Python\_for\_Data\_Analysis\_Wes\_McKinney\_First\_release\_c02

CHAPTER 2

Introductory Examples

This book teaches you the Python tools to work productively with data. While readers

may have many different end goals for their work, the tasks required generally fall into

a number of different broad groups:

Interacting with the outside world

Reading and writing with a variety of file formats and databases.

Preparation

Cleaning, munging, combining, normalizing, reshaping, slicing and dicing, and

transforming data for analysis.

Transformation

Applying mathematical and statistical operations to groups of data sets to derive

new data sets. For example, aggregating a large table by group variables.

Modeling and computation

Connecting your data to statistical models, machine learning algorithms, or other

computational tools

Presentation

Creating interactive or static graphical visualizations or textual summaries

In this chapter I will show you a few data sets and some things we can do with them.

These examples are just intended to pique your interest and thus will only be explained

at a high level. Don’t worry if you have no experience with any of these tools; they will

be discussed in great detail throughout the rest of the book. In the code examples you’ll

see input and output prompts like In [15]:; these are from the IPython shell.

1.usa.gov data from bit.ly

In 2011, URL shortening service bit.ly partnered with the United States government

website usa.gov to provide a feed of anonymous data gathered from users who shorten

links ending with .gov or .mil. As of this writing, in addition to providing a live feed,

hourly snapshots are available as downloadable text files.1

17

www.it-ebooks.infoIn 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 you may see something like

In [15]: path = 'ch02/usagov\_bitly\_data2012-03-16-1331923249.txt'

In [16]: open(path).readline()

Out[16]: '{ "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 numerous built-in and 3rd party modules for converting a JSON string into

a Python dictionary object. Here I’ll use the json module and its loads function invoked

on each line in the sample file I downloaded:

import json

path = 'ch02/usagov\_bitly\_data2012-03-16-1331923249.txt'

records = [json.loads(line) for line in open(path)]

If you’ve never programmed in Python before, the last expression here is called a list

comprehension, which is a concise way of applying an operation (like json.loads) to a

collection of strings or other objects. Conveniently, iterating over an open file handle

gives you a sequence of its lines. The resulting object records is now a list of Python

dicts:

In [18]: records[0]

Out[18]:

{u'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like

Gecko) Chrome/17.0.963.78 Safari/535.11',

u'al': u'en-US,en;q=0.8',

u'c': u'US',

u'cy': u'Danvers',

u'g': u'A6qOVH',

u'gr': u'MA',

u'h': u'wfLQtf',

u'hc': 1331822918,

u'hh': u'1.usa.gov',

u'l': u'orofrog',

u'll': [42.576698, -70.954903],

u'nk': 1,

u'r': u'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',

u't': 1331923247,

u'tz': u'America/New\_York',

u'u': u'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}

1. http://www.usa.gov/About/developer-resources/1usagov.shtml

18 | Chapter 2: Introductory Examples

www.it-ebooks.infoNote that Python indices start at 0 and not 1 like some other languages (like R). It’s

now easy to access individual values within records by passing a string for the key you

wish to access:

In [19]: records[0]['tz']

Out[19]: u'America/New\_York'

The u here in front of the quotation stands for unicode, a standard form of string en-

coding. Note that IPython shows the time zone string object representation here rather

than its print equivalent:

In [20]: print records[0]['tz']

America/New\_York

Counting Time Zones in Pure Python

Suppose we were interested in the most often-occurring time zones in the data set (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 [25]: time\_zones = [rec['tz'] for rec in records]

---------------------------------------------------------------------------

KeyError

Traceback (most recent call last)

/home/wesm/book\_scripts/whetting/<ipython> in <module>()

----> 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 [26]: time\_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [27]: time\_zones[:10]

Out[27]:

[u'America/New\_York',

u'America/Denver',

u'America/New\_York',

u'America/Sao\_Paulo',

u'America/New\_York',

u'America/New\_York',

u'Europe/Warsaw',

u'',

u'',

u'']

Just looking at the first 10 time zones we see that some of them are unknown (empty).

