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 data set in illustrating data manipulation in R.

```
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 data set:

- 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 with largest increases or decreases.
- 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

Using the tools we've looked at so far, most of these kinds of analyses are very straightforward, so I will walk you through many of them. I encourage you to download and explore the data yourself. If you find an interesting pattern in the data, I would love to hear about it.

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 here:

```
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 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 [367]: !head -n 10 names/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 a nicely comma-separated form, it can be loaded into a DataFrame with `pandas.read_csv`:

```
In [368]: import pandas as pd

In [369]: names1880 = pd.read_csv('names/yob1880.txt', names=['name', 'sex', 'births'])

In [370]: names1880
Out[370]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2000 entries, 0 to 1999
Data columns:
name      2000  non-null values
sex       2000  non-null values
births    2000  non-null values
dtypes: int64(1), object(2)
```

These files only contain names with at least 5 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 [371]: names1880.groupby('sex').births.sum()
Out[371]:
sex
F     90993
M    110493
Name: births
```

Since the data set 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. This is easy to do using `pandas.concat`:

```
# 2010 is the last available year right now
years = range(1880, 2011)

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

for year in years:
    path = 'names/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:

Now the `names` DataFrame looks like:

```
In [373]: names
Out[373]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784  non-null values
sex       1690784  non-null values
births    1690784  non-null values
year      1690784  non-null values
dtypes: int64(2), object(2)
```

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 2-4:

```
In [374]: total_births = names.pivot_table('births', rows='year',
   .....:                                  cols='sex', aggfunc=sum)

In [375]: total_births.tail()
Out[375]:
sex         F        M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382

In [376]: total_births.plot(title='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 was 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):
    # Integer division floors
    births = group.births.astype(float)

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

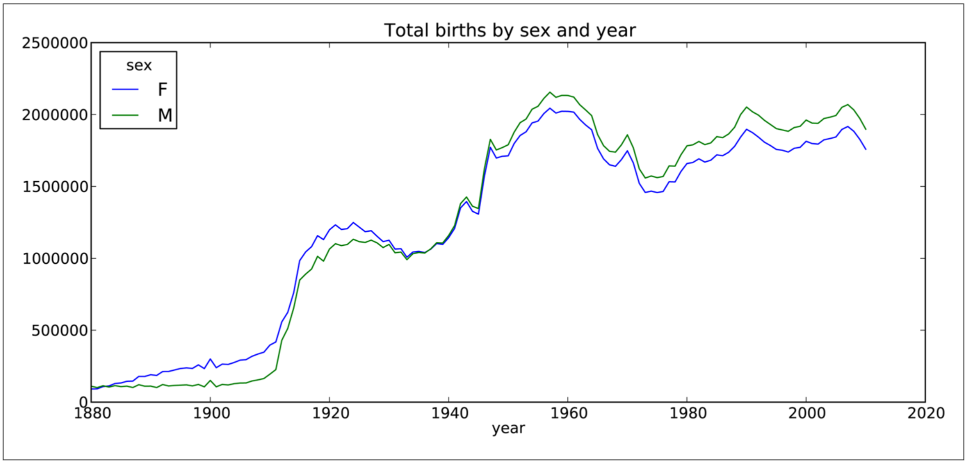

Figure 2-4. Total births by sex and year

> Remember that because `births` is of integer type, we have to cast either the numerator or denominator to floating point to compute a fraction (unless you are using Python 3!).

The resulting complete data set now has the following columns:

```
In [378]: names
Out[378]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784  non-null values
sex       1690784  non-null values
births    1690784  non-null values
year      1690784  non-null values
prop      1690784  non-null values
dtypes: float64(1), int64(2), object(2)
```

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. Since this is floating point data, use `np.allclose` to check that the group sums are sufficiently close to (but perhaps not exactly equal to) 1:

```
In [379]: np.allclose(names.groupby(['year', 'sex']).prop.sum(), 1)
Out[379]: True
```

Now that this is done, I'm going to extract a subset of the data to facilitate further analysis: the top 1000 names for each sex/year combination. This is yet another group operation:

```
def get_top1000(group):
    return group.sort_index(by='births', ascending=False)[:1000]
```

```
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
```

If you prefer a do-it-yourself approach, you could also do:

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

The resulting data set is now quite a bit smaller:

```
In [382]: top1000
Out[382]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 261877 entries, 0 to 261876
Data columns:
name      261877  non-null values
sex       261877  non-null values
births    261877  non-null values
year      261877  non-null values
prop      261877  non-null values
dtypes: float64(1), int64(2), object(2)
```

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

Analyzing Naming Trends

With the full data set and Top 1,000 data set 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 [383]: boys = top1000[top1000.sex == 'M']

In [384]: 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 a bit more useful. Let's form a pivot table of the total number of births by year and name:

```
In [385]: total_births = top1000.pivot_table('births', rows='year', cols='name',
   .....:                                     aggfunc=sum)
```

Now, this can be plotted for a handful of names using DataFrame's `plot` method:

```
In [386]: total_births
Out[386]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6865 entries, Aaden to Zuri
dtypes: float64(6865)

In [387]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]

In [388]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
   .....:             title="Number of births per year")
```

See Figure 2-5 for the result. 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.

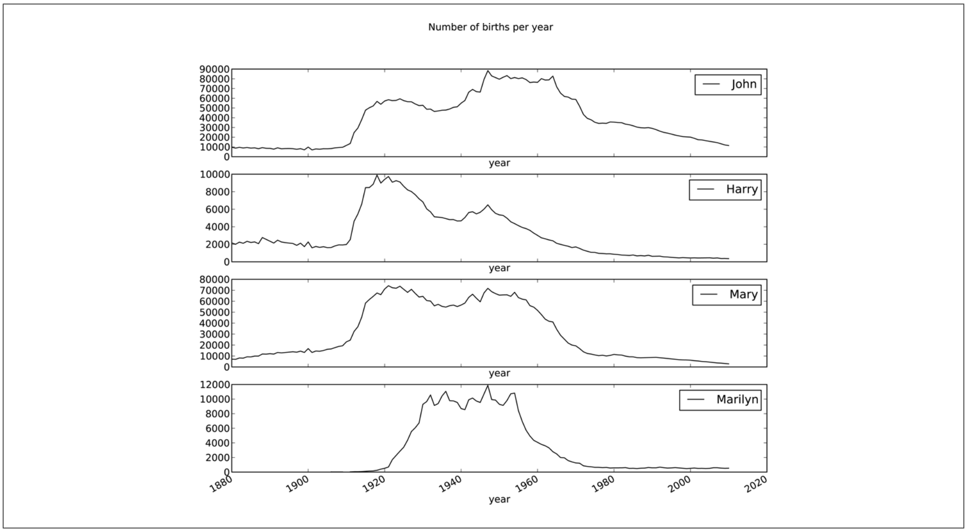

Figure 2-5. A few boy and girl names over time

Measuring the increase in naming diversity

One explanation for the decrease in plots above 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 1000 most popular names, which I aggregate and plot by year and sex:

```
In [390]: table = top1000.pivot_table('prop', rows='year',
   .....:                             cols='sex', aggfunc=sum)

In [391]: table.plot(title='Sum of table1000.prop by year and sex',
   .....:            yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10))
```

See Figure 2-6 for this plot. So 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 [392]: df = boys[boys.year == 2010]

In [393]: df
Out[393]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 260877 to 261876
Data columns:
```

```
name      1000  non-null values
sex       1000  non-null values
births    1000  non-null values
year      1000  non-null values
prop      1000  non-null values
dtypes: float64(1), int64(2), object(2)
```

