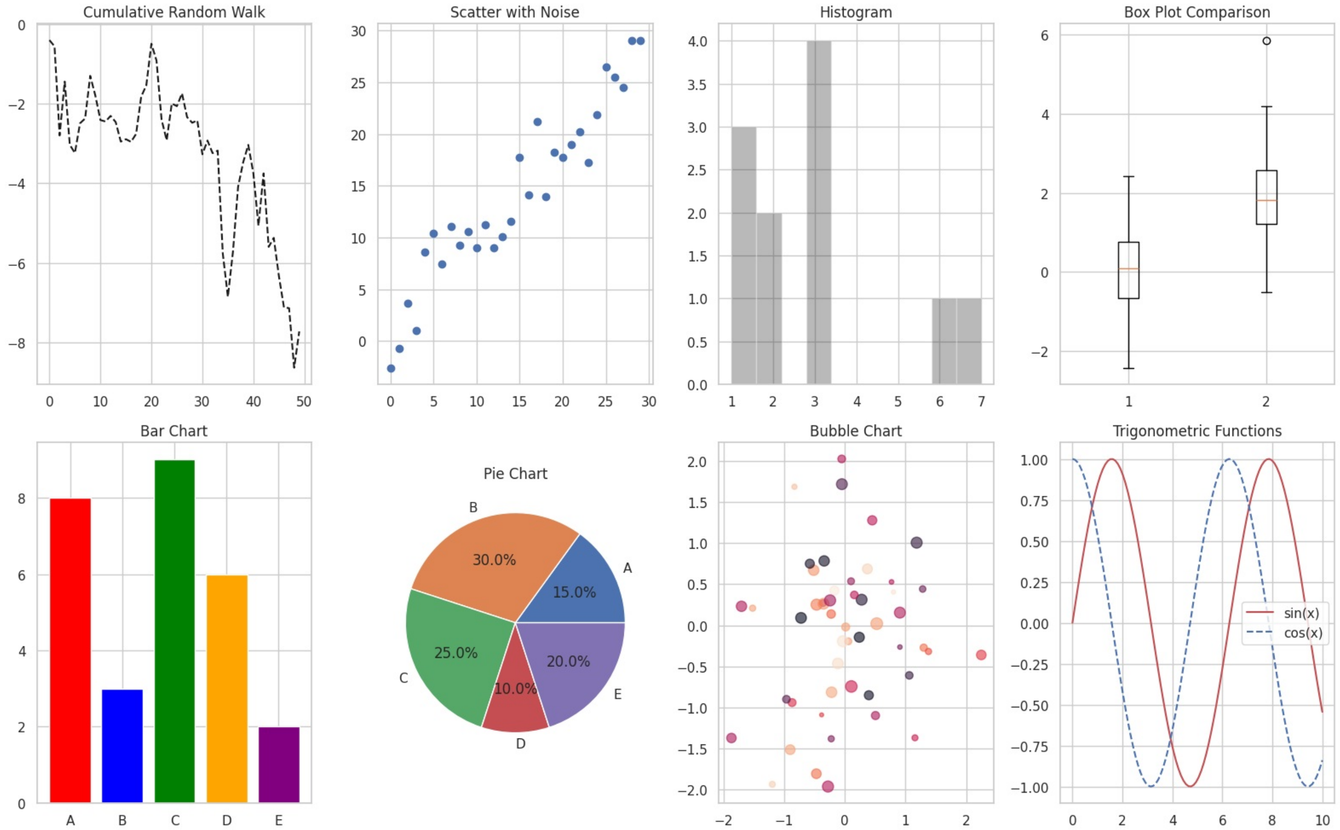
```python
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
ax4 = fig.add_subplot(2, 2, 4)

# Plot on specific subplots instead of the current active one
ax1.plot([1, 1.4, 3.1, -1, 2, -1])
ax2.plot([1, -1, 1, -1, 1, -1, 1])
ax3.plot(np.random.randn(50).cumsum(), 'k--')
ax4.plot(np.random.randn(50).cumprod(),'r:')
plt.show()
```



The 'k--' is a style option instructing matplotlib to plot a black dashed line. The objects returned by fig.add_subplot here are AxesSubplot objects, on which you can directly plot on the other empty subplots by calling each one's instance method

In [53]:

```
import matplotlib.pyplot as plt
import numpy as np

# Increase the figure size significantly
fig = plt.figure(figsize=(16, 10))

# Create 2x4 grid
ax1 = fig.add_subplot(2, 4, 1)
ax2 = fig.add_subplot(2, 4, 2)
ax3 = fig.add_subplot(2, 4, 3)
ax4 = fig.add_subplot(2, 4, 4)
ax5 = fig.add_subplot(2, 4, 5)
ax6 = fig.add_subplot(2, 4, 6)
ax7 = fig.add_subplot(2, 4, 7)
ax8 = fig.add_subplot(2, 4, 8)

# Add plots with titles
ax1.plot(np.random.randn(50).cumsum(), 'k--')
ax1.set_title('Cumulative Random Walk')

ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
ax2.set_title('Scatter with Noise')

ax3.hist([1,2,1,1,3,3,3,3,2,6,7], bins=10, color='k', alpha=0.3)
ax3.set_title('Histogram')

ax4.boxplot([np.random.randn(100), np.random.randn(100) + 2])
ax4.set_title('Box Plot Comparison')

categories = ['A', 'B', 'C', 'D', 'E']
values = np.random.randint(1, 10, size=5)
ax5.bar(categories, values, color=['red', 'blue', 'green', 'orange', 'purple'])
ax5.set_title('Bar Chart')

sizes = [15, 30, 25, 10, 20]
labels = ['A', 'B', 'C', 'D', 'E']
ax6.pie(sizes, labels=labels, autopct='%1.1f%%')
ax6.set_title('Pie Chart')

x = np.random.randn(50)
y = np.random.randn(50)
colors = np.random.rand(50)
sizes = 100 * np.random.rand(50)
ax7.scatter(x, y, c=colors, s=sizes, alpha=0.6)
ax7.set_title('Bubble Chart')

x = np.linspace(0, 10, 100)
ax8.plot(x, np.sin(x), 'r-', label='sin(x)')
ax8.plot(x, np.cos(x), 'b--', label='cos(x)')
ax8.legend()
ax8.set_title('Trigonometric Functions')

# Adjust layout to prevent overlapping
plt.tight_layout()
plt.show()
```



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.

This is very useful, as the axes array can be easily indexed like a two-dimensional array; for example, axes[0, 1]. You can also indicate that subplots should have the same x- or y-axis using sharex and sharey, respectively.

This is especially useful when you're comparing data on the same scale; otherwise, matplotlib autoscales plot limits independently.

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))

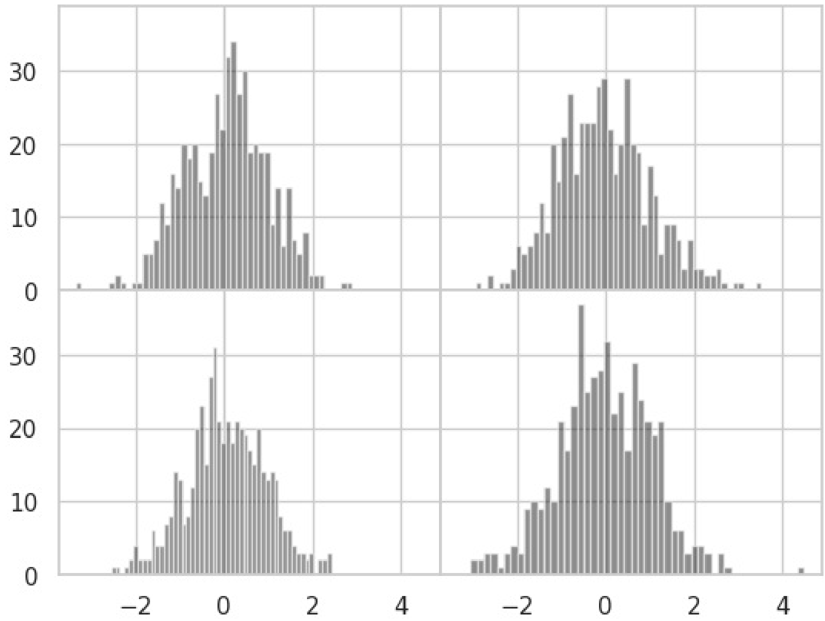
By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots.