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 = {}

1.usa.gov data from bit.ly | 19

www.it-ebooks.infofor x in sequence:

if x in counts:

counts[x] += 1

else:

counts[x] = 1

return counts

If you know a bit more about the Python standard library, you might prefer to 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 [31]: counts = get\_counts(time\_zones)

In [32]: counts['America/New\_York']

Out[32]: 1251

In [33]: len(time\_zones)

Out[33]: 3440

If we wanted the top 10 time zones and their counts, we have to do a little bit of dic-

tionary 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 [35]: top\_counts(counts)

Out[35]:

[(33, u'America/Sao\_Paulo'),

(35, u'Europe/Madrid'),

(36, u'Pacific/Honolulu'),

(37, u'Asia/Tokyo'),

(74, u'Europe/London'),

(191, u'America/Denver'),

(382, u'America/Los\_Angeles'),

(400, u'America/Chicago'),

(521, u''),

(1251, u'America/New\_York')]

20 | Chapter 2: Introductory Examples

www.it-ebooks.infoIf you search the Python standard library, you may find the collections.Counter class,

which makes this task a lot easier:

In [49]: from collections import Counter

In [50]: counts = Counter(time\_zones)

In [51]: counts.most\_common(10)

Out[51]:

[(u'America/New\_York', 1251),

(u'', 521),

(u'America/Chicago', 400),

(u'America/Los\_Angeles', 382),

(u'America/Denver', 191),

(u'Europe/London', 74),

(u'Asia/Tokyo', 37),

(u'Pacific/Honolulu', 36),

(u'Europe/Madrid', 35),

(u'America/Sao\_Paulo', 33)]

Counting Time Zones with pandas

The main pandas data structure is the DataFrame, which you can think of as repre-

senting a table or spreadsheet of data. Creating a DataFrame from the original set of

records is simple:

In [289]: from pandas import DataFrame, Series

In [290]: import pandas as pd

In [291]: frame = DataFrame(records)

In [292]: frame

Out[292]:

<class 'pandas.core.frame.DataFrame'>

Int64Index: 3560 entries, 0 to 3559

Data columns:

\_heartbeat\_

120 non-null values

a

3440 non-null values

al

3094 non-null values

c

2919 non-null values

cy

2919 non-null values

g

3440 non-null values

gr

2919 non-null values

h

3440 non-null values

hc

3440 non-null values

hh

3440 non-null values

kw

93 non-null values

l

3440 non-null values

ll

2919 non-null values

nk

3440 non-null values

r

3440 non-null values

t

3440 non-null values

tz

3440 non-null values

1.usa.gov data from bit.ly | 21

www.it-ebooks.infou

3440 non-null values

dtypes: float64(4), object(14)

In [293]: frame['tz'][:10]

Out[293]:

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

The output shown for the frame is the summary view, shown for large DataFrame ob-

jects. The Series object returned by frame['tz'] has a method value\_counts that gives

us what we’re looking for:

In [294]: tz\_counts = frame['tz'].value\_counts()

In [295]: tz\_counts[:10]

Out[295]:

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

Then, we might want to make a plot of this data using plotting library, matplotlib. You

can do a bit of munging to fill in a substitute value for unknown and missing time zone

data in the records. The fillna function can replace missing (NA) values and unknown

(empty strings) values can be replaced by boolean array indexing:

In [296]: clean\_tz = frame['tz'].fillna('Missing')

In [297]: clean\_tz[clean\_tz == ''] = 'Unknown'

In [298]: tz\_counts = clean\_tz.value\_counts()

In [299]: tz\_counts[:10]

Out[299]:

America/New\_York

1251

Unknown

521

America/Chicago

400

America/Los\_Angeles

382

America/Denver

191

Missing

120

22 | Chapter 2: Introductory Examples

www.it-ebooks.infoEurope/London

Asia/Tokyo

Pacific/Honolulu

Europe/Madrid

74

37

36

35

Making a horizontal bar plot can be accomplished using the plot method on the

counts objects:

In [301]: tz\_counts[:10].plot(kind='barh', rot=0)

See Figure 2-1 for the resulting figure. We’ll explore more tools for working with this

kind of data. For example, the a field contains information about the browser, device,

or application used to perform the URL shortening:

In [302]: frame['a'][1]