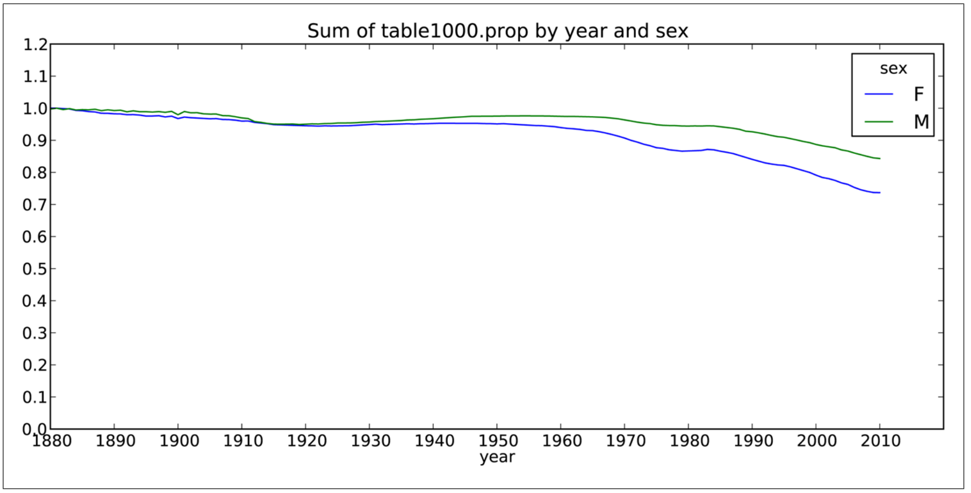

Figure 2-6. Proportion of births represented in top 1000 names by sex

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` 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 [394]: prop_cumsum = df.sort_index(by='prop', ascending=False).prop.cumsum()

In [395]: prop_cumsum[:10]
Out[395]:
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

In [396]: prop_cumsum.searchsorted(0.5)
Out[396]: 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 [397]: df = boys[boys.year == 1900]

In [398]: in1900 = df.sort_index(by='prop', ascending=False).prop.cumsum()

In [399]: in1900.searchsorted(0.5) + 1
Out[399]: 25
```

It should now be fairly straightforward to 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_index(by='prop', ascending=False)
    return group.prop.cumsum().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 2-7):

```
In [401]: diversity.head()
Out[401]:
sex     F   M
year
1880   38  14
1881   38  14
1882   38  15
1883   39  15
1884   39  16

In [402]: diversity.plot(title="Number of popular names in top 50%")
```

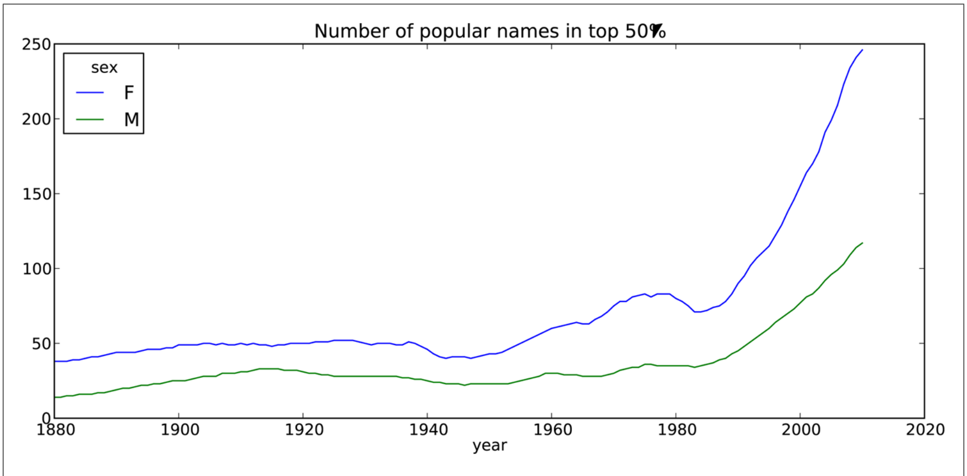

Figure 2-7. Plot of diversity metric by year

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 alternate spellings, is left to the reader.

The "Last letter" Revolution

In 2007, a baby name researcher Laura Wattenberg pointed out on her website (*http://www.babynamewizard.com*) that the distribution of boy names by final letter has changed significantly over the last 100 years. To see this, I first aggregate all of the births in the full data set 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', rows=last_letters,
                          cols=['sex', 'year'], aggfunc=sum)
```