This spacing is all specified relative to the height and width of the plot, so that 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, also available as a top-level function:

```
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.

In [54]:
```
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)
```



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--')
```

This way of specifying both color and line style in a string is provided as a convenience; in practice if you were creating plots programmatically

you might prefer not to have to munge strings together to create plots with the desired style.

The same plot could also have been expressed more explicitly as:

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

There are a number of color abbreviations provided for commonly used colors, but you can use any color on the spectrum by specifying its hex code (e.g., '#CECECE').

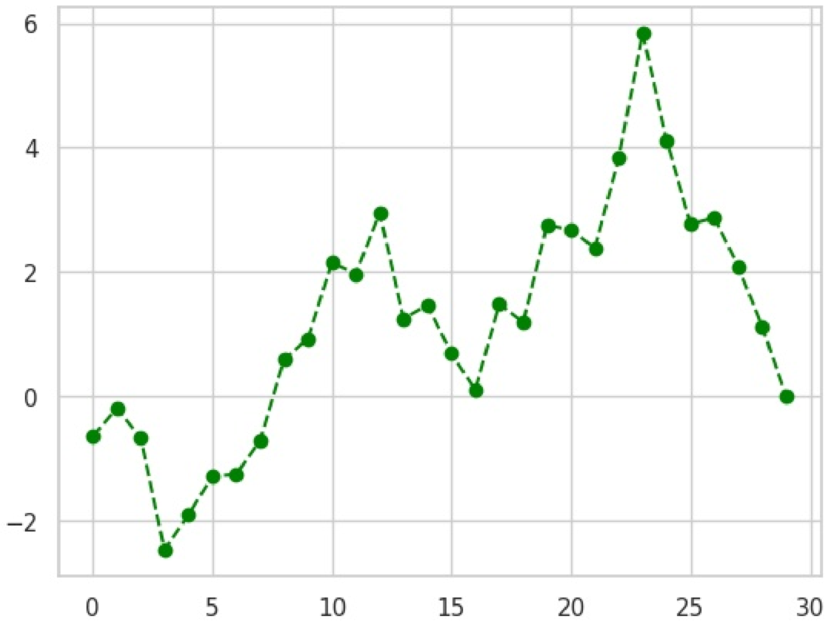
You can see the full set of line styles by looking at the docstring for plot (use plot? in IPython or Jupyter).

```
plt?
```

Line plots can additionally have markers to highlight the actual data points. Since matplotlib creates a continuous line plot, interpolating between points, it can occasionally be unclear where the points lie. The marker can be part of the style string, which must have color followed by marker type and line style

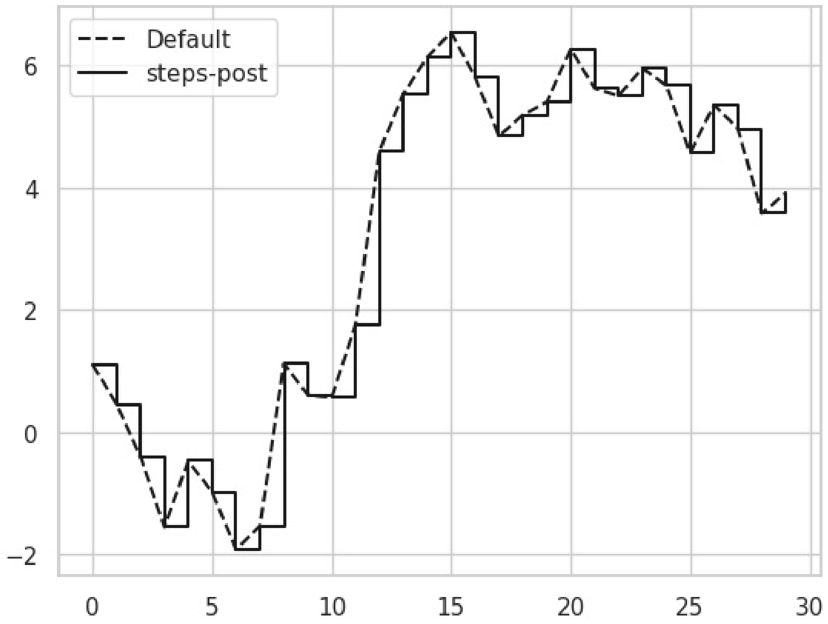
In [55]:
```
fig=plt.figure()
plt.plot(np.random.randn(30).cumsum(), color='green', linestyle='dashed', marker='o')
```

Out[55]:
```
[<matplotlib.lines.Line2D at 0x74da521f1840>]
```



For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the drawstyle option

In [56]:
```
data = np.random.randn(30).cumsum()
fig=plt.figure()
plt.plot(data, 'k--', label='Default')
plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
plt.legend(loc='best')
```
Out[56]:
```
<matplotlib.legend.Legend at 0x74da5040e200>
```



You must call `plt.legend` (or `ax.legend`, if you have a reference to the axes) to create the legend, whether or not you passed the label options when plotting the data.

For most kinds of plot decorations, there are two main ways to do things: using the procedural pyplot interface (i.e., matplotlib.pyplot) and the more object-oriented native matplotlib API.

The pyplot interface, designed for interactive use, consists of methods like xlim, xticks, and xticklabels. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways

• Called with no arguments returns the current parameter value (e.g., plt.xlim() returns the current x-axis plotting range)

• Called with parameters sets the parameter value (e.g., plt.xlim([0, 10]), sets the x-axis range to 0 to 10)

All such methods act on the active or most recently created AxesSubplot. Each of them corresponds to two methods on the subplot object itself; in the case of xlim these are ax.get_xlim and ax.set_xlim.

I prefer to use the subplot instance methods myself in the interest of being explicit (and especially when working with multiple subplots), but you can certainly use whichever you find more convenient.

Setting the title, axis labels, ticks, and ticklabels

To change the x-axis ticks, it’s easiest to use set_xticks and set_xticklabels. The former instructs matplotlib where to place the ticks along the data range; by default these locations will also be the labels. But we can set any other values as the labels using set_xticklabels:

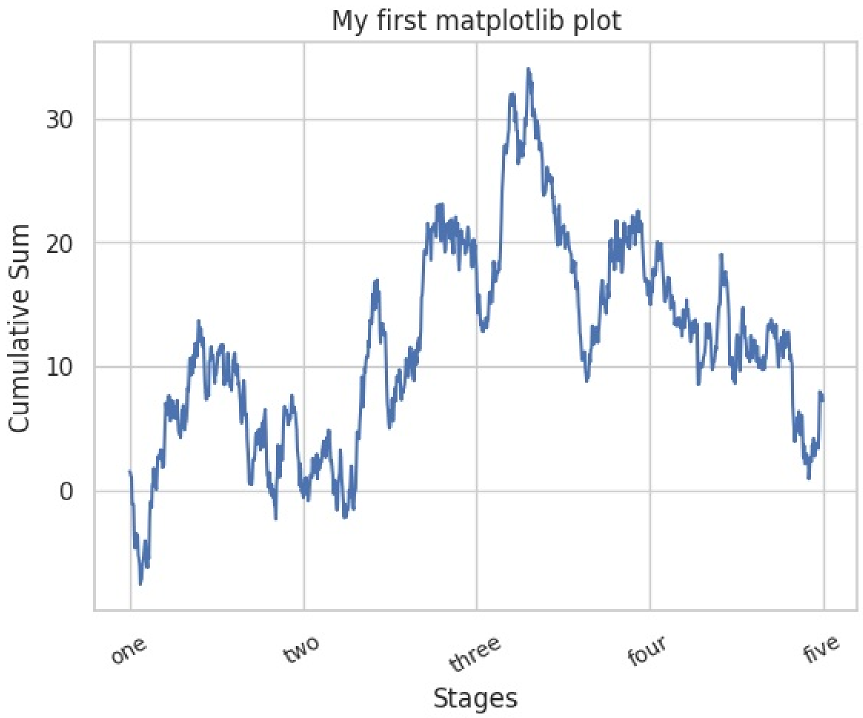
The rotation option sets the x tick labels at a 30-degree rotation. Lastly, set_xlabel gives a name to the x-axis and set_title the subplot title

In [57]:
```
fig=plt.figure()
```

```
ax=fig.add_subplot(1,1,1)
ax.plot(np.random.randn(1000).cumsum())
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')
ax.set_xlabel('Stages')
ax.set_ylabel('Cumulative Sum')
```

