Financial Data Analysis with Python

Instructor: Luping Yu

Apr 12, 2022

Lecture 06. 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:

- help identify outliers or needed data transformations.
- a way of generating ideas for models.

Python has many add-on libraries for making static or dynamic visualizations, but we will be mainly focused on **matplotlib**. It is a plotting package designed for creating (mostly **two-dimensional**) publication-quality plots.

The project was started in 2002 to enable a *MATLAB-like* plotting interface in Python. **matplotlib** supports various GUI backends on all operating systems and additionally can export visualizations to all of the common vector and raster graphics formats (PDF, SVG, JPG, PNG, BMP, GIF, etc.).

The simplest way to follow the code examples in the chapter is to use interactive plotting in the Jupyter notebook. To set this up, execute the following statement in a Jupyter notebook:

In [3]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'
```

Plotting with pandas

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.

Line Plots

Series and DataFrame each have a plot attribute for making some basic plot types. By default, **plot( )** makes line plots:

In [4]:
```
import numpy as np
import pandas as pd
```

```
s = pd.Series(np.random.rand(10), index=np.arange(0, 100, 10))
# numpy.random.rand(): 生成随机数
# np.arange(): 生成等差数列

s
```

Out[4]:

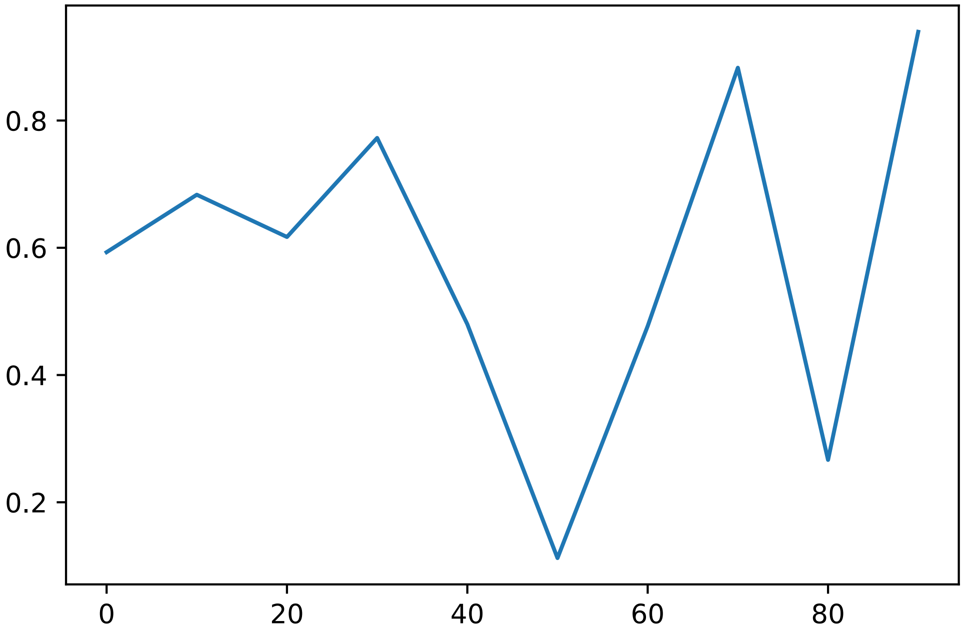
```
0     0.593072
10    0.683394
20    0.616971
30    0.772634
40    0.479885
50    0.112292
60    0.476859
70    0.882975
80    0.266583
90    0.939429
dtype: float64
```

If everything is set up right, a simple line plot should appear:

In [5]:

```
s.plot()
```

Out[5]: <AxesSubplot:>

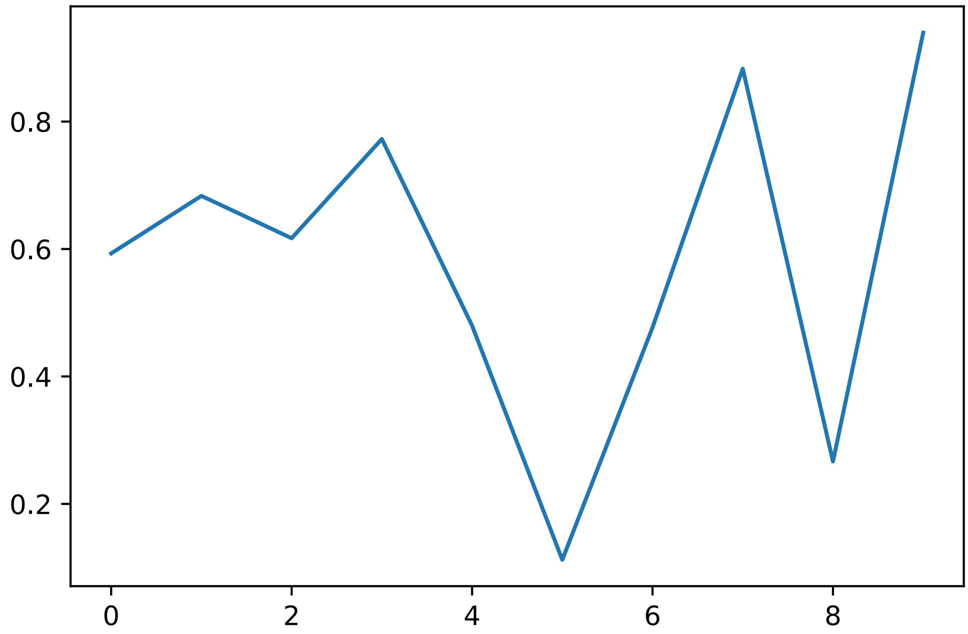


The Series object's **index** is passed to matplotlib for plotting on the **x-axis**, though you can disable this by passing use_index=False.

In [6]:

```
s.plot(use_index=False)
```

Out[6]: <AxesSubplot:>



The x-axis ticks and limits can be adjusted with the **xticks** and **xlim** options, and y-axis respectively with **yticks** and **ylim**. See the following table for a full listing of plot options. I'll comment on a few more of them throughout this section and leave the rest to you to explore.

- Series.plot method arguments:

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

In [8]:
```
df = pd.DataFrame(np.random.rand(10, 2),
                  columns=['A', 'B'],
                  index=np.arange(0, 100, 10))

df
```

Out[8]:

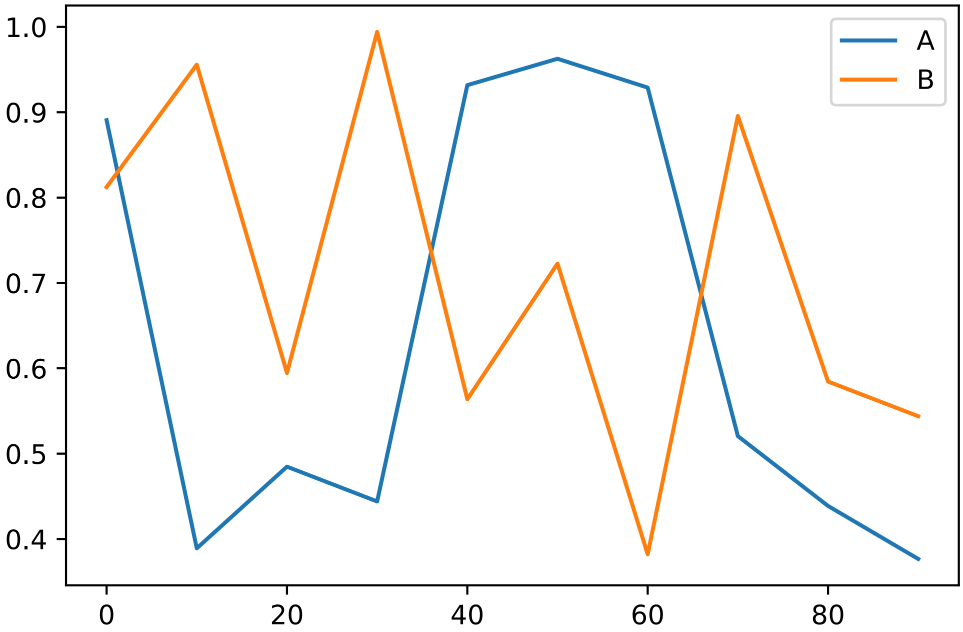
	A	B
0	0.890644	0.812307
10	0.389105	0.955750
20	0.484761	0.594496
30	0.444009	0.994155
40	0.931717	0.563796
50	0.962647	0.722626
60	0.928879	0.382077
70	0.520572	0.895607
80	0.438762	0.584493
90	0.376655	0.543893

In [9]:

```
df.plot()
```

Out[9]:

```
<AxesSubplot:>
```



DataFrame has a number of options allowing some flexibility with how the columns are handled; for example, whether to plot them all on the same subplot or to create separate subplots.

- 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 (Trueby default)

Argument	Description
sort_columns	Plot columns in alphabetical order; by default uses existing column order

Bar Plots

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

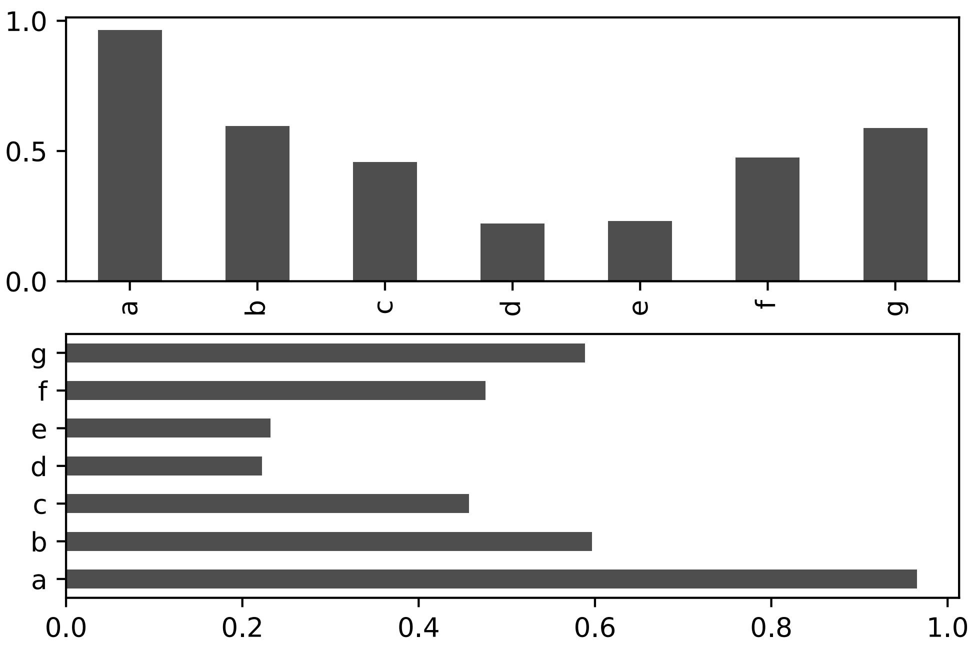
The plot.bar( ) and plot.barh( ) make *vertical* and *horizontal* **bar plots**, respectively. In this case, the Series or DataFrame index will be used as the x (bar) or y (barh) ticks:

In [10]:
```
data = pd.Series(np.random.rand(7), index=list('abcdefg'))

data
```

Out[10]:
```
a    0.964791
b    0.596254
c    0.457071
d    0.222017
e    0.231416
f    0.475800
g    0.588638
dtype: float64
```

In [13]:
```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1)
# 两行一列的子图（子图的行数，子图的列数）

data.plot.bar(ax=axes[0], color='k', alpha=0.7)

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

Out[13]: <AxesSubplot:>



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 [14]:
```
df = pd.DataFrame(np.random.rand(6, 4),
                  index=['one', 'two', 'three', 'four', 'five', 'six'],
                  columns=pd.Index(['A', 'B', 'C', 'D'], name='XMU'))

df
```

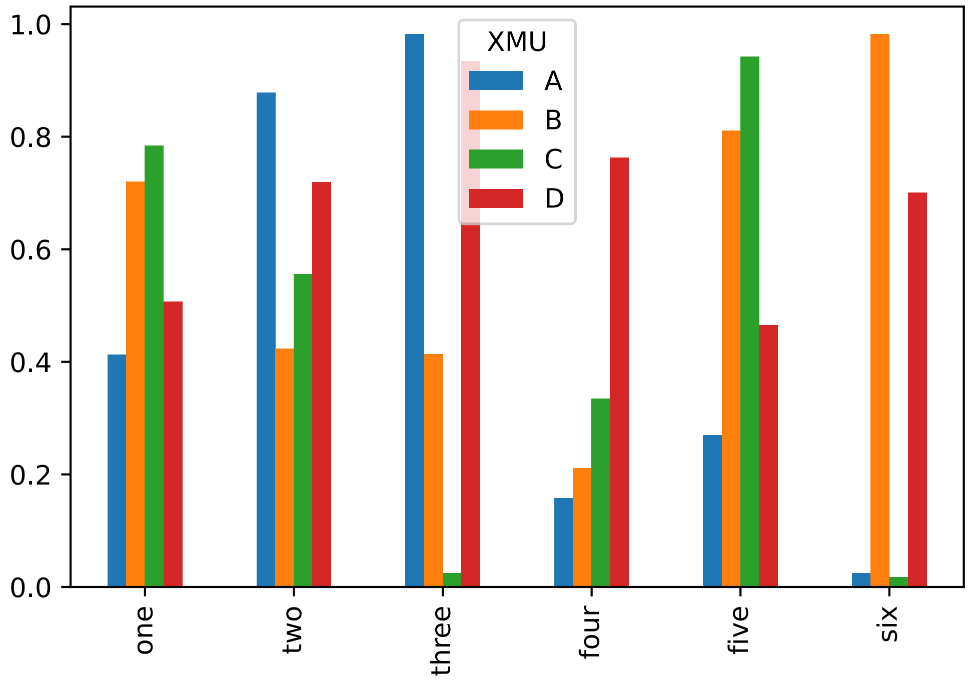
Out[14]:

XMU	A	B	C	D
one	0.412654	0.719823	0.784367	0.506525
two	0.878501	0.422932	0.555934	0.719098
three	0.982058	0.413393	0.024736	0.934065
four	0.158023	0.211044	0.334773	0.763041
five	0.269306	0.810990	0.942117	0.465158
six	0.024565	0.981788	0.017474	0.700778

By default the columns are removed from the DataFrame, though you can leave them in:

In [15]:
```
df.plot.bar()
```

Out[15]: <AxesSubplot:>

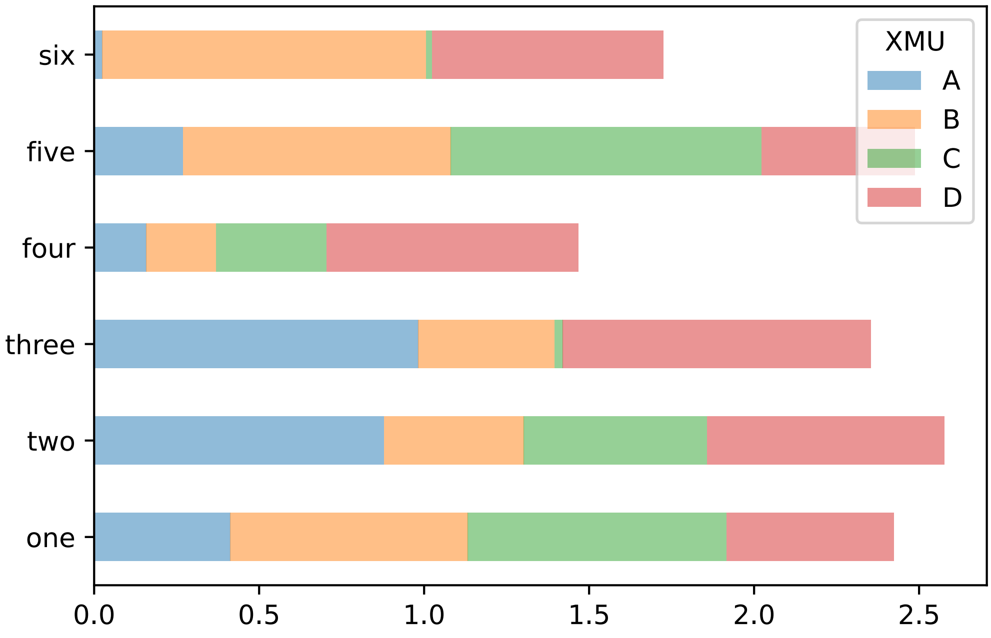


Note that the name "XMU" on the DataFrame's columns is used to title the legend.

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

In [16]:
```
df.plot.barh(stacked=True, alpha=0.5)
```

Out[16]: <AxesSubplot:>



Returning to the tips.csv used earlier in Lecture 04, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day.

We load the data using read_csv and make a *cross-tabulation* by day and party size:

In [17]:
```
tips = pd.read_csv('examples/tips.csv')

tips
```

Out[17]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

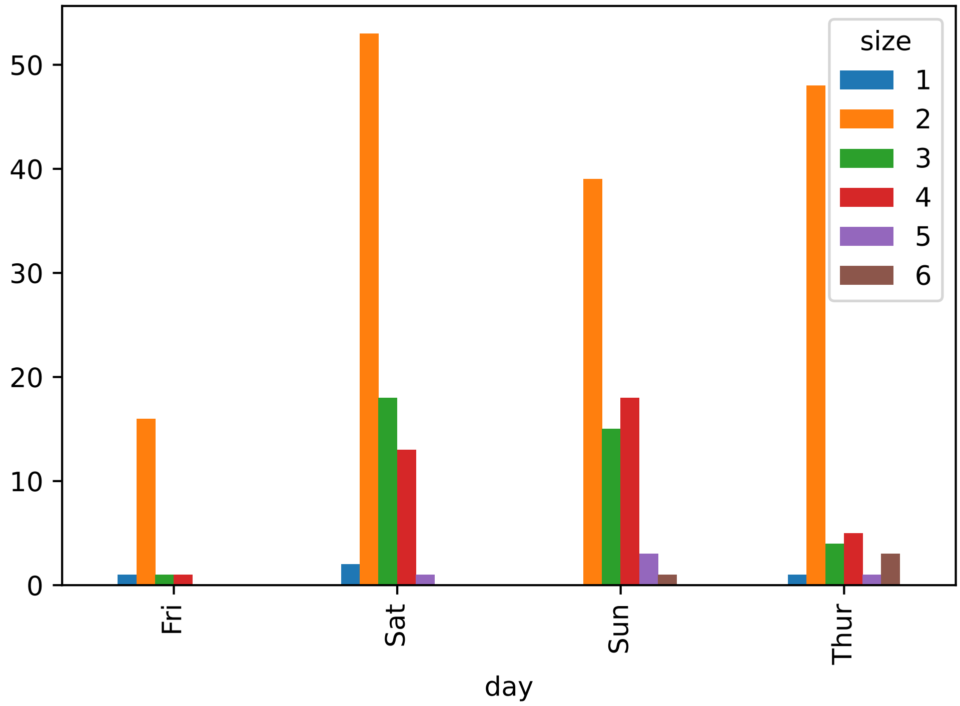
In [18]:
```
party_counts = pd.crosstab(tips['day'], tips['size'])

party_counts
```

Out[18]:

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 [19]:
```
party_counts.plot.bar()
```

Out[19]: <AxesSubplot: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 [21]:
```
tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])

tips.head()
```

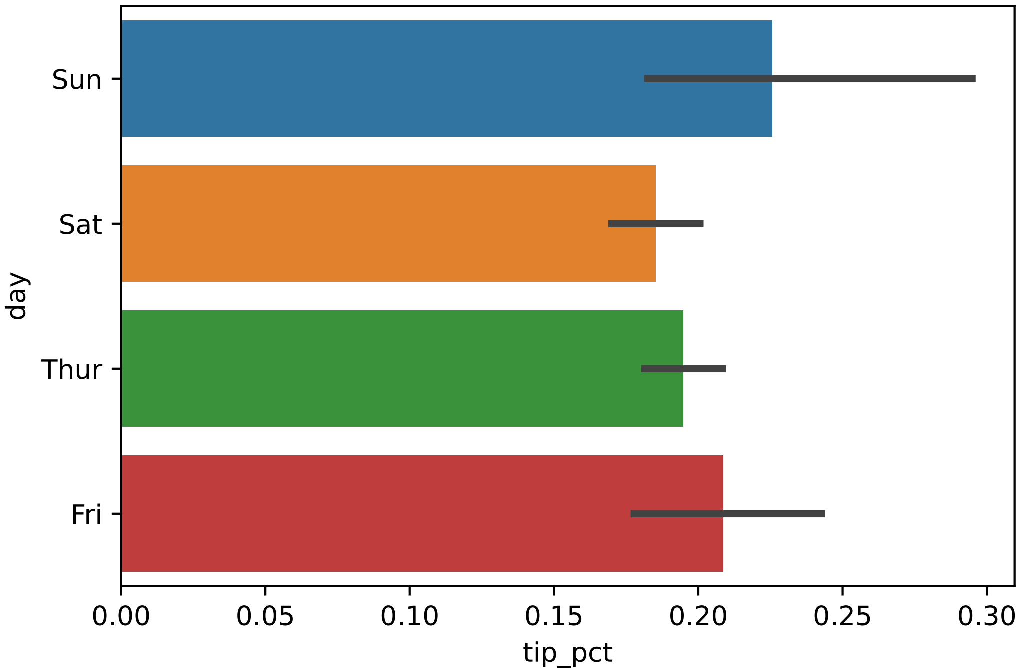
Out[21]:

	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 [22]:
```
import seaborn as sns
```

```
sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

Out[22]: `<AxesSubplot:xlabel='tip_pct', ylabel='day'>`



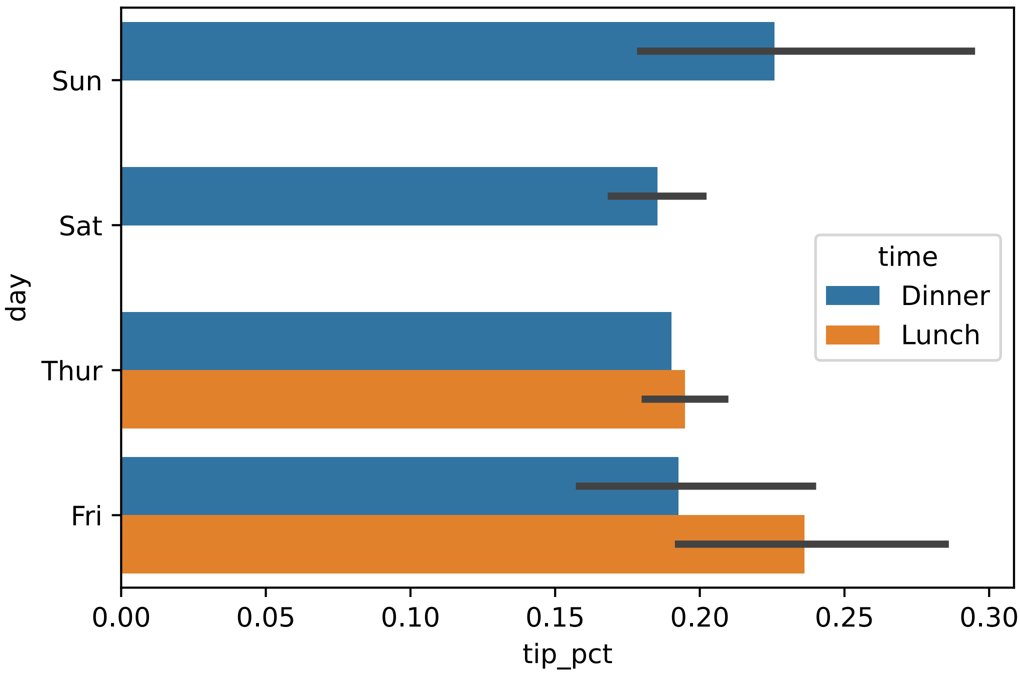
Plotting functions in seaborn take a data argument, which can be a pandas DataFrame. The other arguments refer to column names.

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 [23]:
```
sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

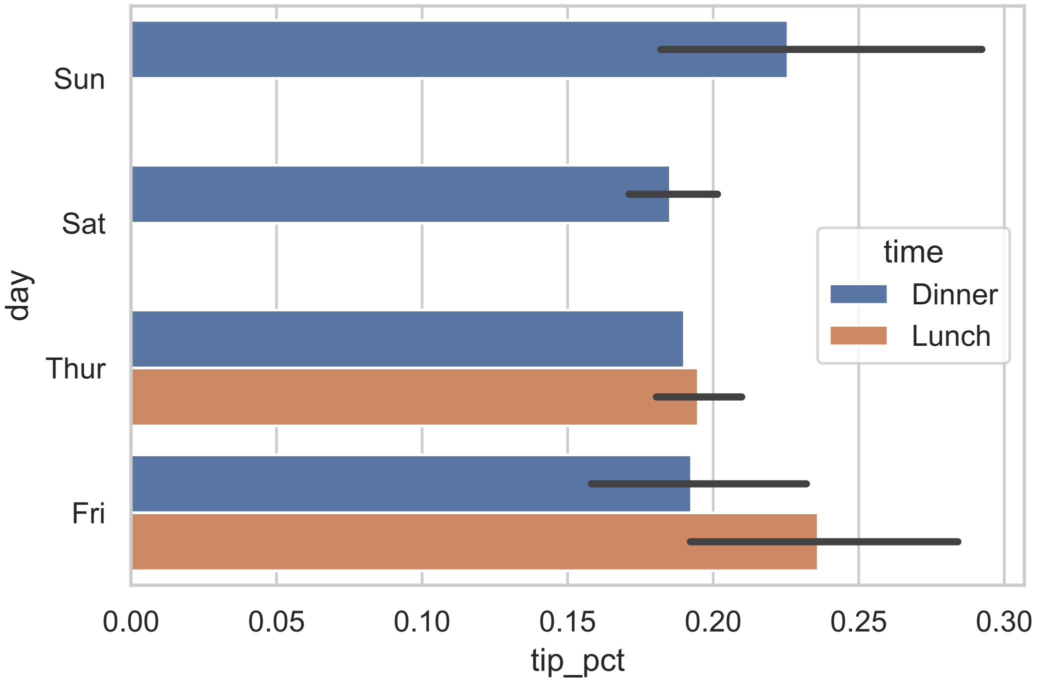
Out[23]: `<AxesSubplot:xlabel='tip_pct', ylabel='day'>`



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:

In [24]:
```
sns.set(style="whitegrid")
#sns.reset_orig()

sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

Out[24]: <AxesSubplot:xlabel='tip_pct', ylabel='day'>



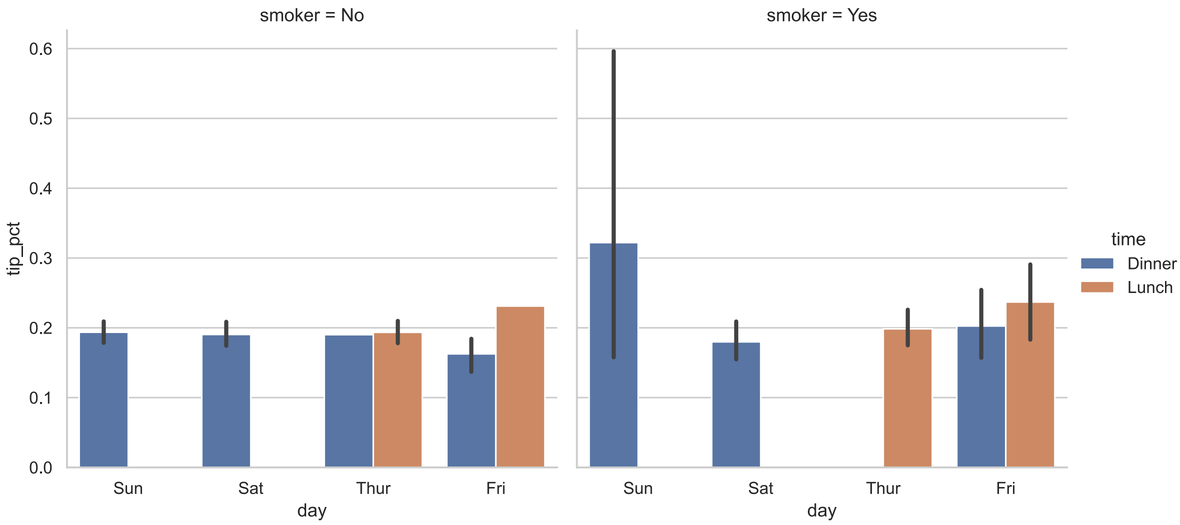
Facet Grids and Categorical Data

What about datasets where we have additional grouping dimensions? One way to visualize data with many categorical variables is to use a **catplot**.

Seaborn has a useful built-in function **catplot** that simplifies making many kinds of plots:

In [25]:
```
sns.catplot(x='day', y='tip_pct', hue='time', col='smoker', kind='bar', data
```

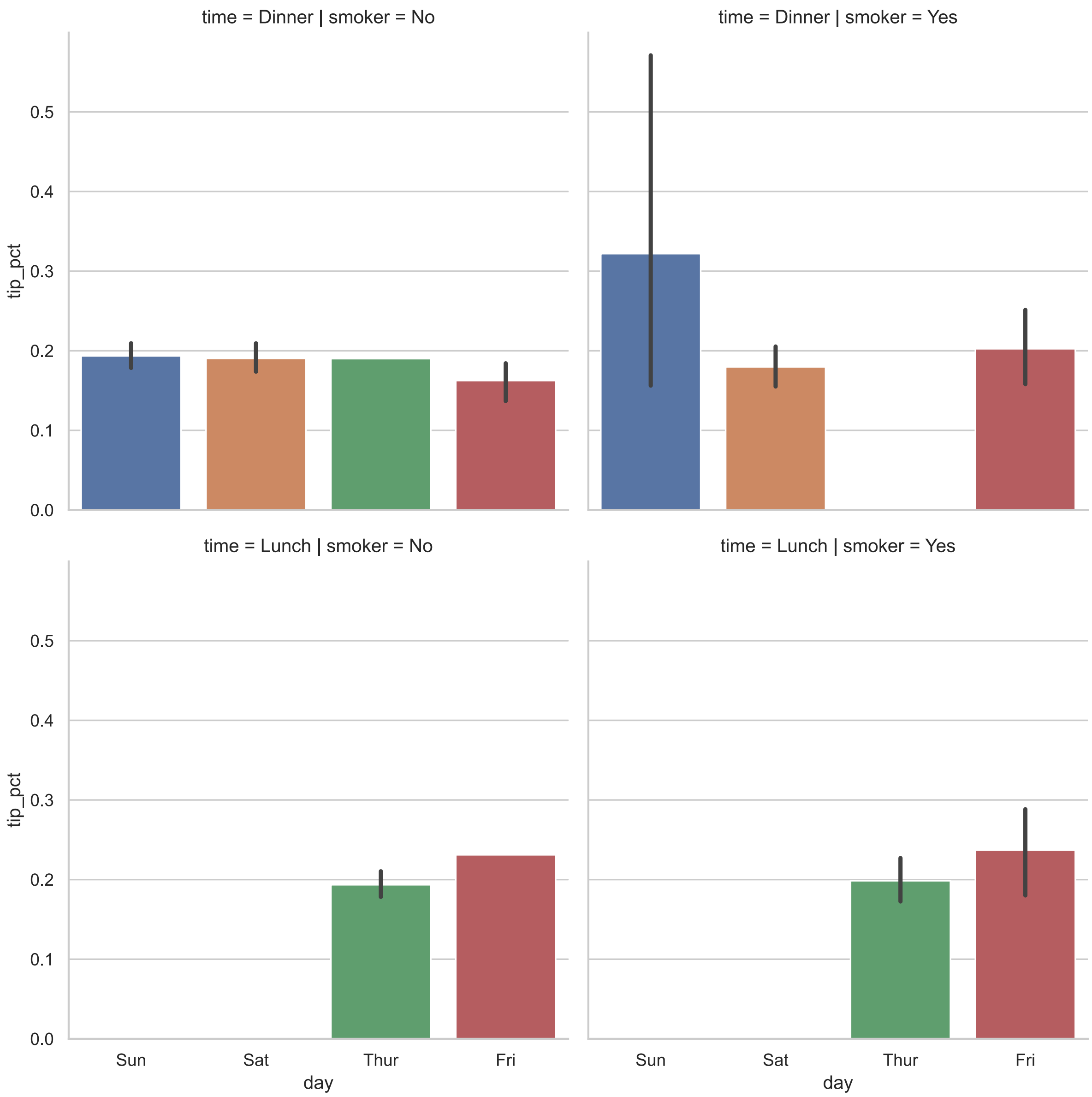
Out[25]: <seaborn.axisgrid.FacetGrid at 0x12794db20>



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 [26]:
```
sns.catplot(x='day', y='tip_pct', row='time', col='smoker', kind='bar', data
```

Out[26]:
```
<seaborn.axisgrid.FacetGrid at 0x127c15d30>
```

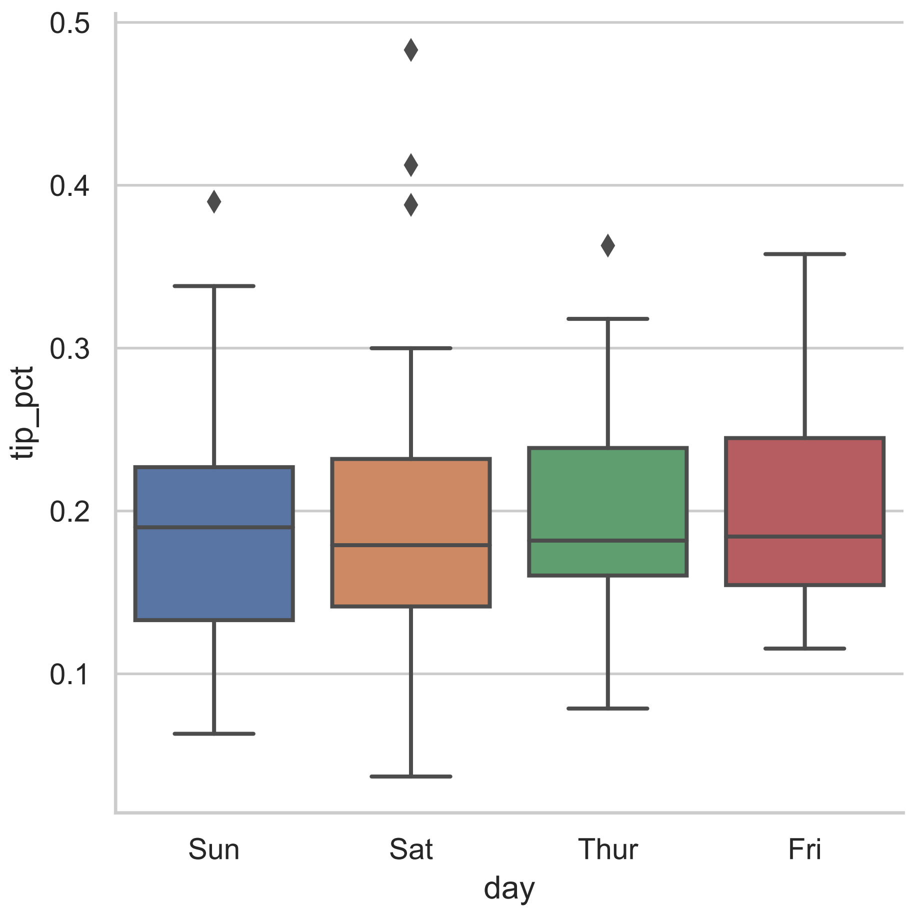


catplot supports other plot types that may be useful depending on what you are trying to display.

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

In [27]:
```
sns.catplot(x='day', y='tip_pct', kind='box', data=tips[tips.tip_pct < 0.5])
```

Out[27]: <seaborn.axisgrid.FacetGrid at 0x127cf8c70>



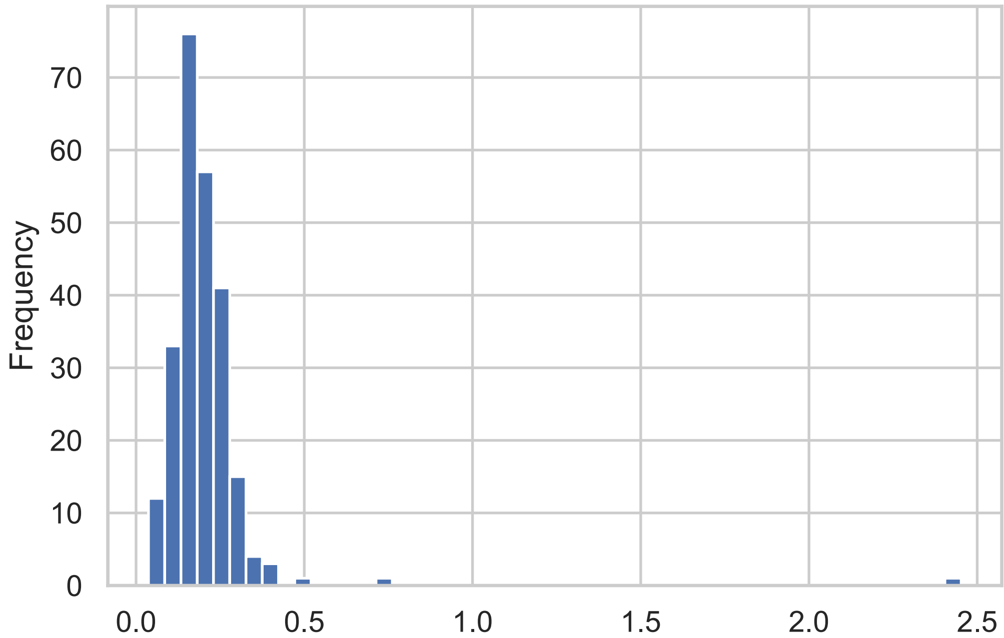
Histograms and Density Plots

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 [28]:
```
tips['tip_pct'].plot.hist(bins=50)
```

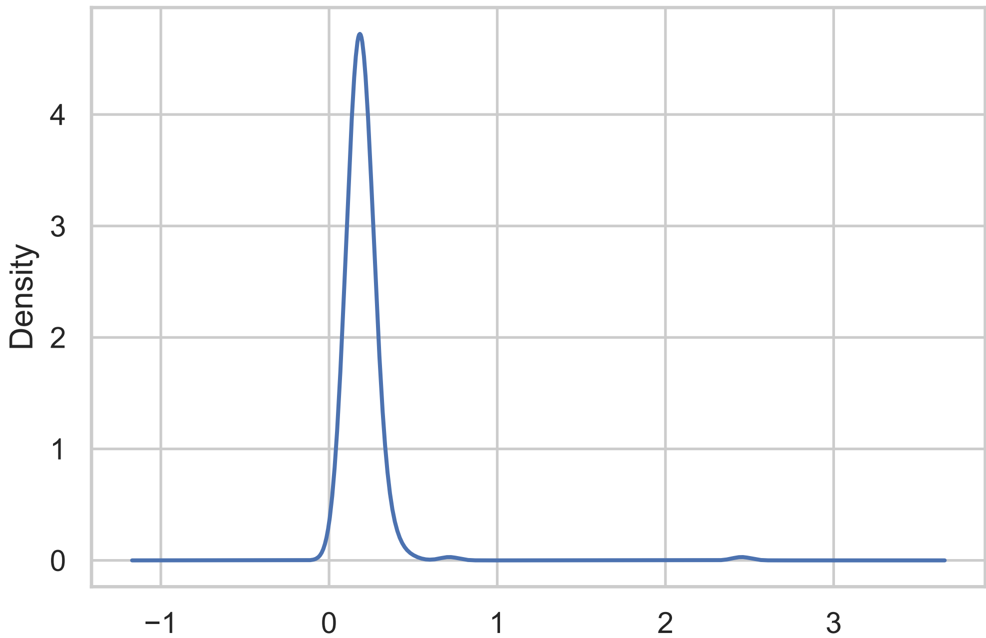
Out[28]: <AxesSubplot:ylabel='Frequency'>



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:

In [29]:
```
tips['tip_pct'].plot.density()
```

Out[29]:
```
<AxesSubplot:ylabel='Density'>
```



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

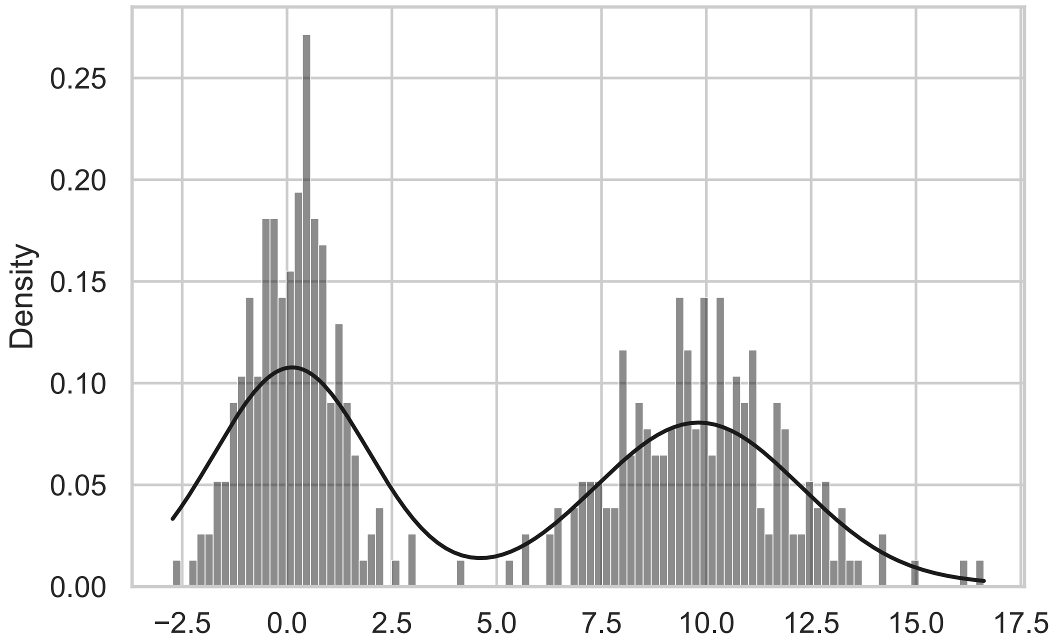
As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions:

In [30]:
```
comp1 = np.random.normal(0, 1, size=200)
comp2 = np.random.normal(10, 2, size=200)

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

values
```

Out[30]:
```
0       -0.850069
1       -0.406853
2        0.192401
3        1.186091
4        2.187075
           ...
395     11.184493
396      9.705715
397     11.069138
398     12.788098
399      8.398379
Length: 400, dtype: float64
```

In [31]:
```
sns.histplot(values, bins=100, color='k', kde=True, stat="density")
```

Out[31]:
```
<AxesSubplot: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.

For example, here we load the macrodata.csv, select a few variables, then compute log differences:

In [32]:
```
macro = pd.read_csv('examples/macrodata.csv')

macro
```

Out[32]:

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	m1	t
0	1959.0	1.0	2710.349	1707.4	286.898	470.045	1886.9	28.980	139.7	
1	1959.0	2.0	2778.801	1733.7	310.859	481.301	1919.7	29.150	141.7	
2	1959.0	3.0	2775.488	1751.8	289.226	491.260	1916.4	29.350	140.5	
3	1959.0	4.0	2785.204	1753.7	299.356	484.052	1931.3	29.370	140.0	
4	1960.0	1.0	2847.699	1770.5	331.722	462.199	1955.5	29.540	139.6	
...	...	...	...	...	...	...	...	...	...	
198	2008.0	3.0	13324.600	9267.7	1990.693	991.551	9838.3	216.889	1474.7	
199	2008.0	4.0	13141.920	9195.3	1857.661	1007.273	9920.4	212.174	1576.5	
200	2009.0	1.0	12925.410	9209.2	1558.494	996.287	9926.4	212.671	1592.8	
201	2009.0	2.0	12901.504	9189.0	1456.678	1023.528	10077.5	214.469	1653.6	
202	2009.0	3.0	12990.341	9256.0	1486.398	1044.088	10040.6	216.385	1673.9	

203 rows × 14 columns

In [33]:

```
data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]

trans_data = np.log(data).diff().dropna()
# np.log(): Natural logarithm
# diff(): First discrete difference of element

trans_data
```

Out[33]:

	cpi	m1	tbilrate	unemp
1	0.005849	0.014215	0.088193	-0.128617
2	0.006838	-0.008505	0.215321	0.038466
3	0.000681	-0.003565	0.125317	0.055060
4	0.005772	-0.002861	-0.212805	-0.074108
5	0.000338	0.004289	-0.266946	0.000000
...	...	...	...	...
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

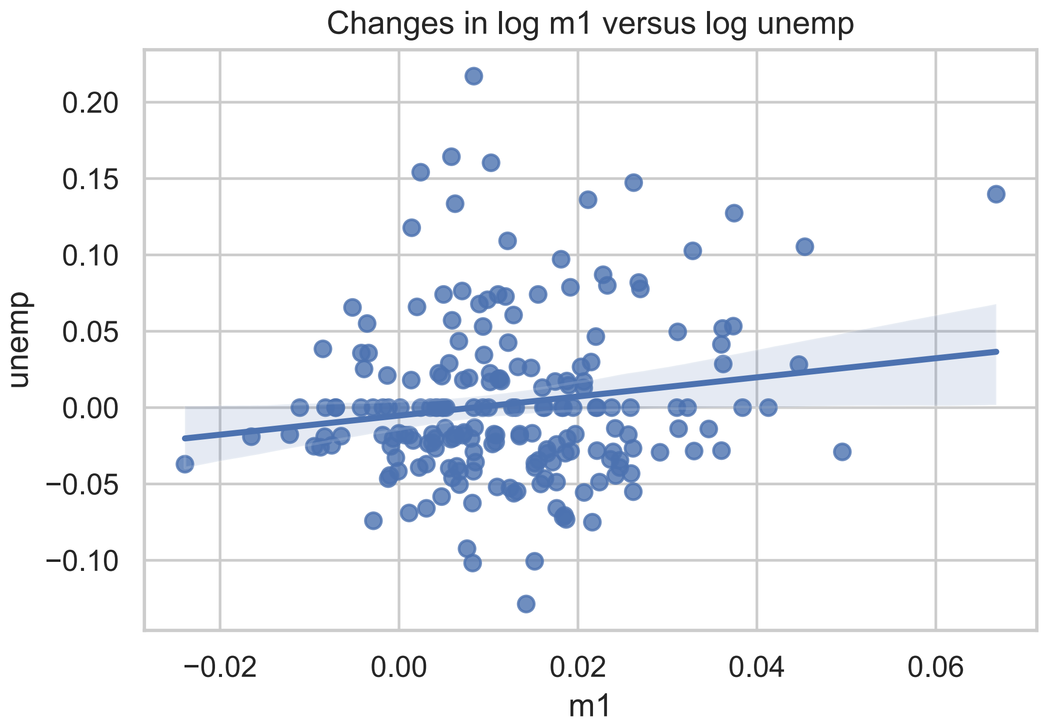
202 rows × 4 columns

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

In [34]:

```
sns.regplot(x='m1', y='unemp', data=trans_data)

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

Out[34]: `Text(0.5, 1.0, 'Changes in log m1 versus log unemp')`



Conclusion

The goal of this chapter was to get your feet wet with some basic data visualization using **pandas**, **matplotlib**, and **seaborn**.

If visually communicating the results of data analysis is important in your work, I encourage you to seek out resources to learn more about effective data visualization.

It is an active field of research and you can practice with many excellent learning resources available online and in print form.