Out[302]: u'GoogleMaps/RochesterNY'

In [303]: frame['a'][50]

Out[303]: u'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

In [304]: frame['a'][51]

Out[304]: u'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P925/V10e Build/FRG83G) AppleWebKit/533.1 (K

Figure 2-1. Top time zones in the 1.usa.gov sample data

Parsing all of the interesting information in these “agent” strings may seem like a

daunting task. Luckily, once you have mastered Python’s built-in string functions and

regular expression capabilities, it is really not so bad. For example, we could split off

the first token in the string (corresponding roughly to the browser capability) and make

another summary of the user behavior:

In [305]: results = Series([x.split()[0] for x in frame.a.dropna()])

In [306]: results[:5]

Out[306]:

0

Mozilla/5.0

1

GoogleMaps/RochesterNY

2

Mozilla/4.0

3

Mozilla/5.0

4

Mozilla/5.0

1.usa.gov data from bit.ly | 23

www.it-ebooks.infoIn [307]: results.value\_counts()[:8]

Out[307]:

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

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, I’ll exclude these

from the data:

In [308]: cframe = frame[frame.a.notnull()]

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

In [309]: operating\_system = np.where(cframe['a'].str.contains('Windows'),

.....:

'Windows', 'Not Windows')

In [310]: operating\_system[:5]

Out[310]:

0

Windows

1

Not Windows

2

Windows

3

Not Windows

4

Windows

Name: a

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

systems:

In [311]: by\_tz\_os = cframe.groupby(['tz', operating\_system])

The group counts, analogous to the value\_counts function above, can be computed

using size. This result is then reshaped into a table with unstack:

In [312]: agg\_counts = by\_tz\_os.size().unstack().fillna(0)

In [313]: agg\_counts[:10]

Out[313]:

a

tz

Africa/Cairo

Africa/Casablanca

Africa/Ceuta

Africa/Johannesburg

Africa/Lusaka

America/Anchorage

America/Argentina/Buenos\_Aires

Not WindowsWindows

245

0

0

0

0

0

4

1276

3

1

2

1

1

1

0

24 | Chapter 2: Introductory Examples

www.it-ebooks.infoAmerica/Argentina/Cordoba

America/Argentina/Mendoza

0

0

1

1

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 [314]: indexer = agg\_counts.sum(1).argsort()

In [315]: indexer[:10]

Out[315]:

tz

24

20

21

92

87

53

54

57

26

55

Africa/Cairo

Africa/Casablanca

Africa/Ceuta

Africa/Johannesburg

Africa/Lusaka

America/Anchorage

America/Argentina/Buenos\_Aires

America/Argentina/Cordoba

America/Argentina/Mendoza

I then use take to select the rows in that order, then slice off the last 10 rows:

In [316]: count\_subset = agg\_counts.take(indexer)[-10:]

In [317]: count\_subset

Out[317]:

a

Not Windows

tz

America/Sao\_Paulo

13

Europe/Madrid

16

Pacific/Honolulu

0

Asia/Tokyo

2

Europe/London

43

America/Denver

132

America/Los\_Angeles

130

America/Chicago

115

245

America/New\_York

339

Windows

20

19

36

35

31

59

252

285

276

912

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 stacked=True (see Figure 2-2) :

In [319]: count\_subset.plot(kind='barh', stacked=True)

The plot doesn’t make it easy to see the relative percentage of Windows users in the

smaller groups, but the rows can easily be normalized to sum to 1 then plotted again

(see Figure 2-3):

In [321]: normed\_subset = count\_subset.div(count\_subset.sum(1), axis=0)

In [322]: normed\_subset.plot(kind='barh', stacked=True)

1.usa.gov data from bit.ly | 25

www.it-ebooks.infoFigure 2-2. Top time zones by Windows and non-Windows users

Figure 2-3. Percentage Windows and non-Windows users in top-occurring time zones

All of the methods employed here will be examined in great detail throughout the rest

of the book.

