CHAPTER 14
Data Analysis Examples

Now that we've reached the end of this book's main chapters, we're going to take a look at a number of real-world datasets. For each dataset, we'll use the techniques presented in this book to extract meaning from the raw data. The demonstrated techniques can be applied to all manner of other datasets, including your own. This chapter contains a collection of miscellaneous example datasets that you can use for practice with the tools in this book.

The example datasets are found in the book's accompanying GitHub repository.

14.1 1.USA.gov Data from Bitly

In 2011, URL shortening service Bitly partnered with the US government website USA.gov to provide a feed of anonymous data gathered from users who shorten links ending with *.gov* or *.mil*. In 2011, a live feed as well as hourly snapshots were available as downloadable text files. This service is shut down at the time of this writing (2017), but we preserved one of the data files for the book's examples.

In the case of the hourly snapshots, each line in each file contains a common form of web data known as JSON, which stands for JavaScript Object Notation. For example, if we read just the first line of a file we may see something like this:

```
In [5]: path = 'datasets/bitly_usagov/example.txt'

In [6]: open(path).readline()
Out[6]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\/535.11
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US", "nk": 1,
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l":
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http:\\/\\/www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf", "u":
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\n'
```

Python has both built-in and third-party libraries for converting a JSON string into a Python dictionary object. Here we'll use the `json` module and its `loads` function invoked on each line in the sample file we downloaded:

```
import json
path = 'datasets/bitly_usagov/example.txt'
records = [json.loads(line) for line in open(path)]
```

The resulting object `records` is now a list of Python dicts:

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like Gecko)
Chrome/17.0.963.78 Safari/535.11',
 'al': 'en-US,en;q=0.8',
 'c': 'US',
 'cy': 'Danvers',
 'g': 'A6qOVH',
 'gr': 'MA',
 'h': 'wfLQtf',
 'hc': 1331822918,
 'hh': '1.usa.gov',
 'l': 'orofrog',
 'll': [42.576698, -70.954903],
 'nk': 1,
 'r': 'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
 't': 1331923247,
 'tz': 'America/New_York',
 'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Counting Time Zones in Pure Python

Suppose we were interested in finding the most often-occurring time zones in the dataset (the `tz` field). There are many ways we could do this. First, let's extract a list of time zones again using a list comprehension:

```
In [12]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-db4fbd348da9> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
<ipython-input-12-db4fbd348da9> in <listcomp>(.0)
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

Oops! Turns out that not all of the records have a time zone field. This is easy to handle, as we can add the check `if 'tz' in rec` at the end of the list comprehension:

```
In [13]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [14]: time_zones[:10]
Out[14]:
```

```
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

Just looking at the first 10 time zones, we see that some of them are unknown (empty string). You can filter these out also, but I'll leave them in for now. Now, to produce counts by time zone I'll show two approaches: the harder way (using just the Python standard library) and the easier way (using pandas). One way to do the counting is to use a dict to store counts while we iterate through the time zones:

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

Using more advanced tools in the Python standard library, you can write the same thing more briefly:

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # values will initialize to 0
    for x in sequence:
        counts[x] += 1
    return counts
```

I put this logic in a function just to make it more reusable. To use it on the time zones, just pass the `time_zones` list:

```
In [17]: counts = get_counts(time_zones)

In [18]: counts['America/New_York']
Out[18]: 1251

In [19]: len(time_zones)
Out[19]: 3440
```

If we wanted the top 10 time zones and their counts, we can do a bit of dictionary acrobatics:

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
```

```
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

We have then:

```
In [21]: top_counts(counts)
Out[21]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
 (36, 'Pacific/Honolulu'),
 (37, 'Asia/Tokyo'),
 (74, 'Europe/London'),
 (191, 'America/Denver'),
 (382, 'America/Los_Angeles'),
 (400, 'America/Chicago'),
 (521, ''),
 (1251, 'America/New_York')]
```

If you search the Python standard library, you may find the `collections.Counter` class, which makes this task a lot easier:

```
In [22]: from collections import Counter

In [23]: counts = Counter(time_zones)

In [24]: counts.most_common(10)
Out[24]:
[('America/New_York', 1251),
 ('', 521),
 ('America/Chicago', 400),
 ('America/Los_Angeles', 382),
 ('America/Denver', 191),
 ('Europe/London', 74),
 ('Asia/Tokyo', 37),
 ('Pacific/Honolulu', 36),
 ('Europe/Madrid', 35),
 ('America/Sao_Paulo', 33)]
```

Counting Time Zones with pandas

Creating a DataFrame from the original set of records is as easy as passing the list of records to `pandas.DataFrame`:

```
In [25]: import pandas as pd

In [26]: frame = pd.DataFrame(records)

In [27]: frame.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3560 entries, 0 to 3559
Data columns (total 18 columns):
_heartbeat_    120 non-null float64
a              3440 non-null object
```

```
al              3094 non-null object
c               2919 non-null object
cy              2919 non-null object
g               3440 non-null object
gr              2919 non-null object
h               3440 non-null object
hc              3440 non-null float64
hh              3440 non-null object
kw              93 non-null object
l               3440 non-null object
ll              2919 non-null object
nk              3440 non-null float64
r               3440 non-null object
t               3440 non-null float64
tz              3440 non-null object
u               3440 non-null object
dtypes: float64(4), object(14)
memory usage: 500.7+ KB

In [28]: frame['tz'][:10]
Out[28]:
0     America/New_York
1       America/Denver
2     America/New_York
3    America/Sao_Paulo
4     America/New_York
5     America/New_York
6        Europe/Warsaw
7
8
9
Name: tz, dtype: object
```

The output shown for the `frame` is the *summary view*, shown for large DataFrame objects. We can then use the `value_counts` method for Series:

```
In [29]: tz_counts = frame['tz'].value_counts()

In [30]: tz_counts[:10]
Out[30]:
America/New_York       1251
                        521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
America/Sao_Paulo        33
Name: tz, dtype: int64
```

We can visualize this data using matplotlib. You can do a bit of munging to fill in a substitute value for unknown and missing time zone data in the records. We replace the missing values with the `fillna` method and use boolean array indexing for the empty strings:

```
In [31]: clean_tz = frame['tz'].fillna('Missing')

In [32]: clean_tz[clean_tz == ''] = 'Unknown'

In [33]: tz_counts = clean_tz.value_counts()

In [34]: tz_counts[:10]
Out[34]:
America/New_York       1251
Unknown                 521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Missing                 120
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
Name: tz, dtype: int64
```

At this point, we can use the seaborn package to make a horizontal bar plot (see Figure 14-1 for the resulting visualization):

```
In [36]: import seaborn as sns

In [37]: subset = tz_counts[:10]

In [38]: sns.barplot(y=subset.index, x=subset.values)
```

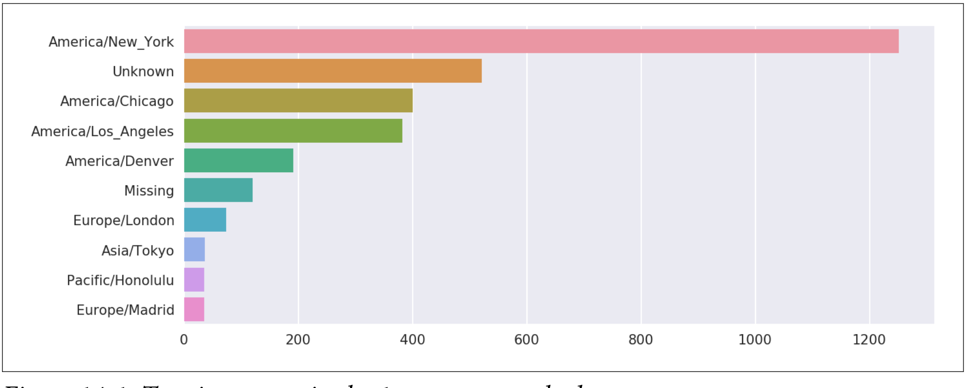

Figure 14-1. Top time zones in the 1.usa.gov sample data

The `a` field contains information about the browser, device, or application used to perform the URL shortening:

```
In [39]: frame['a'][1]
Out[39]: 'GoogleMaps/RochesterNY'

In [40]: frame['a'][50]
Out[40]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

In [41]: frame['a'][51][:50]  # long line
Out[41]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P9'
```

Parsing all of the interesting information in these "agent" strings may seem like a daunting task. One possible strategy is to split off the first token in the string (corresponding roughly to the browser capability) and make another summary of the user behavior:

```
In [42]: results = pd.Series([x.split()[0] for x in frame.a.dropna()])

In [43]: results[:5]
Out[43]:
0               Mozilla/5.0
1    GoogleMaps/RochesterNY
2               Mozilla/4.0
3               Mozilla/5.0
4               Mozilla/5.0
dtype: object

In [44]: results.value_counts()[:8]
Out[44]:
Mozilla/5.0                 2594
Mozilla/4.0                  601
GoogleMaps/RochesterNY       121
Opera/9.80                    34
TEST_INTERNET_AGENT           24
GoogleProducer                21
Mozilla/6.0                    5
BlackBerry8520/5.0.0.681       4
dtype: int64
```

Now, suppose you wanted to decompose the top time zones into Windows and non-Windows users. As a simplification, let's say that a user is on Windows if the string `'Windows'` is in the agent string. Since some of the agents are missing, we'll exclude these from the data:

```
In [45]: cframe = frame[frame.a.notnull()]
```

We want to then compute a value for whether each row is Windows or not:

```
In [47]: cframe['os'] = np.where(cframe['a'].str.contains('Windows'),
   ....:                         'Windows', 'Not Windows')

In [48]: cframe['os'][:5]
```

```
Out[48]:
0        Windows
1    Not Windows
2        Windows
3    Not Windows
4        Windows
Name: os, dtype: object
```

Then, you can group the data by its time zone column and this new list of operating systems:

```
In [49]: by_tz_os = cframe.groupby(['tz', 'os'])
```

The group counts, analogous to the `value_counts` function, can be computed with `size`. This result is then reshaped into a table with `unstack`:

```
In [50]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [51]: agg_counts[:10]
Out[51]:
os                              Not Windows  Windows
tz
                                      245.0    276.0
Africa/Cairo                            0.0      3.0
Africa/Casablanca                       0.0      1.0
Africa/Ceuta                            0.0      2.0
Africa/Johannesburg                     0.0      1.0
Africa/Lusaka                           0.0      1.0
America/Anchorage                       4.0      1.0
America/Argentina/Buenos_Aires          1.0      0.0
America/Argentina/Cordoba               0.0      1.0
America/Argentina/Mendoza               0.0      1.0
```

Finally, let's select the top overall time zones. To do so, I construct an indirect index array from the row counts in `agg_counts`:

```
# Use to sort in ascending order
In [52]: indexer = agg_counts.sum(1).argsort()

In [53]: indexer[:10]
Out[53]:
tz
                                  24
Africa/Cairo                      20
Africa/Casablanca                 21
Africa/Ceuta                      92
Africa/Johannesburg               87
Africa/Lusaka                     53
America/Anchorage                 54
America/Argentina/Buenos_Aires    57
America/Argentina/Cordoba         26
America/Argentina/Mendoza         55
dtype: int64
```

I use `take` to select the rows in that order, then slice off the last 10 rows (largest values):