Out[57]:

```
Text(0, 0.5, 'Cumulative Sum')
```

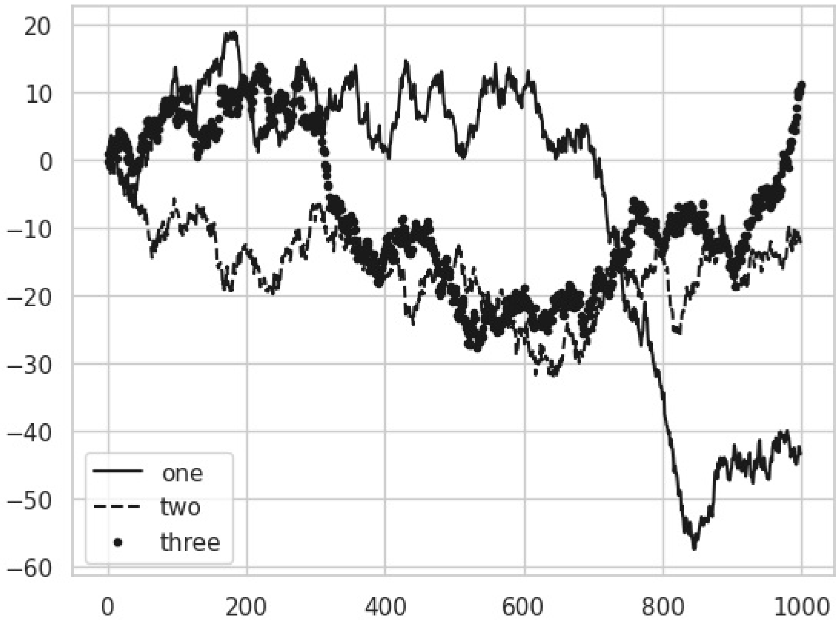


In [58]:

```
from numpy.random import randn
fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
ax.plot(randn(1000).cumsum(), 'k', label='one')
ax.plot(randn(1000).cumsum(), 'k--', label='two')
ax.plot(randn(1000).cumsum(), 'k.', label='three')
ax.legend(loc='best')
```

Out[58]:

```
<matplotlib.legend.Legend at 0x74da500b7010>
```



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)
```

In [59]:

```
import pandas as pd
from datetime import datetime
```

```python
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
data = pd.read_csv('spx.csv', index_col=0, parse_dates=True)
spx = data['Open']
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')
]
for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75),xytext=(date, spx.asof(date) + 225),arrowprops=dict(facec
    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')


# There are a couple of important points to highlight in this plot: the ax.annotate method can draw labels at t
# We use the set_xlim and set_ylim methods to manually set the start and end boundaries for the plot rather tha
# Lastly, ax.set_title adds a main title to the plot.
```

Out[59]:

```
Text(0.5, 1.0, 'Important dates in the 2008-2009 financial crisis')
```

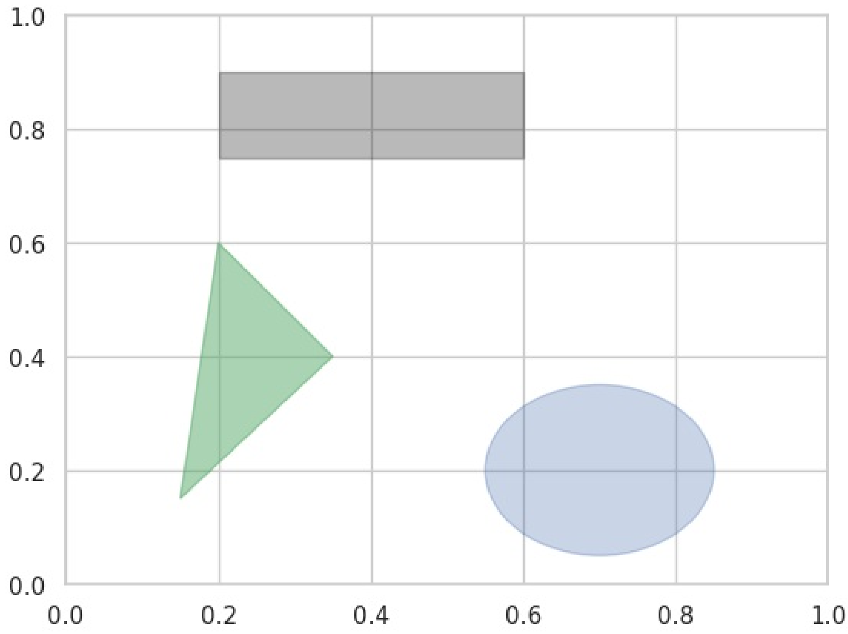
Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as patches.

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

To add a shape to a plot, you create the patch object shp and add it to a subplot by calling ax.add_patch(shp)

In [60]:

```python
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)
poly=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(poly)
plt.savefig('shapes.png',dpi=400,bbox_inches='tight')
plt.savefig('shapes.pdf',dpi=400,bbox_inches='tight')
```



You can save the active figure to file using plt.savefig. This method is equivalent to the figure object's savefig instance method.

The file type is inferred from the file extension. So if you used .pdf instead, you would get a PDF. There are a couple of important options that I use frequently for publishing graphics: dpi, which controls the dots-per-inch resolution, and bbox_inches, which can trim the whitespace around the actual figure.

To get the same plot as a PNG with minimal whitespace around the plot and at 400 DPI, refer the above code.

savefig doesn't have to write to disk; it can also write to any file-like object, such as a BytesIO

In [61]:
```
from io import BytesIO
buffer = BytesIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

```
<Figure size 640x480 with 0 Axes>
```

matplotlib Configuration

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 keyword arguments indicating the new parameters. An easy way to write down the options in your program is as a dict:

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

For more extensive customization and to see a list of all the options, matplotlib comes with a configuration file matplotlibrc in the matplotlib/mpl-data directory.

If you customize this file and place it in your home directory titled .matplotlibrc, it will be loaded each time you use matplotlib

Plotting with pandas and seaborn

matplotlib can be a fairly low-level tool. You assemble a plot from its base components: the data display (i.e., the type of plot: line, bar, box, scatter, contour, etc.),legend, title, tick labels, and other annotations.

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 DataFrame and Series objects.

Another library is seaborn, a statistical graphics library created by Michael Waskom. Seaborn simplifies creating many common visualization types.

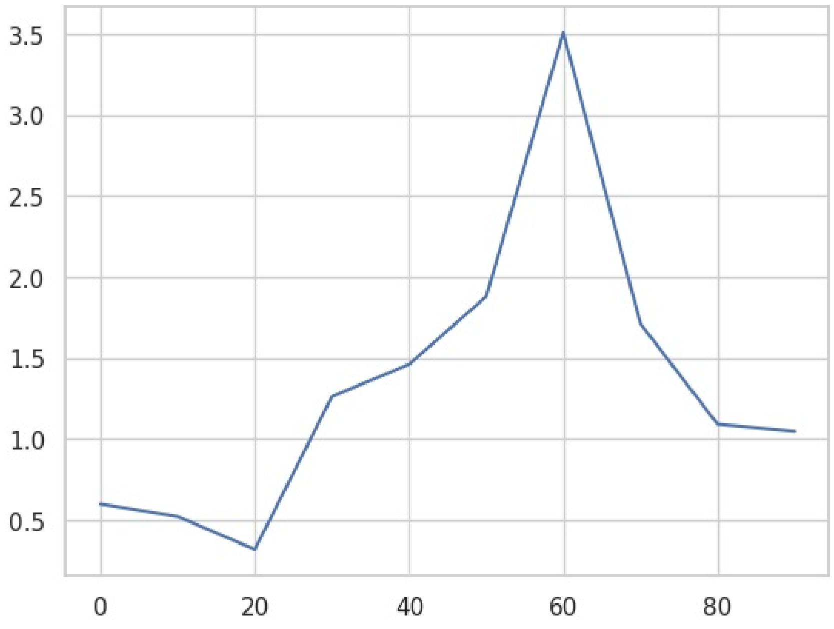
Importing seaborn modifies the default matplotlib color schemes and plot styles to improve readability and aesthetics. Even if you do not use the seaborn API, you may prefer to import seaborn as a simple way to improve the visual aesthetics of general matplotlib plots