MovieLens 1M Data Set

GroupLens Research (http://www.grouplens.org/node/73) provides a number of collec-

tions of movie ratings data collected from users of MovieLens in the late 1990s and

26 | Chapter 2: Introductory Examples

www.it-ebooks.infoearly 2000s. The data provide movie ratings, movie metadata (genres and year), and

demographic data about the users (age, zip code, gender, and occupation). Such data

is often of interest in the development of recommendation systems based on machine

learning algorithms. While I will not be exploring machine learning techniques in great

detail in this book, I will show you how to slice and dice data sets like these into the

exact form you need.

The MovieLens 1M data set contains 1 million ratings collected from 6000 users on

4000 movies. It’s spread across 3 tables: ratings, user information, and movie infor-

mation. After extracting the data from the zip file, each table can be loaded into a pandas

DataFrame object using pandas.read\_table:

import pandas as pd

unames = ['user\_id', 'gender', 'age', 'occupation', 'zip']

users = pd.read\_table('ml-1m/users.dat', sep='::', header=None,

names=unames)

rnames = ['user\_id', 'movie\_id', 'rating', 'timestamp']

ratings = pd.read\_table('ml-1m/ratings.dat', sep='::', header=None,

names=rnames)

mnames = ['movie\_id', 'title', 'genres']

movies = pd.read\_table('ml-1m/movies.dat', sep='::', header=None,

names=mnames)

You can verify that everything succeeded by looking at the first few rows of each Da-

taFrame with Python's slice syntax:

In [334]: users[:5]

Out[334]:

user\_id gender age

0

1

F

1

1

2

M

56

2

3

M

25

3

4

M

45

4

5

M

25

occupation

10

16

15

7

20

In [335]: ratings[:5]

Out[335]:

user\_id movie\_id rating

0

1

1193

5

1

1

661

3

2

1

914

3

3

1

3408

4

4

1

2355

5

In [336]: movies[:5]

Out[336]:

movie\_id

0

1

1

2

2

3

3

4

zip

48067

70072

55117

02460

55455

timestamp

978300760

978302109

978301968

978300275

978824291

title

Toy Story (1995)

Jumanji (1995)

Grumpier Old Men (1995)

Waiting to Exhale (1995)

genres

Animation|Children's|Comedy

Adventure|Children's|Fantasy

Comedy|Romance

Comedy|Drama

MovieLens 1M Data Set | 27

www.it-ebooks.info4

5

Father of the Bride Part II (1995)

Comedy

In [337]: ratings

Out[337]:

<class 'pandas.core.frame.DataFrame'>

Int64Index: 1000209 entries, 0 to 1000208

Data columns:

user\_id

1000209 non-null values

movie\_id

1000209 non-null values

rating

1000209 non-null values

timestamp

1000209 non-null values

dtypes: int64(4)

Note that ages and occupations are coded as integers indicating groups described in

the data set’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 then merging that result with the movies data. pandas infers which columns to

use as the merge (or join) keys based on overlapping names:

In [338]: data = pd.merge(pd.merge(ratings, users), movies)

In [339]: data

Out[339]:

<class 'pandas.core.frame.DataFrame'>

Int64Index: 1000209 entries, 0 to 1000208

Data columns:

user\_id

1000209 non-null values

movie\_id

1000209 non-null values

rating

1000209 non-null values

timestamp

1000209 non-null values

gender

1000209 non-null values

age

1000209 non-null values

occupation

1000209 non-null values

zip

1000209 non-null values

title

1000209 non-null values

genres

1000209 non-null values

dtypes: int64(6), object(4)

In [340]: data.ix[0]

Out[340]:

user\_id

1

movie\_id

1

rating

5

timestamp

978824268

gender

F

age

1

occupation

10

zip

48067

title

Toy Story (1995)

genres

Animation|Children's|Comedy

Name: 0

28 | Chapter 2: Introductory Examples

www.it-ebooks.infoIn this form, aggregating the ratings grouped by one or more user or movie attributes

is straightforward once you build some familiarity with pandas. To get mean movie

ratings for each film grouped by gender, we can use the pivot\_table method:

In [341]: mean\_ratings = data.pivot\_table('rating', rows='title',

.....:

cols='gender', aggfunc='mean')

In [342]: mean\_ratings[:5]

Out[342]:

gender

title

$1,000,000 Duck (1971)