```
In [54]: count_subset = agg_counts.take(indexer[-10:])

In [55]: count_subset
Out[55]:
os                   Not Windows  Windows
tz
America/Sao_Paulo           13.0     20.0
Europe/Madrid               16.0     19.0
Pacific/Honolulu             0.0     36.0
Asia/Tokyo                   2.0     35.0
Europe/London               43.0     31.0
America/Denver             132.0     59.0
America/Los_Angeles        130.0    252.0
America/Chicago            115.0    285.0
                           245.0    276.0
America/New_York           339.0    912.0
```

pandas has a convenience method called `nlargest` that does the same thing:

```
In [56]: agg_counts.sum(1).nlargest(10)
Out[56]:
tz
America/New_York       1251.0
                        521.0
America/Chicago         400.0
America/Los_Angeles     382.0
America/Denver          191.0
Europe/London            74.0
Asia/Tokyo               37.0
Pacific/Honolulu         36.0
Europe/Madrid            35.0
America/Sao_Paulo        33.0
dtype: float64
```

Then, as shown in the preceding code block, this can be plotted in a bar plot; I'll make it a stacked bar plot by passing an additional argument to seaborn's `barplot` function (see Figure 14-2):

```
# Rearrange the data for plotting
In [58]: count_subset = count_subset.stack()

In [59]: count_subset.name = 'total'

In [60]: count_subset = count_subset.reset_index()

In [61]: count_subset[:10]
Out[61]:
                  tz           os  total
0  America/Sao_Paulo  Not Windows   13.0
1  America/Sao_Paulo      Windows   20.0
```

```
2      Europe/Madrid  Not Windows   16.0
3      Europe/Madrid      Windows   19.0
4   Pacific/Honolulu  Not Windows    0.0
5   Pacific/Honolulu      Windows   36.0
6         Asia/Tokyo  Not Windows    2.0
7         Asia/Tokyo      Windows   35.0
8      Europe/London  Not Windows   43.0
9      Europe/London      Windows   31.0
```

```
In [62]: sns.barplot(x='total', y='tz', hue='os',  data=count_subset)
```

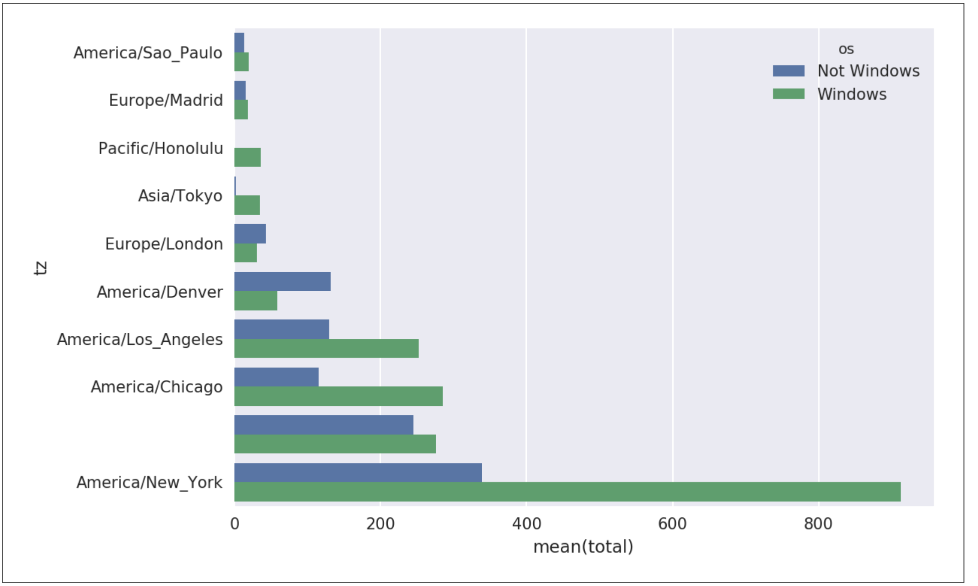

Figure 14-2. Top time zones by Windows and non-Windows users

The plot doesn't make it easy to see the relative percentage of Windows users in the smaller groups, so let's normalize the group percentages to sum to 1:

```
def norm_total(group):
    group['normed_total'] = group.total / group.total.sum()
    return group

results = count_subset.groupby('tz').apply(norm_total)
```

Then plot this in Figure 14-3:

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='os',  data=results)
```

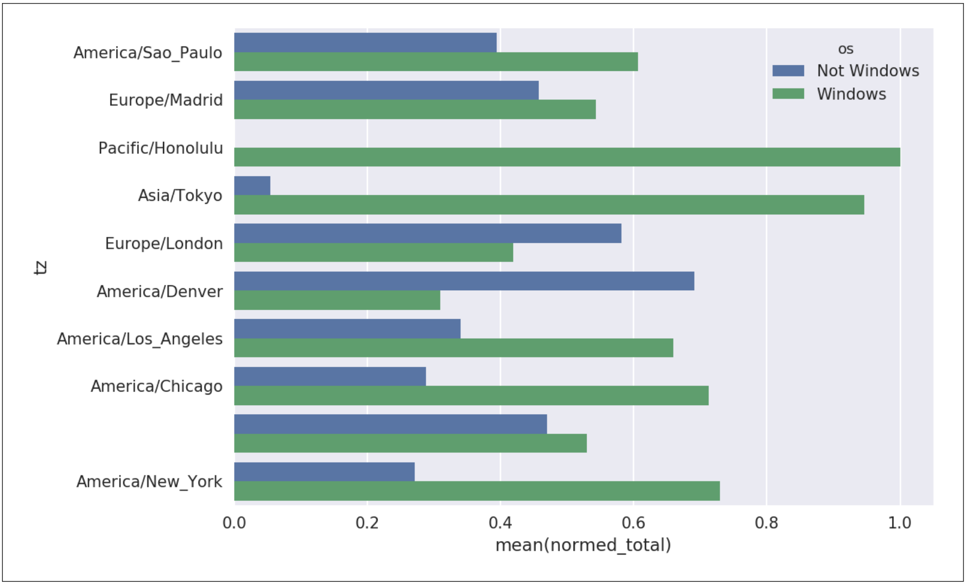

Figure 14-3. Percentage Windows and non-Windows users in top-occurring time zones

We could have computed the normalized sum more efficiently by using the `transform` method with `groupby`:

```
In [66]: g = count_subset.groupby('tz')

In [67]: results2 = count_subset.total / g.total.transform('sum')
```

14.2 MovieLens 1M Dataset

GroupLens Research provides a number of collections of movie ratings data collected from users of MovieLens in the late 1990s and early 2000s. The data provide movie ratings, movie metadata (genres and year), and demographic data about the users (age, zip code, gender identification, and occupation). Such data is often of interest in the development of recommendation systems based on machine learning algorithms. While we do not explore machine learning techniques in detail in this book, I will show you how to slice and dice datasets like these into the exact form you need.

The MovieLens 1M dataset contains 1 million ratings collected from 6,000 users on 4,000 movies. It's spread across three tables: ratings, user information, and movie information. After extracting the data from the ZIP file, we can load each table into a pandas DataFrame object using `pandas.read_table`:

```python
import pandas as pd

# Make display smaller
pd.options.display.max_rows = 10

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('datasets/movielens/users.dat', sep='::',
                      header=None, names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('datasets/movielens/ratings.dat', sep='::',
                        header=None, names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
                       header=None, names=mnames)
```

You can verify that everything succeeded by looking at the first few rows of each DataFrame with Python's slice syntax:

```
In [69]: users[:5]
Out[69]:
   user_id gender  age  occupation    zip
0        1      F    1          10  48067
1        2      M   56          16  70072
2        3      M   25          15  55117
3        4      M   45           7  02460
4        5      M   25          20  55455

In [70]: ratings[:5]
Out[70]:
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968
3        1      3408       4  978300275
4        1      2355       5  978824291

In [71]: movies[:5]
Out[71]:
   movie_id                               title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
3         4            Waiting to Exhale (1995)                  Comedy|Drama
4         5  Father of the Bride Part II (1995)                        Comedy

In [72]: ratings
Out[72]:
         user_id  movie_id  rating  timestamp
0              1      1193       5  978300760
1              1       661       3  978302109
2              1       914       3  978301968
```

```
3                 1      3408       4  978300275
4                 1      2355       5  978824291
...             ...       ...     ...        ...
1000204        6040      1091       1  956716541
1000205        6040      1094       5  956704887
1000206        6040       562       5  956704746
1000207        6040      1096       4  956715648
1000208        6040      1097       4  956715569
[1000209 rows x 4 columns]
```

Note that ages and occupations are coded as integers indicating groups described in the dataset's *README* file. Analyzing the data spread across three tables is not a simple task; for example, suppose you wanted to compute mean ratings for a particular movie by sex and age. As you will see, this is much easier to do with all of the data merged together into a single table. Using pandas's `merge` function, we first merge `ratings` with `users` and then merge that result with the `movies` data. pandas infers which columns to use as the merge (or *join*) keys based on overlapping names:

```
In [73]: data = pd.merge(pd.merge(ratings, users), movies)

In [74]: data
Out[74]:
         user_id  movie_id  rating  timestamp gender  age  occupation    zip  \
0              1      1193       5  978300760      F    1          10  48067
1              2      1193       5  978298413      M   56          16  70072
2             12      1193       4  978220179      M   25          12  32793
3             15      1193       4  978199279      M   25           7  22903
4             17      1193       5  978158471      M   50           1  95350
...          ...       ...     ...        ...    ...  ...         ...    ...
1000204     5949      2198       5  958846401      M   18          17  47901
1000205     5675      2703       3  976029116      M   35          14  30030
1000206     5780      2845       1  958153068      M   18          17  92886
1000207     5851      3607       5  957756608      F   18          20  55410
1000208     5938      2909       4  957273353      M   25           1  35401
                                           title                genres
0             One Flew Over the Cuckoo's Nest (1975)             Drama
1             One Flew Over the Cuckoo's Nest (1975)             Drama
2             One Flew Over the Cuckoo's Nest (1975)             Drama
3             One Flew Over the Cuckoo's Nest (1975)             Drama
4             One Flew Over the Cuckoo's Nest (1975)             Drama
...                                          ...                   ...
1000204                       Modulations (1998)           Documentary
1000205                    Broken Vessels (1998)                 Drama
1000206                        White Boys (1999)                 Drama
1000207                 One Little Indian (1973)  Comedy|Drama|Western
1000208  Five Wives, Three Secretaries and Me (1998)       Documentary
[1000209 rows x 10 columns]

In [75]: data.iloc[0]
Out[75]:
user_id                                            1
```

```
movie_id                                      1193
rating                                           5
timestamp                                978300760
gender                                           F
age                                              1
occupation                                      10
zip                                          48067
title        One Flew Over the Cuckoo's Nest (1975)
genres                                       Drama
Name: 0, dtype: object
```