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

```
In [404]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')

In [405]: subtable.head()
Out[405]:
sex               F                       M
year           1910    1960    2010    1910    1960    2010
last_letter
a            108376  691247  670605     977    5204   28438
b               NaN     694     450     411    3912   38859
c                 5      49     946     482   15476   23125
d              6750    3729    2607   22111  262112   44398
e            133569  435013  313833   28655  178823  129012
```

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 [406]: subtable.sum()
Out[406]:
sex  year
F    1910     396416
     1960    2022062
     2010    1759010
M    1910     194198
     1960    2132588
     2010    1898382

In [407]: letter_prop = subtable / subtable.sum().astype(float)
```

With the letter proportions now in hand, I can make bar plots for each sex broken down by year. See Figure 2-8:

```
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)
```

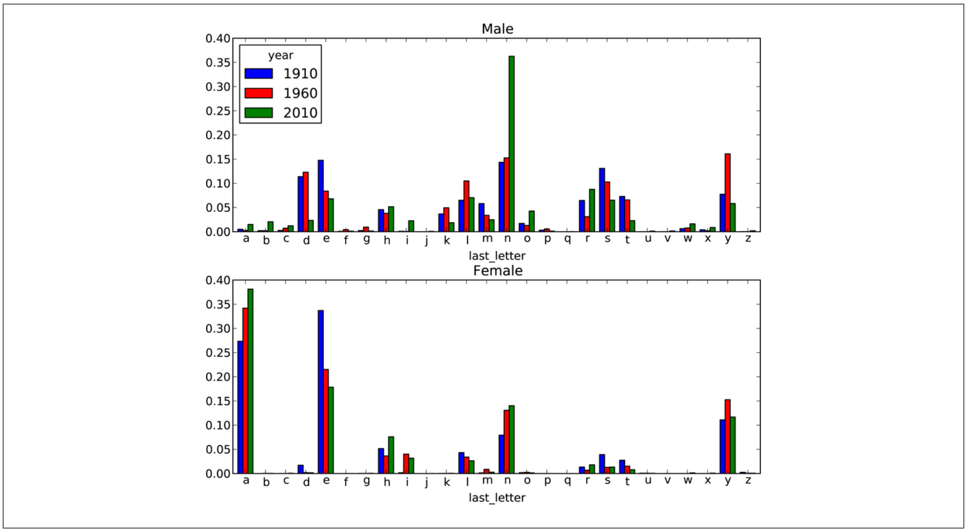

Figure 2-8. Proportion of boy and girl names ending in each letter

As you can see, boy names ending in "n" have experienced significant growth since the 1960s. Going back to the full table created above, 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 [410]: letter_prop = table / table.sum().astype(float)

In [411]: dny_ts = letter_prop.ix[['d', 'n', 'y'], 'M'].T

In [412]: dny_ts.head()
Out[412]:
             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 2-9):