Line Plots

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

In [62]:
```
fig = plt.figure()
df = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))
df.plot()
```

Out[62]: <Axes: >

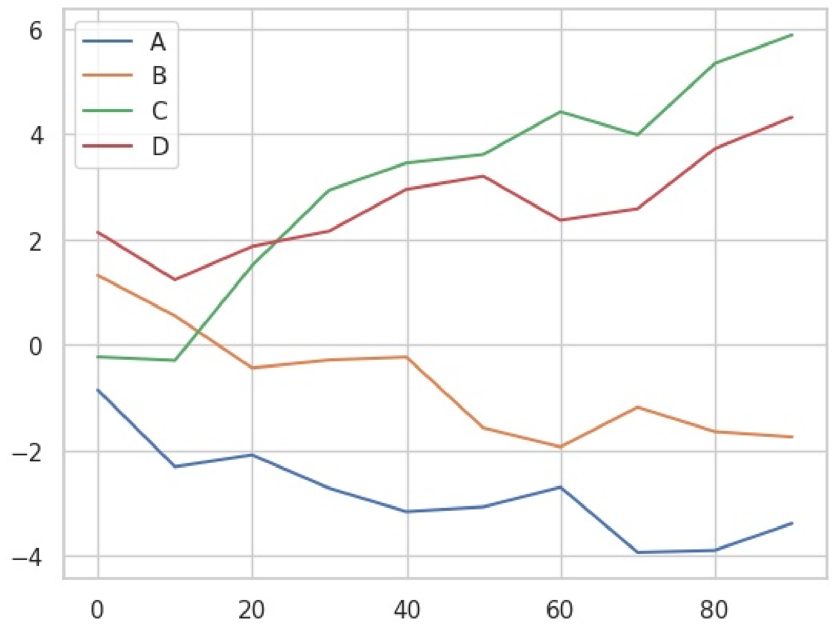


The Series object's index is passed to matplotlib for plotting on the x-axis, though you can disable this by passing use_index=False. The x-axis ticks and limits can be adjusted with the xticks and xlim options, and y-axis respectively with yticks and ylim.

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

In [63]:
```
df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),columns=['A', 'B', 'C', 'D'],index=np.arange(0, 100, 10))
df.plot()
```

Out[63]: <Axes: >



The plot attribute contains a "family" of methods for different plot types. For example, df.plot() is equivalent to df.plot.line().

Additional keyword arguments to plot are passed through to the respective matplotlib plotting function, so you can further customize these plots by learning more about the matplotlib API.

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 --> Description

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 defaul it uses existing column order

Bar Plots

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

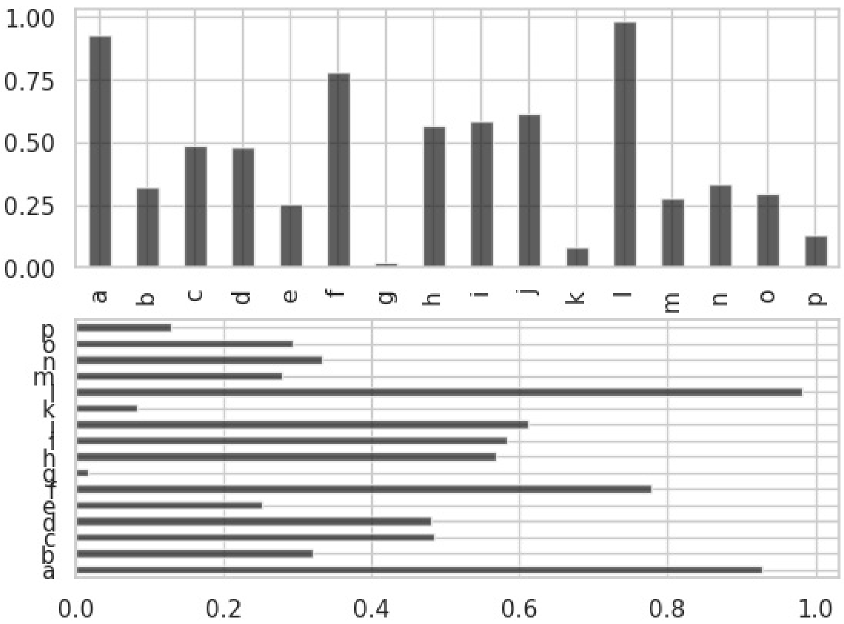
In this case, the Series or DataFrame index will be used as the x (bar) or y (barh) ticks

In [64]:

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

Out[64]:

```
<Axes: >
```

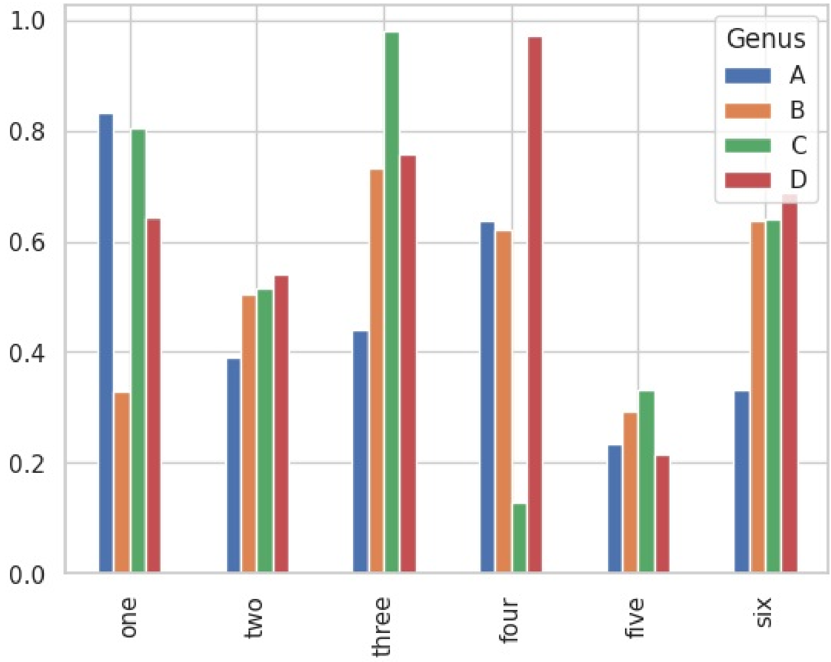


The options color='k' and alpha=0.7 set the color of the plots to black and use partial transparency on the filling.

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

In [65]:
```
df = pd.DataFrame(np.random.rand(6, 4),index=['one', 'two', 'three', 'four', 'five', 'six'],\
                  columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
df.plot.bar()
```

Out[65]:
```
<Axes: >
```

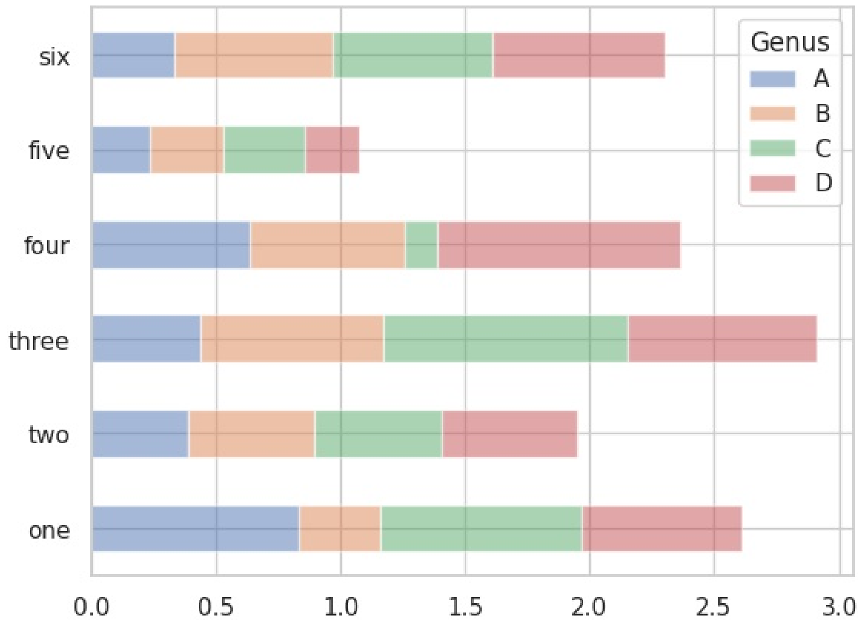


Note that the name “Genus” on the DataFrame’s columns is used to title the legend.