To get mean movie ratings for each film grouped by gender, we can use the `pivot_table` method:

```
In [76]: mean_ratings = data.pivot_table('rating', index='title',
   ....:                                 columns='gender', aggfunc='mean')

In [77]: mean_ratings[:5]
Out[77]:
gender                                F         M
title
$1,000,000 Duck (1971)         3.375000  2.761905
'Night Mother (1986)           3.388889  3.352941
'Til There Was You (1997)      2.675676  2.733333
'burbs, The (1989)             2.793478  2.962085
...And Justice for All (1979)  3.828571  3.689024
```

This produced another DataFrame containing mean ratings with movie titles as row labels (the “index”) and gender as column labels. I first filter down to movies that received at least 250 ratings (a completely arbitrary number); to do this, I then group the data by title and use `size()` to get a Series of group sizes for each title:

```
In [78]: ratings_by_title = data.groupby('title').size()

In [79]: ratings_by_title[:10]
Out[79]:
title
$1,000,000 Duck (1971)                37
'Night Mother (1986)                  70
'Til There Was You (1997)             52
'burbs, The (1989)                   303
...And Justice for All (1979)        199
1-900 (1994)                           2
10 Things I Hate About You (1999)    700
101 Dalmatians (1961)                565
101 Dalmatians (1996)                364
12 Angry Men (1957)                  616
dtype: int64

In [80]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [81]: active_titles
```

```
Out[81]:
Index([''burbs, The (1989)', '10 Things I Hate About You (1999)',
       '101 Dalmatians (1961)', '101 Dalmatians (1996)', '12 Angry Men (1957)',
       '13th Warrior, The (1999)', '2 Days in the Valley (1996)',
       '20,000 Leagues Under the Sea (1954)', '2001: A Space Odyssey (1968)',
       '2010 (1984)',
       ...
       'X-Men (2000)', 'Year of Living Dangerously (1982)',
       'Yellow Submarine (1968)', 'You've Got Mail (1998)',
       'Young Frankenstein (1974)', 'Young Guns (1988)',
       'Young Guns II (1990)', 'Young Sherlock Holmes (1985)',
       'Zero Effect (1998)', 'eXistenZ (1999)'],
      dtype='object', name='title', length=1216)
```

The index of titles receiving at least 250 ratings can then be used to select rows from `mean_ratings`:

```
# Select rows on the index
In [82]: mean_ratings = mean_ratings.loc[active_titles]

In [83]: mean_ratings
Out[83]:
gender                                    F         M
title
'burbs, The (1989)                 2.793478  2.962085
10 Things I Hate About You (1999)  3.646552  3.311966
101 Dalmatians (1961)              3.791444  3.500000
101 Dalmatians (1996)              3.240000  2.911215
12 Angry Men (1957)                4.184397  4.328421
...                                     ...       ...
Young Guns (1988)                  3.371795  3.425620
Young Guns II (1990)               2.934783  2.904025
Young Sherlock Holmes (1985)       3.514706  3.363344
Zero Effect (1998)                 3.864407  3.723140
eXistenZ (1999)                    3.098592  3.289086
[1216 rows x 2 columns]
```

To see the top films among female viewers, we can sort by the `F` column in descending order:

```
In [85]: top_female_ratings = mean_ratings.sort_values(by='F', ascending=False)

In [86]: top_female_ratings[:10]
Out[86]:
gender                                                         F         M
title
Close Shave, A (1995)                                   4.644444  4.473795
Wrong Trousers, The (1993)                              4.588235  4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)           4.572650  4.464589
Wallace & Gromit: The Best of Aardman Animation...      4.563107  4.385075
Schindler's List (1993)                                 4.562602  4.491415
Shawshank Redemption, The (1994)                        4.539075  4.560625
Grand Day Out, A (1992)                                 4.537879  4.293255
```

```
To Kill a Mockingbird (1962)                     4.536667  4.372611
Creature Comforts (1990)                         4.513889  4.272277
Usual Suspects, The (1995)                       4.513317  4.518248
```

Measuring Rating Disagreement

Suppose you wanted to find the movies that are most divisive between male and female viewers. One way is to add a column to `mean_ratings` containing the difference in means, then sort by that:

```
In [87]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Sorting by `'diff'` yields the movies with the greatest rating difference so that we can see which ones were preferred by women:

```
In [88]: sorted_by_diff = mean_ratings.sort_values(by='diff')

In [89]: sorted_by_diff[:10]
Out[89]:
gender                                        F         M      diff
title
Dirty Dancing (1987)                   3.790378  2.959596 -0.830782
Jumpin' Jack Flash (1986)              3.254717  2.578358 -0.676359
Grease (1978)                          3.975265  3.367041 -0.608224
Little Women (1994)                    3.870588  3.321739 -0.548849
Steel Magnolias (1989)                 3.901734  3.365957 -0.535777
Anastasia (1997)                       3.800000  3.281609 -0.518391
Rocky Horror Picture Show, The (1975)  3.673016  3.160131 -0.512885
Color Purple, The (1985)               4.158192  3.659341 -0.498851
Age of Innocence, The (1993)           3.827068  3.339506 -0.487561
Free Willy (1993)                      2.921348  2.438776 -0.482573
```

Reversing the order of the rows and again slicing off the top 10 rows, we get the movies preferred by men that women didn't rate as highly:

```
# Reverse order of rows, take first 10 rows
In [90]: sorted_by_diff[::-1][:10]
Out[90]:
gender                                         F         M      diff
title
Good, The Bad and The Ugly, The (1966)  3.494949  4.221300  0.726351
Kentucky Fried Movie, The (1977)        2.878788  3.555147  0.676359
Dumb & Dumber (1994)                    2.697987  3.336595  0.638608
Longest Day, The (1962)                 3.411765  4.031447  0.619682
Cable Guy, The (1996)                   2.250000  2.863787  0.613787
Evil Dead II (Dead By Dawn) (1987)      3.297297  3.909283  0.611985
Hidden, The (1987)                      3.137931  3.745098  0.607167
Rocky III (1982)                        2.361702  2.943503  0.581801
Caddyshack (1980)                       3.396135  3.969737  0.573602
For a Few Dollars More (1965)           3.409091  3.953795  0.544704
```

Suppose instead you wanted the movies that elicited the most disagreement among viewers, independent of gender identification. Disagreement can be measured by the variance or standard deviation of the ratings:

```
# Standard deviation of rating grouped by title
In [91]: rating_std_by_title = data.groupby('title')['rating'].std()

# Filter down to active_titles
In [92]: rating_std_by_title = rating_std_by_title.loc[active_titles]

# Order Series by value in descending order
In [93]: rating_std_by_title.sort_values(ascending=False)[:10]
Out[93]:
title
Dumb & Dumber (1994)                     1.321333
Blair Witch Project, The (1999)          1.316368
Natural Born Killers (1994)              1.307198
Tank Girl (1995)                         1.277695
Rocky Horror Picture Show, The (1975)    1.260177
Eyes Wide Shut (1999)                    1.259624
Evita (1996)                             1.253631
Billy Madison (1995)                     1.249970
Fear and Loathing in Las Vegas (1998)    1.246408
Bicentennial Man (1999)                  1.245533
Name: rating, dtype: float64
```

You may have noticed that movie genres are given as a pipe-separated (`|`) string. If you wanted to do some analysis by genre, more work would be required to transform the genre information into a more usable form.

14.3 US Baby Names 1880–2010

The United States Social Security Administration (SSA) has made available data on the frequency of baby names from 1880 through the present. Hadley Wickham, an author of several popular R packages, has often made use of this dataset in illustrating data manipulation in R.

We need to do some data wrangling to load this dataset, but once we do that we will have a DataFrame that looks like this:

```
In [4]: names.head(10)
Out[4]:
        name sex  births  year
0       Mary   F    7065  1880
1       Anna   F    2604  1880
2       Emma   F    2003  1880
3  Elizabeth   F    1939  1880
4     Minnie   F    1746  1880
5   Margaret   F    1578  1880
6        Ida   F    1472  1880
```

```
7      Alice   F    1414  1880
8     Bertha   F    1320  1880
9      Sarah   F    1288  1880
```

There are many things you might want to do with the dataset:

- Visualize the proportion of babies given a particular name (your own, or another name) over time
- Determine the relative rank of a name
- Determine the most popular names in each year or the names whose popularity has advanced or declined the most
- Analyze trends in names: vowels, consonants, length, overall diversity, changes in spelling, first and last letters
- Analyze external sources of trends: biblical names, celebrities, demographic changes

With the tools in this book, many of these kinds of analyses are within reach, so I will walk you through some of them.

As of this writing, the US Social Security Administration makes available data files, one per year, containing the total number of births for each sex/name combination. The raw archive of these files can be obtained from *http://www.ssa.gov/oact/babynames/limits.html*.

In the event that this page has been moved by the time you're reading this, it can most likely be located again by an internet search. After downloading the "National data" file *names.zip* and unzipping it, you will have a directory containing a series of files like *yob1880.txt*. I use the Unix `head` command to look at the first 10 lines of one of the files (on Windows, you can use the `more` command or open it in a text editor):

```
In [94]: !head -n 10 datasets/babynames/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

As this is already in a nicely comma-separated form, it can be loaded into a DataFrame with `pandas.read_csv`:

```
In [95]: import pandas as pd

In [96]: names1880 = pd.read_csv('datasets/babynames/yob1880.txt',
   ....:                         names=['name', 'sex', 'births'])

In [97]: names1880
Out[97]:
           name sex  births
0          Mary   F    7065
1          Anna   F    2604
2          Emma   F    2003
3     Elizabeth   F    1939
4        Minnie   F    1746
...         ...  ..     ...
1995     Woodie   M       5
1996     Worthy   M       5
1997     Wright   M       5
1998       York   M       5
1999  Zachariah   M       5
[2000 rows x 3 columns]
```

These files only contain names with at least five occurrences in each year, so for simplicity's sake we can use the sum of the births column by sex as the total number of births in that year:

```
In [98]: names1880.groupby('sex').births.sum()
Out[98]:
sex
F     90993
M    110493
Name: births, dtype: int64
```

Since the dataset is split into files by year, one of the first things to do is to assemble all of the data into a single DataFrame and further to add a `year` field. You can do this using `pandas.concat`:

```
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
    path = 'datasets/babynames/yob%d.txt' % year
    frame = pd.read_csv(path, names=columns)

    frame['year'] = year
    pieces.append(frame)

# Concatenate everything into a single DataFrame
names = pd.concat(pieces, ignore_index=True)
```

There are a couple things to note here. First, remember that concat glues the DataFrame objects together row-wise by default. Secondly, you have to pass ignore_index=True because we're not interested in preserving the original row numbers returned from read_csv. So we now have a very large DataFrame containing all of the names data:

```
In [100]: names
Out[100]:
                name sex  births  year
0               Mary   F    7065  1880
1               Anna   F    2604  1880
2               Emma   F    2003  1880
3          Elizabeth   F    1939  1880
4             Minnie   F    1746  1880
...              ...  ..     ...   ...
1690779      Zymaire   M       5  2010
1690780       Zyonne   M       5  2010
1690781    Zyquarius   M       5  2010
1690782        Zyran   M       5  2010
1690783        Zzyzx   M       5  2010
[1690784 rows x 4 columns]
```

