In [2]:
```
%matplotlib inline
# import naming conventions
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# (further reading on mpl imports: http://bit.ly/197aGoq )
```

Part 1: data structures

There are two* main structures in `pandas` : Series (1-dimensional labeled array) and DataFrame (2-dimensional labeled structure).

* there is also a TimeSeries (a flavor of Series that contains datetimes), Panel (3-dimensional), and Panel4D (4-dimensional). The last two are 'less used,' according to the docs. I haven't experimented with them yet.

Series (1D)

Series can hold any data type, and the axis label is called an index. Series is dict-like in that you can get and set values by index label.

In [3]:
```
s1 = pd.Series([5,7,9,'y',10,11])
s1
```

Out[3]:
```
0     5
1     7
2     9
3     y
4    10
5    11
dtype: object
```

In [4]:
```
# by default (without specifying them explicitly), the index label is just an
 int
s1[0]
```

Out[4]:
```
5
```

DataFrame (2D)

Columns can be of different data types. Index and column names are optional. If individual Series have different indexes, the DataFrame index will be the union of the individual ones.

Can create from:

- dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Series
- another DataFrame

N.B.: there are other helper methods for constructing DataFrames from varying data types; see the docs (http://pandas.pydata.org/pandas-docs/stable/dsintro.html#alternate-constructors) for more options.

In [25]:
```
# create a couple more Series
s2, s3, s4 = pd.Series(np.random.randn(6)), pd.Series(np.random.randn(6)), pd.
Series(np.random.randn(6))
```

In [27]:
```
# combine multiple Series into a DataFrame with column labels
df1 = pd.DataFrame({'A': s1, 'B': s2, 'C': s3, 'D': s4})

df1
```

Out[27]:

	A	B	C	D
0	5	1.059393	-0.666090	-0.263345
1	7	-0.080123	0.974164	1.835602
2	9	-0.146537	-0.626024	-1.120802
3	y	0.810360	-0.923507	0.153765
4	10	-0.037260	-0.136711	0.996121
5	11	0.548880	1.557192	-0.553245

In [32]:
```python
# when Series are different lengths, DataFrame fills in gaps with NaN
s4 = pd.Series(np.random.randn(8))  # whoaaaaaa this Series has extra entries!

df1 = pd.DataFrame({'A': s1, 'B': s2, 'C': s3, 'D': s4})

df1
```

Out[32]:

	A	B	C	D
0	5	1.059393	-0.666090	0.189001
1	7	-0.080123	0.974164	1.140012
2	9	-0.146537	-0.626024	-1.172670
3	y	0.810360	-0.923507	0.741262
4	10	-0.037260	-0.136711	-0.243563
5	11	0.548880	1.557192	-0.614017
6	NaN	NaN	NaN	-0.825578
7	NaN	NaN	NaN	-0.430320

In [37]:
```python
# create a DataFrame from numpy array
df2 = pd.DataFrame(np.random.randn(6,5))

df2                # can only have one 'pretty' output per cell (if it's the last
command)

#print df2         # otherwise, can print arb number of results w/o pretty forma
t
#print df1         # (uncomment both of these print statements)
```

Out[37]:

	0	1	2	3	4
0	-1.288270	1.145167	0.448429	-0.280620	0.746102
1	-0.920075	-2.125817	0.219931	-0.320564	0.940035
2	-1.749811	-1.722389	0.008619	-0.513473	0.513615
3	-0.262157	-0.425945	0.676044	-0.734886	0.132838
4	-0.008849	0.170730	-1.743279	-0.549093	-1.840107
5	1.082913	0.855540	0.053807	0.528531	-0.133769

Can inspect your DataFrames with head() and tail() methods - takes a number of lines as an argument.

Without specifiying them, DataFrames have default index and column name attributes.

In [41]:
```
# recall current dataframe to show the top 3 rows
df2.head(3)
```

Out[41]:

	0	1	2	3	4
0	-1.288270	1.145167	0.448429	-0.280620	0.746102
1	-0.920075	-2.125817	0.219931	-0.320564	0.940035
2	-1.749811	-1.722389	0.008619	-0.513473	0.513615

But you can assign to those attributes of the DataFrame...

In [50]:
```
cols = ['a', 'b', 'c', 'd', 'e']

# assign columns attribute (names)
df2.columns = cols

# create an index:
#  generate a sequence of dates with pandas' data_range() method,
#  then assign the index attribute
dates = pd.date_range(start='2020-10-22 13:45:27', freq='h', periods=6)
df2.index = dates

df2
```

Out[50]:

	a	b	c	d	e
2020-10-22 13:45:27	-1.288270	1.145167	0.448429	-0.280620	0.746102
2020-10-22 14:45:27	-0.920075	-2.125817	0.219931	-0.320564	0.940035
2020-10-22 15:45:27	-1.749811	-1.722389	0.008619	-0.513473	0.513615
2020-10-22 16:45:27	-0.262157	-0.425945	0.676044	-0.734886	0.132838
2020-10-22 17:45:27	-0.008849	0.170730	-1.743279	-0.549093	-1.840107
2020-10-22 18:45:27	1.082913	0.855540	0.053807	0.528531	-0.133769

In [53]:
```
# an aside: inspecting the dates object...
print ('what is a date_range object?\n\n', dates)
```

```
what is a date_range object?

 DatetimeIndex(['2020-10-22 13:45:27', '2020-10-22 14:45:27',
               '2020-10-22 15:45:27', '2020-10-22 16:45:27',
               '2020-10-22 17:45:27', '2020-10-22 18:45:27'],
              dtype='datetime64[ns]', freq='H')
```

Do some indexing / subsetting...

In [54]:
```python
# select a row by index label by using .loc
df2.loc['2020-10-22 13:45:27']
```

Out[54]:
```
a   -1.288270
b    1.145167
c    0.448429
d   -0.280620
e    0.746102
Name: 2020-10-22 13:45:27, dtype: float64
```

In [55]:
```python
# select a single element
df2.loc['2020-10-22 14:45:27','c']
```

Out[55]:
```
0.21993097804011166
```

In [57]:
```python
# new dataframe with random numbers
df1 = pd.DataFrame(np.random.randn(6,4), index=list('123456'),columns=list('AB
CD'))

df1
```

Out[57]:

	A	B	C	D
1	0.390347	-0.842599	-0.416640	0.913335
2	-0.394141	1.907458	-0.164556	0.456193
3	2.765845	0.423942	-0.768414	0.506957
4	-1.646204	-1.638822	0.512350	0.203116
5	0.704281	-0.441898	0.466234	0.400223
6	-0.345562	-0.714986	-0.254086	-0.128000

In [63]:
```python
# address two separate rows, and a range of all columns
df1.loc[['5','6'],'A':'Z']
```

Out[63]:

	A	B	C	D
5	0.704281	-0.441898	0.466234	0.400223
6	-0.345562	-0.714986	-0.254086	-0.128000

part 2: data

In the `data/` directory is the sample of parsed twitter data that floats around with gnacs. To create the string of column names, I just used the explain option with all other options.

In [68]:
```
gnacs_x = "id|postedTime|body|None1|['twitter_entiteis:urls:url']|['None']|['actor1:languages_list-items']|gnip:language:value|twitter_lang|[u'geo:coordinates_list-items']|geo2:type|None2|None3|None4|None5|actor2:utcOffset|None|None6|None7|None8|None9|None10|None11|None12|None13|actor3:displayName|actor:preferredUsername|actor4:id|gnip:klout_score|actor5:followersCount|actor6:friendsCount|actor7:listedCount|actor8:statusesCount|Tweet|None14|None15|None16"
colnames = gnacs_x.split('|')
```

In [71]:
```
# prevent the automatic compression of wide dataframes (add scroll bar)
pd.set_option("display.max_columns", None)

# get some data, inspect
df1 = pd.read_csv('C:\\Users\\wafa\\Documents\\big_data_analytics\\week2\\twitter_sample.csv', sep='|', names=colnames)

df1.tail(5)
```

Out[71]:

	id	postedTime	body
90	tag:search.twitter.com,2005:351835321081659392	2013-07-01T22:50:51.000Z	@xhazzasdimples Probabile AHAHHAHAHAHAHAHAHAHA...
91	tag:search.twitter.com,2005:351835321442385921	2013-07-01T22:50:52.000Z	Mandei a foto do Piqué e o fc n respondeu até ...
92	tag:search.twitter.com,2005:351835321425608704	2013-07-01T22:50:52.000Z	O matheus André ta me falando aqui , tem quase...
93	tag:search.twitter.com,2005:351835320859369475	2013-07-01T22:50:51.000Z	@Mulayhim hatha bs one exam. Other exams y6l3o...
94	tag:search.twitter.com,2005:351835321471746048	2013-07-01T22:50:52.000Z	😕 Hmm...

Since there are so many explain fields that come back with 'None', let's just get rid of them for now.

(In the future, we might try to find a way to make that field more descriptive, too.)

In [316]:
```
# n.b.: this is an *in-place* delete -- unusual for a pandas structure
#del df1['None']
# The command below is how the docs suggest carrying this out (creating a new df).
#   But, it doesn't seem to work -- possibly due to multiple cols with same name. Oh well.
#new_df = df1.drop('None..', axis=1)  # return new df
new_df = df1[df1.columns.drop(list(df1.filter(regex='None')))]
```

In [317]:
```
# have a peek
new_df.tail(5)
```

Out[317]:

	id	postedTime	body
90	tag:search.twitter.com,2005:351835321081659392	2013-07-01T22:50:51.000Z	@xhazzasdimples Probabile AHAHHAHAHAHAHAHAHAHA...
91	tag:search.twitter.com,2005:351835321442385921	2013-07-01T22:50:52.000Z	Mandei a foto do Piqué e o fc n respondeu até ...
92	tag:search.twitter.com,2005:351835321425608704	2013-07-01T22:50:52.000Z	O matheus André ta me falando aqui , tem quase...
93	tag:search.twitter.com,2005:351835320859369475	2013-07-01T22:50:51.000Z	@Mulayhim hatha bs one exam. Other exams y6l3o...
94	tag:search.twitter.com,2005:351835321471746048	2013-07-01T22:50:52.000Z	😕 Hmm...

slicing & combining

Subsetting a DataFrame is very similar to the syntax in R. There are two ways to select columns: 'dot' (attribute) notation, and 'square bracket' (index) notation. Sometimes, the column names will dictate which you have to use.

In [79]:
```
# inspect those rows with twitter-classified lang 'en' (scroll the right to se
e)
new_df[new_df.twitter_lang == 'en'].head(3)

# the colons in the column name below won't allow dot-access to the column, so
we can quote them and still filter.
#df1[df1["gnip:language:value"] == 'en'].head(3)
```

Out[79]:

	id	postedTime	body	['twitter_entiteis:url:
8	tag:search.twitter.com,2005:351835318028222465	2013-07-01T22:50:51.000Z	@pafcdan Aww good! X	
9	tag:search.twitter.com,2005:351835318346981377	2013-07-01T22:50:51.000Z	Newest hobby: sending videos back and forth of...	
11	tag:search.twitter.com,2005:351835318024028161	2013-07-01T22:50:51.000Z	~FINALLY OFF OF WORK~	

Let's get a subset of this dataframe that has numerical values so we can eventually do some stuff.

In [80]:
```python
# create new dataframe from numerical columns
df2 = df1[["gnip:klout_score","actor5:followersCount", "actor6:friendsCount",
"actor7:listedCount", "actor8:statusesCount"]]

df2.head()
```

Out[80]:

	gnip:klout_score	actor5:followersCount	actor6:friendsCount	actor7:listedCount	actor8:statuse
0	35	178	129	0	
1	32	144	215	0	
2	18	37	54	0	
3	50	438	174	1	
4	21	12	6	0	

In [81]:
```python
# because I happen to know the answer, let's check data types of the column
s...
df2.dtypes
```

Out[81]:
```
gnip:klout_score         object
actor5:followersCount     int64
actor6:friendsCount       int64
actor7:listedCount        int64
actor8:statusesCount      int64
dtype: object
```

The `object` type means that the column has multiple types of data in it. This is a good opportunity to 'fix' a section of the DataFrame by way of a function & the `map()` function

In [318]:
```python
# convert ints / strings to floats, give up on anything else (call it 0.0)
def floatify(val):
    if val == None or val == 'None':
        return 0.0
    else:
        return float(val)
```

In [319]:
```python
# assigning to an existing column overwrites that column
df2['gnip:klout_score'] = df2['gnip:klout_score'].map(floatify).copy()

# check again
df2.dtypes
```

Out[319]:
```
gnip:klout_score         float64
actor5:followersCount    float64
actor6:friendsCount      float64
actor7:listedCount       float64
actor8:statusesCount     float64
fol/fr                   float64
score/followers          float64
dtype: object
```

In [85]:
```
# use all floats just for fun.
#  this only works if the elements can all be converted to floats (e.g. ints o
r something python can handle)
df2 = df2.astype(float)

df2.dtypes
```

Out[85]:
```
gnip:klout_score          float64
actor5:followersCount     float64
actor6:friendsCount       float64
actor7:listedCount        float64
actor8:statusesCount      float64
dtype: object
```

Since they're all numbers now, we can do math and also add new columns to the DataFrame. Combining values from separate columns occurs on a row-by-row basis, as expected.

In [88]:
```
# look at some activity ratios - add col to df
df2['score/followers'] = df2['gnip:klout_score'] / df2['actor5:followersCount'
]

df2.head()

# can also use the built-in describe() method to get quick descriptive stats o
n the dataframe
#df2.describe()
```

Out[88]:

	gnip:klout_score	actor5:followersCount	actor6:friendsCount	actor7:listedCount	actor8:statuse
0	35.0	178.0	129.0	0.0	
1	32.0	144.0	215.0	0.0	
2	18.0	37.0	54.0	0.0	
3	50.0	438.0	174.0	1.0	1
4	21.0	12.0	6.0	0.0	

grouping

groupby() is used for the split-apply-combine process. I'm led to believe that this is one of the stronger aspects of `pandas` ' approach to DataFrames (versus R's), but haven't yet had a chance to really see the power.

In [91]:
```
new_df.head()
```

Out[91]:

	id	postedTime	body	['twitter_entit
0	tag:search.twitter.com,2005:351835317671690241	2013-07-01T22:50:51.000Z	kavga edelim ama konuşalım	
1	tag:search.twitter.com,2005:351835317604593666	2013-07-01T22:50:51.000Z	@shane_joersz wooooow	
2	tag:search.twitter.com,2005:351835317747191808	2013-07-01T22:50:51.000Z	お前との肌のふれあいなんぞ求めてない。自重しろ。	
3	tag:search.twitter.com,2005:351835317608792064	2013-07-01T22:50:51.000Z	@Gabo_navoficial yo tambien creo en ti mi char...	
4	tag:search.twitter.com,2005:351835317755592705	2013-07-01T22:50:51.000Z	только ты об этом не знаешь... http://t.co/MOH...	

Use a groupby to collect all rows by language value, and subsequently use some of the methods available to `GroupBy` DataFrames. Note that the `GroupBy` methods will only act on (and the method call only return) values for columns where numerical calculation makes sense.

In [96]:
```
# subset new_df, create new df with only 'popular' accounts -- those matching
 the filter condition given
pop_df = new_df[new_df["actor5:followersCount"] >= 150]
pop_df.head(4)
```

Out[96]:

	id	postedTime	body	['twitter_entit
0	tag:search.twitter.com,2005:351835317671690241	2013-07-01T22:50:51.000Z	kavga edelim ama konuşalım	
3	tag:search.twitter.com,2005:351835317608792064	2013-07-01T22:50:51.000Z	@Gabo_navoficial yo tambien creo en ti mi char...	
5	tag:search.twitter.com,2005:351835317801730048	2013-07-01T22:50:51.000Z	I'm at Büyükçekmece Sahil w/ @emineetrk http:/...	['http://t.co/3
6	tag:search.twitter.com,2005:351835317554257920	2013-07-01T22:50:51.000Z	Dile Al Amor >>>	

In [320]:

```python
# fix the klout scores again
pop_df['gnip:klout_score'] = pop_df['gnip:klout_score'].map(floatify).copy()
pop_df.head(4)
```

Out[320]:

	id	postedTime	body	['twitter_entit
0	tag:search.twitter.com,2005:351835317671690241	2013-07-01T22:50:51.000Z	kavga edelim ama konuşalım	
3	tag:search.twitter.com,2005:351835317608792064	2013-07-01T22:50:51.000Z	@Gabo_navoficial yo tambien creo en ti mi char...	
5	tag:search.twitter.com,2005:351835317801730048	2013-07-01T22:50:51.000Z	I'm at Büyükçekmece Sahil w/ @emineetrk http:/...	['http://t.co/3
6	tag:search.twitter.com,2005:351835317554257920	2013-07-01T22:50:51.000Z	Dile Al Amor >>>	

In [104]:
```
# use GroupBy methods for stats on each group:
print(pop_df.groupby("gnip:klout_score").size())      # number of elements per group
print(pop_df.groupby("gnip:klout_score").sum())       # sum of elements in each group (obviously doesn't make sense for some cols)
print(pop_df.groupby("gnip:klout_score").mean())      # algebraic mean of elements per group
```

```
gnip:klout_score
0.0     1
18.0    1
28.0    1
29.0    1
30.0    2
31.0    1
32.0    1
33.0    5
34.0    2
35.0    3
36.0    2
37.0    5
38.0    4
39.0    3
40.0    7
41.0    7
42.0    3
43.0    2
44.0    7
45.0    2
46.0    1
47.0    1
48.0    2
49.0    1
50.0    1
53.0    2
57.0    1
62.0    1
64.0    1
dtype: int64
                   actor4:id  actor5:followersCount  actor6:friendsCount  \
gnip:klout_score
0.0                 45421764                    349                  244
18.0               242505369                    290                  683
28.0                81327848                    152                  335
29.0               891890964                    150                  157
30.0               975212764                    744                  671
31.0               190517290                    182                  706
32.0                27737035                    342                 2001
33.0              1387808164                   1340                 1573
34.0               973531054                   1395                 1242
35.0              1095757042                   1232                 1140
36.0              1167902723                    312                  460
37.0              3419043249                   2443                 3563
38.0              1217499951                   1207                 1701
39.0              1359425302                   2202                 2378
40.0              3094534273                   5853                 4466
41.0              3178644422                   1479                 1374
42.0               730117140                   1695                 1041
43.0              1012834821                    660                  498
44.0              2226256675                  11006                 6123
45.0               633297900                   1465                 1235
46.0               554205628                   1999                  293
47.0               220404906                    421                  345
48.0              1568877390                   2764                 2141
49.0               732586412                    159                  145
```

50.0	461188787	438	174
53.0	1126677797	6263	4430
57.0	125565884	192	235
62.0	1160945754	11873	69
64.0	29619102	40543	116

	actor7:listedCount	actor8:statusesCount
gnip:klout_score		
0.0	1	12236
18.0	0	540
28.0	3	2151
29.0	0	6252
30.0	1	23685
31.0	1	2417
32.0	0	468
33.0	14	34906
34.0	0	38306
35.0	1	43370
36.0	0	6273
37.0	2	82012
38.0	1	59149
39.0	2	25344
40.0	65	95264
41.0	0	39447
42.0	3	58312
43.0	0	8499
44.0	20	132726
45.0	3	117371
46.0	27	60236
47.0	0	15986
48.0	1	36416
49.0	0	8925
50.0	1	17636
53.0	19	50304
57.0	3	4073
62.0	56	1991
64.0	486	60465

	actor4:id	actor5:followersCount	actor6:friendsCount	\
gnip:klout_score				
0.0	4.542176e+07	349.000000	244.000000	
18.0	2.425054e+08	290.000000	683.000000	
28.0	8.132785e+07	152.000000	335.000000	
29.0	8.918910e+08	150.000000	157.000000	
30.0	4.876064e+08	372.000000	335.500000	
31.0	1.905173e+08	182.000000	706.000000	
32.0	2.773704e+07	342.000000	2001.000000	
33.0	2.775616e+08	268.000000	314.600000	
34.0	4.867655e+08	697.500000	621.000000	
35.0	3.652523e+08	410.666667	380.000000	
36.0	5.839514e+08	156.000000	230.000000	
37.0	6.838086e+08	488.600000	712.600000	
38.0	3.043750e+08	301.750000	425.250000	
39.0	4.531418e+08	734.000000	792.666667	
40.0	4.420763e+08	836.142857	638.000000	
41.0	4.540921e+08	211.285714	196.285714	
42.0	2.433724e+08	565.000000	347.000000	
43.0	5.064174e+08	330.000000	249.000000	

44.0	3.180367e+08	1572.285714	874.714286
45.0	3.166490e+08	732.500000	617.500000
46.0	5.542056e+08	1999.000000	293.000000
47.0	2.204049e+08	421.000000	345.000000
48.0	7.844387e+08	1382.000000	1070.500000
49.0	7.325864e+08	159.000000	145.000000
50.0	4.611888e+08	438.000000	174.000000
53.0	5.633389e+08	3131.500000	2215.000000
57.0	1.255659e+08	192.000000	235.000000
62.0	1.160946e+09	11873.000000	69.000000
64.0	2.961910e+07	40543.000000	116.000000

	actor7:listedCount	actor8:statusesCount
gnip:klout_score		
0.0	1.000000	12236.000000
18.0	0.000000	540.000000
28.0	3.000000	2151.000000
29.0	0.000000	6252.000000
30.0	0.500000	11842.500000
31.0	1.000000	2417.000000
32.0	0.000000	468.000000
33.0	2.800000	6981.200000
34.0	0.000000	19153.000000
35.0	0.333333	14456.666667
36.0	0.000000	3136.500000
37.0	0.400000	16402.400000
38.0	0.250000	14787.250000
39.0	0.666667	8448.000000
40.0	9.285714	13609.142857
41.0	0.000000	5635.285714
42.0	1.000000	19437.333333
43.0	0.000000	4249.500000
44.0	2.857143	18960.857143
45.0	1.500000	58685.500000
46.0	27.000000	60236.000000
47.0	0.000000	15986.000000
48.0	0.500000	18208.000000
49.0	0.000000	8925.000000
50.0	1.000000	17636.000000
53.0	9.500000	25152.000000
57.0	3.000000	4073.000000
62.0	56.000000	1991.000000
64.0	486.000000	60465.000000