'Night Mother (1986)

'Til There Was You (1997)

'burbs, The (1989)

...And Justice for All (1979)

FM

3.375000

3.388889

2.675676

2.793478

3.8285712.761905

3.352941

2.733333

2.962085

3.689024

This produced another DataFrame containing mean ratings with movie totals as row

labels and gender as column labels. First, I’m going to filter down to movies that re-

ceived at least 250 ratings (a completely arbitrary number); to do this, I group the data

by title and use size() to get a Series of group sizes for each title:

In [343]: ratings\_by\_title = data.groupby('title').size()

In [344]: ratings\_by\_title[:10]

Out[344]:

title

$1,000,000 Duck (1971)

'Night Mother (1986)

'Til There Was You (1997)

'burbs, The (1989)

...And Justice for All (1979)

1-900 (1994)

10 Things I Hate About You (1999)

101 Dalmatians (1961)

101 Dalmatians (1996)

12 Angry Men (1957)

37

70

52

303

199

2

700

565

364

616

In [345]: active\_titles = ratings\_by\_title.index[ratings\_by\_title >= 250]

In [346]: active\_titles

Out[346]:

Index(['burbs, The (1989), 10 Things I Hate About You (1999),

101 Dalmatians (1961), ..., Young Sherlock Holmes (1985),

Zero Effect (1998), eXistenZ (1999)], dtype=object)

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

mean\_ratings above:

In [347]: mean\_ratings = mean\_ratings.ix[active\_titles]

In [348]: mean\_ratings

Out[348]:

<class 'pandas.core.frame.DataFrame'>

Index: 1216 entries, 'burbs, The (1989) to eXistenZ (1999)

MovieLens 1M Data Set | 29

www.it-ebooks.infoData columns:

F

1216 non-null values

M

1216 non-null values

dtypes: float64(2)

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

order:

In [350]: top\_female\_ratings = mean\_ratings.sort\_index(by='F', ascending=False)

In [351]: top\_female\_ratings[:10]

Out[351]:

gender

Close Shave, A (1995)

Wrong Trousers, The (1993)

Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)

Wallace & Gromit: The Best of Aardman Animation (1996)

Schindler's List (1993)

Shawshank Redemption, The (1994)

Grand Day Out, A (1992)

To Kill a Mockingbird (1962)

Creature Comforts (1990)

Usual Suspects, The (1995)

F

4.644444

4.588235

4.572650

4.563107

4.562602

4.539075

4.537879

4.536667

4.513889

4.513317

M

4.473795

4.478261

4.464589

4.385075

4.491415

4.560625

4.293255

4.372611

4.272277

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 [352]: mean\_ratings['diff'] = mean\_ratings['M'] - mean\_ratings['F']

Sorting by 'diff' gives us the movies with the greatest rating difference and which were

preferred by women:

In [353]: sorted\_by\_diff = mean\_ratings.sort\_index(by='diff')

In [354]: sorted\_by\_diff[:15]

Out[354]:

gender

Dirty Dancing (1987)

Jumpin' Jack Flash (1986)

Grease (1978)

Little Women (1994)

Steel Magnolias (1989)

Anastasia (1997)

Rocky Horror Picture Show, The (1975)

Color Purple, The (1985)

Age of Innocence, The (1993)

Free Willy (1993)

French Kiss (1995)

Little Shop of Horrors, The (1960)

Guys and Dolls (1955)

Mary Poppins (1964)

Patch Adams (1998)

F

3.790378

3.254717

3.975265

3.870588

3.901734

3.800000

3.673016

4.158192

3.827068

2.921348

3.535714

3.650000

4.051724

4.197740

3.473282

M

diff

2.959596 -0.830782

2.578358 -0.676359

3.367041 -0.608224

3.321739 -0.548849

3.365957 -0.535777

3.281609 -0.518391

3.160131 -0.512885

3.659341 -0.498851

3.339506 -0.487561

2.438776 -0.482573

3.056962 -0.478752

3.179688 -0.470312

3.583333 -0.468391

3.730594 -0.467147

3.008746 -0.464536

30 | Chapter 2: Introductory Examples

www.it-ebooks.infoReversing the order of the rows and again slicing off the top 15 rows, we get the movies

preferred by men that women didn’t rate as highly:

# Reverse order of rows, take first 15 rows

In [355]: sorted\_by\_diff[::-1][:15]

Out[355]:

gender

F

Good, The Bad and The Ugly, The (1966) 3.494949

Kentucky Fried Movie, The (1977)

2.878788

Dumb & Dumber (1994)

2.697987

Longest Day, The (1962)

3.411765

Cable Guy, The (1996)

2.250000

Evil Dead II (Dead By Dawn) (1987)

3.297297

Hidden, The (1987)

3.137931

Rocky III (1982)

2.361702

Caddyshack (1980)

3.396135

For a Few Dollars More (1965)

3.409091

Porky's (1981)

2.296875

Animal House (1978)

3.628906

Exorcist, The (1973)

3.537634

Fright Night (1985)

2.973684

Barb Wire (1996)

1.585366

M

4.221300

3.555147

3.336595

4.031447

2.863787

3.909283

3.745098

2.943503

3.969737

3.953795

2.836364

4.167192

4.067239

3.500000

2.100386

diff

0.726351

0.676359

0.638608

0.619682

0.613787

0.611985

0.607167

0.581801

0.573602

0.544704

0.539489

0.538286

0.529605

0.526316

0.515020

Suppose instead you wanted the movies that elicited the most disagreement among

viewers, independent of gender. Disagreement can be measured by the variance or

standard deviation of the ratings:

# Standard deviation of rating grouped by title

In [356]: rating\_std\_by\_title = data.groupby('title')['rating'].std()

# Filter down to active\_titles

In [357]: rating\_std\_by\_title = rating\_std\_by\_title.ix[active\_titles]

# Order Series by value in descending order

In [358]: rating\_std\_by\_title.order(ascending=False)[:10]

Out[358]:

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

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. I will revisit this data later in the book to

illustrate such a transformation.

MovieLens 1M Data Set | 31

www.it-ebooks.infoUS 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

0

Mary

F

7065

1

Anna

F

2604

2

Emma

F

2003

3 Elizabeth

F

1939

4

Minnie

F

1746

5 Margaret

F

1578

6

Ida

F

1472

7

Alice

F

1414

8

Bertha

F

1320

9

Sarah

F

1288

year

1880

1880

1880

1880

1880

1880

1880

1880

1880

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 straight-

forward, 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):

32 | Chapter 2: Introductory Examples

www.it-ebooks.infoIn [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 simplic-

ity’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)

US Baby Names 1880-2010 | 33

www.it-ebooks.info# 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)

34 | Chapter 2: Introductory Examples

www.it-ebooks.infoFigure 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]

US Baby Names 1880-2010 | 35

www.it-ebooks.infogrouped = 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")

36 | Chapter 2: Introductory Examples

www.it-ebooks.infoSee 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 met-

ric 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:

US Baby Names 1880-2010 | 37

www.it-ebooks.infoname

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

38 | Chapter 2: Introductory Examples

www.it-ebooks.infoSince arrays are zero-indexed, adding 1 to this result gives you a result of 117. By con-

trast, 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 com-

bination; 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

US Baby Names 1880-2010 | 39

www.it-ebooks.infoAs 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

year

1910

1960

2010

last\_letter

a

108376 691247 670605

b

NaN

694

450

c

5

49

946

d

6750

3729

2607

e

133569 435013 313833

M

191019602010

977

411

482

22111

286555204

3912

15476

262112

17882328438

38859

23125

44398

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

40 | Chapter 2: Introductory Examples

www.it-ebooks.infofig, 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

year

1880 0.083055 0.153213

1881 0.083247 0.153214

1882 0.085340 0.149560

1883 0.084066 0.151646

1884 0.086120 0.149915

y

0.075760

0.077451

0.077537

0.079144

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()

US Baby Names 1880-2010 | 41

www.it-ebooks.infoFigure 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:

42 | Chapter 2: Introductory Examples

www.it-ebooks.infoIn [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.

Conclusions and The Path Ahead | 43

www.it-ebooks.infowww.it-ebooks.infoCHAPTER 3

IPython: An Interactive