With this data in hand, we can already start aggregating the data at the year and sex level using groupby or pivot_table (see Figure 14-4):

```
In [101]: total_births = names.pivot_table('births', index='year',
   .....:                                  columns='sex', aggfunc=sum)

In [102]: total_births.tail()
Out[102]:
sex         F        M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382

In [103]: total_births.plot(title='Total births by sex and year')
```

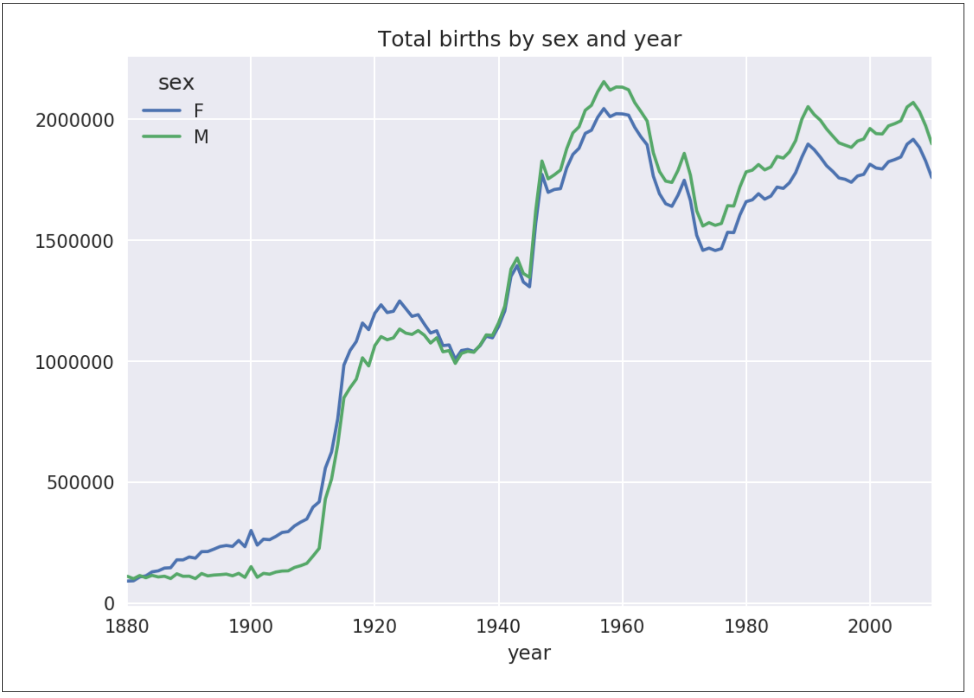

Figure 14-4. Total births by sex and year

Next, let's insert a column `prop` with the fraction of babies given each name relative to the total number of births. A `prop` value of `0.02` would indicate that 2 out of every 100 babies were given a particular name. Thus, we group the data by year and sex, then add the new column to each group:

```
def add_prop(group):
    group['prop'] = group.births / group.births.sum()
    return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```

The resulting complete dataset now has the following columns:

```
In [105]: names
Out[105]:
                 name sex  births  year      prop
0                Mary   F    7065  1880  0.077643
1                Anna   F    2604  1880  0.028618
2                Emma   F    2003  1880  0.022013
3           Elizabeth   F    1939  1880  0.021309
4              Minnie   F    1746  1880  0.019188
...               ...  ..     ...   ...       ...
1690779       Zymaire   M       5  2010  0.000003
1690780        Zyonne   M       5  2010  0.000003
1690781     Zyquarius   M       5  2010  0.000003
```

```
1690782       Zyran    M        5  2010  0.000003
1690783       Zzyzx    M        5  2010  0.000003
[1690784 rows x 5 columns]
```

When performing a group operation like this, it's often valuable to do a sanity check, like verifying that the prop column sums to 1 within all the groups:

```
In [106]: names.groupby(['year', 'sex']).prop.sum()
Out[106]:
year  sex
1880  F      1.0
      M      1.0
1881  F      1.0
      M      1.0
1882  F      1.0
            ...
2008  M      1.0
2009  F      1.0
      M      1.0
2010  F      1.0
      M      1.0
Name: prop, Length: 262, dtype: float64
```

Now that this is done, I'm going to extract a subset of the data to facilitate further analysis: the top 1,000 names for each sex/year combination. This is yet another group operation:

```
def get_top1000(group):
    return group.sort_values(by='births', ascending=False)[:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
# Drop the group index, not needed
top1000.reset_index(inplace=True, drop=True)
```

If you prefer a do-it-yourself approach, try this instead:

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_values(by='births', ascending=False)[:1000])
top1000 = pd.concat(pieces, ignore_index=True)
```

The resulting dataset is now quite a bit smaller:

```
In [108]: top1000
Out[108]:
             name sex  births  year      prop
0            Mary   F    7065  1880  0.077643
1            Anna   F    2604  1880  0.028618
2            Emma   F    2003  1880  0.022013
3       Elizabeth   F    1939  1880  0.021309
4          Minnie   F    1746  1880  0.019188
...           ...  ..     ...   ...       ...
261872     Camilo   M     194  2010  0.000102
261873     Destin   M     194  2010  0.000102
```

```
261874      Jaquan    M     194  2010  0.000102
261875      Jaydan    M     194  2010  0.000102
261876      Maxton    M     193  2010  0.000102
[261877 rows x 5 columns]
```

We'll use this Top 1,000 dataset in the following investigations into the data.

Analyzing Naming Trends

With the full dataset and Top 1,000 dataset in hand, we can start analyzing various naming trends of interest. Splitting the Top 1,000 names into the boy and girl portions is easy to do first:

```
In [109]: boys = top1000[top1000.sex == 'M']

In [110]: girls = top1000[top1000.sex == 'F']
```

Simple time series, like the number of Johns or Marys for each year, can be plotted but require a bit of munging to be more useful. Let's form a pivot table of the total number of births by year and name:

```
In [111]: total_births = top1000.pivot_table('births', index='year',
   .....:                                    columns='name',
   .....:                                    aggfunc=sum)
```

Now, this can be plotted for a handful of names with DataFrame's `plot` method (Figure 14-5 shows the result):

```
In [112]: total_births.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
memory usage: 6.9 MB

In [113]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]

In [114]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
   .....:             title="Number of births per year")
```

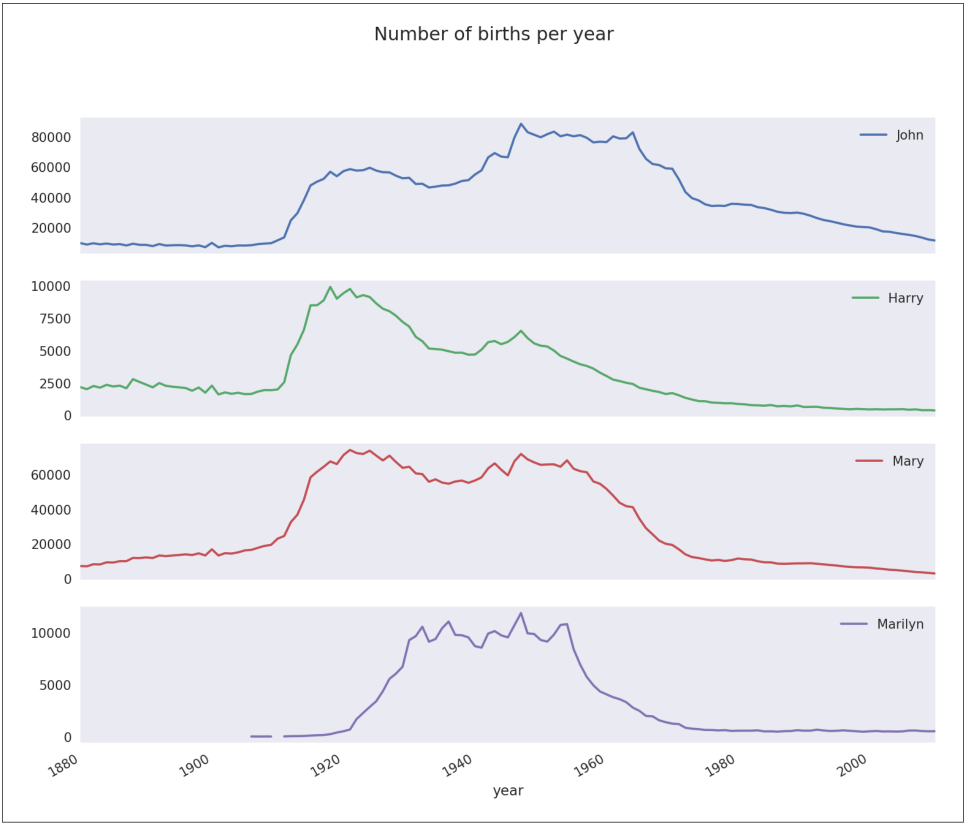

Figure 14-5. A few boy and girl names over time

On looking at this, you might conclude that these names have grown out of favor with the American population. But the story is actually more complicated than that, as will be explored in the next section.

Measuring the increase in naming diversity

One explanation for the decrease in plots is that fewer parents are choosing common names for their children. This hypothesis can be explored and confirmed in the data. One measure is the proportion of births represented by the top 1,000 most popular names, which I aggregate and plot by year and sex (Figure 14-6 shows the resulting plot):

```
In [116]: table = top1000.pivot_table('prop', index='year',
   .....:                                columns='sex', aggfunc=sum)

In [117]: table.plot(title='Sum of table1000.prop by year and sex',
   .....:            yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10)
)
```

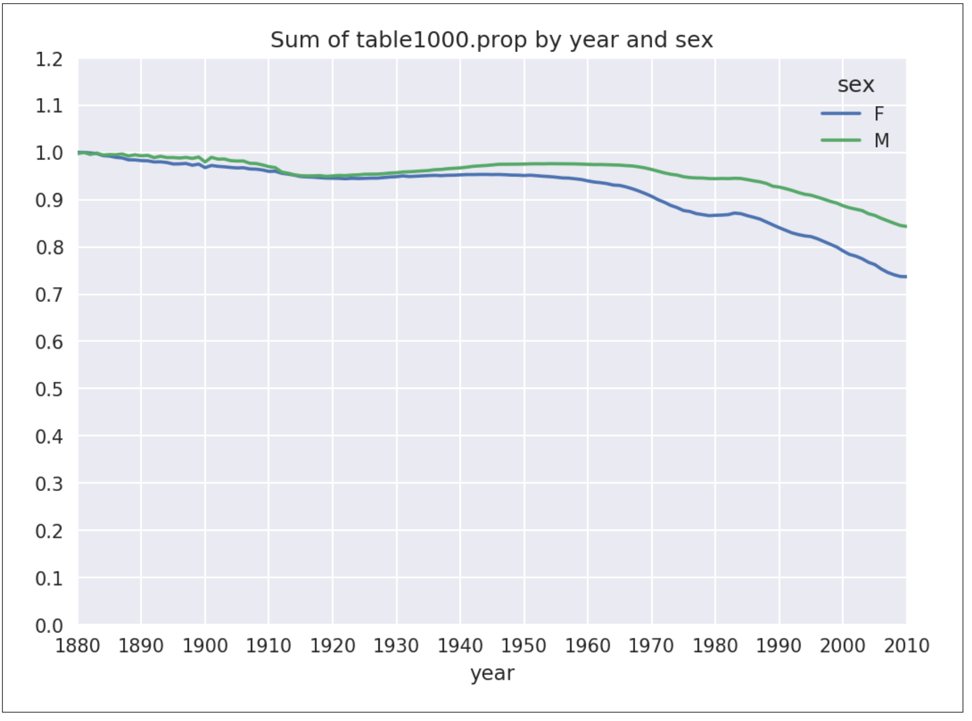

Figure 14-6. Proportion of births represented in top 1000 names by sex

You can see that, indeed, there appears to be increasing name diversity (decreasing total proportion in the top 1,000). Another interesting metric is the number of distinct names, taken in order of popularity from highest to lowest, in the top 50% of births. This number is a bit more tricky to compute. Let's consider just the boy names from 2010:

```
In [118]: df = boys[boys.year == 2010]

In [119]: df
Out[119]:
           name sex  births  year      prop
260877    Jacob   M   21875  2010  0.011523
260878    Ethan   M   17866  2010  0.009411
260879  Michael   M   17133  2010  0.009025
260880   Jayden   M   17030  2010  0.008971
260881  William   M   16870  2010  0.008887
```

```
...           ...  ..     ...   ...        ...
261872    Camilo    M     194  2010  0.000102
261873    Destin    M     194  2010  0.000102
261874    Jaquan    M     194  2010  0.000102
261875    Jaydan    M     194  2010  0.000102
261876    Maxton    M     193  2010  0.000102
[1000 rows x 5 columns]
```

After sorting `prop` in descending order, we want to know how many of the most popular names it takes to reach 50%. You could write a `for` loop to do this, but a vectorized NumPy way is a bit more clever. Taking the cumulative sum, `cumsum`, of `prop` and then calling the method `searchsorted` returns the position in the cumulative sum at which `0.5` would need to be inserted to keep it in sorted order:

```
In [120]: prop_cumsum = df.sort_values(by='prop', ascending=False).prop.cumsum()

In [121]: prop_cumsum[:10]
Out[121]:
260877    0.011523
260878    0.020934
260879    0.029959
260880    0.038930
260881    0.047817
260882    0.056579
260883    0.065155
260884    0.073414
260885    0.081528
260886    0.089621
Name: prop, dtype: float64

In [122]: prop_cumsum.values.searchsorted(0.5)
Out[122]: 116
```

Since arrays are zero-indexed, adding 1 to this result gives you a result of 117. By contrast, in 1900 this number was much smaller:

```
In [123]: df = boys[boys.year == 1900]

In [124]: in1900 = df.sort_values(by='prop', ascending=False).prop.cumsum()

In [125]: in1900.values.searchsorted(0.5) + 1
Out[125]: 25
```

You can now apply this operation to each year/sex combination, `groupby` those fields, and `apply` a function returning the count for each group:

```
def get_quantile_count(group, q=0.5):
    group = group.sort_values(by='prop', ascending=False)
    return group.prop.cumsum().values.searchsorted(q) + 1

diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

This resulting DataFrame `diversity` now has two time series, one for each sex, indexed by year. This can be inspected in IPython and plotted as before (see Figure 14-7):

```
In [128]: diversity.head()
Out[128]:
sex     F   M
year
1880   38  14
1881   38  14
1882   38  15
1883   39  15
1884   39  16

In [129]: diversity.plot(title="Number of popular names in top 50%")
```

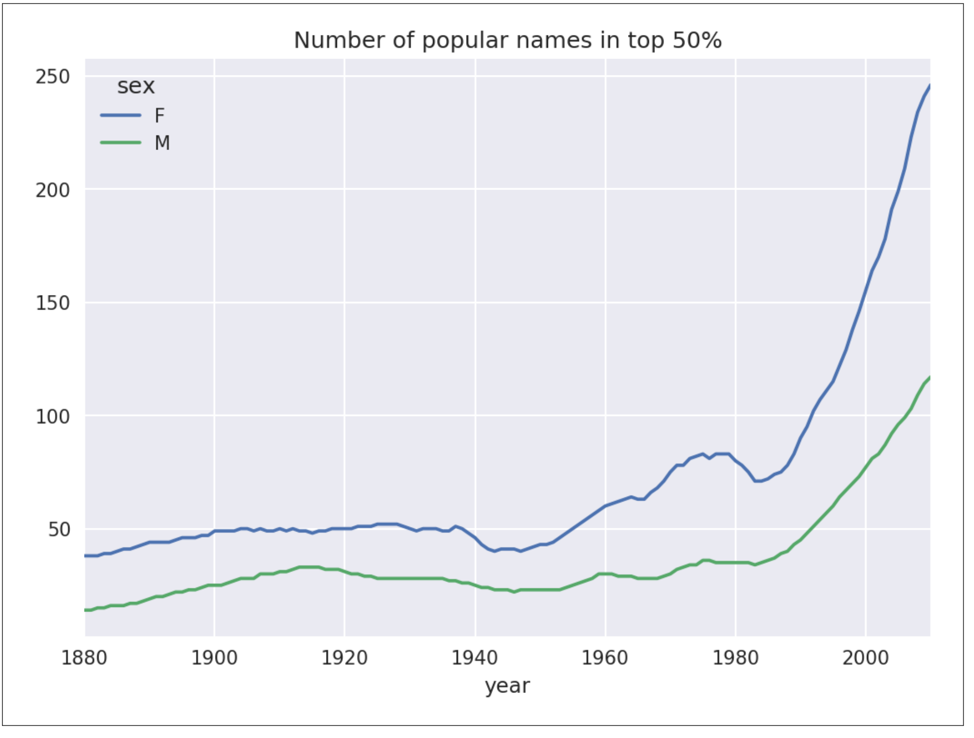

Figure 14-7. Plot of diversity metric by year

As you can see, girl names have always been more diverse than boy names, and they have only become more so over time. Further analysis of what exactly is driving the diversity, like the increase of alternative spellings, is left to the reader.

The "last letter" revolution

In 2007, baby name researcher Laura Wattenberg pointed out on her website that the distribution of boy names by final letter has changed significantly over the last 100 years. To see this, we first aggregate all of the births in the full dataset by year, sex, and final letter:

```
# extract last letter from name column
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'

table = names.pivot_table('births', index=last_letters,
                          columns=['sex', 'year'], aggfunc=sum)
```

Then we select out three representative years spanning the history and print the first few rows:

```
In [131]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')

In [132]: subtable.head()
Out[132]:
sex                 F                            M
year             1910      1960      2010     1910      1960      2010
last_letter
a            108376.0  691247.0  670605.0    977.0    5204.0   28438.0
b                 NaN     694.0     450.0    411.0    3912.0   38859.0
c                 5.0      49.0     946.0    482.0   15476.0   23125.0
d              6750.0    3729.0    2607.0  22111.0  262112.0   44398.0
e            133569.0  435013.0  313833.0  28655.0  178823.0  129012.0
```

Next, normalize the table by total births to compute a new table containing proportion of total births for each sex ending in each letter:

```
In [133]: subtable.sum()
Out[133]:
sex  year
F    1910     396416.0
     1960    2022062.0
     2010    1759010.0
M    1910     194198.0
     1960    2132588.0
     2010    1898382.0
dtype: float64

In [134]: letter_prop = subtable / subtable.sum()

In [135]: letter_prop
Out[135]:
sex                 F                             M
year             1910      1960      2010      1910      1960      2010
last_letter
a            0.273390  0.341853  0.381240  0.005031  0.002440  0.014980
```

```
b             NaN  0.000343  0.000256  0.002116  0.001834  0.020470
c        0.000013  0.000024  0.000538  0.002482  0.007257  0.012181
d        0.017028  0.001844  0.001482  0.113858  0.122908  0.023387
e        0.336941  0.215133  0.178415  0.147556  0.083853  0.067959
...           ...       ...       ...       ...       ...       ...
v             NaN  0.000060  0.000117  0.000113  0.000037  0.001434
w        0.000020  0.000031  0.001182  0.006329  0.007711  0.016148
x        0.000015  0.000037  0.000727  0.003965  0.001851  0.008614
y        0.110972  0.152569  0.116828  0.077349  0.160987  0.058168
z        0.002439  0.000659  0.000704  0.000170  0.000184  0.001831
[26 rows x 6 columns]
```

With the letter proportions now in hand, we can make bar plots for each sex broken down by year (see Figure 14-8):

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female',
                      legend=False)
```

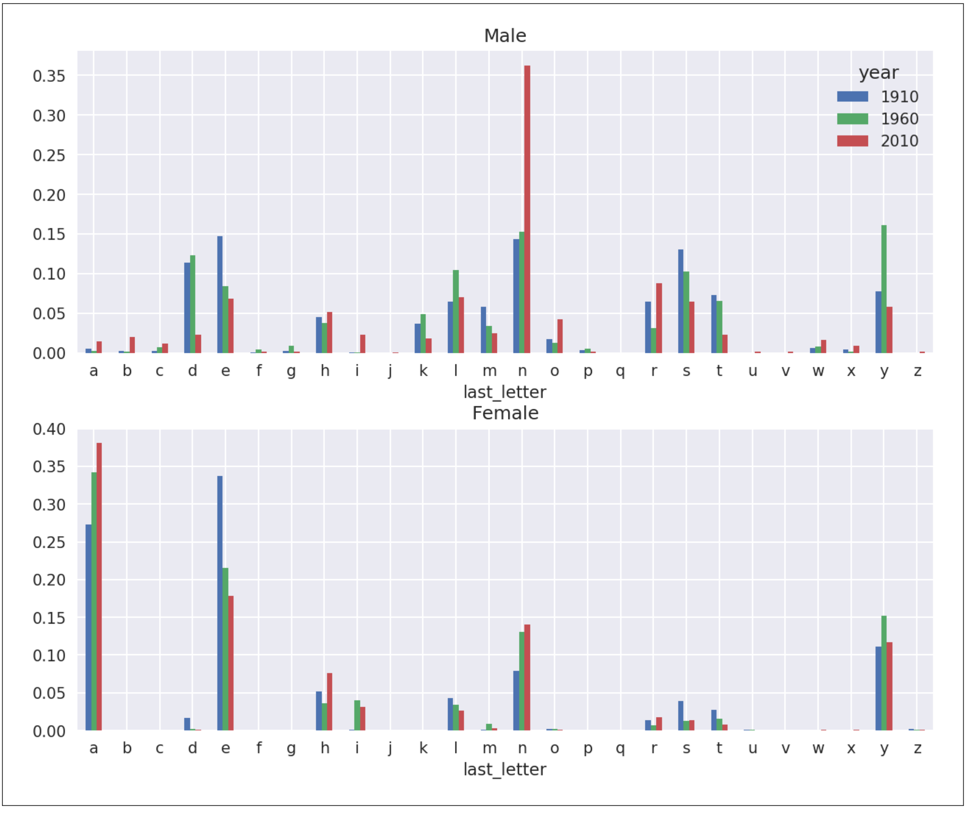

Figure 14-8. Proportion of boy and girl names ending in each letter

As you can see, boy names ending in *n* have experienced significant growth since the 1960s. Going back to the full table created before, I again normalize by year and sex and select a subset of letters for the boy names, finally transposing to make each column a time series:

```
In [138]: letter_prop = table / table.sum()

In [139]: dny_ts = letter_prop.loc[['d', 'n', 'y'], 'M'].T

In [140]: dny_ts.head()
Out[140]:
last_letter         d         n         y
year
1880         0.083055  0.153213  0.075760
1881         0.083247  0.153214  0.077451
1882         0.085340  0.149560  0.077537
1883         0.084066  0.151646  0.079144
1884         0.086120  0.149915  0.080405
```

With this DataFrame of time series in hand, I can make a plot of the trends over time again with its `plot` method (see Figure 14-9):

```
In [143]: dny_ts.plot()
```

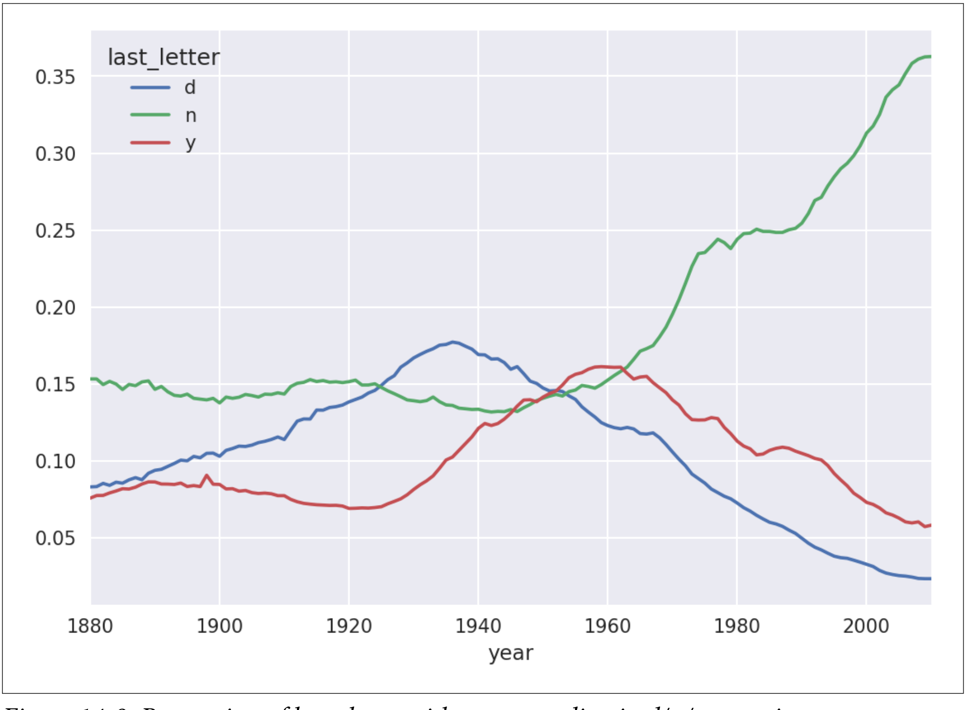

Figure 14-9. Proportion of boys born with names ending in d/n/y over time

Boy names that became girl names (and vice versa)

Another fun trend is looking at boy names that were more popular with one sex earlier in the sample but have "changed sexes" in the present. One example is the name Lesley or Leslie. Going back to the `top1000` DataFrame, I compute a list of names occurring in the dataset starting with "lesl":

```
In [144]: all_names = pd.Series(top1000.name.unique())

In [145]: lesley_like = all_names[all_names.str.lower().str.contains('lesl')]

In [146]: lesley_like
Out[146]:
632     Leslie
2294    Lesley
4262    Leslee
4728     Lesli
6103     Lesly
dtype: object
```

From there, we can filter down to just those names and sum births grouped by name to see the relative frequencies:

```
In [147]: filtered = top1000[top1000.name.isin(lesley_like)]

In [148]: filtered.groupby('name').births.sum()
Out[148]:
name
Leslee      1082
Lesley     35022
Lesli        929
Leslie    370429
Lesly      10067
Name: births, dtype: int64
```

Next, let's aggregate by sex and year and normalize within year:

```
In [149]: table = filtered.pivot_table('births', index='year',
   .....:                              columns='sex', aggfunc='sum')

In [150]: table = table.div(table.sum(1), axis=0)

In [151]: table.tail()
Out[151]:
sex     F   M
year
2006  1.0 NaN
2007  1.0 NaN
2008  1.0 NaN
2009  1.0 NaN
2010  1.0 NaN
```

Lastly, it's now possible to make a plot of the breakdown by sex over time (Figure 14-10):

```
In [153]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

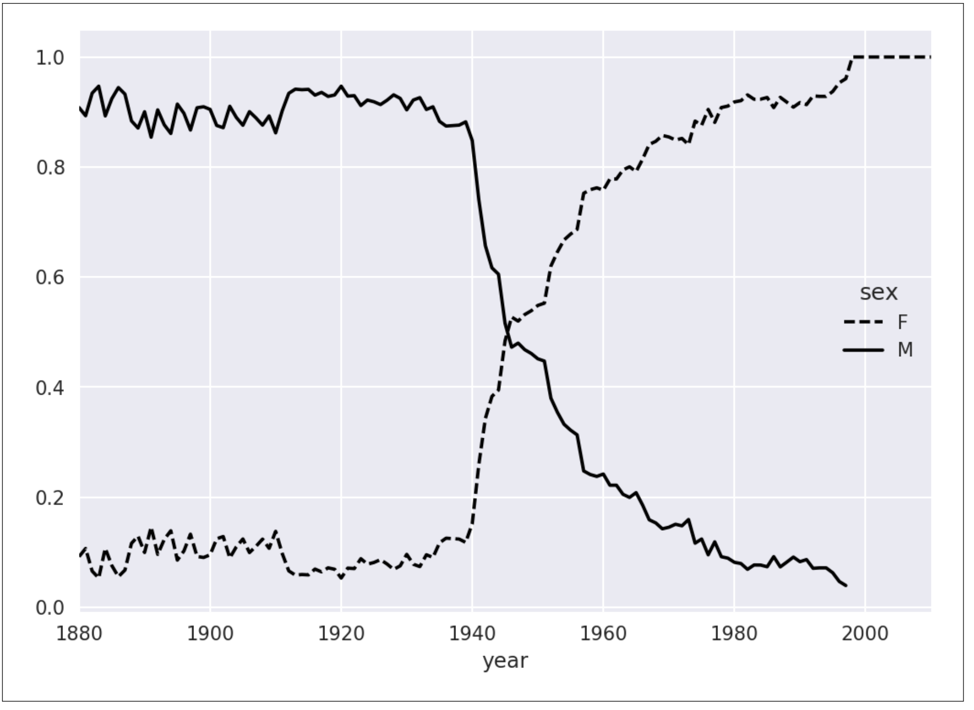

Figure 14-10. Proportion of male/female Lesley-like names over time

14.4 USDA Food Database

The US Department of Agriculture makes available a database of food nutrient information. Programmer Ashley Williams made available a version of this database in JSON format. The records look like this:

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,
Wing, meat and skin with breading",
  "tags": ["KFC"],
  "manufacturer": "Kentucky Fried Chicken",
  "group": "Fast Foods",
  "portions": [
    {
      "amount": 1,
      "unit": "wing, with skin",
      "grams": 68.0
```

```
      },

      ...
    ],
    "nutrients": [
      {
        "value": 20.8,
        "units": "g",
        "description": "Protein",
        "group": "Composition"
      },

      ...
    ]
  }
```

Each food has a number of identifying attributes along with two lists of nutrients and portion sizes. Data in this form is not particularly amenable to analysis, so we need to do some work to wrangle the data into a better form.

After downloading and extracting the data from the link, you can load it into Python with any JSON library of your choosing. I'll use the built-in Python `json` module:

```
In [154]: import json

In [155]: db = json.load(open('datasets/usda_food/database.json'))

In [156]: len(db)
Out[156]: 6636
```

Each entry in `db` is a dict containing all the data for a single food. The `'nutrients'` field is a list of dicts, one for each nutrient:

```
In [157]: db[0].keys()
Out[157]: dict_keys(['id', 'description', 'tags', 'manufacturer', 'group', 'porti
ons', 'nutrients'])

In [158]: db[0]['nutrients'][0]
Out[158]:
{'description': 'Protein',
 'group': 'Composition',
 'units': 'g',
 'value': 25.18}

In [159]: nutrients = pd.DataFrame(db[0]['nutrients'])

In [160]: nutrients[:7]
Out[160]:
                     description        group units    value
0                        Protein  Composition     g    25.18
1              Total lipid (fat)  Composition     g    29.20
2    Carbohydrate, by difference  Composition     g     3.06
3                            Ash        Other     g     3.28
```

```
4                        Energy       Energy  kcal   376.00
5                         Water  Composition     g    39.28
6                        Energy       Energy    kJ  1573.00
```

When converting a list of dicts to a DataFrame, we can specify a list of fields to extract. We'll take the food names, group, ID, and manufacturer:

```
In [161]: info_keys = ['description', 'group', 'id', 'manufacturer']

In [162]: info = pd.DataFrame(db, columns=info_keys)

In [163]: info[:5]
Out[163]:
                          description                   group    id  \
0                     Cheese, caraway  Dairy and Egg Products  1008
1                     Cheese, cheddar  Dairy and Egg Products  1009
2                        Cheese, edam  Dairy and Egg Products  1018
3                        Cheese, feta  Dairy and Egg Products  1019
4  Cheese, mozzarella, part skim milk  Dairy and Egg Products  1028
  manufacturer
0
1
2
3
4

In [164]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
description     6636 non-null object
group           6636 non-null object
id              6636 non-null int64
manufacturer    5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

You can see the distribution of food groups with `value_counts`:

```
In [165]: pd.value_counts(info.group)[:10]
Out[165]:
Vegetables and Vegetable Products    812
Beef Products                        618
Baked Products                       496
Breakfast Cereals                    403
Fast Foods                           365
Legumes and Legume Products          365
Lamb, Veal, and Game Products        345
Sweets                               341
Pork Products                        328
Fruits and Fruit Juices              328
Name: group, dtype: int64
```

Now, to do some analysis on all of the nutrient data, it's easiest to assemble the nutrients for each food into a single large table. To do so, we need to take several steps. First, I'll convert each list of food nutrients to a DataFrame, add a column for the food `id`, and append the DataFrame to a list. Then, these can be concatenated together with `concat`:

If all goes well, `nutrients` should look like this:

```
In [167]: nutrients
Out[167]:
                               description        group units    value     id
0                                  Protein  Composition     g   25.180   1008
1                        Total lipid (fat)  Composition     g   29.200   1008
2              Carbohydrate, by difference  Composition     g    3.060   1008
3                                      Ash        Other     g    3.280   1008
4                                   Energy       Energy  kcal  376.000   1008
...                                    ...          ...   ...      ...    ...
389350                 Vitamin B-12, added     Vitamins   mcg    0.000  43546
389351                         Cholesterol        Other    mg    0.000  43546
389352        Fatty acids, total saturated        Other     g    0.072  43546
389353  Fatty acids, total monounsaturated        Other     g    0.028  43546
389354  Fatty acids, total polyunsaturated        Other     g    0.041  43546
[389355 rows x 5 columns]
```

I noticed that there are duplicates in this DataFrame, so it makes things easier to drop them:

```
In [168]: nutrients.duplicated().sum()  # number of duplicates
Out[168]: 14179

In [169]: nutrients = nutrients.drop_duplicates()
```

Since `'group'` and `'description'` are in both DataFrame objects, we can rename for clarity:

```
In [170]: col_mapping = {'description' : 'food',
   .....:                'group'       : 'fgroup'}

In [171]: info = info.rename(columns=col_mapping, copy=False)

In [172]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
food            6636 non-null object
fgroup          6636 non-null object
id              6636 non-null int64
manufacturer    5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB

In [173]: col_mapping = {'description' : 'nutrient',
```

```
   .....:                   'group' : 'nutgroup'}

In [174]: nutrients = nutrients.rename(columns=col_mapping, copy=False)

In [175]: nutrients
Out[175]:
                               nutrient     nutgroup units    value     id
0                               Protein  Composition     g   25.180   1008
1                     Total lipid (fat)  Composition     g   29.200   1008
2           Carbohydrate, by difference  Composition     g    3.060   1008
3                                   Ash        Other     g    3.280   1008
4                                Energy       Energy  kcal  376.000   1008
...                                 ...          ...   ...      ...    ...
389350              Vitamin B-12, added     Vitamins   mcg    0.000  43546
389351                      Cholesterol        Other    mg    0.000  43546
389352       Fatty acids, total saturated        Other     g    0.072  43546
389353  Fatty acids, total monounsaturated        Other     g    0.028  43546
389354  Fatty acids, total polyunsaturated        Other     g    0.041  43546
[375176 rows x 5 columns]
```

With all of this done, we're ready to merge `info` with `nutrients`:

```
In [176]: ndata = pd.merge(nutrients, info, on='id', how='outer')

In [177]: ndata.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns (total 8 columns):
nutrient        375176 non-null object
nutgroup        375176 non-null object
units           375176 non-null object
value           375176 non-null float64
id              375176 non-null int64
food            375176 non-null object
fgroup          375176 non-null object
manufacturer    293054 non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB

In [178]: ndata.iloc[30000]
Out[178]:
nutrient                                       Glycine
nutgroup                                   Amino Acids
units                                                g
value                                             0.04
id                                                6158
food            Soup, tomato bisque, canned, condensed
fgroup                      Soups, Sauces, and Gravies
manufacturer
Name: 30000, dtype: object
```

We could now make a plot of median values by food group and nutrient type (see Figure 14-11):

```
In [180]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)

In [181]: result['Zinc, Zn'].sort_values().plot(kind='barh')
```

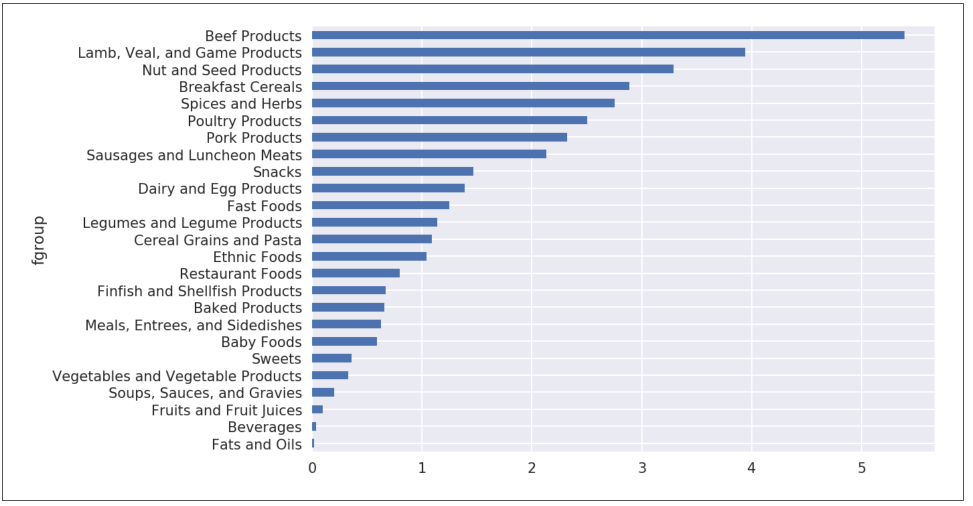

Figure 14-11. Median zinc values by nutrient group

With a little cleverness, you can find which food is most dense in each nutrient:

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.loc[x.value.idxmax()]
get_minimum = lambda x: x.loc[x.value.idxmin()]

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# make the food a little smaller
max_foods.food = max_foods.food.str[:50]
```

The resulting DataFrame is a bit too large to display in the book; here is only the `'Amino Acids'` nutrient group:

```
In [183]: max_foods.loc['Amino Acids']['food']
Out[183]:
nutrient
Alanine                          Gelatins, dry powder, unsweetened
Arginine                              Seeds, sesame flour, low-fat
Aspartic acid                                  Soy protein isolate
Cystine               Seeds, cottonseed flour, low fat (glandless)
Glutamic acid                                  Soy protein isolate
                                   ...
Serine         Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Threonine      Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Tryptophan      Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine       Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
```

```
Valine          Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Name: food, Length: 19, dtype: object
```

14.5 2012 Federal Election Commission Database

The US Federal Election Commission publishes data on contributions to political campaigns. This includes contributor names, occupation and employer, address, and contribution amount. An interesting dataset is from the 2012 US presidential election. A version of the dataset I downloaded in June 2012 is a 150 megabyte CSV file *P00000001-ALL.csv* (see the book's data repository), which can be loaded with `pandas.read_csv`:

```
In [184]: fec = pd.read_csv('datasets/fec/P00000001-ALL.csv')

In [185]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
cmte_id              1001731 non-null object
cand_id              1001731 non-null object
cand_nm              1001731 non-null object
contbr_nm            1001731 non-null object
contbr_city          1001712 non-null object
contbr_st            1001727 non-null object
contbr_zip           1001620 non-null object
contbr_employer      988002 non-null object
contbr_occupation    993301 non-null object
contb_receipt_amt    1001731 non-null float64
contb_receipt_dt     1001731 non-null object
receipt_desc         14166 non-null object
memo_cd              92482 non-null object
memo_text            97770 non-null object
form_tp              1001731 non-null object
file_num             1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB
```

A sample record in the DataFrame looks like this:

```
In [186]: fec.iloc[123456]
Out[186]:
cmte_id             C00431445
cand_id             P80003338
cand_nm         Obama, Barack
contbr_nm         ELLMAN, IRA
contbr_city             TEMPE
                     ...
receipt_desc              NaN
memo_cd                   NaN
memo_text                 NaN
form_tp                 SA17A
```

```
file_num                    772372
Name: 123456, Length: 16, dtype: object
```

You may think of some ways to start slicing and dicing this data to extract informative statistics about donors and patterns in the campaign contributions. I'll show you a number of different analyses that apply techniques in this book.

You can see that there are no political party affiliations in the data, so this would be useful to add. You can get a list of all the unique political candidates using `unique`:

```
In [187]: unique_cands = fec.cand_nm.unique()

In [188]: unique_cands
Out[188]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
       "Roemer, Charles E. 'Buddy' III", 'Pawlenty, Timothy',
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick', 'Cain, Herman',
       'Gingrich, Newt', 'McCotter, Thaddeus G', 'Huntsman, Jon',
       'Perry, Rick'], dtype=object)

In [189]: unique_cands[2]
Out[189]: 'Obama, Barack'
```

One way to indicate party affiliation is using a dict:[1]

```
parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
           'Pawlenty, Timothy': 'Republican',
           'Perry, Rick': 'Republican',
           "Roemer, Charles E. 'Buddy' III": 'Republican',
           'Romney, Mitt': 'Republican',
           'Santorum, Rick': 'Republican'}
```

Now, using this mapping and the `map` method on Series objects, you can compute an array of political parties from the candidate names:

```
In [191]: fec.cand_nm[123456:123461]
Out[191]:
123456    Obama, Barack
123457    Obama, Barack
123458    Obama, Barack
123459    Obama, Barack
123460    Obama, Barack
```

1 This makes the simplifying assumption that Gary Johnson is a Republican even though he later became the Libertarian party candidate.

```
Name: cand_nm, dtype: object

In [192]: fec.cand_nm[123456:123461].map(parties)
Out[192]:
123456    Democrat
123457    Democrat
123458    Democrat
123459    Democrat
123460    Democrat
Name: cand_nm, dtype: object

# Add it as a column
In [193]: fec['party'] = fec.cand_nm.map(parties)

In [194]: fec['party'].value_counts()
Out[194]:
Democrat      593746
Republican    407985
Name: party, dtype: int64
```

A couple of data preparation points. First, this data includes both contributions and refunds (negative contribution amount):

```
In [195]: (fec.contb_receipt_amt > 0).value_counts()
Out[195]:
True     991475
False     10256
Name: contb_receipt_amt, dtype: int64
```

To simplify the analysis, I'll restrict the dataset to positive contributions:

```
In [196]: fec = fec[fec.contb_receipt_amt > 0]
```

Since Barack Obama and Mitt Romney were the main two candidates, I'll also prepare a subset that just has contributions to their campaigns:

```
In [197]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney, Mitt'])]
```

Donation Statistics by Occupation and Employer

Donations by occupation is another oft-studied statistic. For example, lawyers (attorneys) tend to donate more money to Democrats, while business executives tend to donate more to Republicans. You have no reason to believe me; you can see for yourself in the data. First, the total number of donations by occupation is easy:

```
In [198]: fec.contbr_occupation.value_counts()[:10]
Out[198]:
RETIRED                                   233990
INFORMATION REQUESTED                      35107
ATTORNEY                                   34286
HOMEMAKER                                  29931
PHYSICIAN                                  23432
INFORMATION REQUESTED PER BEST EFFORTS     21138
```

```
ENGINEER                                  14334
TEACHER                                   13990
CONSULTANT                                13273
PROFESSOR                                 12555
Name: contbr_occupation, dtype: int64
```

You will notice by looking at the occupations that many refer to the same basic job type, or there are several variants of the same thing. The following code snippet illustrates a technique for cleaning up a few of them by mapping from one occupation to another; note the "trick" of using `dict.get` to allow occupations with no mapping to "pass through":

```
occ_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
   'C.E.O.': 'CEO'
}

# If no mapping provided, return x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
```

I'll also do the same thing for employers:

```
emp_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'SELF' : 'SELF-EMPLOYED',
   'SELF EMPLOYED' : 'SELF-EMPLOYED',
}

# If no mapping provided, return x
f = lambda x: emp_mapping.get(x, x)
fec.contbr_employer = fec.contbr_employer.map(f)
```

Now, you can use `pivot_table` to aggregate the data by party and occupation, then filter down to the subset that donated at least $2 million overall:

```
In [201]: by_occupation = fec.pivot_table('contb_receipt_amt',
   .....:                                 index='contbr_occupation',
   .....:                                 columns='party', aggfunc='sum')

In [202]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]

In [203]: over_2mm
Out[203]:
party                 Democrat    Republican
contbr_occupation
ATTORNEY           11141982.97  7.477194e+06
CEO                 2074974.79  4.211041e+06
CONSULTANT          2459912.71  2.544725e+06
ENGINEER             951525.55  1.818374e+06
```

```
EXECUTIVE          1355161.05  4.138850e+06
...                       ...           ...
PRESIDENT          1878509.95  4.720924e+06
PROFESSOR          2165071.08  2.967027e+05
REAL ESTATE         528902.09  1.625902e+06
RETIRED           25305116.38  2.356124e+07
SELF-EMPLOYED       672393.40  1.640253e+06
[17 rows x 2 columns]
```

It can be easier to look at this data graphically as a bar plot (`'barh'` means horizontal bar plot; see Figure 14-12):

```
In [205]: over_2mm.plot(kind='barh')
```

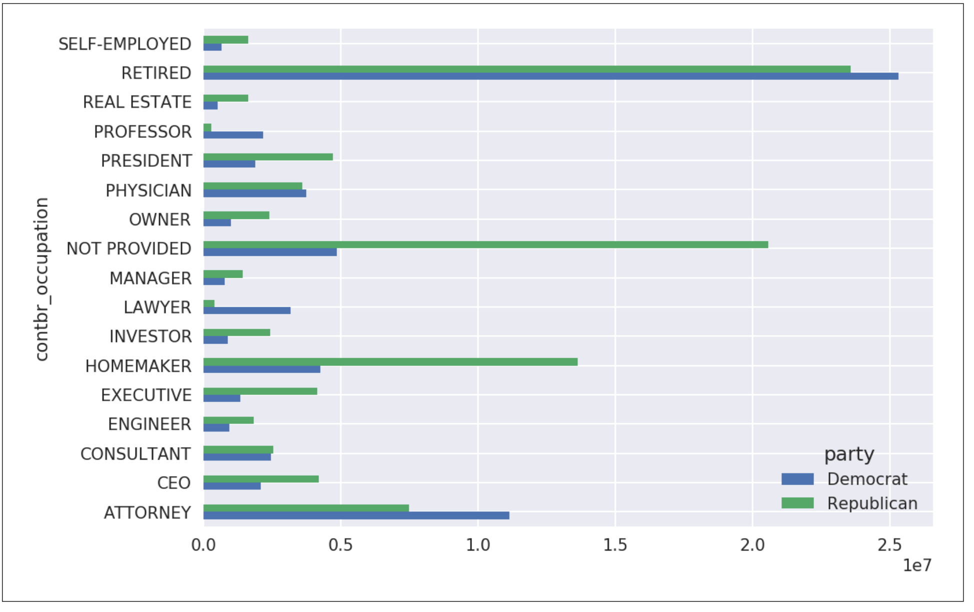

Figure 14-12. Total donations by party for top occupations

You might be interested in the top donor occupations or top companies that donated to Obama and Romney. To do this, you can group by candidate name and use a variant of the `top` method from earlier in the chapter:

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key)['contb_receipt_amt'].sum()
    return totals.nlargest(n)
```

Then aggregate by occupation and employer:

```
In [207]: grouped = fec_mrbo.groupby('cand_nm')

In [208]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)
Out[208]:
```

```
cand_nm        contbr_occupation
Obama, Barack  RETIRED                   25305116.38
               ATTORNEY                  11141982.97
               INFORMATION REQUESTED      4866973.96
               HOMEMAKER                  4248875.80
               PHYSICIAN                  3735124.94
                                             ...
Romney, Mitt   HOMEMAKER                  8147446.22
               ATTORNEY                   5364718.82
               PRESIDENT                  2491244.89
               EXECUTIVE                  2300947.03
               C.E.O.                     1968386.11
Name: contb_receipt_amt, Length: 14, dtype: float64

In [209]: grouped.apply(get_top_amounts, 'contbr_employer', n=10)
Out[209]:
cand_nm        contbr_employer
Obama, Barack  RETIRED                   22694358.85
               SELF-EMPLOYED             17080985.96
               NOT EMPLOYED               8586308.70
               INFORMATION REQUESTED      5053480.37
               HOMEMAKER                  2605408.54
                                             ...
Romney, Mitt   CREDIT SUISSE               281150.00
               MORGAN STANLEY              267266.00
               GOLDMAN SACH & CO.          238250.00
               BARCLAYS CAPITAL            162750.00
               H.I.G. CAPITAL              139500.00
Name: contb_receipt_amt, Length: 20, dtype: float64
```

Bucketing Donation Amounts

A useful way to analyze this data is to use the cut function to discretize the contributor amounts into buckets by contribution size:

```
In [210]: bins = np.array([0, 1, 10, 100, 1000, 10000,
   .....:                  100000, 1000000, 10000000])

In [211]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)

In [212]: labels
Out[212]:
411         (10, 100]
412       (100, 1000]
413       (100, 1000]
414         (10, 100]
415         (10, 100]
              ...
701381      (10, 100]
701382    (100, 1000]
701383        (1, 10]
701384      (10, 100]
```

```
701385    (100, 1000]
Name: contb_receipt_amt, Length: 694282, dtype: category
Categories (8, interval[int64]): [(0, 1] < (1, 10] < (10, 100] < (100, 1000] < (1
000, 10000] <
                                  (10000, 100000] < (100000, 1000000] < (1000000,
 10000000]]
```

We can then group the data for Obama and Romney by name and bin label to get a histogram by donation size:

```
In [213]: grouped = fec_mrbo.groupby(['cand_nm', labels])

In [214]: grouped.size().unstack(0)
Out[214]:
cand_nm              Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                       493.0          77.0
(1, 10]                    40070.0        3681.0
(10, 100]                 372280.0       31853.0
(100, 1000]               153991.0       43357.0
(1000, 10000]              22284.0       26186.0
(10000, 100000]                2.0           1.0
(100000, 1000000]              3.0           NaN
(1000000, 10000000]            4.0           NaN
```

This data shows that Obama received a significantly larger number of small donations than Romney. You can also sum the contribution amounts and normalize within buckets to visualize percentage of total donations of each size by candidate (Figure 14-13 shows the resulting plot):

```
In [216]: bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)

In [217]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)

In [218]: normed_sums
Out[218]:
cand_nm              Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                    0.805182      0.194818
(1, 10]                   0.918767      0.081233
(10, 100]                 0.910769      0.089231
(100, 1000]               0.710176      0.289824
(1000, 10000]             0.447326      0.552674
(10000, 100000]           0.823120      0.176880
(100000, 1000000]         1.000000           NaN
(1000000, 10000000]       1.000000           NaN

In [219]: normed_sums[:-2].plot(kind='barh')
```

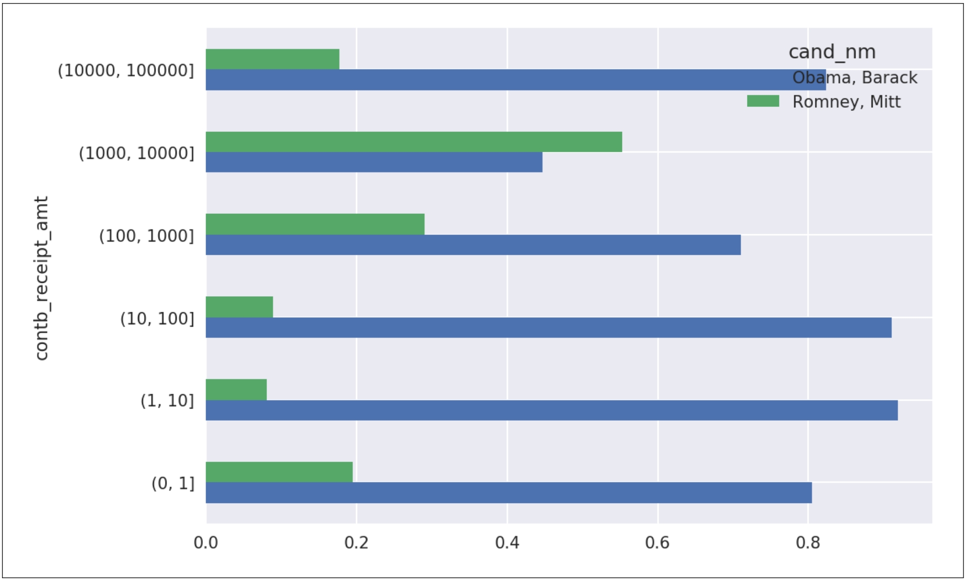

Figure 14-13. Percentage of total donations received by candidates for each donation size

I excluded the two largest bins as these are not donations by individuals.

This analysis can be refined and improved in many ways. For example, you could aggregate donations by donor name and zip code to adjust for donors who gave many small amounts versus one or more large donations. I encourage you to download and explore the dataset yourself.

Donation Statistics by State

Aggregating the data by candidate and state is a routine affair:

```
In [220]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])

In [221]: totals = grouped.contb_receipt_amt.sum().unstack(0).fillna(0)

In [222]: totals = totals[totals.sum(1) > 100000]

In [223]: totals[:10]
Out[223]:
cand_nm    Obama, Barack  Romney, Mitt
contbr_st
AK             281840.15      86204.24
AL             543123.48     527303.51
AR             359247.28     105556.00
AZ            1506476.98    1888436.23
CA           23824984.24   11237636.60
CO            2132429.49    1506714.12
CT            2068291.26    3499475.45
```

```
DC              4373538.80    1025137.50
DE               336669.14      82712.00
FL              7318178.58    8338458.81
```

If you divide each row by the total contribution amount, you get the relative percentage of total donations by state for each candidate:

```
In [224]: percent = totals.div(totals.sum(1), axis=0)

In [225]: percent[:10]
Out[225]:
cand_nm     Obama, Barack  Romney, Mitt
contbr_st
AK               0.765778      0.234222
AL               0.507390      0.492610
AR               0.772902      0.227098
AZ               0.443745      0.556255
CA               0.679498      0.320502
CO               0.585970      0.414030
CT               0.371476      0.628524
DC               0.810113      0.189887
DE               0.802776      0.197224
FL               0.467417      0.532583
```

14.6 Conclusion

We've reached the end of the book's main chapters. I have included some additional content you may find useful in the appendixes.

In the five years since the first edition of this book was published, Python has become a popular and widespread language for data analysis. The programming skills you have developed here will stay relevant for a long time into the future. I hope the programming tools and libraries we've explored serve you well in your work.