```
In [414]: dny_ts.plot()
```

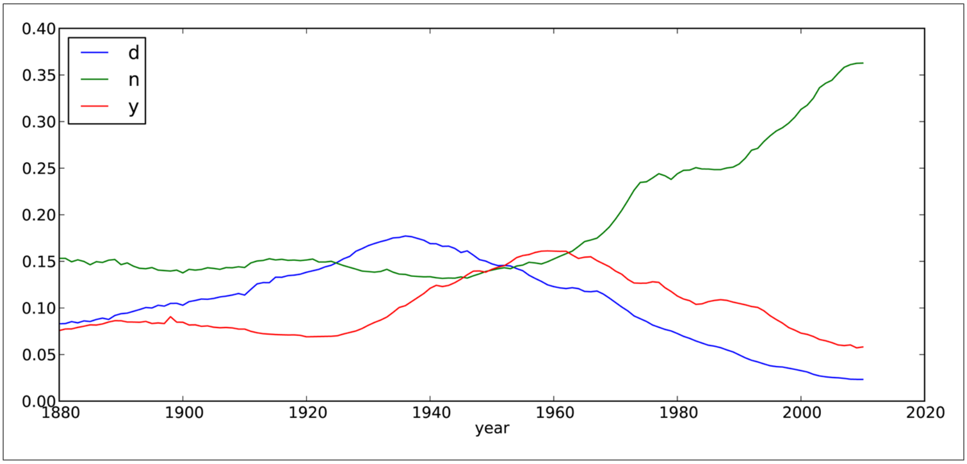

Figure 2-9. Proportion of boys born with names ending in d/n/y over time

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` dataset, I compute a list of names occurring in the dataset starting with `'lesl'`:

```
In [415]: all_names = top1000.name.unique()

In [416]: mask = np.array(['lesl' in x.lower() for x in all_names])

In [417]: lesley_like = all_names[mask]

In [418]: lesley_like
Out[418]: array([Leslie, Lesley, Leslee, Lesli, 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 [419]: filtered = top1000[top1000.name.isin(lesley_like)]

In [420]: filtered.groupby('name').births.sum()
Out[420]:
name
Leslee      1082
Lesley     35022
Lesli        929
Leslie    370429
Lesly      10067
Name: births
```

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

```
In [421]: table = filtered.pivot_table('births', rows='year',
   .....:                              cols='sex', aggfunc='sum')

In [422]: table = table.div(table.sum(1), axis=0)

In [423]: table.tail()
Out[423]:
sex   F   M
year
2006  1 NaN
2007  1 NaN
2008  1 NaN
2009  1 NaN
2010  1 NaN
```

Lastly, it's now easy to make a plot of the breakdown by sex over time (Figure 2-10):

```
In [425]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

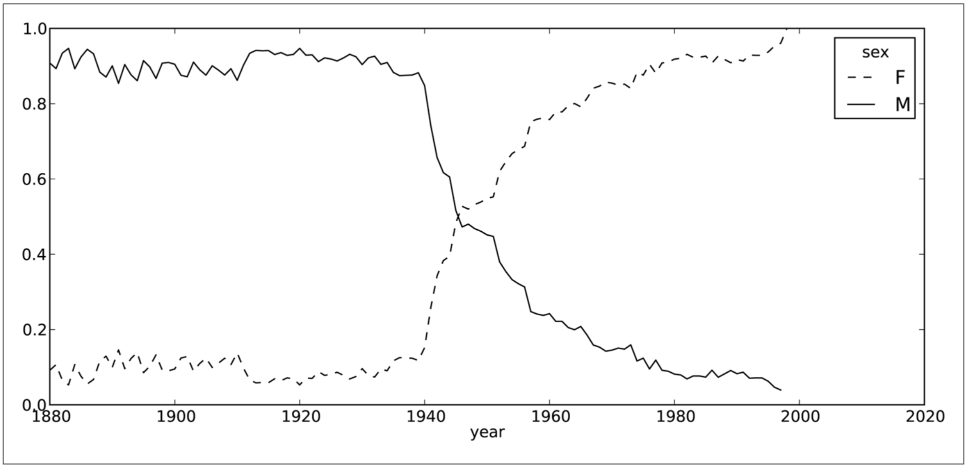

Figure 2-10. Proportion of male/female Lesley-like names over time

Conclusions and The Path Ahead

The examples in this chapter are rather simple, but they're here to give you a bit of a flavor of what sorts of things you can expect in the upcoming chapters. The focus of this book is on *tools* as opposed to presenting more sophisticated analytical methods. Mastering the techniques in this book will enable you to implement your own analyses (assuming you know what you want to do!) in short order.