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

In [66]:
```
df.plot.barh(stacked=True, alpha=0.5)
```

Out[66]:
```
<Axes: >
```



A useful recipe for bar plots is to visualize a Series's value frequency using value_counts: `s.value_counts().plot.bar()` .

Returning to the tipping dataset used earlier in the book, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day.

I load the data using read_csv and make a cross-tabulation by day and party size

In [67]:
```
tips = pd.read_csv('tips.csv')
party_counts = pd.crosstab(tips['day'], tips['size'])
print(party_counts)
```

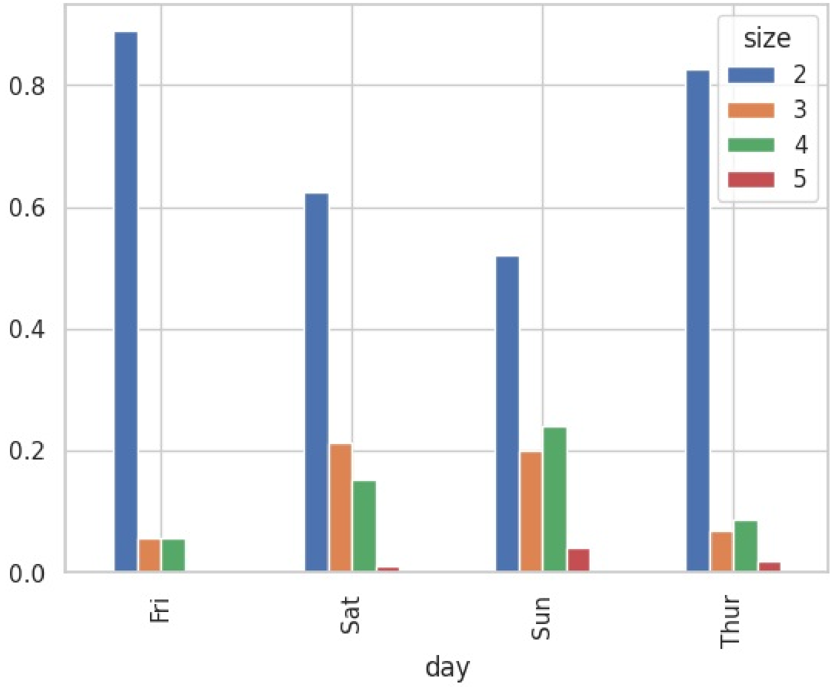
```
size  1   2   3   4  5  6
day
Fri   1  16   1   1  0  0
Sat   2  53  18  13  1  0
Sun   0  39  15  18  3  1
Thur  1  48   4   5  1  3
```

In [68]:
```
party_counts = party_counts.loc[:, 2:5]
party_pcts = party_counts.div(party_counts.sum(1), axis=0)
print(party_pcts)
```

```
size         2         3         4         5
day
Fri   0.888889  0.055556  0.055556  0.000000
Sat   0.623529  0.211765  0.152941  0.011765
Sun   0.520000  0.200000  0.240000  0.040000
Thur  0.827586  0.068966  0.086207  0.017241
```

In [69]:
```
party_pcts.plot.bar()
```

Out[69]:
```
<Axes: xlabel='day'>
```



So you can see that party sizes appear to increase on the weekend in this dataset.

With data that requires aggregation or summarization before making a plot, using the seaborn package can make things much simpler.

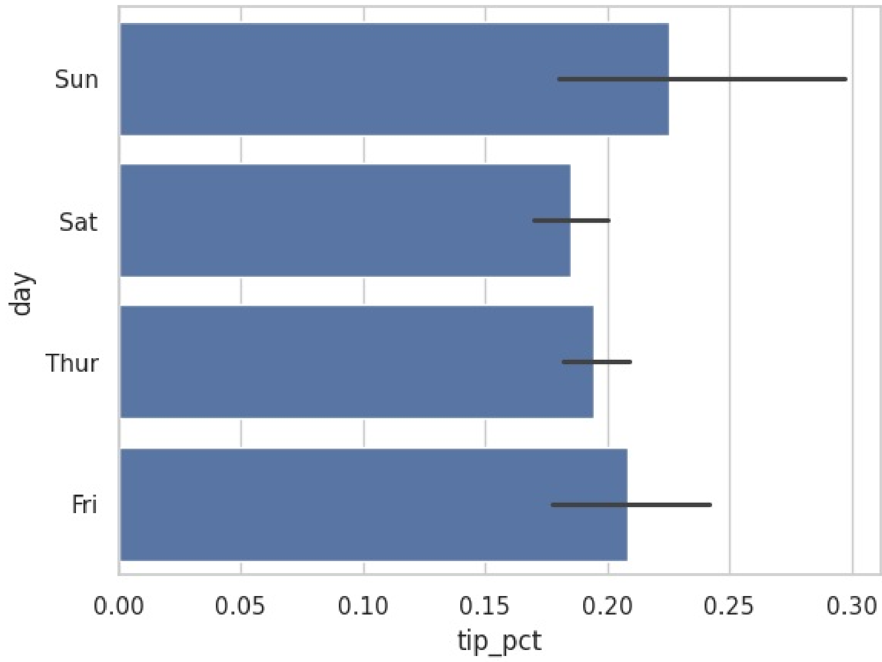
Let's look now at the tipping percentage by day with seaborn

In [70]:
```
import seaborn as sns
tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])
print(tips.head())
```

```
   total_bill   tip     sex smoker  day    time  size   tip_pct
0       16.99  1.01  Female     No  Sun  Dinner     2  0.063204
1       10.34  1.66    Male     No  Sun  Dinner     3  0.191244
2       21.01  3.50    Male     No  Sun  Dinner     3  0.199886
3       23.68  3.31    Male     No  Sun  Dinner     2  0.162494
4       24.59  3.61  Female     No  Sun  Dinner     4  0.172069
```

In [71]:
```
fig=plt.figure()
sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

Out[71]:
```
<Axes: xlabel='tip_pct', ylabel='day'>
```



Plotting functions in seaborn take a data argument, which can be a pandas DataFrame. The other arguments refer to column names.

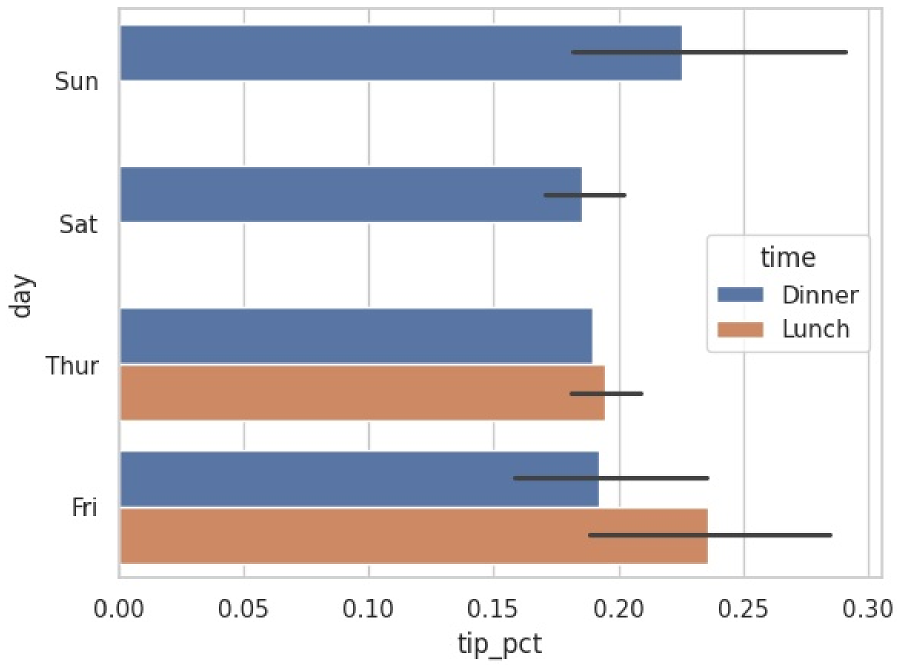
Because there are multiple observations for each value in the day, the bars are the average value of tip_pct.

The black lines drawn on the bars represent the 95% confidence interval (this can be configured through optional arguments).

seaborn.barplot has a hue option that enables us to split by an additional categorical value

In [72]:
```
fig=plt.figure()
sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

Out[72]: <Axes: xlabel='tip_pct', ylabel='day'>



Notice that seaborn has automatically changed the aesthetics of plots: the default color palette, plot background, and grid line colors.