In [119]:
```
# though this looks like a normal dataframe, the DataFrameGroupBy object has a
heirarchical index
#  this means it may not act as you might expect.
lang_gb = pop_df[['twitter_lang',\
             'gnip:klout_score',\
             'actor5:followersCount',\
             'actor6:friendsCount',\
             ]].groupby('twitter_lang')


# note the new index 'twitter_lang' -- in this case, .head(n) returns <= n ele
ments for each index
lang_gb.head(2)

# see that they type is DataFrameGroupBy object
#lang_gb
```

Out[119]:

	twitter_lang	gnip:klout_score	actor5:followersCount	actor6:friendsCount
0	tr	35.0	178	129
3	es	50.0	438	174
5	tr	41.0	226	346
6	pt	42.0	247	64
8	en	38.0	380	860
9	en	41.0	160	135
10	es	62.0	11873	69
16	und	53.0	1179	628
18	he	37.0	151	284
20	it	46.0	1999	293
22	id	43.0	258	302
24	pl	48.0	2037	1984
27	pt	38.0	231	352
29	id	33.0	404	378
33	ko	39.0	170	120
38	und	35.0	228	185
42	ar	40.0	458	413
64	ja	28.0	152	335
79	lv	41.0	297	237
88	vi	44.0	767	486
90	it	53.0	5084	3802

```
In [122]: # to get a DataFrame object that responds more like I'm used to, create a new
           one using the
          #   aggregate method, which results in a single-index DataFrame
          lang_gb_mean = lang_gb.aggregate(np.mean)

          lang_gb_mean.head()

          # verify the single index
          #lang_gb_mean.index
```

Out[122]:

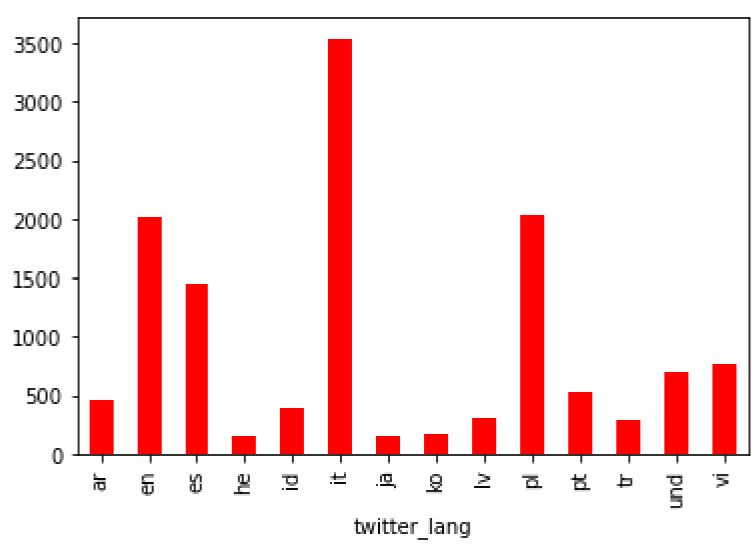
	gnip:klout_score	actor5:followersCount	actor6:friendsCount
twitter_lang			
ar	40.000000	458.000000	413.000000
en	39.400000	2019.633333	635.666667
es	40.076923	1452.769231	458.538462
he	37.000000	151.000000	284.000000
id	43.250000	387.000000	245.500000

part 3: plotting

As far as I can tell, plotting in Python was not fun in the past. Below is some easy, base matplotlib, but 'nice' graphics take *a lot* of code. This situation is changing quite quickly now, with the success of `ggplot2` in the R world and the attempts to a) make `matplotlib` look less sucky, and b) implement the Grammar of Graphics in Python.

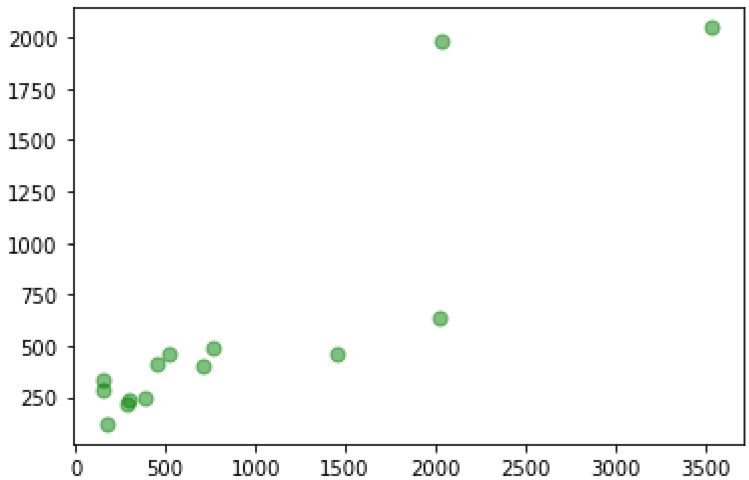
```
In [130]: # .plot() is a pandas wrapper for matplotlib's plt.plot()
          lang_gb_mean['actor5:followersCount'].plot(kind='bar', color='r')
```

```
Out[130]: <matplotlib.axes._subplots.AxesSubplot at 0x1fb5099b8b0>
```



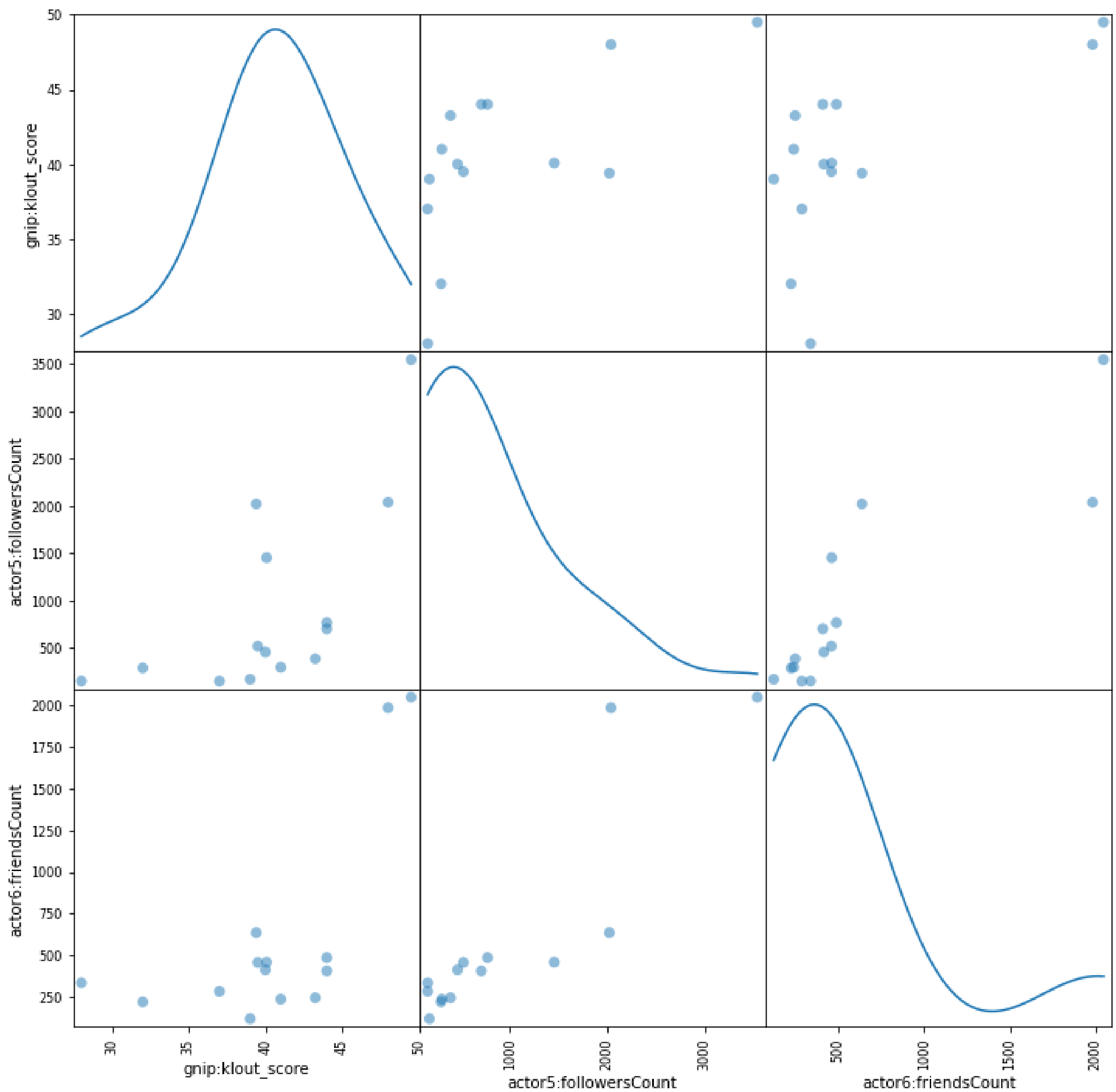
In [134]:
```python
# more base matplotlib
plt.scatter(x=lang_gb_mean['actor5:followersCount'],\
            y=lang_gb_mean['actor6:friendsCount'],\
            alpha=0.5,\
            s=50,\
            color='green',\
            marker='o')
```

Out[134]:
```
<matplotlib.collections.PathCollection at 0x1fb4f72dbe0>
```



In [136]:
```
# now read the docs and copypasta a neat-looking plot
from pandas.plotting import scatter_matrix

scatter_matrix(lang_gb_mean, alpha=0.5, figsize=(12,12), diagonal='kde', s=200
)
```

Out[136]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000001FB51C2E3A0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001FB51F32B20>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001FB5312FF70
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x000001FB53167400>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001FB53192850>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001FB531BFBE0
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x000001FB50E79670>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001FB51C2EC70>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x000001FB51C92BE0
>]],
      dtype=object)
```



Finally, a short taste of some other plotting libraries. My munging + plotting skillz in this world are still a work in progress, so I will definitely return to this section with an actual use-case in the future. For now, we'll make up some data for illustrative purposes.

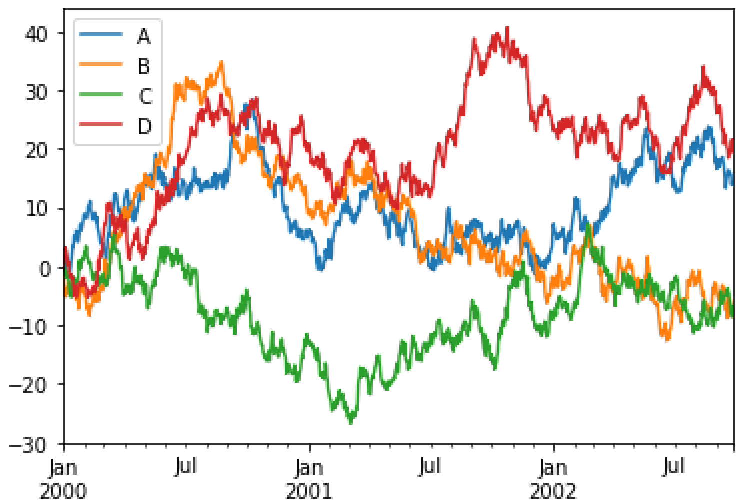
In [137]:
```
# make up some data with large-scale patterns and a datetime index
df = pd.DataFrame(np.random.randn(1000, 4), index=pd.date_range('1/1/2000', pe
riods=1000), columns=list('ABCD'))
df = df.cumsum()
df.head()
```

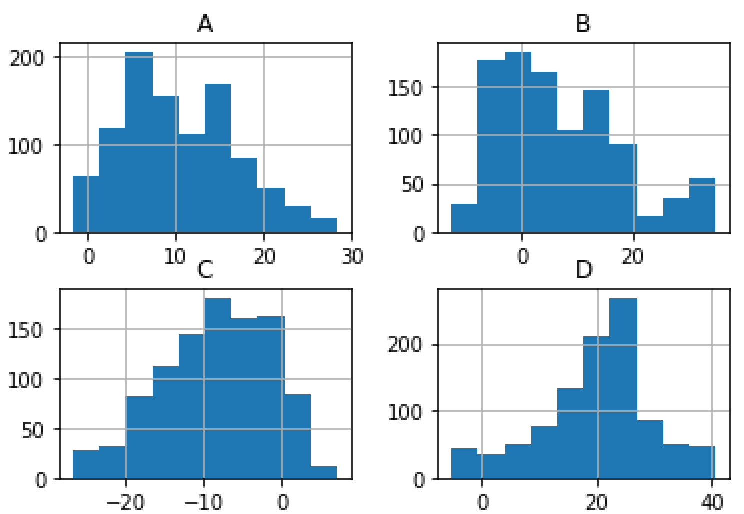
Out[137]:

	A	B	C	D
2000-01-01	-0.064655	-0.503366	-2.004935	-0.077975
2000-01-02	-0.196966	-3.354750	1.259366	0.147783
2000-01-03	-0.531685	-4.268116	1.313652	-0.571963
2000-01-04	0.280180	-5.542810	0.421919	-0.173888
2000-01-05	-0.348305	-6.217966	0.524750	-0.404553

```
In [36]: df.plot()
         df.hist()
```

```
Out[36]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x11c498518>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1e8c0ba8>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1a1e904128>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1e93d128>]],
               dtype=object)
```





Visualizing the distribution of a dataset

When dealing with a set of data, often the first thing you'll want to do is get a sense for how the variables are distributed. This chapter of the tutorial will give a brief introduction to some of the tools in seaborn for examining univariate and bivariate distributions. You may also want to look at the :ref: `categorical plots <categorical_tutorial>` chapter for examples of functions that make it easy to compare the distribution of a variable across levels of other variables.

```
In [37]: %matplotlib inline
```

In [38]:
```
import numpy as np
import pandas as pd
from scipy import stats, integrate
import matplotlib.pyplot as plt
```

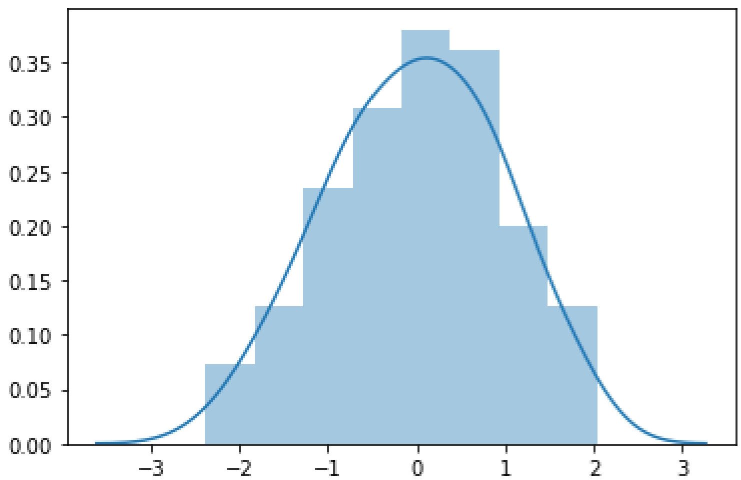
In [39]:
```
import seaborn as sns
sns.set(color_codes=True)
```

In [40]:
```
np.random.seed(sum(map(ord, "distributions")))
```

Plotting univariate distributions

The most convenient way to take a quick look at a univariate distribution in seaborn is the *** `distplot` function. By default, this will draw a `histogram <https://en.wikipedia.org/wiki/Histogram>` _ and fit a `kernel density estimate <https://en.wikipedia.org/wiki/Kernel_density_estimation>` _ (KDE).

In [142]:
```
import seaborn as sns
x = np.random.normal(size=100)
sns.distplot(x);
```

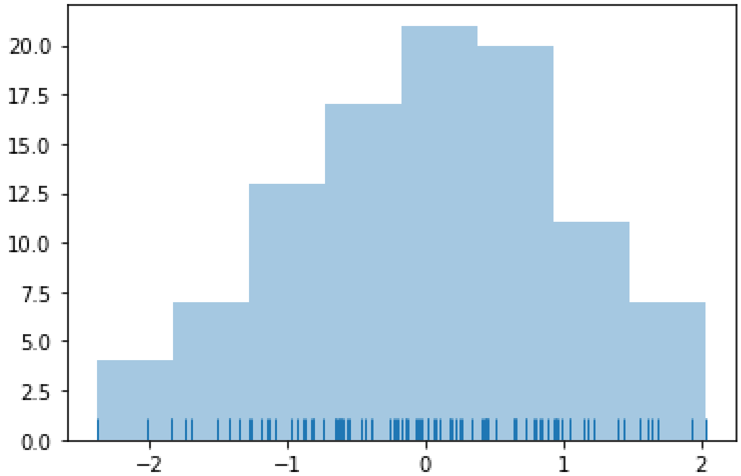


Histograms

Histograms are likely familiar, and a **`hist`** function already exists in matplotlib. A histogram represents the distribution of data by forming bins along the range of the data and then drawing bars to show the number of observations that fall in each bin.

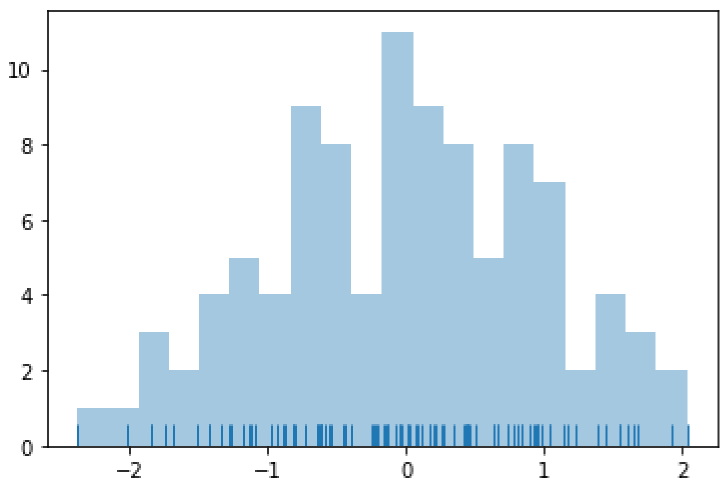
To illustrate this, let's remove the density curve and add a rug plot, which draws a small vertical tick at each observation. You can make the rug plot itself with the **`rugplot`** function, but it is also available in **`distplot`** :

```
In [146]: sns.distplot(x, kde=False, rug=True);
```



When drawing histograms, the main choice you have is the number of bins to use and where to place them. **`distplot`** uses a simple rule to make a good guess for what the right number is by default, but trying more or fewer bins might reveal other features in the data:

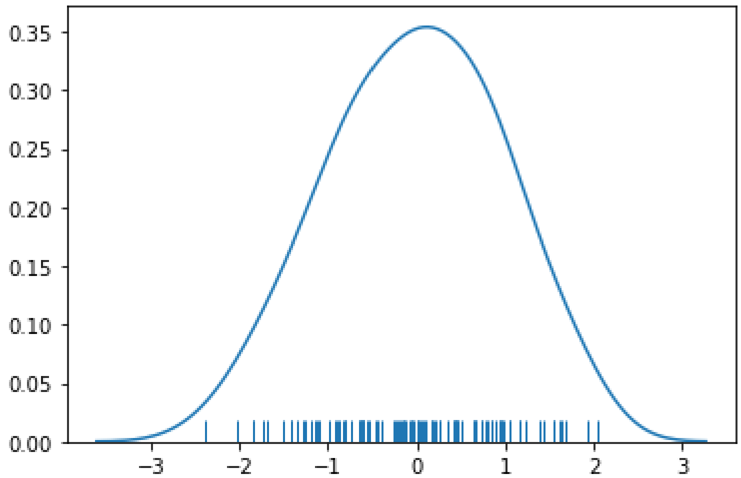
```
In [147]: sns.distplot(x, bins=20, kde=False, rug=True);
```



Kernel density estimaton

The kernel density estimate may be less familiar, but it can be a useful tool for plotting the shape of a distribution. Like the histogram, the KDE plots encodes the density of observations on one axis with height along the other axis:

In [148]:
```
sns.distplot(x, hist=False, rug=True);
```

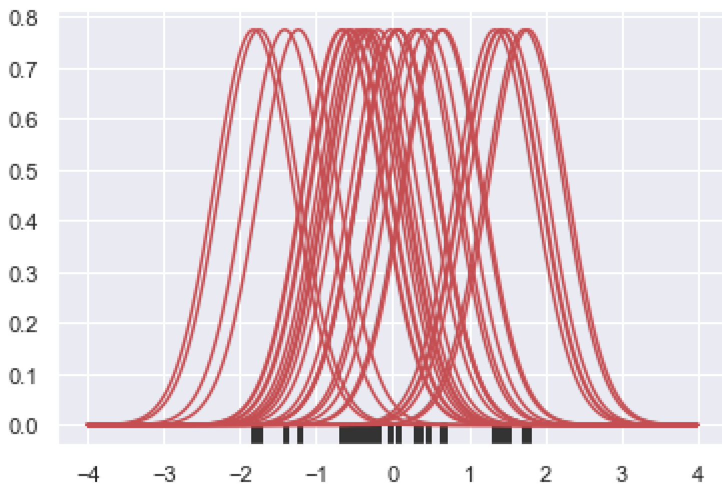


Drawing a KDE is more computationally involved than drawing a histogram. What happens is that each observation is first replaced with a normal (Gaussian) curve centered at that value:

In [151]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

In [161]:

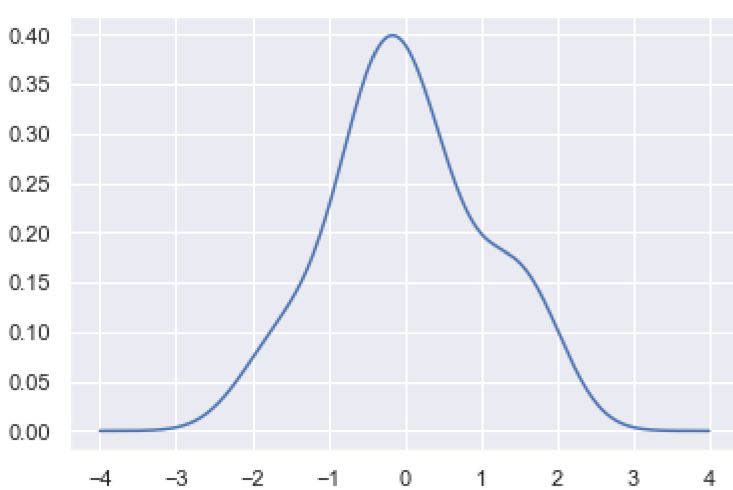
```
from scipy.stats import norm
x = np.random.normal(0, 1, size=30)
bandwidth = 1.06 * x.std() * x.size ** (-1 / 5.)
support = np.linspace(-4, 4, 200)

kernels = []
for x_i in x:

    kernel = norm(x_i, bandwidth).pdf(support)
    kernels.append(kernel)
    plt.plot(support, kernel, color="r")

sns.rugplot(x, color=".2", linewidth=3);
```



Next, these curves are summed to compute the value of the density at each point in the support grid. The resulting curve is then normalized so that the area under it is equal to 1:

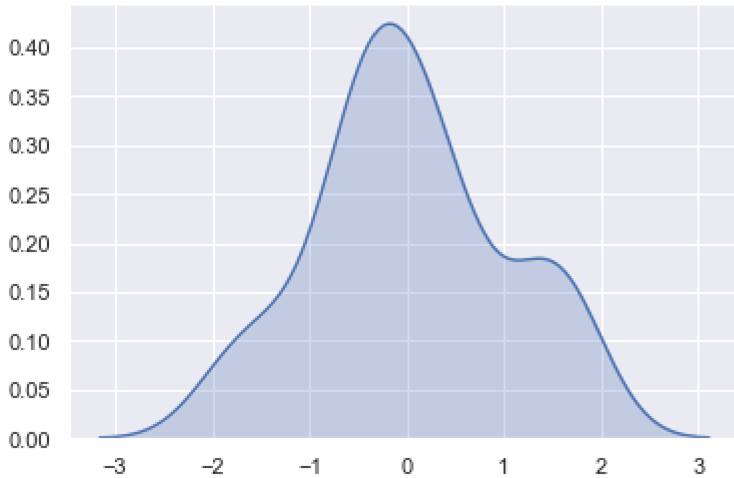
In [167]:

```
from scipy import integrate
density = np.sum(kernels, axis=0)
density /= integrate.trapz(density, support)
plt.plot(support, density);
```



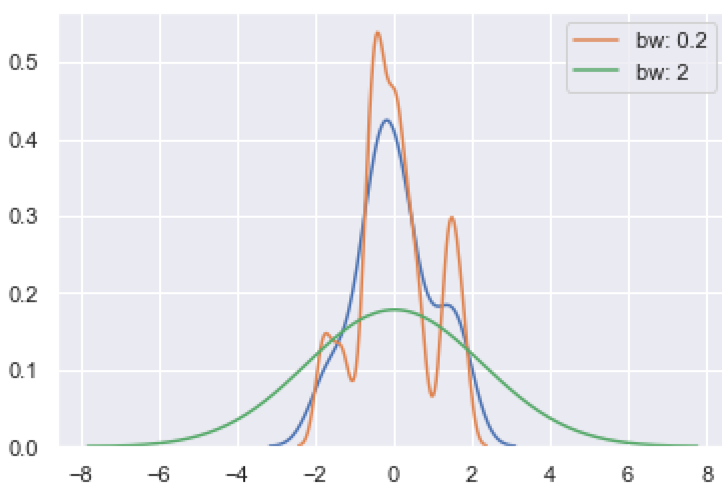
We can see that if we use the `kdeplot` function in seaborn, we get the same curve. This function is used by `distplot`, but it provides a more direct interface with easier access to other options when you just want the density estimate:

In [168]:
```
sns.kdeplot(x, shade=True);
```



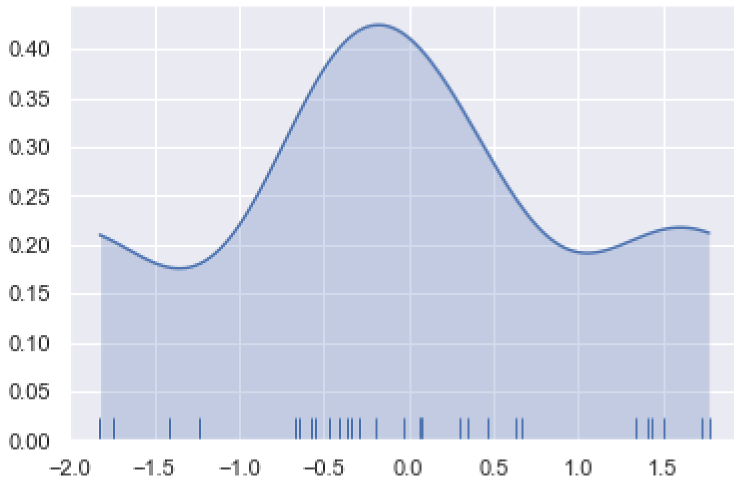
The bandwidth (`bw`) parameter of the KDE controls how tightly the estimation is fit to the data, much like the bin size in a histogram. It corresponds to the width of the kernels we plotted above. The default behavior tries to guess a good value using a common reference rule, but it may be helpful to try larger or smaller values:

In [169]:
```
sns.kdeplot(x)
sns.kdeplot(x, bw=.2, label="bw: 0.2")
sns.kdeplot(x, bw=2, label="bw: 2")
plt.legend();
```



As you can see above, the nature of the Gaussian KDE process means that estimation extends past the largest and smallest values in the dataset. It's possible to control how far past the extreme values the curve is drawn with the `cut` parameter; however, this only influences how the curve is drawn and not how it is fit:

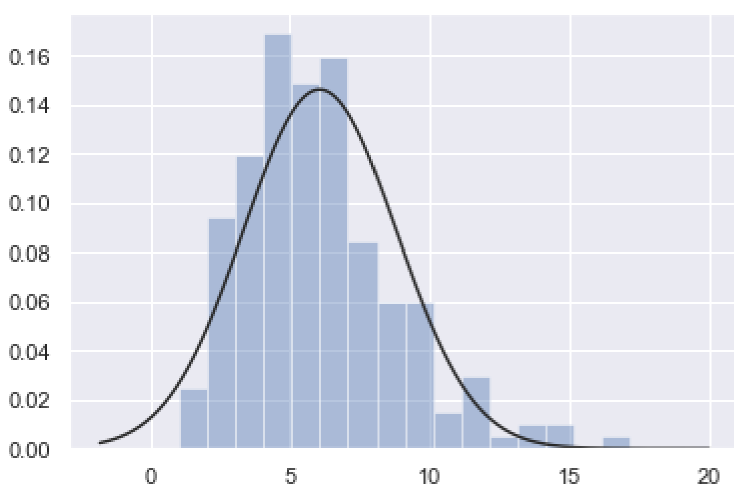
In [170]:
```
sns.kdeplot(x, shade=True, cut=0)
sns.rugplot(x);
```



Fitting parametric distributions

You can also use `distplot` to fit a parametric distribution to a dataset and visually evaluate how closely it corresponds to the observed data:

In [175]:
```
from scipy.stats import norm
import seaborn as sns, numpy as np
x = np.random.gamma(6, size=200)
sns.distplot(x, kde=False, fit=norm);
```



Plotting bivariate distributions

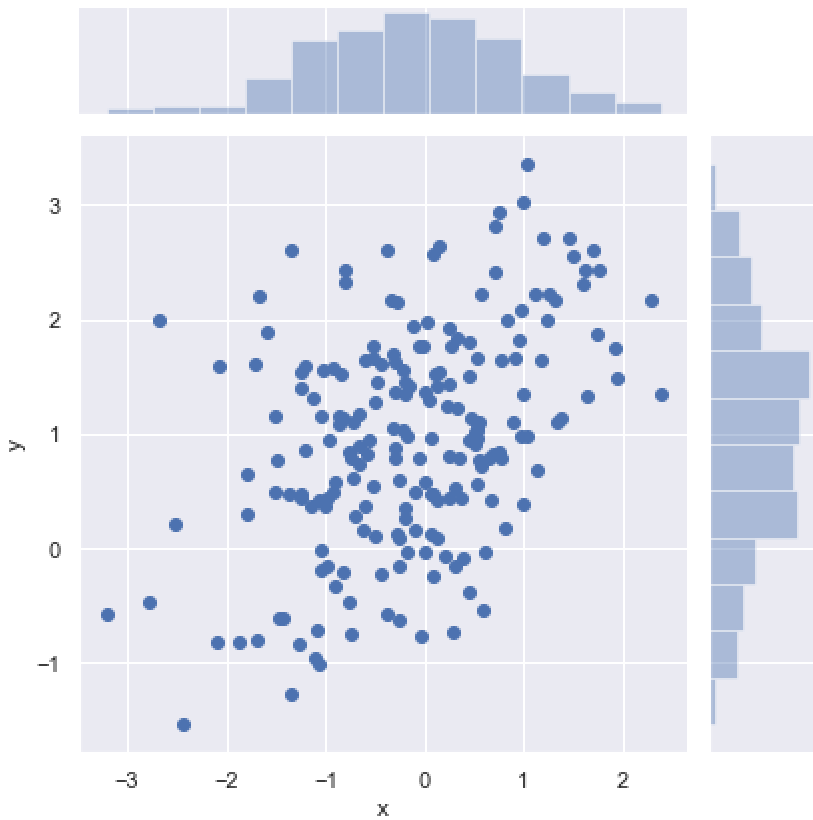
It can also be useful to visualize a bivariate distribution of two variables. The easiest way to do this in seaborn is to just use the *** `jointplot` function, which creates a multi-panel figure that shows both the bivariate (or joint) relationship between two variables along with the univariate (or marginal) distribution of each on separate axes.

In [176]:
```
mean, cov = [0, 1], [(1, .5), (.5, 1)]
data = np.random.multivariate_normal(mean, cov, 200)
df = pd.DataFrame(data, columns=["x", "y"])
```

Scatterplots

The most familiar way to visualize a bivariate distribution is a scatterplot, where each observation is shown with point at the *x* and *y* values. This is analgous to a rug plot on two dimensions. You can draw a scatterplot with the matplotlib **`plt.scatter`** function, and it is also the default kind of plot shown by the **`jointplot`** function:

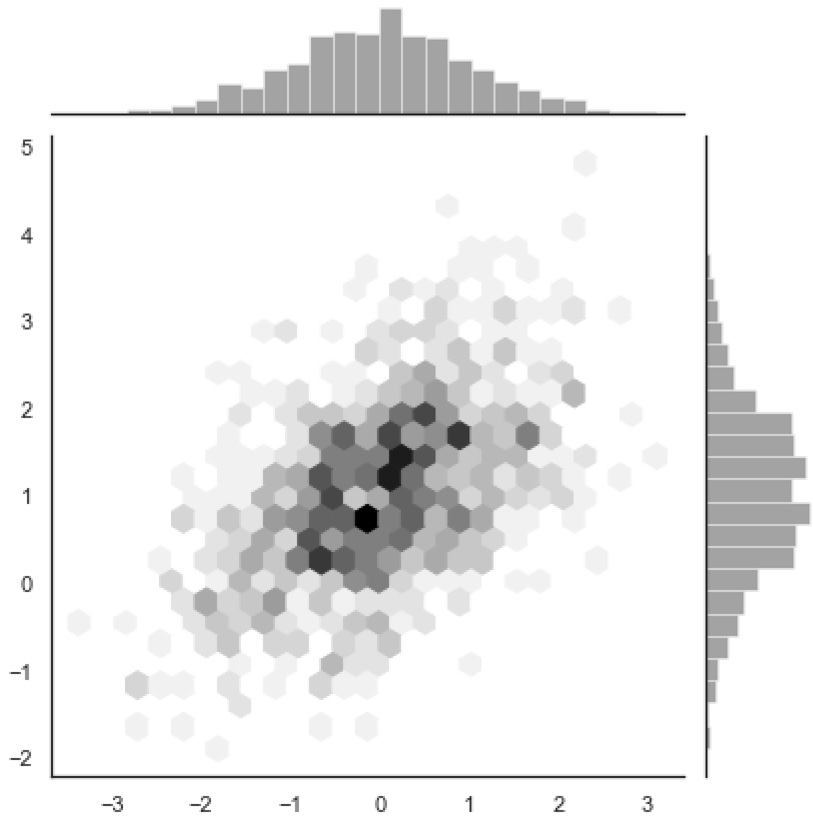
In [177]:
```
sns.jointplot(x="x", y="y", data=df);
```



Hexbin plots

The bivariate analogue of a histogram is known as a **"hexbin"** plot, because it shows the counts of observations that fall within hexagonal bins. This plot works best with relatively large datasets. It's available through the matplotlib **`plt.hexbin`** function and as a style in **`jointplot`** . It looks best with a white background:

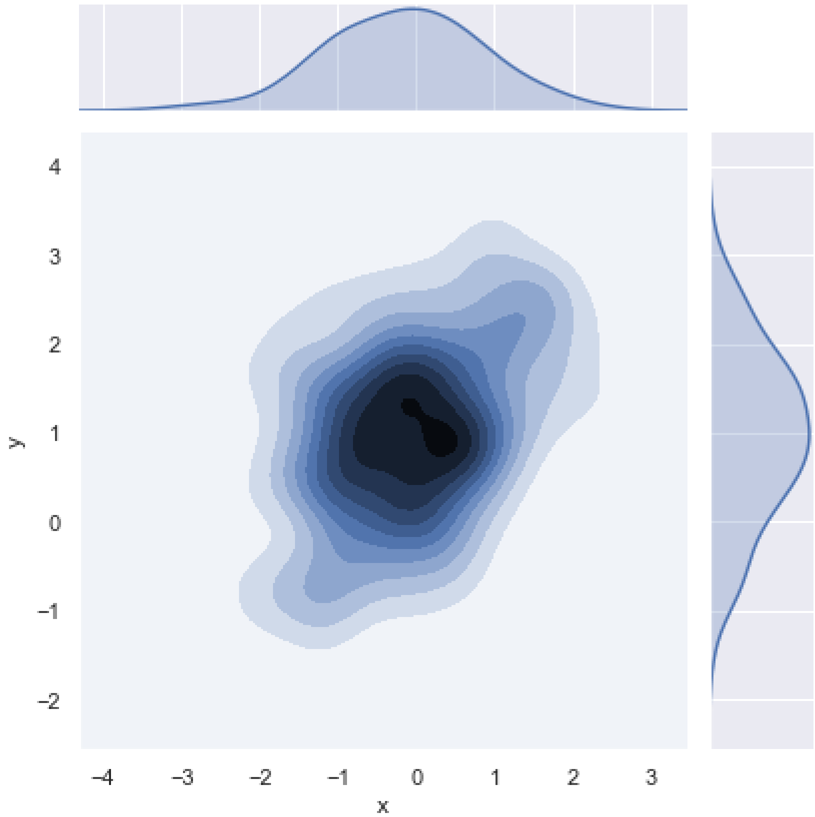
In [178]:
```
x, y = np.random.multivariate_normal(mean, cov, 1000).T
with sns.axes_style("white"):
    sns.jointplot(x=x, y=y, kind="hex", color="k");
```



Kernel density estimation

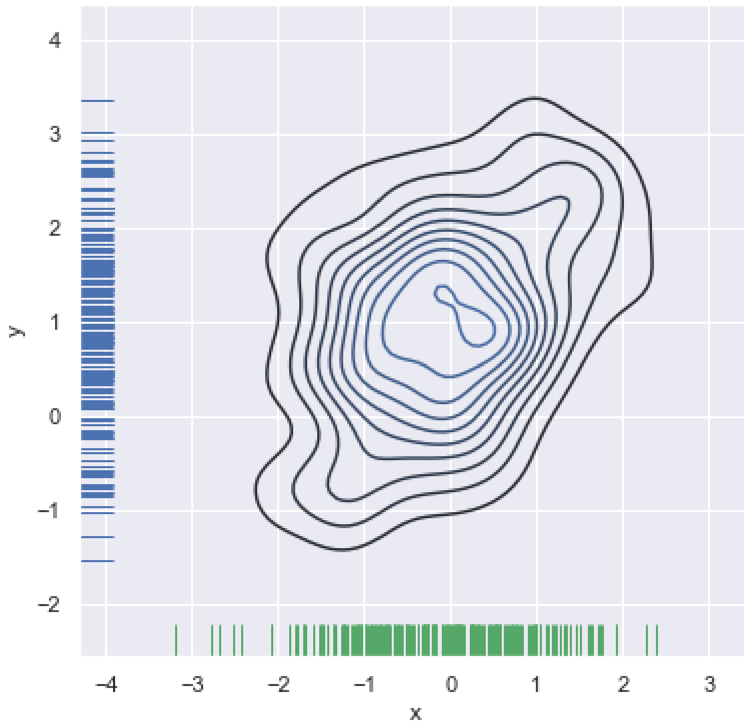
It is also posible to use the kernel density estimation procedure described above to visualize a bivariate distribution. In seaborn, this kind of plot is shown with a contour plot and is available as a style in `jointplot` :

In [179]:
```
sns.jointplot(x="x", y="y", data=df, kind="kde");
```



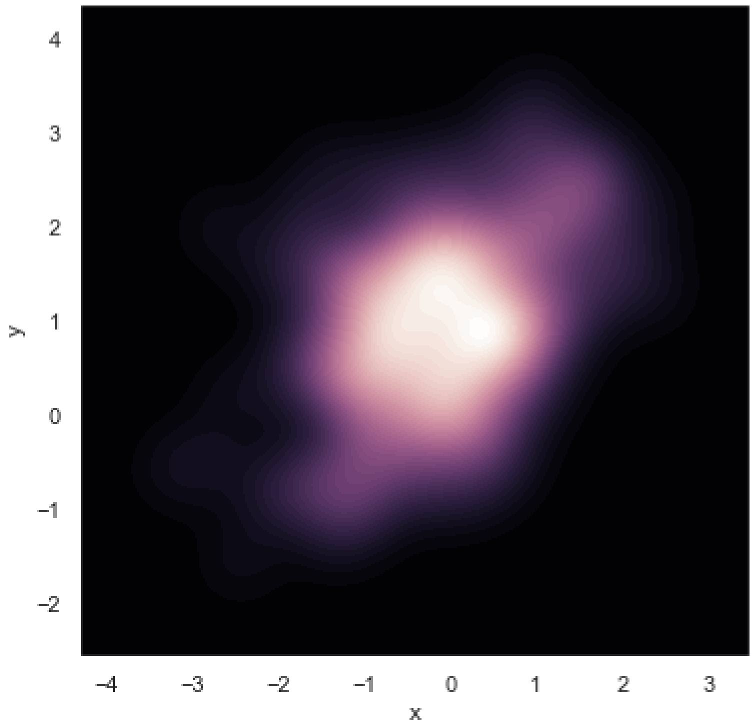
You can also draw a two-dimensional kernel density plot with the **`kdeplot`** function. This allows you to draw this kind of plot onto a specific (and possibly already existing) matplotlib axes, whereas the **`jointplot`** function manages its own figure:

In [180]:
```
f, ax = plt.subplots(figsize=(6, 6))
sns.kdeplot(df.x, df.y, ax=ax)
sns.rugplot(df.x, color="g", ax=ax)
sns.rugplot(df.y, vertical=True, ax=ax);
```



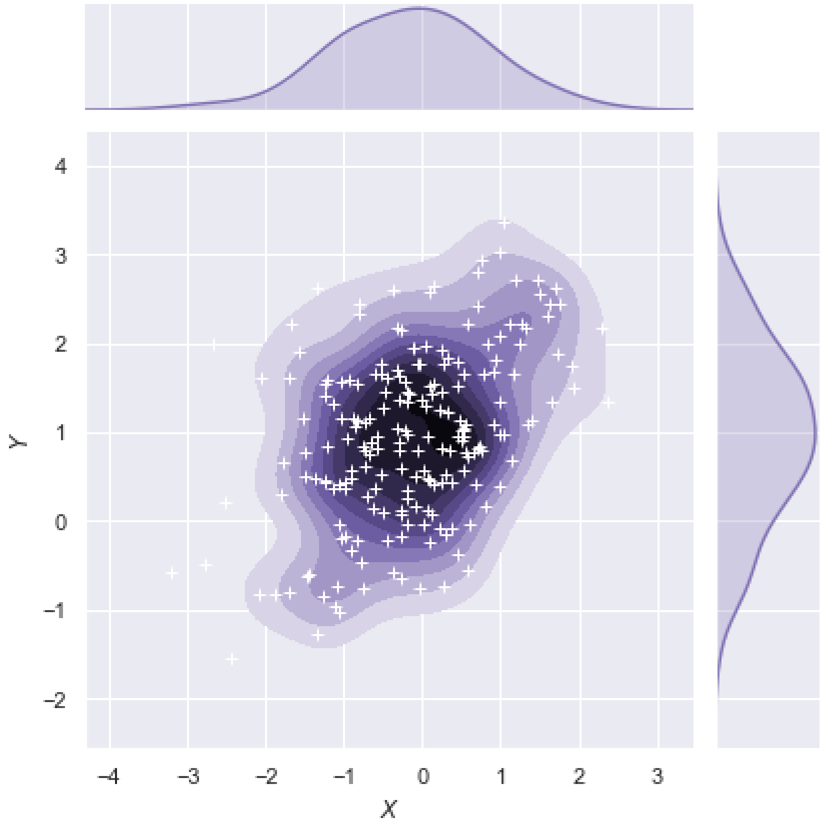
If you wish to show the bivariate density more continuously, you can simply increase the number of contour levels:

In [181]:
```
f, ax = plt.subplots(figsize=(6, 6))
cmap = sns.cubehelix_palette(as_cmap=True, dark=0, light=1, reverse=True)
sns.kdeplot(df.x, df.y, cmap=cmap, n_levels=60, shade=True);
```



The **`jointplot`** function uses a **`JointGrid`** to manage the figure. For more flexibility, you may want to draw your figure by using **`JointGrid`** directly. **`jointplot` returns the** `JointGrid` ** object after plotting, which you can use to add more layers or to tweak other aspects of the visualization:

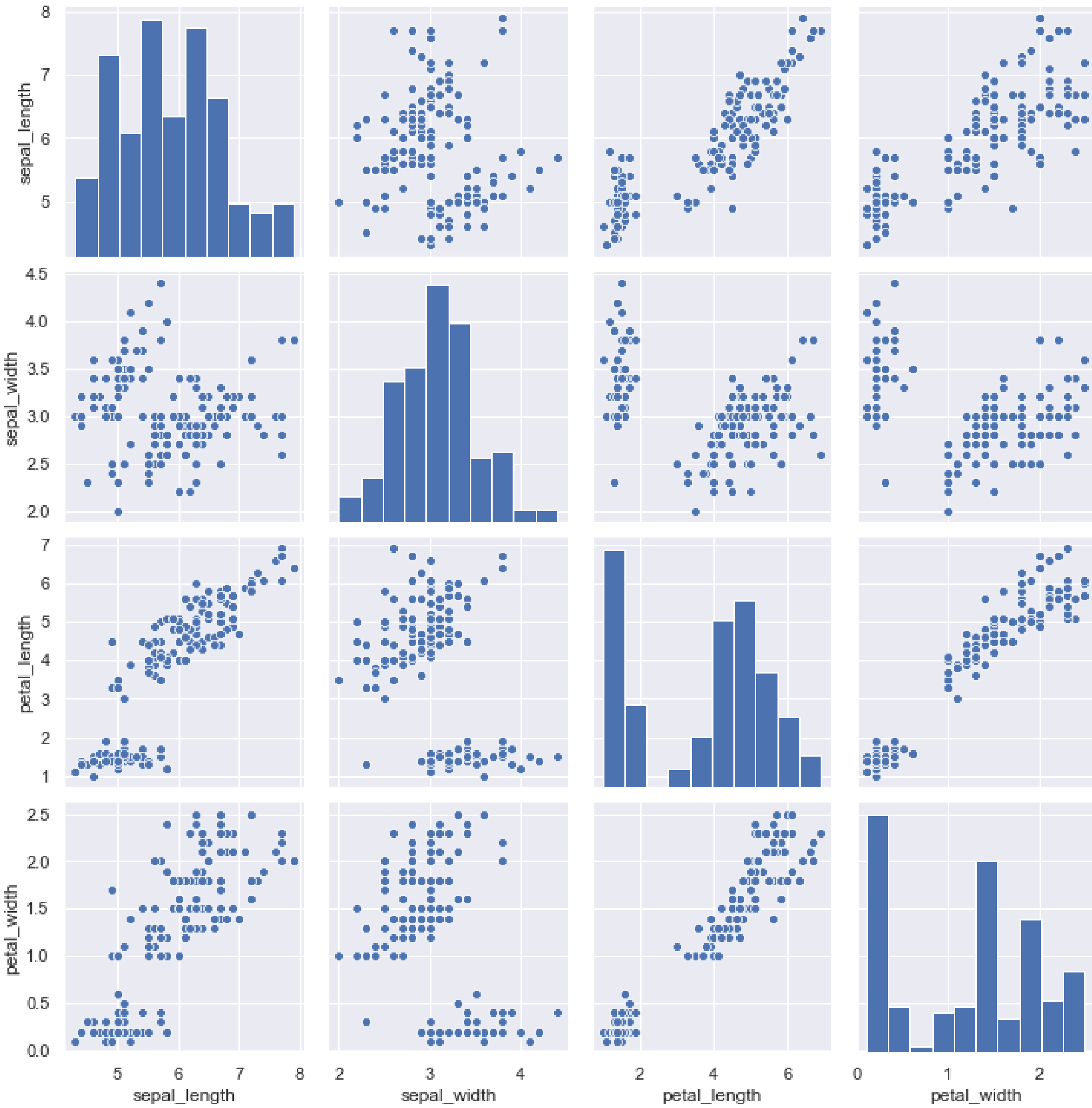
In [182]:
```
g = sns.jointplot(x="x", y="y", data=df, kind="kde", color="m")
g.plot_joint(plt.scatter, c="w", s=30, linewidth=1, marker="+")
g.ax_joint.collections[0].set_alpha(0)
g.set_axis_labels("$X$", "$Y$");
```



Visualizing pairwise relationships in a dataset

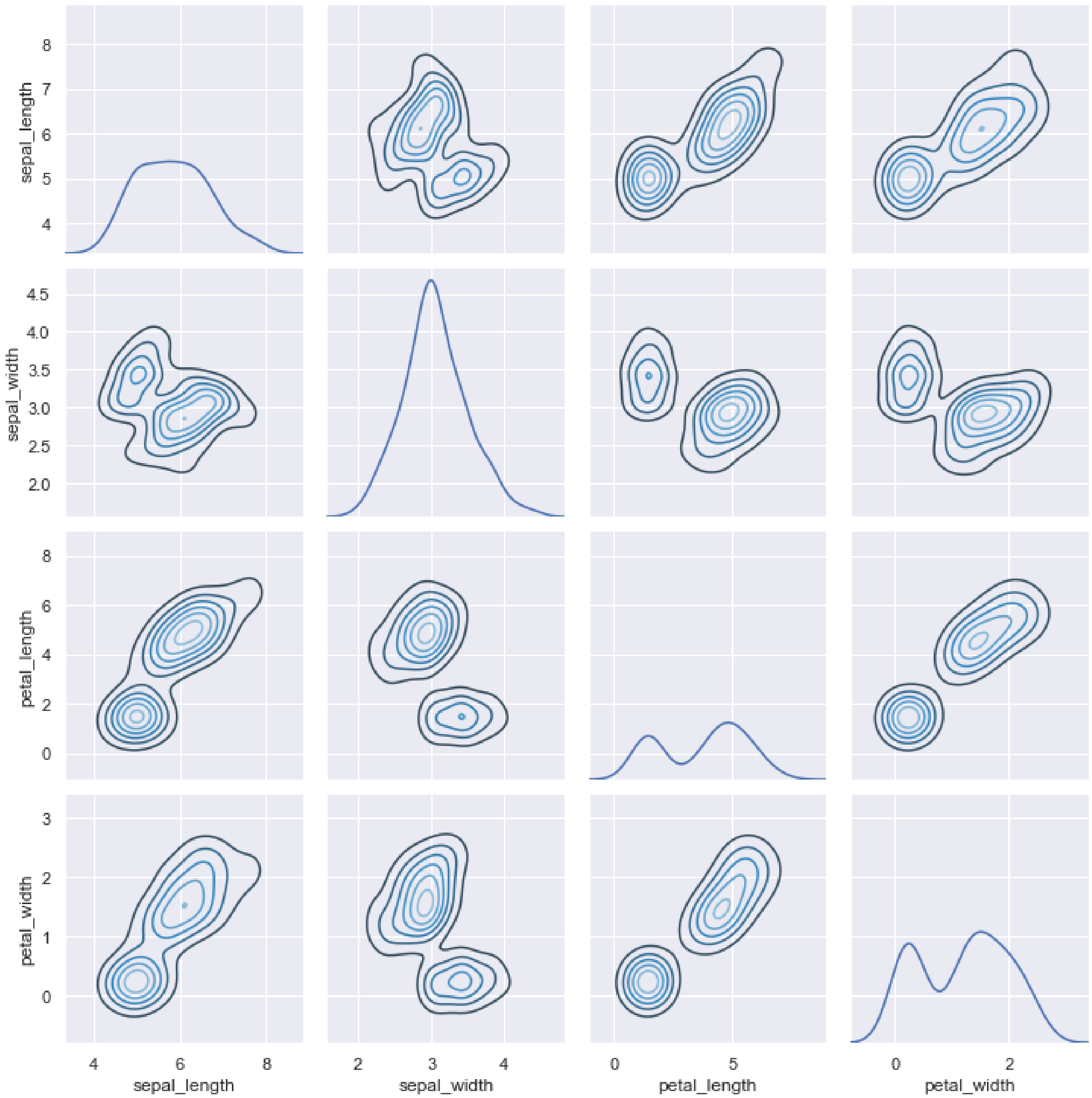
To plot multiple pairwise bivariate distributions in a dataset, you can use the **`pairplot`** function. This creates a matrix of axes and shows the relationship for each pair of columns in a DataFrame. By default, it also draws the univariate distribution of each variable on the diagonal Axes:

In [190]:
```
import seaborn as sns
iris = sns.load_dataset("iris")
sns.pairplot(iris);
```



Much like the relationship between **`jointplot`** and **`JointGrid`**, the **`pairplot`** function is built on top of a **`PairGrid`** object, which can be used directly for more flexibility:

In [192]:
```
g = sns.PairGrid(iris)
g.map_diag(sns.kdeplot)
g.map_offdiag(sns.kdeplot, cmap="Blues_d", n_levels=6);
```



In [ ]:
```

```

Visualizing linear relationships

In [60]:
```
%matplotlib inline
```

In [61]:
```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
```

In [193]:
```
import seaborn as sns
sns.set(color_codes=True)
```

In [194]:
```
np.random.seed(sum(map(ord, "regression")))
```

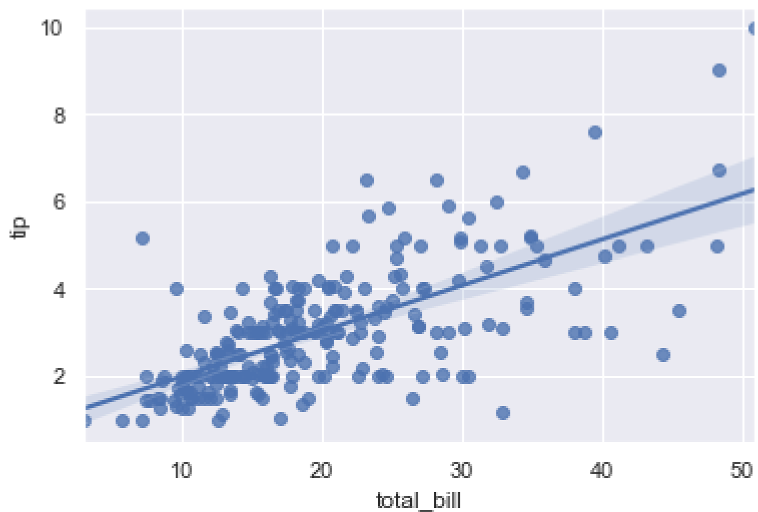
In [195]:
```
tips = sns.load_dataset("tips")
```

Functions to draw linear regression models

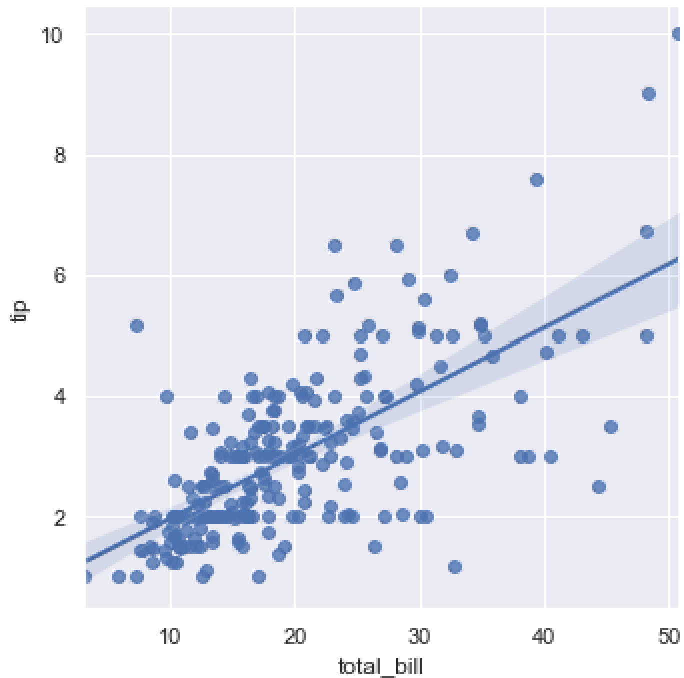
Two main functions in seaborn are used to visualize a linear relationship as determined through regression. These functions, **`regplot`** and **`lmplot`** are closely related, and share much of their core functionality. It is important to understand the ways they differ, however, so that you can quickly choose the correct tool for particular job.

In the simplest invocation, both functions draw a scatterplot of two variables, `x` and `y` , and then fit the regression model `y ~ x` and plot the resulting regression line and a 95% confidence interval for that regression:

In [196]:
```
sns.regplot(x="total_bill", y="tip", data=tips);
```



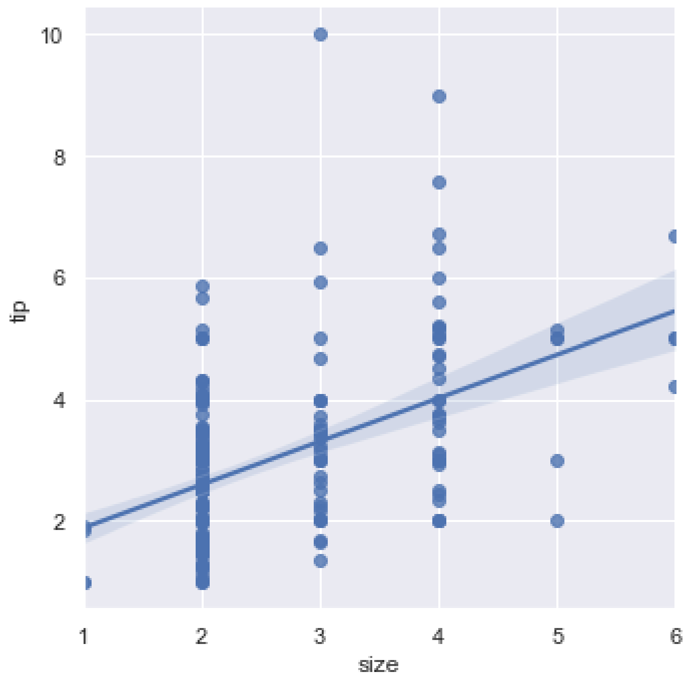
In [197]:
```
sns.lmplot(x="total_bill", y="tip", data=tips);
```



You should note that the resulting plots are identical, except that the figure shapes are different. We will explain why this is shortly. For now, the other main difference to know about is that **`regplot`** accepts the `x` and `y` variables in a variety of formats including simple numpy arrays, pandas `Series` objects, or as references to variables in a pandas `DataFrame` object passed to `data`. In contrast, **`lmplot`** has `data` as a required parameter and the `x` and `y` variables must be specified as strings. This data format is called "long-form" or `"tidy" <http://vita.had.co.nz/papers/tidy-data.pdf>` _ data. Other than this input flexibility, **`regplot`** possesses a subset of **`lmplot`** 's features, so we will demonstrate them using the latter.

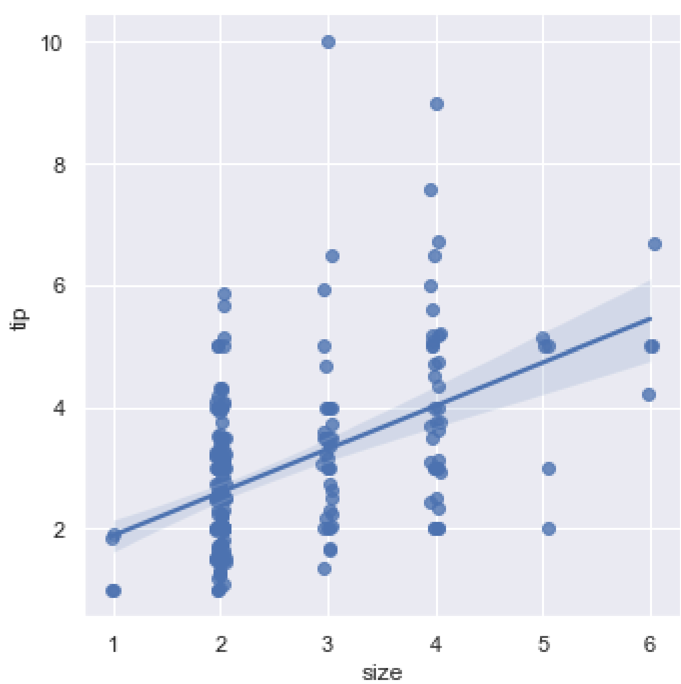
It's possible to fit a linear regression when one of the variables takes discrete values, however, the simple scatterplot produced by this kind of dataset is often not optimal:

```
In [203]: sns.lmplot(x="size", y="tip", data=tips);
```



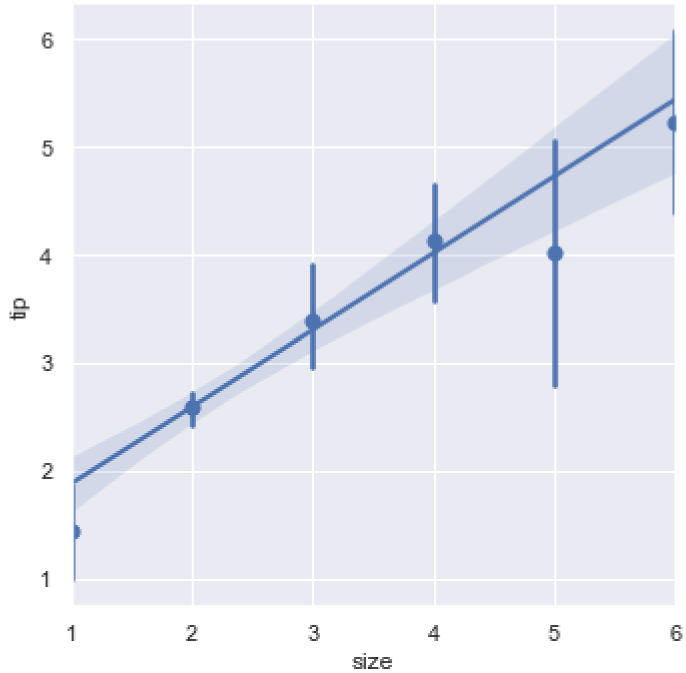
One option is to add some random noise ("jitter") to the discrete values to make the distribution of those values more clear. Note that jitter is applied only to the scatterplot data and does not influence the regression line fit itself:

```
In [204]: sns.lmplot(x="size", y="tip", data=tips, x_jitter=.05);
```



A second option is to collapse over the observations in each discrete bin to plot an estimate of central tendency along with a confidence interval:

In [205]:
```
sns.lmplot(x="size", y="tip", data=tips, x_estimator=np.mean);
```

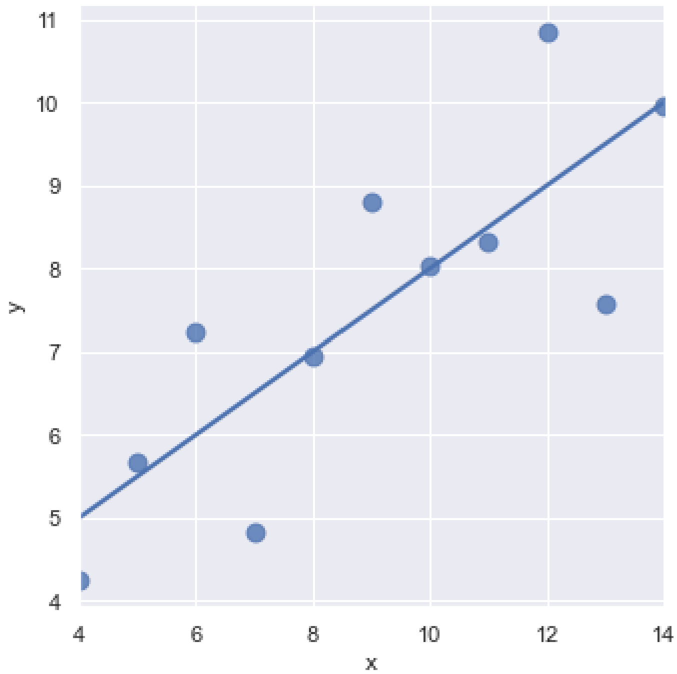


Fitting different kinds of models

The simple linear regression model used above is very simple to fit, however, it is not appropriate for some kinds of datasets. The `Anscombe's quartet <https://en.wikipedia.org/wiki/Anscombe%27s_quartet> _` dataset shows a few examples where simple linear regression provides an identical estimate of a relationship where simple visual inspection clearly shows differences. For example, in the first case, the linear regression is a good model:

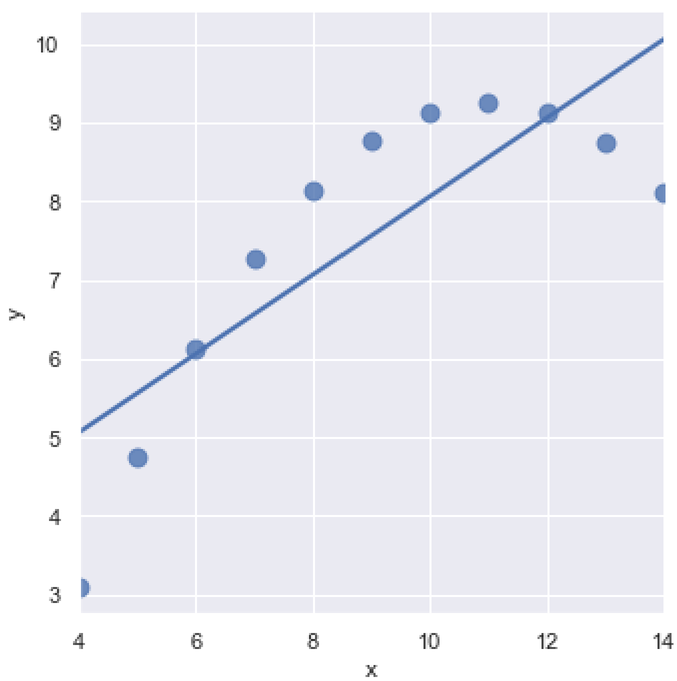
In [206]:
```
anscombe = sns.load_dataset("anscombe")
```

In [207]:
```
sns.lmplot(x="x", y="y", data=anscombe.query("dataset == 'I'"),
           ci=None, scatter_kws={"s": 80});
```



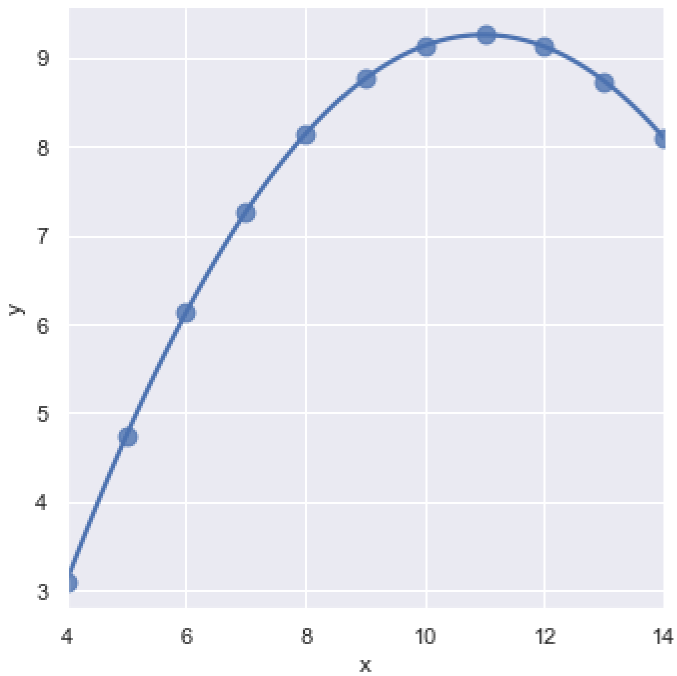
The linear relationship in the second dataset is the same, but the plot clearly shows that this is not a good model:

In [210]:
```
sns.lmplot(x="x", y="y", data=anscombe.query("dataset == 'II'"),
           robust = True, ci=None, scatter_kws={"s": 80});
```



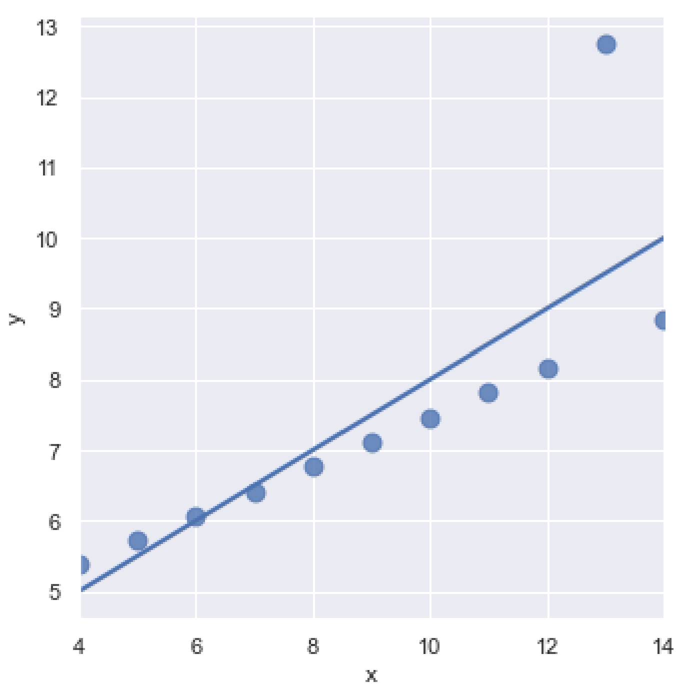
In the presence of these kind of higher-order relationships, **`lmplot`** and **`regplot`** can fit a polynomial regression model to explore simple kinds of nonlinear trends in the dataset:

In [211]:
```
sns.lmplot(x="x", y="y", data=anscombe.query("dataset == 'II'"),
           order=2, ci=None, scatter_kws={"s": 80});
```



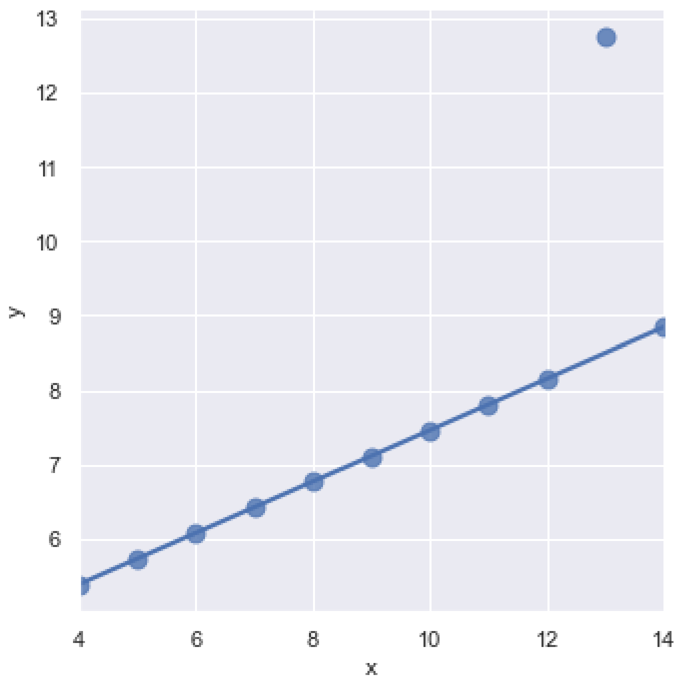
A different problem is posed by "outlier" observations that deviate for some reason other than the main relationship under study:

In [212]:
```
sns.lmplot(x="x", y="y", data=anscombe.query("dataset == 'III'"),
           ci=None, scatter_kws={"s": 80});
```



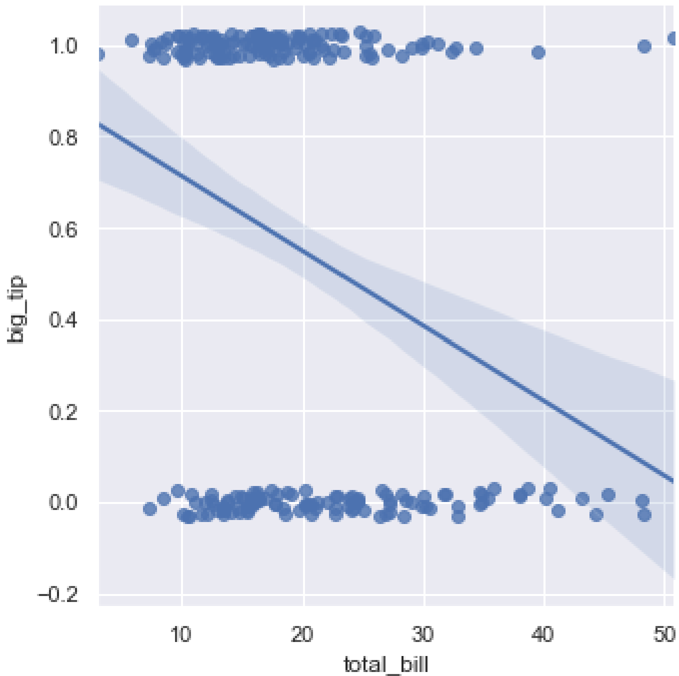
In the presence of outliers, it can be useful to fit a robust regression, which uses a different loss function to downweight relatively large residuals:

In [213]:
```
sns.lmplot(x="x", y="y", data=anscombe.query("dataset == 'III'"),
           robust = True,  ci=None, scatter_kws={"s": 80});
```



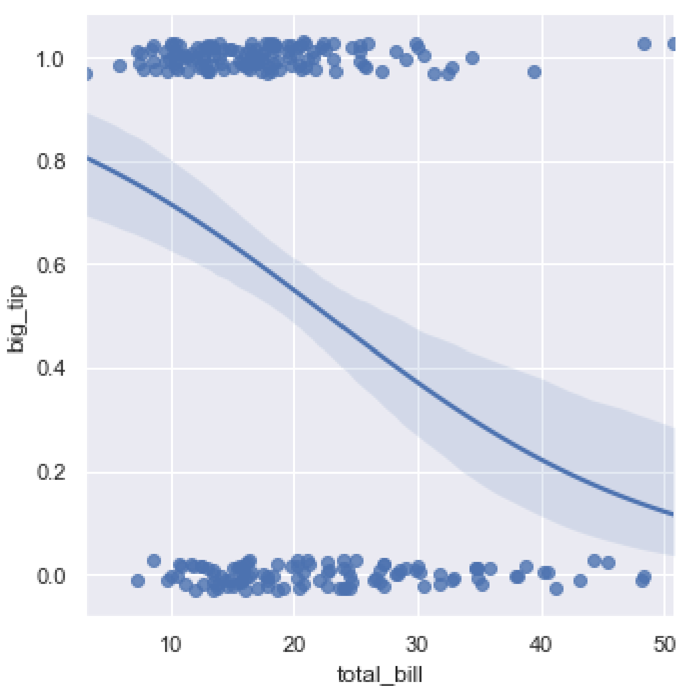
When the y variable is binary, simple linear regression also "works" but provides implausible predictions:

In [214]:
```
tips["big_tip"] = (tips.tip / tips.total_bill) > .15
sns.lmplot(x="total_bill", y="big_tip", data=tips,
           y_jitter=.03);
```



The solution in this case is to fit a logistic regression, such that the regression line shows the estimated probability of `y = 1` for a given value of `x` :

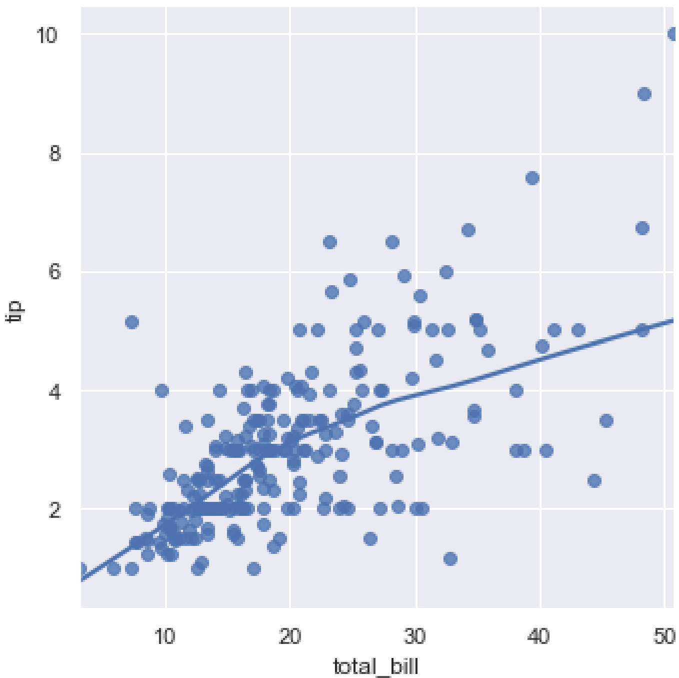
In [215]:
```
sns.lmplot(x="total_bill", y="big_tip", data=tips,
           logistic=True, y_jitter=.03);
```



Note that the logistic regression estimate is considerably more computationally intensive (this is true of robust regression as well) than simple regression, and as the confidence interval around the regression line is computed using a bootstrap procedure, you may wish to turn this off for faster iteration (using `ci=None`).

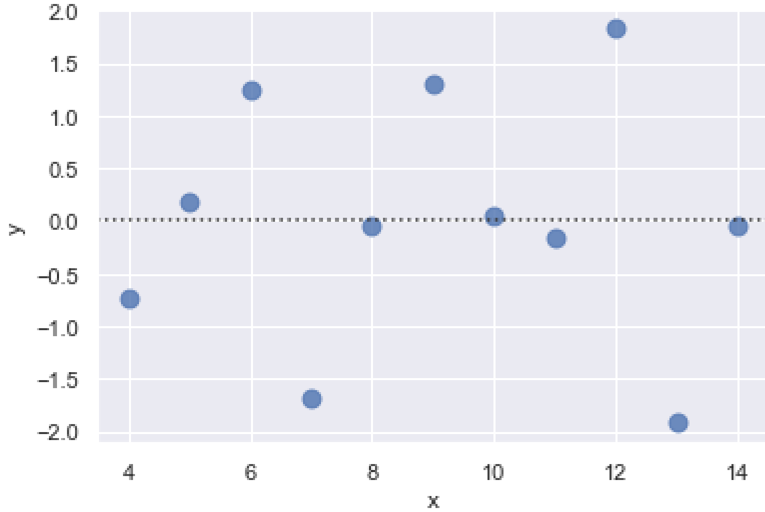
An altogether different approach is to fit a nonparametric regression using a `lowess smoother <https://en.wikipedia.org/wiki/Local_regression>` _. This approach has the fewest assumptions, although it is computationally intensive and so currently confidence intervals are not computed at all:

In [216]:
```
sns.lmplot(x="total_bill", y="tip", data=tips,
           lowess=True);
```



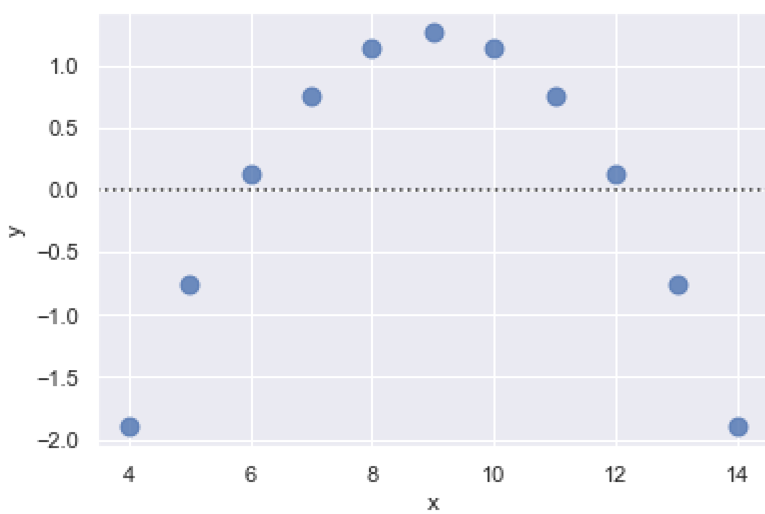
The **`residplot`** function can be a useful tool for checking whether the simple regression model is appropriate for a dataset. It fits and removes a simple linear regression and then plots the residual values for each observation. Ideally, these values should be randomly scattered around `y = 0` :

In [217]:
```
sns.residplot(x="x", y="y", data=anscombe.query("dataset == 'I'"),
              scatter_kws={"s": 80});
```



If there is structure in the residuals, it suggests that simple linear regression is not appropriate:

In [218]:
```
sns.residplot(x="x", y="y", data=anscombe.query("dataset == 'II'"),
              scatter_kws={"s": 80});
```

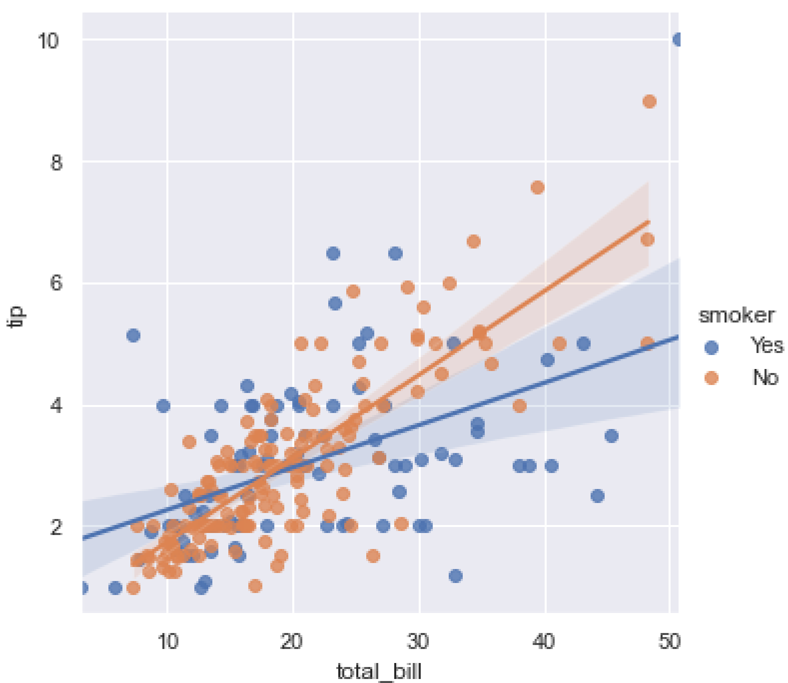


Conditioning on other variables

The plots above show many ways to explore the relationship between a pair of variables. Often, however, a more interesting question is "how does the relationship between these two variables change as a function of a third variable?" This is where the difference between `regplot` and `lmplot` appears. While `regplot` always shows a single relationship, `lmplot` combines `regplot` with `FacetGrid` to provide an easy interface to show a linear regression on "faceted" plots that allow you to explore interactions with up to three additional categorical variables.

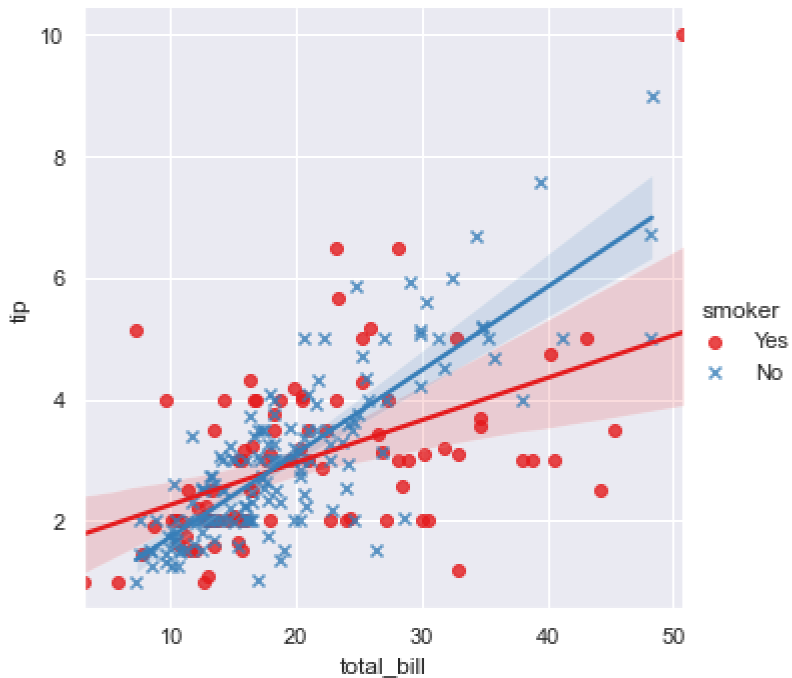
The best way to separate out a relationship is to plot both levels on the same axes and to use color to distinguish them:

In [219]:
```
sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips);
```



In addition to color, it's possible to use different scatterplot markers to make plots the reproduce to black and white better. You also have full control over the colors used:

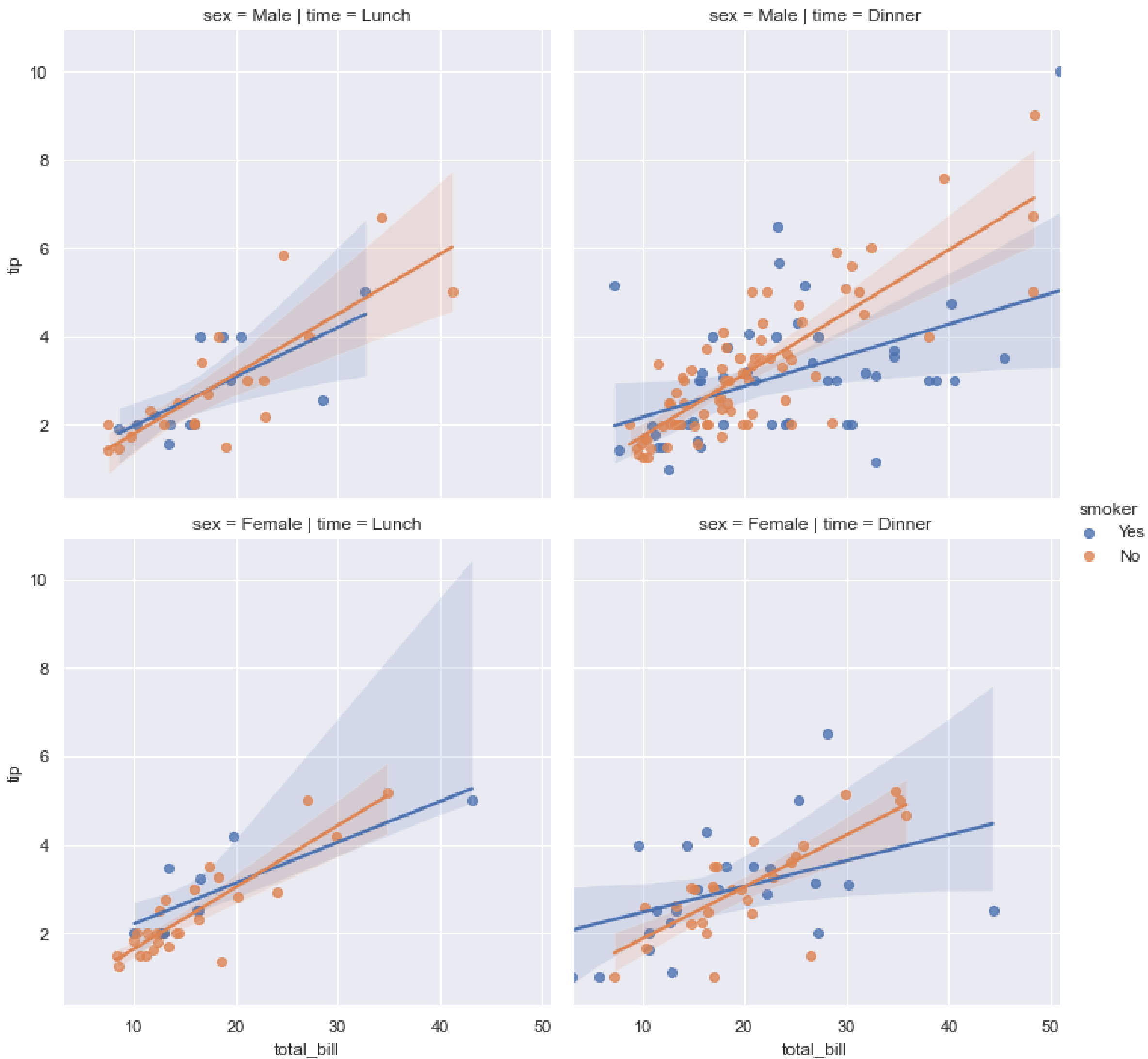
In [220]:
```
sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips,