You can switch between different plot appearances using seaborn.set `sns.set(style="whitegrid")`

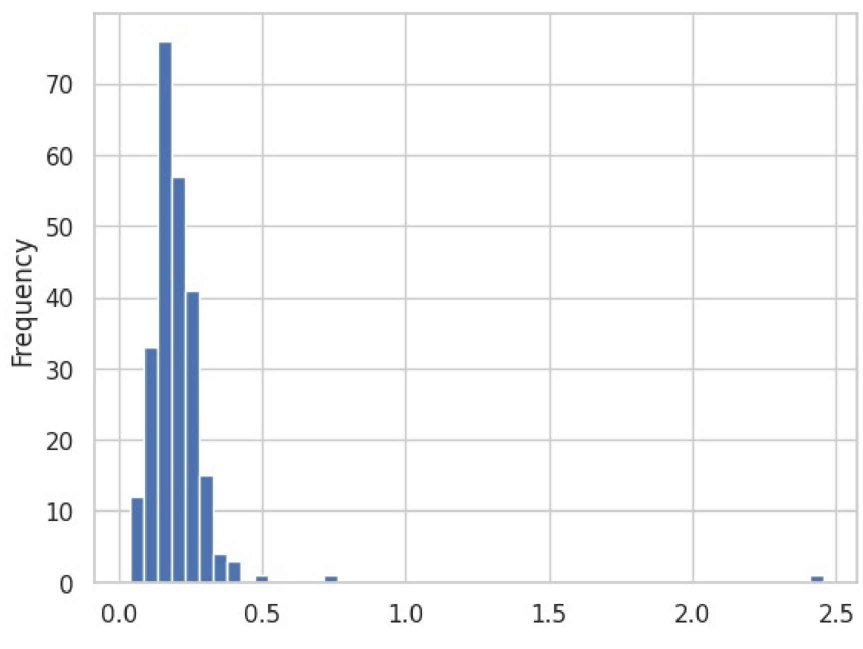
Histograms and Density Plots

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.

Using the tipping data from before, we can make a histogram of tip percentages of the total bill using the `plot.hist` method on the Series

In [73]:
```
fig=plt.figure()
tips['tip_pct'].plot.hist(bins=50)
```

Out[73]: <Axes: ylabel='Frequency'>

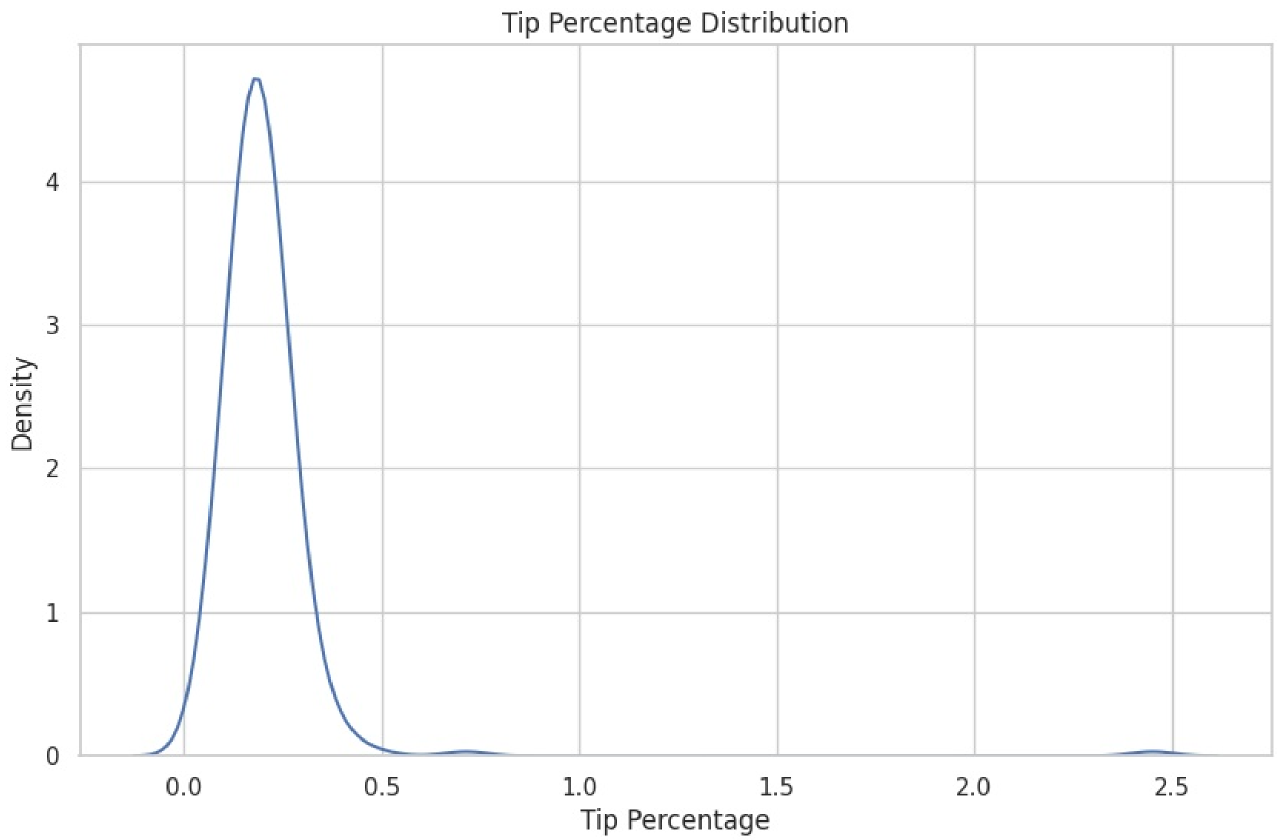


A related plot type is a density plot, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data.

The usual procedure is to approximate this distribution as a mixture of “kernels”—that is, simpler distributions like the normal distribution.

Thus, density plots are also known as kernel density estimate (KDE) plots. Using plot.kde makes a density plot using the conventional mixture-of-normals estimate

In [75]:
```python
import seaborn as sns
import matplotlib.pyplot as plt

# Seaborn handles KDE plotting well and has fewer dependency issues
plt.figure(figsize=(10, 6))
sns.kdeplot(data=tips['tip_pct'])
plt.title('Tip Percentage Distribution')
plt.xlabel('Tip Percentage')
plt.ylabel('Density')
plt.show()
```



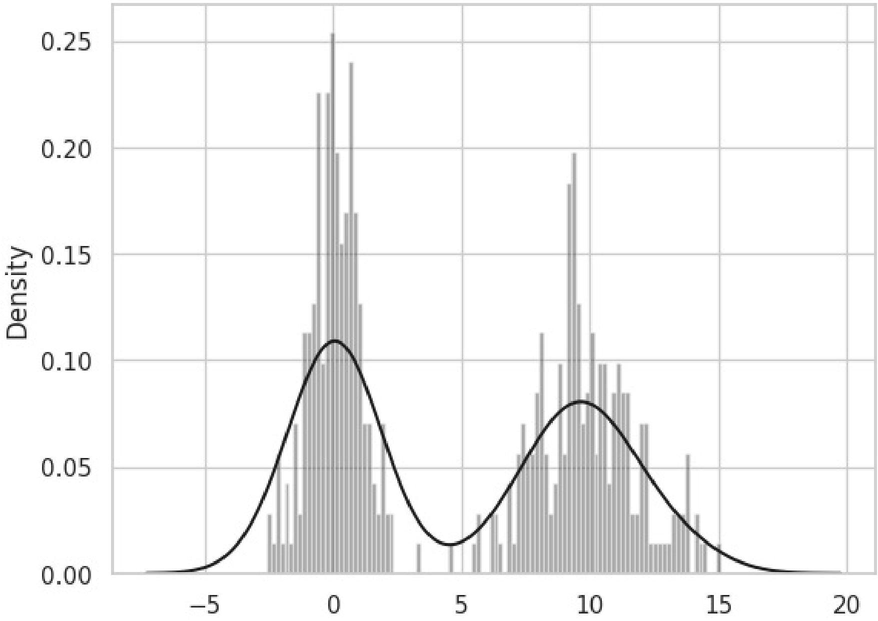
Seaborn makes histograms and density plots even easier through its distplot method, which can plot both a histogram and a continuous density estimate simultaneously.

As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions

In [79]:
```python
fig=plt.figure()
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')
```

```
/tmp/ipykernel_15881/3472363609.py:5: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

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

Out[79]:
```
<Axes: ylabel='Density'>
```



Scatter or Point Plots

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

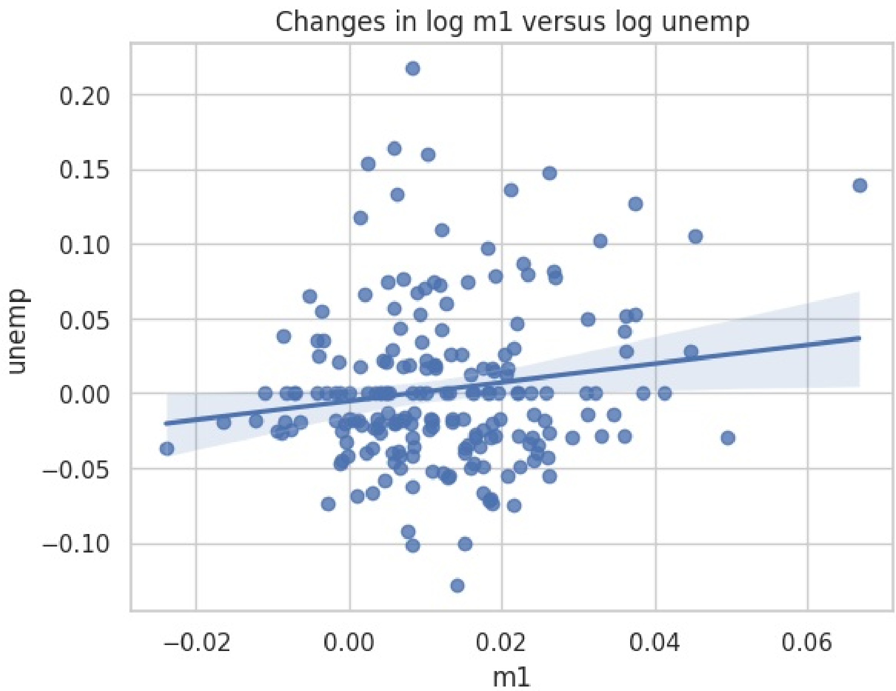
In [80]:
```
macro = pd.read_csv('macrodata.csv')
data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]
trans_data = np.log(data).diff().dropna()
print(trans_data[-5:])
```