           markers=["o", "x"], palette="Set1");
```



To add another variable, you can draw multiple "facets" which each level of the variable appearing in the rows or columns of the grid:

In [221]:
```
sns.lmplot(x="total_bill", y="tip", hue="smoker", col="time", data=tips);
```

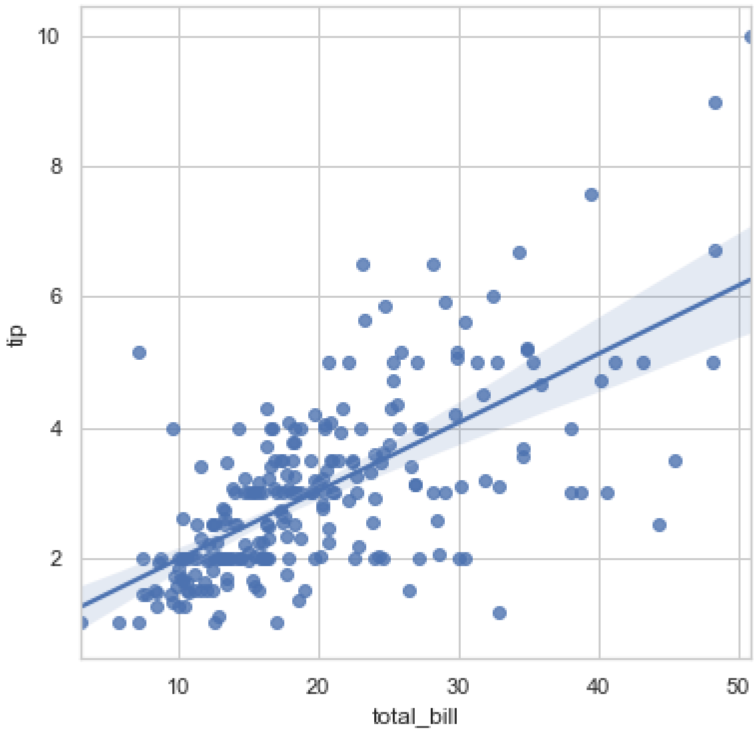
In [223]:
```
sns.lmplot(x="total_bill", y="tip", hue="smoker",
           col="time", row="sex", data=tips);
```



Controlling the size and shape of the plot

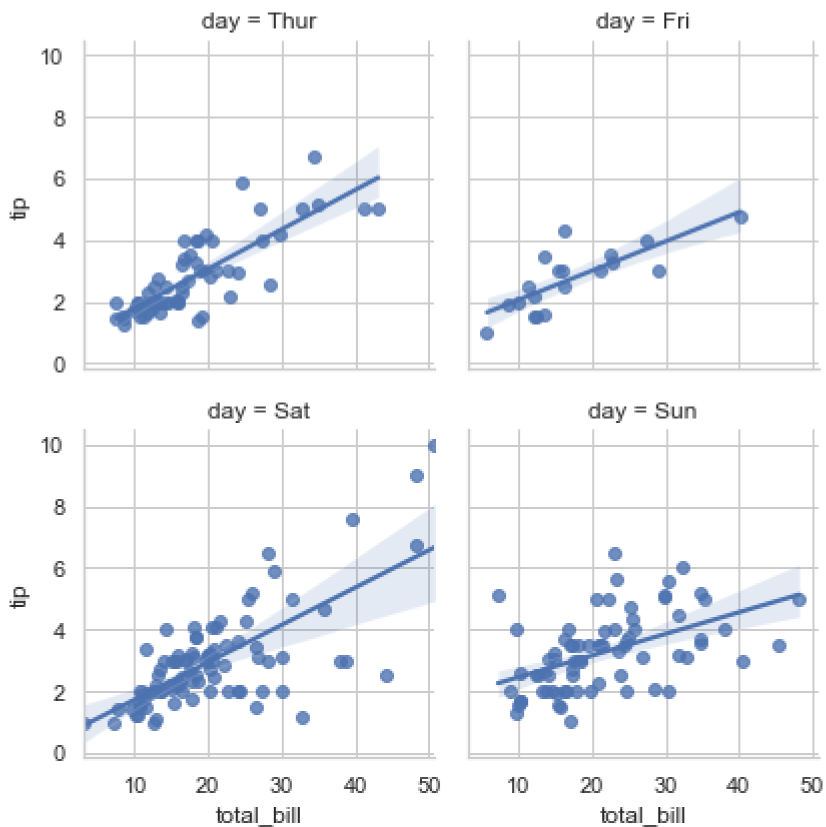
Before we noted that the default plots made by `regplot` and `lmplot` look the same but on axes that have a different size and shape. This is because `regplot` is an "axes-level" function draws onto a specific axes. This means that you can make multi-panel figures yourself and control exactly where the regression plot goes. If no axes object is explictly provided, it simply uses the "currently active" axes, which is why the default plot has the same size and shape as most other matplotlib functions. To control the size, you need to create a figure object yourself.

In [292]:
```
f, ax = plt.subplots(figsize=(6, 6))
sns.regplot(x="total_bill", y="tip", data=tips, ax=ax);
```

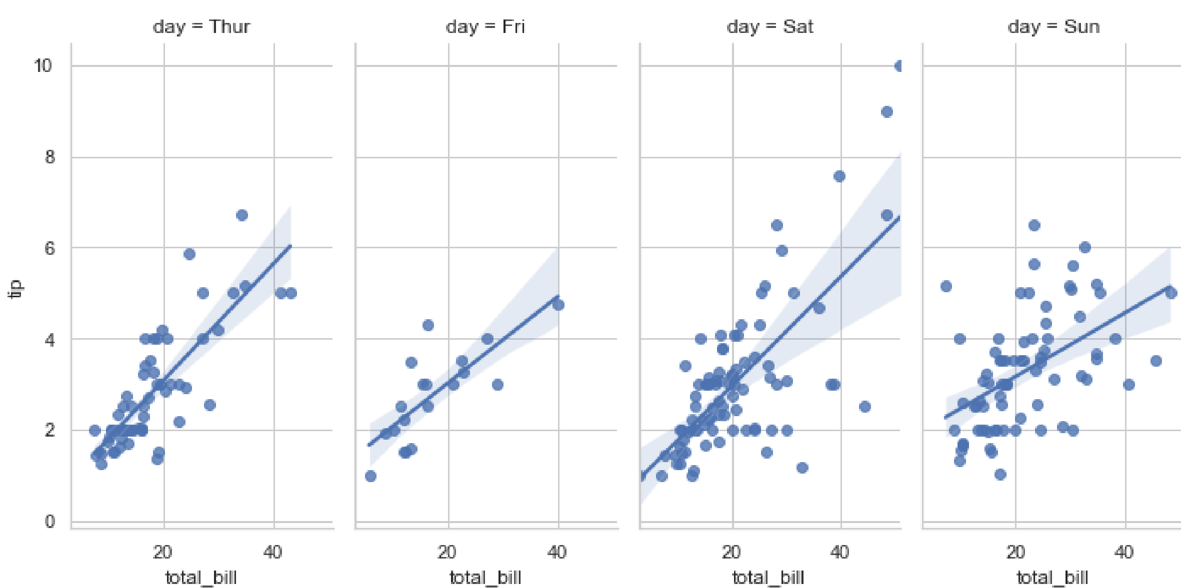


In contrast, the size and shape of the **`lmplot`** figure is controlled through the `FacetGrid` interface using the `size` and `aspect` parameters, which apply to each *facet* in the plot, not to the overall figure itself:

In [304]:
```
sns.lmplot(x="total_bill", y="tip", col="day", data=tips,
           col_wrap=2, height=3);
```



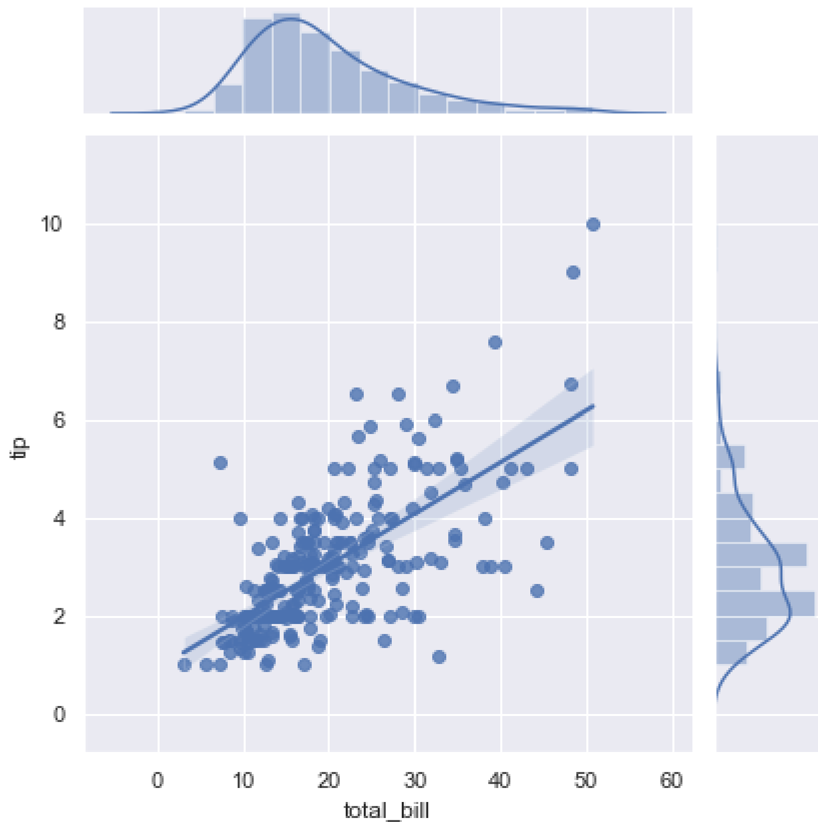
In [308]:
```
sns.lmplot(x="total_bill", y="tip", col="day", data=tips,
           aspect=.5);
```



Plotting a regression in other contexts

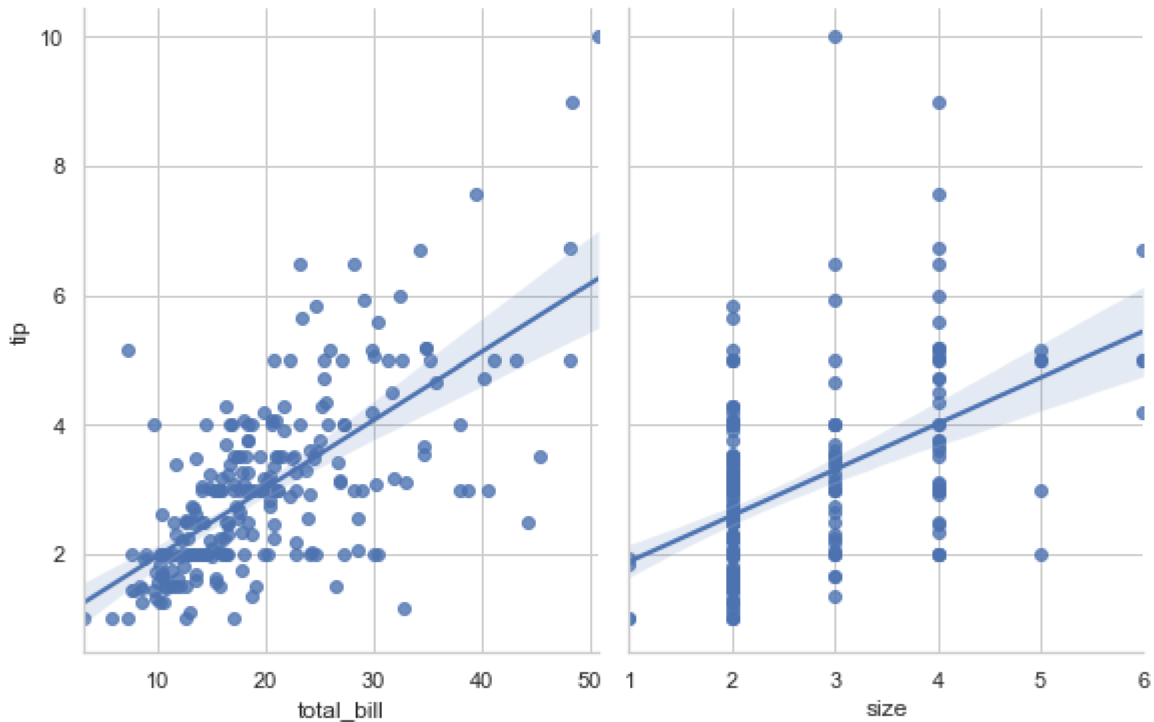
A few other seaborn functions use **`regplot`** in the context of a larger, more complex plot. The first is the **`jointplot`** function that we introduced in the :ref: `distributions tutorial <distribution_tutorial>` . In addition to the plot styles previously discussed, **`jointplot`** can use **`regplot`** to show the linear regression fit on the joint axes by passing `kind="reg"` :

In [228]:
```
sns.jointplot(x="total_bill", y="tip", data=tips, kind="reg");
```



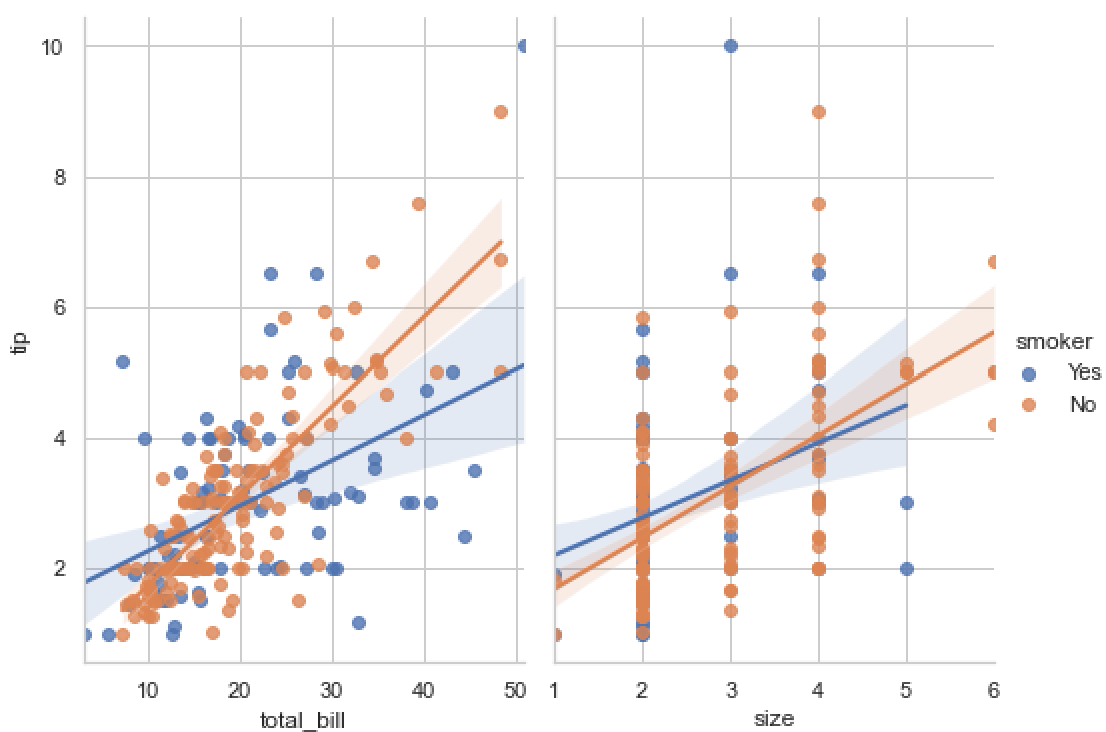
Using the **`pairplot`** function with `kind="reg"` combines **`regplot`** and **`PairGrid`** to show the linear relationship between variables in a dataset. Take care to note how this is different from **`lmplot`** . In the figure below, the two axes don't show the same relationship conditioned on two levels of a third variable; rather, **`PairGrid`** is used to show multiple relationships between different pairings of the variables in a dataset:

In [309]:
```
sns.pairplot(tips, x_vars=["total_bill", "size"], y_vars=["tip"],
             height=5, aspect=.8, kind="reg");
```



Like **lmplot** , but unlike **jointplot** , conditioning on an additional categorical variable is built into **pairplot** using the hue parameter:

In [311]:
```
sns.pairplot(tips, x_vars=["total_bill", "size"], y_vars=["tip"],
             hue="smoker", height=5, aspect=.7, kind="reg");
```



Plotting with categorical data

We can use scatterplots and regression model fits to visualize the relationship between two variables and how it changes across levels of additional categorical variables. However, what if one of the main variables you are interested in is categorical? In this case, the scatterplot and regression model approach won't work. There are several options, however, for visualizing such a relationship.

It's useful to divide seaborn's categorical plots into three groups: those that show each observation at each level of the categorical variable, those that show an abstract representation of each *distribution* of observations, and those that apply a statistical estimation to show a measure of central tendency and confidence interval. The first includes the functions **swarmplot** and **stripplot**, the second includes **boxplot** and **violinplot**, and the third includes **barplot** and **pointplot**. These functions all share a basic API for how they accept data, although each has specific parameters that control the particulars of the visualization that is applied to that data.

Much like the relationship between **regplot** and **lmplot**, in seaborn there are both relatively low-level and relatively high-level approaches for making categorical plots. The functions named above are all low-level in that they plot onto a specific matplotlib axes. There is also the higher-level **factorplot**, which combines these functions with a **FacetGrid** to apply a categorical plot across a grid of figure panels.

In [231]:
```
%matplotlib inline
```

In [232]:
```
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
```

In [233]:
```
import seaborn as sns
sns.set(style="whitegrid", color_codes=True)
```

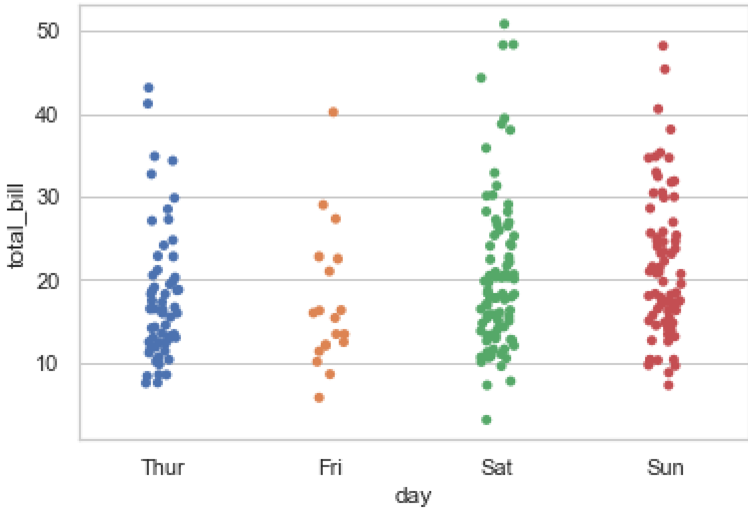
In [234]:
```
np.random.seed(sum(map(ord, "categorical")))
```

In [235]:
```
titanic = sns.load_dataset("titanic")
tips = sns.load_dataset("tips")
iris = sns.load_dataset("iris")
```

Categorical scatterplots

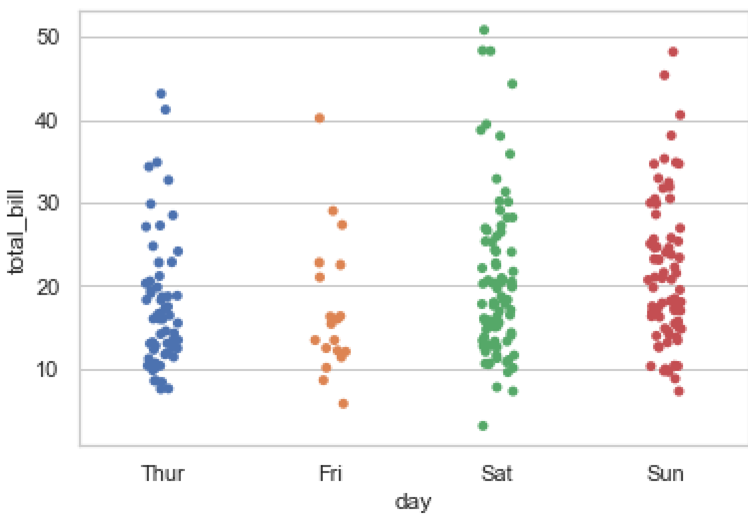
A simple way to show the the values of some quantitative variable across the levels of a categorical variable uses *** `stripplot` , which generalizes a scatterplot to the case where one of the variables is categorical:

In [236]:
```
sns.stripplot(x="day", y="total_bill", data=tips);
```



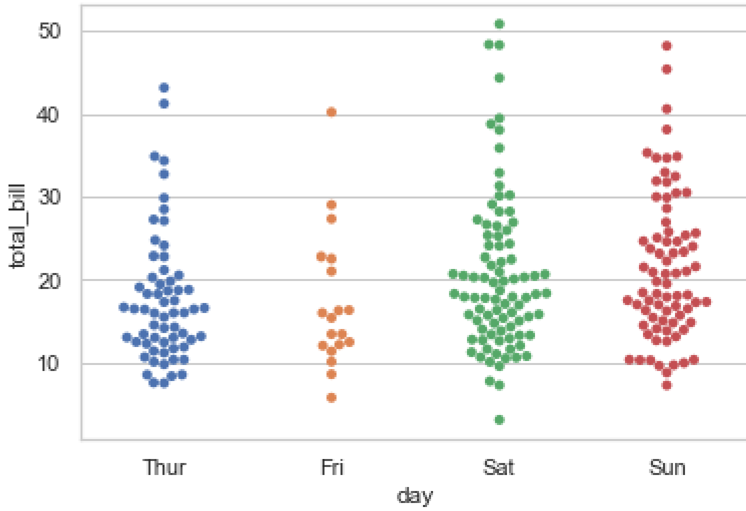
In a strip plot, the scatterplot points will usually overlap. This makes it difficult to see the full distribution of data. One easy solution is to adjust the positions (only along the categorical axis) using some random "jitter":

In [237]:
```
sns.stripplot(x="day", y="total_bill", data=tips, jitter=True);
```



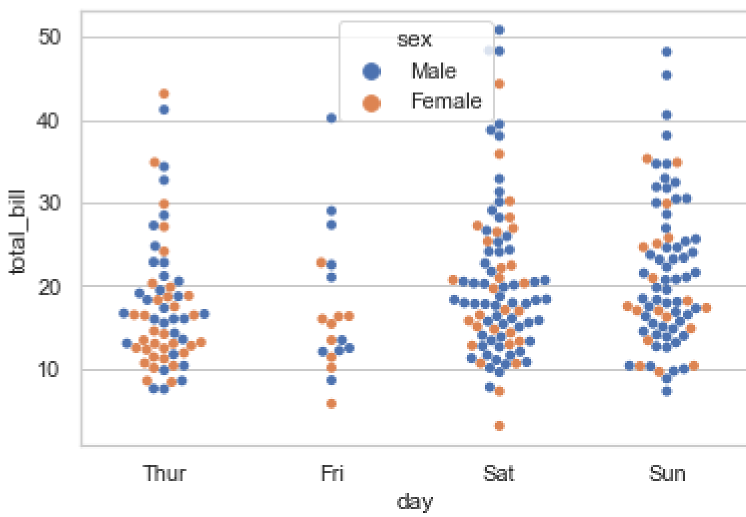
A different approach would be to use the function *** `swarmplot` , which positions each scatterplot point on the categorical axis with an algorithm that avoids overlapping points:

In [238]:
```
sns.swarmplot(x="day", y="total_bill", data=tips);
```



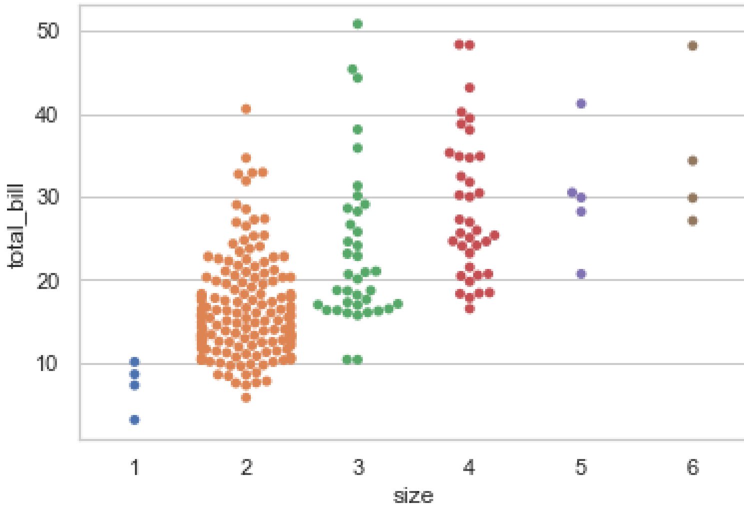
It's also possible to add a nested categorical variable with the `hue` parameter. Above the color and position on the categorical axis are redundant, but now each provides information about one of the two variables:

In [240]:
```
sns.swarmplot(x="day", y="total_bill", hue="sex", data=tips);
```



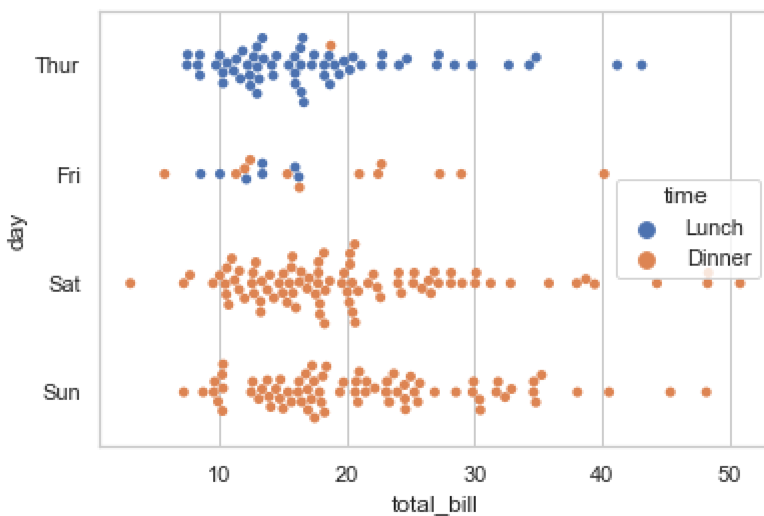
In general, the seaborn categorical plotting functions try to infer the order of categories from the data. If your data have a pandas `Categorical` datatype, then the default order of the categories can be set there. For other datatypes, string-typed categories will be plotted in the order they appear in the DataFrame, but categories that look numerical will be sorted:

In [241]:
```
sns.swarmplot(x="size", y="total_bill", data=tips);
```



With these plots, it's often helpful to put the categorical variable on the vertical axis (this is particularly useful when the category names are relatively long or there are many categories). You can force an orientation using the `orient` keyword, but usually plot orientation can be inferred from the datatypes of the variables passed to `x` and/or `y` :

In [242]:
```
sns.swarmplot(x="total_bill", y="day", hue="time", data=tips);
```



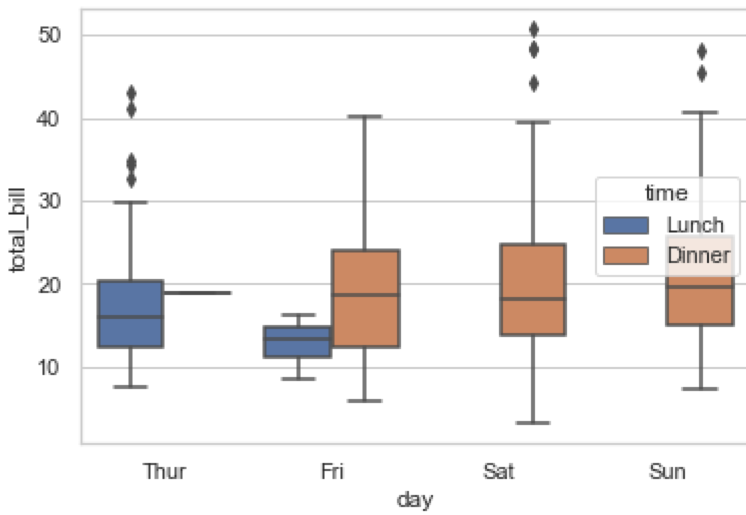
Distributions of observations within categories

At a certain point, the categorical scatterplot approach becomes limited in the information it can provide about the distribution of values within each category. There are several ways to summarize this information in ways that facilitate easy comparisons across the category levels. These generalize some of the approaches we discussed in the :ref: `chapter <distribution_tutorial>` to the case where we want to quickly compare across several distributions.

Boxplots ^^^^^^^^

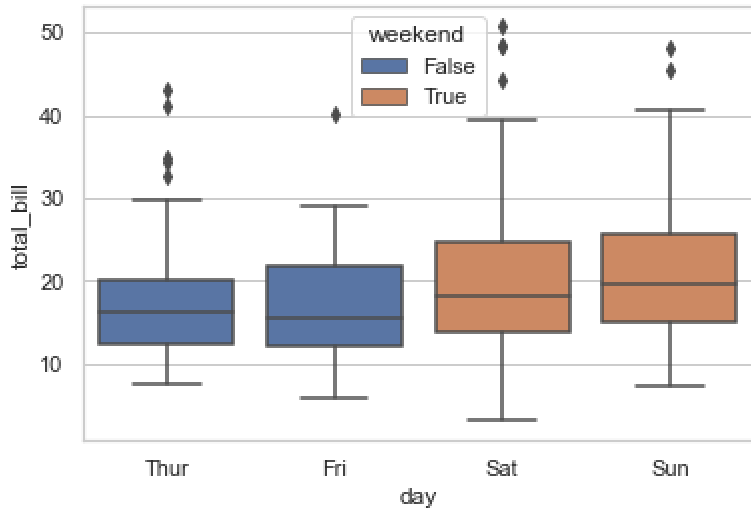
The first is the familiar *** `boxplot` . This kind of plot shows the three quartile values of the distribution along with extreme values. The "whiskers" extend to points that lie within 1.5 IQRs of the lower and upper quartile, and then observations that fall outside this range are displayed independently. Importantly, this means that each value in the boxplot corresponds to an actual observation in the data:

In [244]:
```
sns.boxplot(x="day", y="total_bill", hue="time", data=tips);
```



For boxplots, the assumption when using a `hue` variable is that it is nested within the `x` or `y` variable. This means that by default, the boxes for different levels of `hue` will be offset, as you can see above. If your `hue` variable is not nested, you can set the `dodge` parameter to disable offsetting:

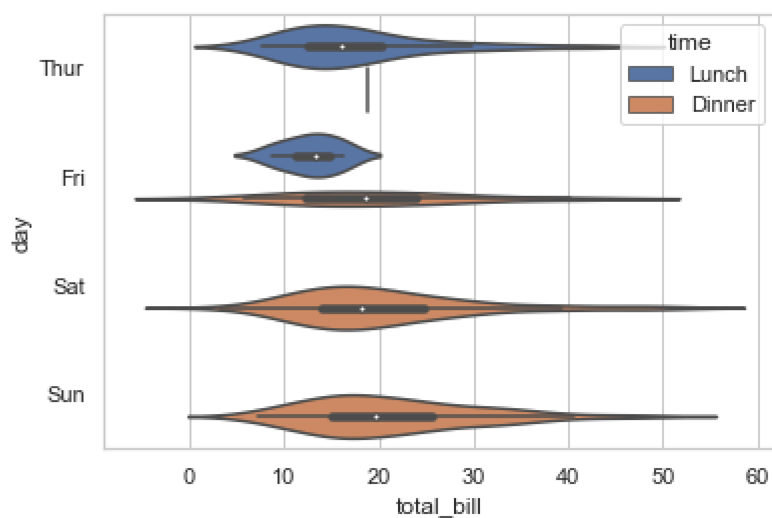
In [245]:
```
tips["weekend"] = tips["day"].isin(["Sat", "Sun"])
sns.boxplot(x="day", y="total_bill", hue="weekend", data=tips, dodge=False);
```



Violinplots ^^^^^^^^^^^

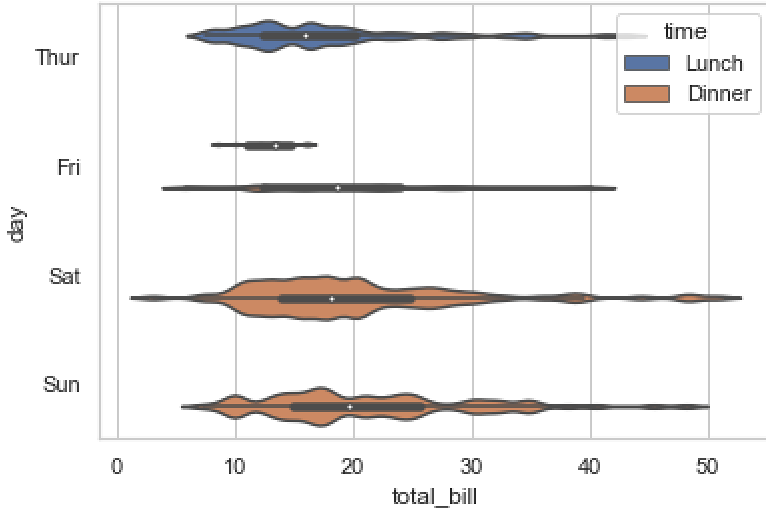
A different approach is a *** `violinplot` , which combines a boxplot with the kernel density estimation procedure described in the :ref: `distributions <distribution_tutorial>` tutorial:

In [246]:
```
sns.violinplot(x="total_bill", y="day", hue="time", data=tips);
```



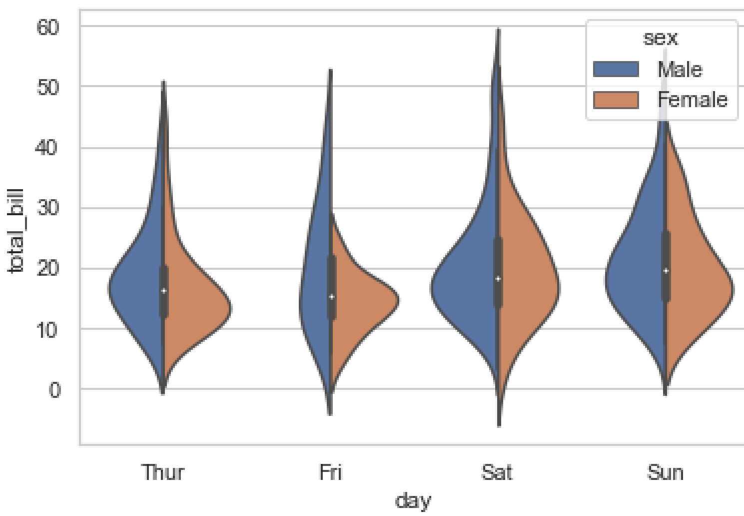
This approach uses the kernel density estimate to provide a better description of the distribution of values. Additionally, the quartile and whikser values from the boxplot are shown inside the violin. Because the violinplot uses a KDE, there are some other parameters that may need tweaking, adding some complexity relative to the straightforward boxplot:

In [247]:
```
sns.violinplot(x="total_bill", y="day", hue="time", data=tips,
               bw=.1, scale="count", scale_hue=False);
```



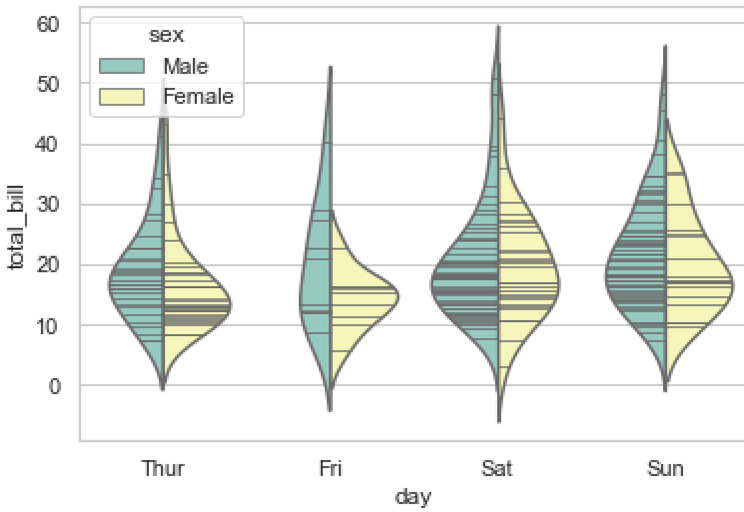
It's also possible to "split" the violins when the hue parameter has only two levels, which can allow for a more efficient use of space:

In [248]:
```
sns.violinplot(x="day", y="total_bill", hue="sex", data=tips, split=True);
```



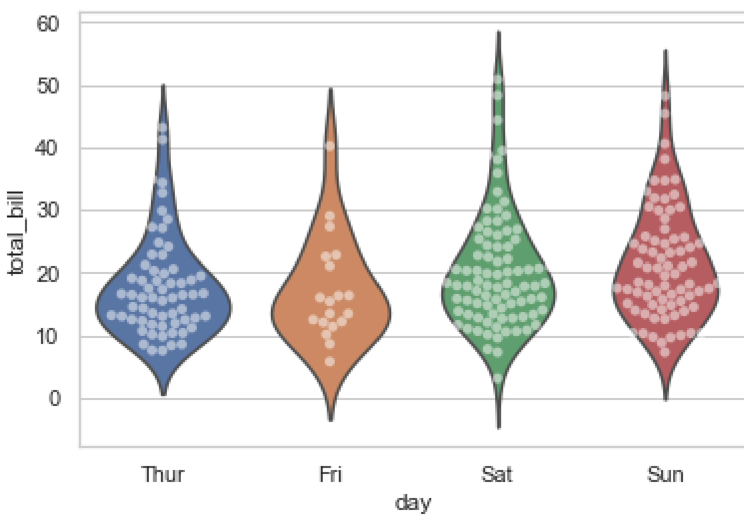
Finally, there are several options for the plot that is drawn on the interior of the violins, including ways to show each individual observation instead of the summary boxplot values:

In [249]:
```
sns.violinplot(x="day", y="total_bill", hue="sex", data=tips,
               split=True, inner="stick", palette="Set3");
```



It can also be useful to combine ***swarmplot* or** `swarmplot` with ***violinplot* or** `boxplot` to show each observation along with a summary of the distribution:

In [250]:
```
sns.violinplot(x="day", y="total_bill", data=tips, inner=None)
sns.swarmplot(x="day", y="total_bill", data=tips, color="w", alpha=.5);
```



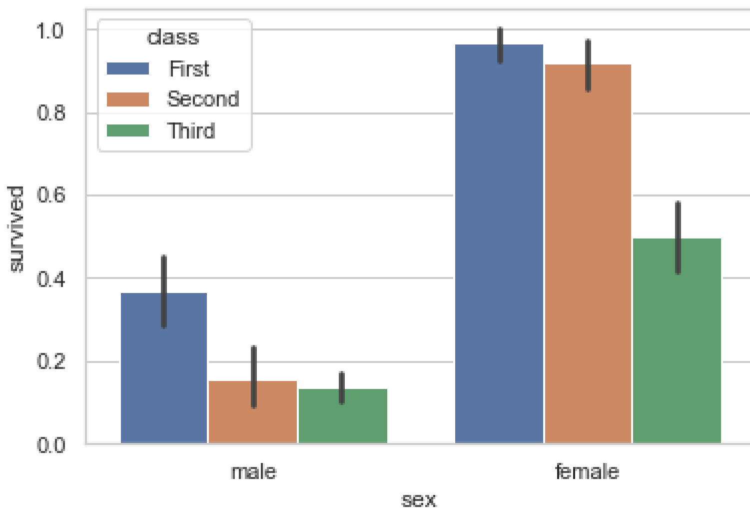
Statistical estimation within categories

Often, rather than showing the distribution within each category, you might want to show the central tendency of the values. Seaborn has two main ways to show this information, but importantly, the basic API for these functions is identical to that for the ones discussed above.

Bar plots

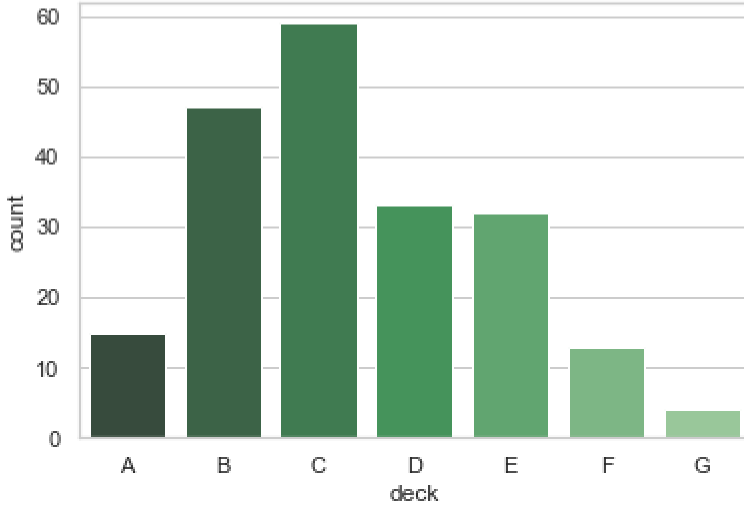
A familiar style of plot that accomplishes this goal is a bar plot. In seaborn, the ***barplot*** function operates on a full dataset and shows an arbitrary estimate, using the mean by default. When there are multiple observations in each category, it also uses bootstrapping to compute a confidence interval around the estimate and plots that using error bars:

In [251]:
```
sns.barplot(x="sex", y="survived", hue="class", data=titanic);
```



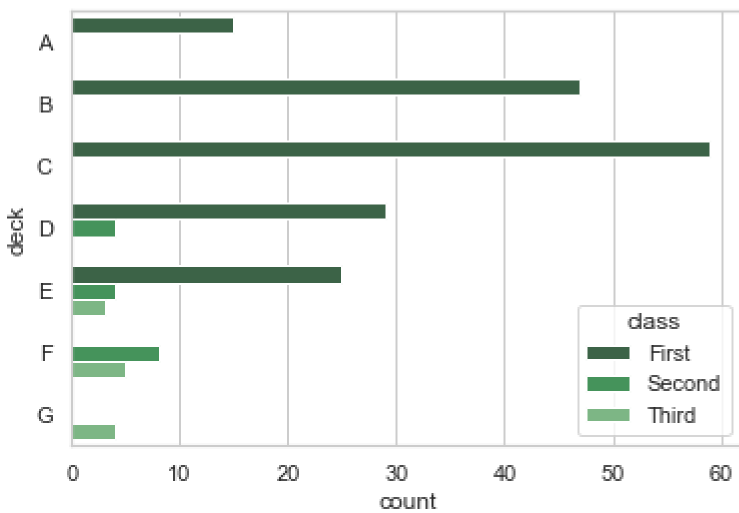
A special case for the bar plot is when you want to show the number of observations in each category rather than computing a statistic for a second variable. This is similar to a histogram over a categorical, rather than quantitative, variable. In seaborn, it's easy to do so with the *** `countplot` function:

In [252]:
```
sns.countplot(x="deck", data=titanic, palette="Greens_d");
```



Both ***barplot*** and ***countplot*** can be invoked with all of the options discussed above, along with others that are demonstrated in the detailed documentation for each function:

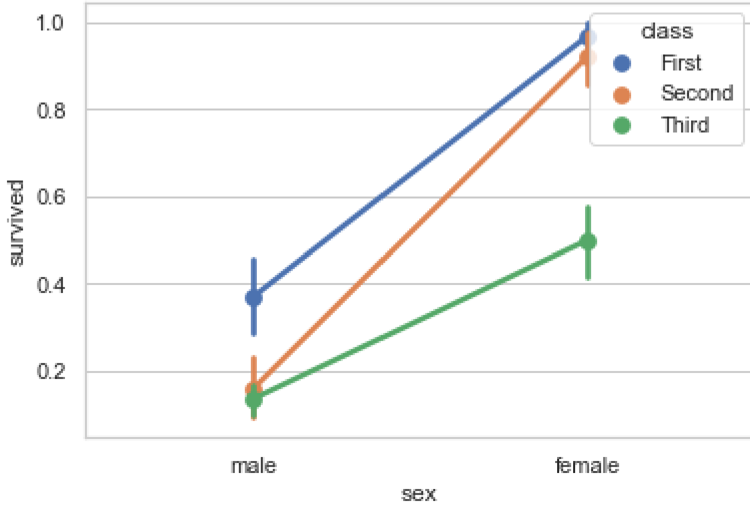
In [253]:
```
sns.countplot(y="deck", hue="class", data=titanic, palette="Greens_d");
```



Point plots

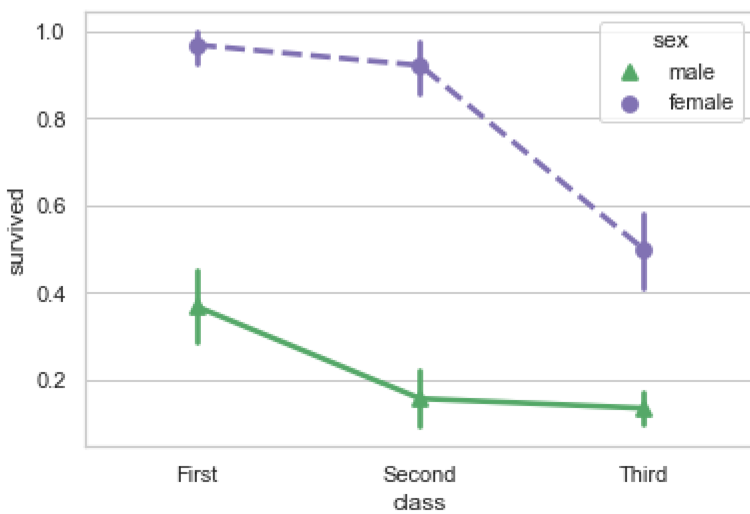
An alternative style for visualizing the same information is offered by the ***pointplot*** function. This function also encodes the value of the estimate with height on the other axis, but rather than show a full bar it just plots the point estimate and confidence interval. Additionally, pointplot connects points from the same `hue` category. This makes it easy to see how the main relationship is changing as a function of a second variable, because your eyes are quite good at picking up on differences of slopes:

In [254]:
```
sns.pointplot(x="sex", y="survived", hue="class", data=titanic);
```



To make figures that reproduce well in black and white, it can be good to use different markers and line styles for the levels of the `hue` category:

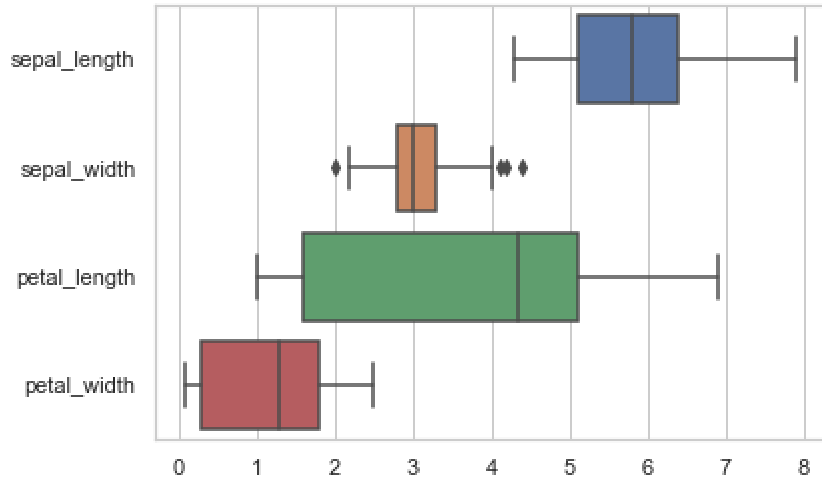
In [255]:
```
sns.pointplot(x="class", y="survived", hue="sex", data=titanic,
              palette={"male": "g", "female": "m"},
              markers=["^", "o"], linestyles=["-", "--"]);
```



Plotting "wide-form" data

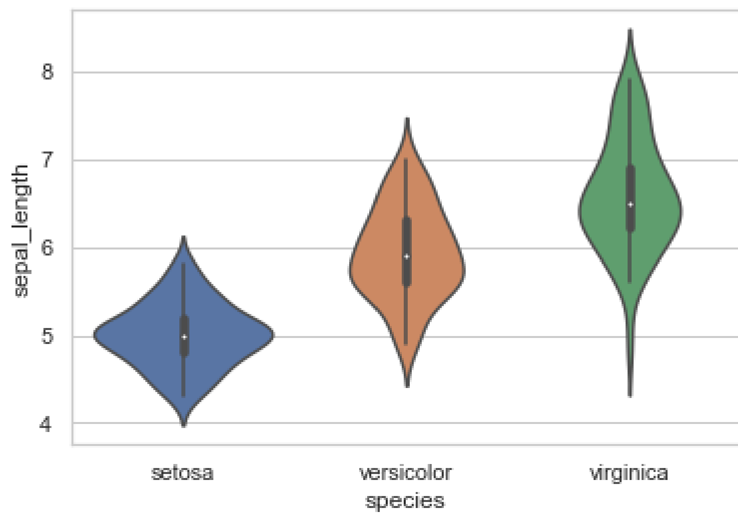
While using "long-form" or "tidy" data is preferred, these functions can also by applied to "wide-form" data in a variety of formats, including pandas DataFrames or two-dimensional numpy arrays. These objects should be passed directly to the `data` parameter:

In [256]:
```
sns.boxplot(data=iris, orient="h");
```



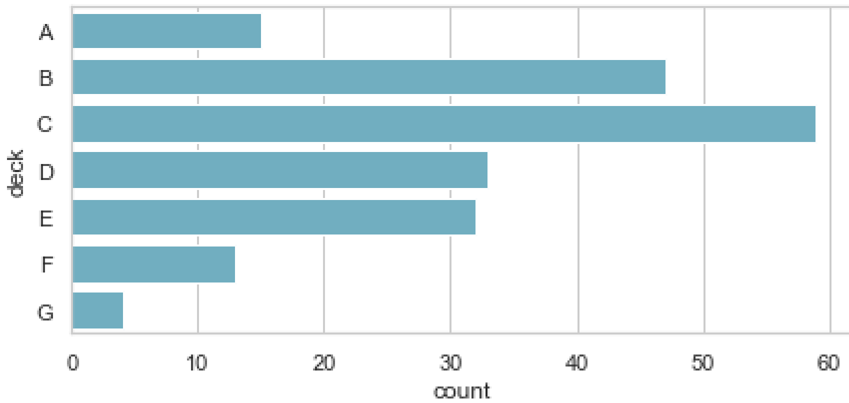
Additionally, these functions accept vectors of Pandas or numpy objects rather than variables in a `DataFrame` :

In [257]:
```
sns.violinplot(x=iris.species, y=iris.sepal_length);
```



To control the size and shape of plots made by the functions discussed above, you must set up the figure yourself using matplotlib commands. Of course, this also means that the plots can happily coexist in a multi-panel figure with other kinds of plots:

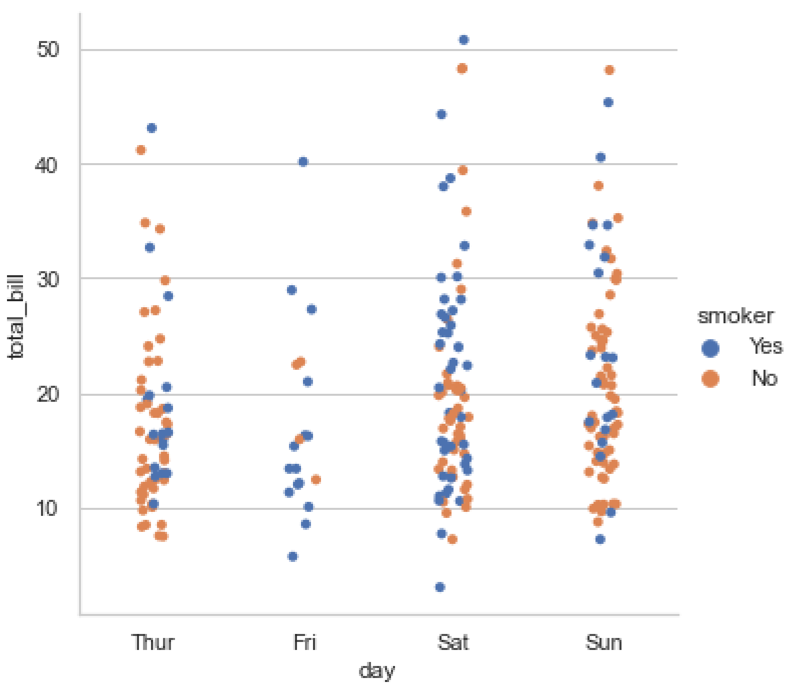
In [258]:
```
f, ax = plt.subplots(figsize=(7, 3))
sns.countplot(y="deck", data=titanic, color="c");
```



Drawing multi-panel categorical plots

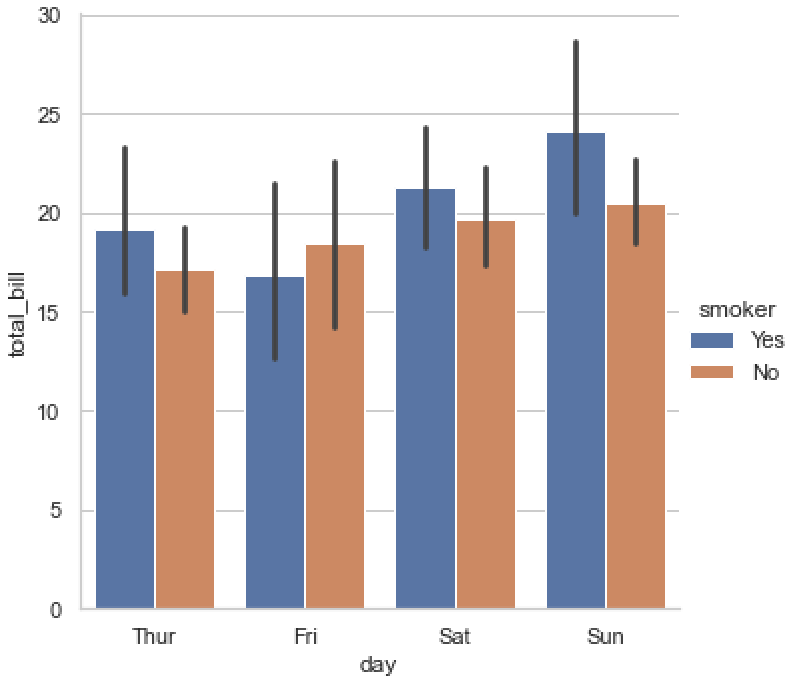
As we mentioned above, there are two ways to draw categorical plots in seaborn. Similar to the duality in the regression plots, you can either use the functions introduced above, or the higher-level function ***factorplot*** , which combines these functions with a ***FacetGrid*** to add the ability to examine additional categories through the larger structure of the figure. By default, ***factorplot*** produces a ***pointplot*** :

In [280]:
```
sns.catplot(x="day", y="total_bill", hue="smoker", data=tips);
```



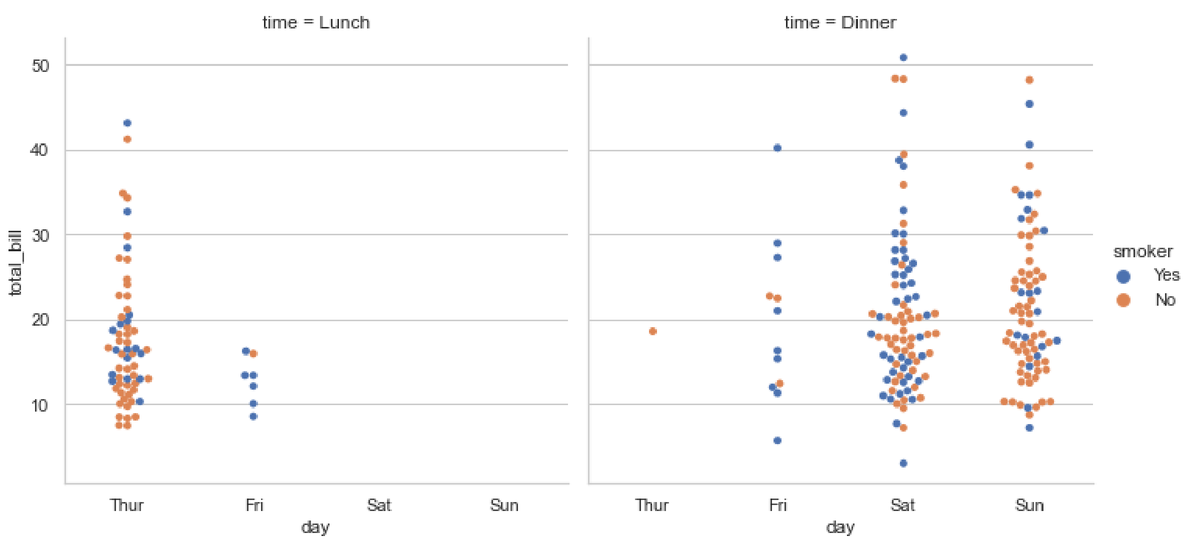
However, the `kind` parameter lets you chose any of the kinds of plots discussed above:

In [262]:
```
sns.factorplot(x="day", y="total_bill", hue="smoker", data=tips, kind="bar");
```



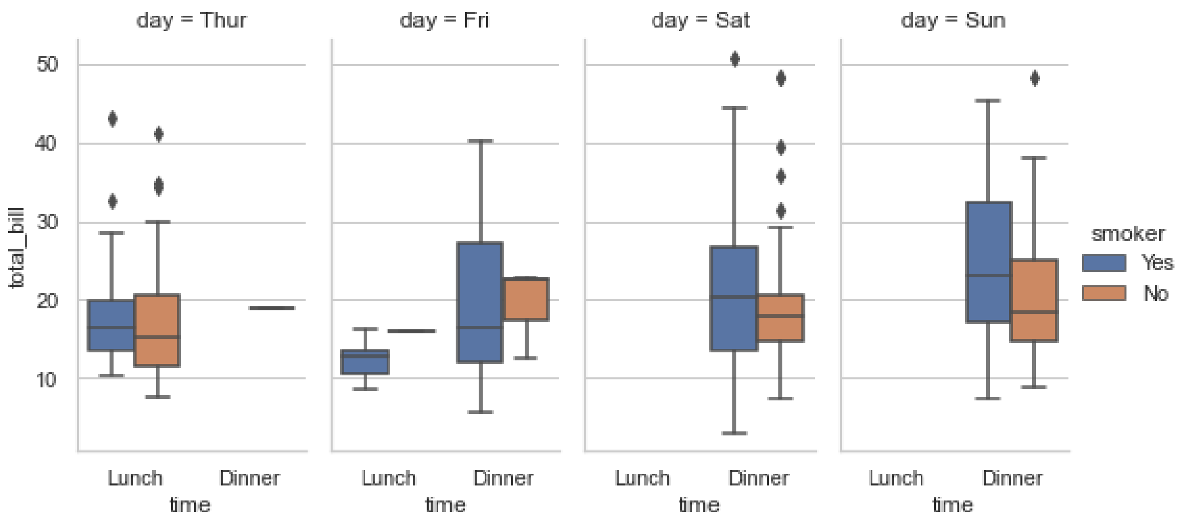
The main advantage of using a ***factorplot*** is that it is very easy to "facet" the plot and investigate the role of other categorical variables:

In [263]:
```
sns.factorplot(x="day", y="total_bill", hue="smoker",
               col="time", data=tips, kind="swarm");
```



Any kind of plot can be drawn. Because of the way :class: `FacetGrid` works, to change the size and shape of the figure you need to specify the `size` and `aspect` arguments, which apply to each facet:

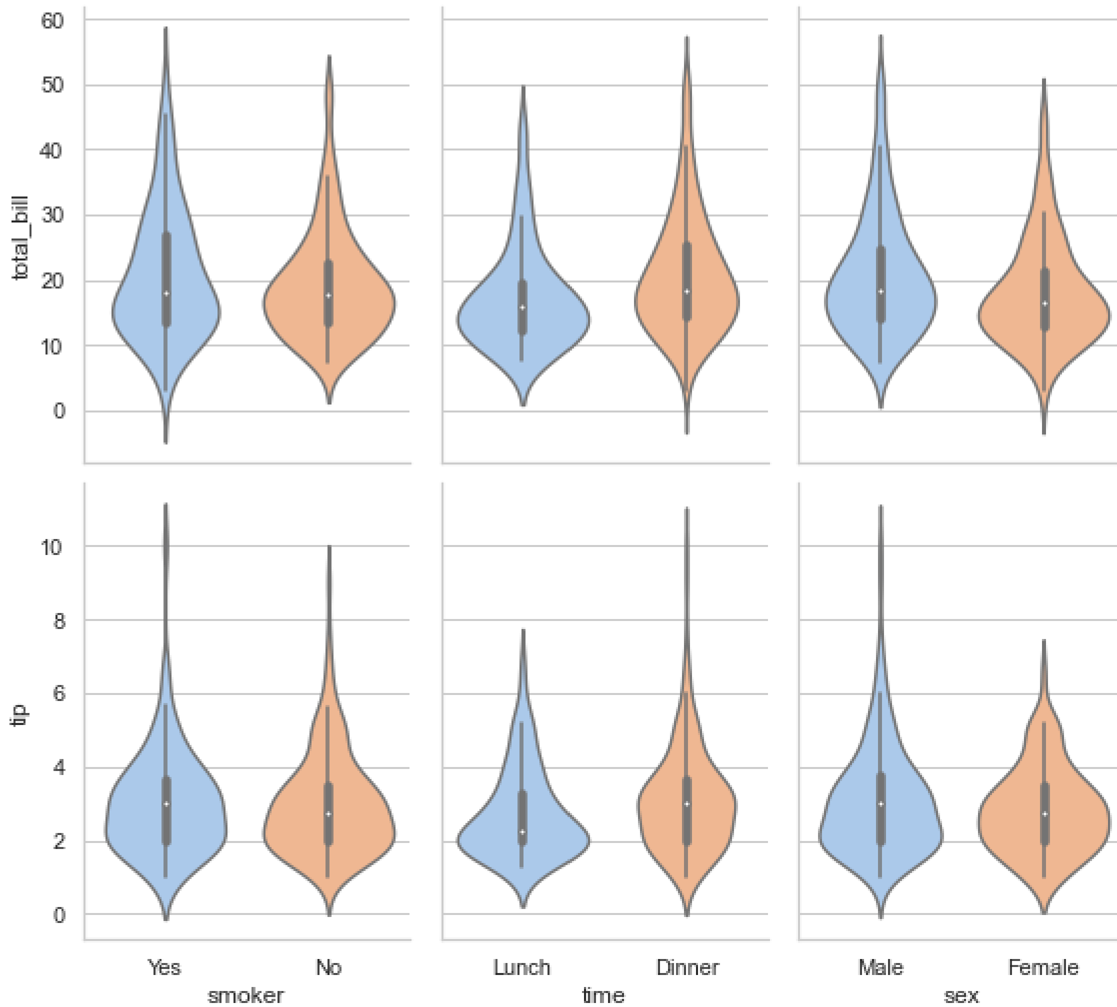
```
In [276]: sns.catplot(x="time", y="total_bill", hue="smoker",
                      col="day", data=tips, kind="box", height=4, aspect=.5);
```



It is important to note that you could also make this plot by using ***`boxplot`*** and :class: `FacetGrid` directly. However, special care must be taken to ensure that the order of the categorical variables is enforced in each facet, either by using data with a `Categorical` datatype or by passing `order` and `hue_order` .

Because of the generalized API of the categorical plots, they should be easy to apply to other more complex contexts. For example, they are easily combined with a :class: `PairGrid` to show categorical relationships across several different variables:

In [313]:
```
g = sns.PairGrid(tips,
                 x_vars=["smoker", "time", "sex"],
                 y_vars=["total_bill", "tip"],
                 aspect=.75, height=3.5)
g.map(sns.violinplot, palette="pastel");
```



End Of Assignment