```
          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
```

We can then use seaborn's regplot method, which makes a scatter plot and fits a linear regression line

In [81]:
```
fig=plt.figure()
sns.regplot(x='m1',y='unemp', data=trans_data)
plt.title('Changes in log %s versus log %s' % ('m1', 'unemp'))
```

Out[81]:
```
Text(0.5, 1.0, 'Changes in log m1 versus log unemp')
```

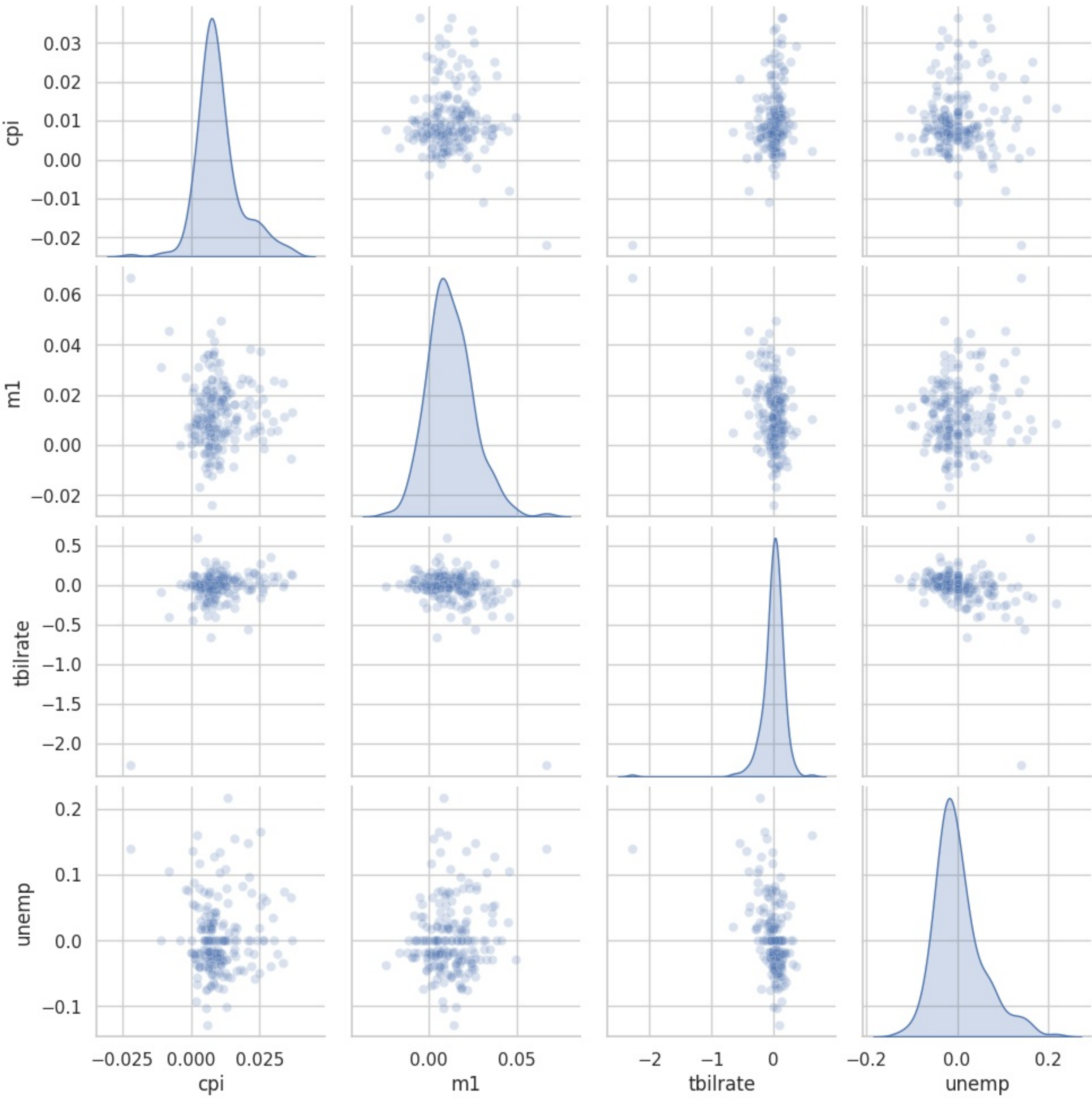


In exploratory data analysis it's helpful to be able to look at all the scatter plots among a group of variables; this is known as a pairs plot or scatter plot matrix.

Making such a plot from scratch is a bit of work, so seaborn has a convenient pairplot function, which supports placing histograms or density estimates of each variable along the diagonal

In [82]:
```
sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha': 0.2})
```

Out[82]: <seaborn.axisgrid.PairGrid at 0x74da4b116dd0>



You may notice the plot_kws argument. This enables us to pass down configuration options to the individual plotting calls on the off-diagonal elements.

Check out the seaborn.pairplot docstring for more granular configuration options

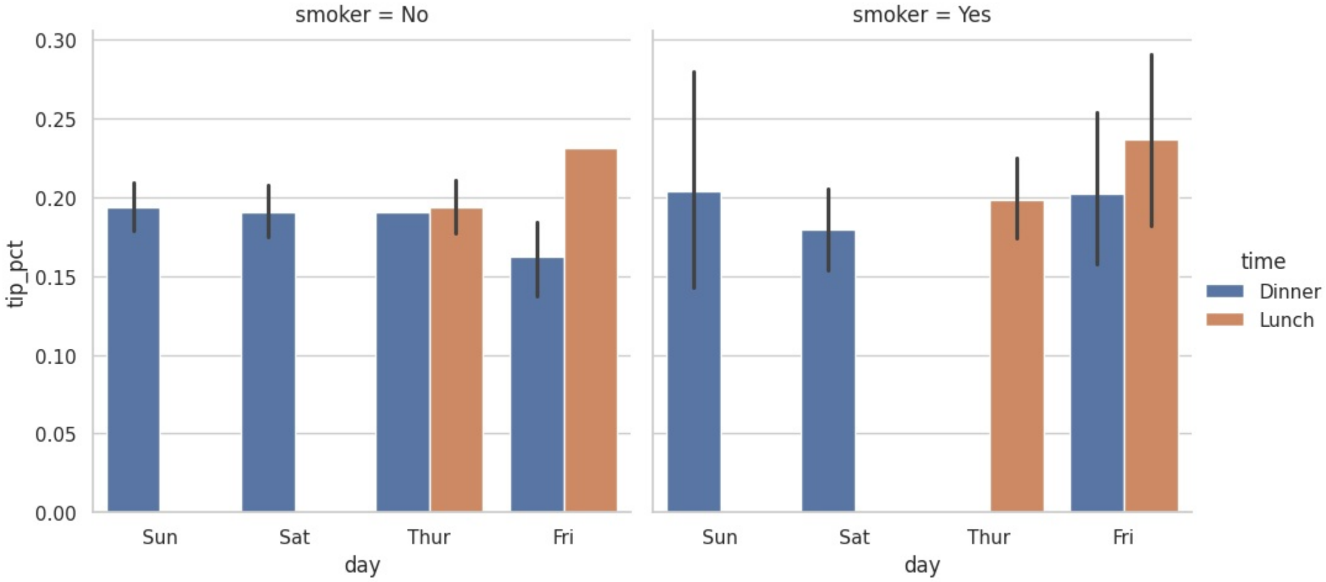
Facet Grids and Categorical Data

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 catplot that simplifies making many kinds of faceted plots

In [83]:
```
sns.catplot(x='day', y='tip_pct', hue='time', col='smoker',kind='bar', data=tips[tips.tip_pct < 1])
```

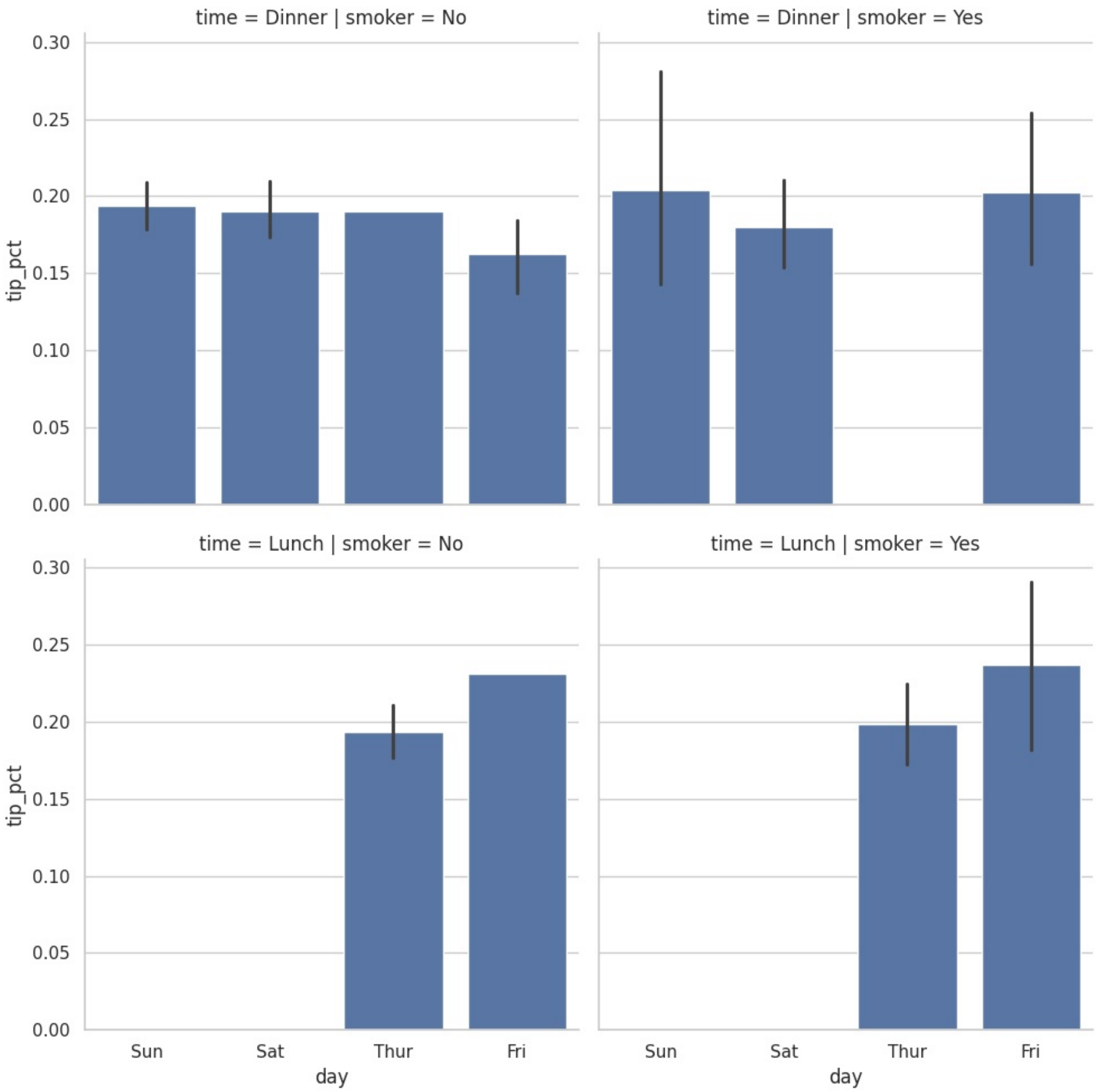
Out[83]: <seaborn.axisgrid.FacetGrid at 0x74da4b44c7f0>



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

In [84]:
```
sns.catplot(x='day', y='tip_pct', row='time',col='smoker',kind='bar', data=tips[tips.tip_pct < 1])
```

Out[84]:
```
<seaborn.axisgrid.FacetGrid at 0x74da4ae2a0e0>
```



catplot supports other plot types that may be useful depending on what you are trying to display.

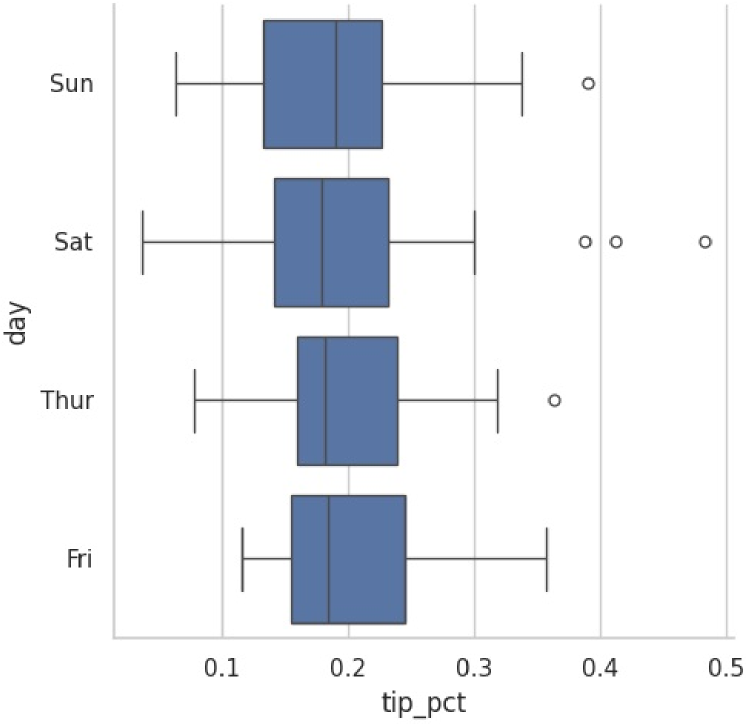
For example, box plots (which show the median, quartiles, and outliers) can be an effective visualization type

In [85]:

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

Out[85]:

```
<seaborn.axisgrid.FacetGrid at 0x74da4a9e1960>
```



With tools like Bokeh and Plotly, it's now possible to specify dynamic, interactive graphics in Python that are destined for a web browser.

For creating static graphics for print or web, I recommend defaulting to matplotlib and add-on libraries like pandas and seaborn for your needs.

In [ ]:

```
# Code with expected output
result = 2 + 3 * 4
print(result)
```

Output:

```
14
```

In [ ]: