Handling Data

Before you start working on the visual part of any visualization, you actually need data. The data is what makes a visualization interesting. If you don't have interesting data, you just end up with a forgettable graph or a pretty but useless picture. Where can you find good data? How can you access it?

When you have your data, it needs to be formatted so that you can load it into your software. Maybe you got the data as a comma-delimited text file or an Excel spreadsheet, and you need to convert it to something such as XML, or vice versa. Maybe the data you want is accessible point-by-point from a web application, but you want an entire spreadsheet.

Learn to access and process data, and your visualization skills will follow.

Gather Data

Data is the core of any visualization. Fortunately, there are a lot of places to find it. You can get it from experts in the area you're interested in, a variety of online applications, or you can gather it yourself.

Provided by Others

This route is common, especially if you're a freelance designer or work in a graphics department of a larger organization. This is a good thing a lot of the time because someone else did all the data gathering work for you, but you still need to be careful. A lot of mistakes can happen along the way before that nicely formatted spreadsheet gets into your hands.

When you share data with spreadsheets, the most common mistake to look for is typos. Are there any missing zeros? Did your client or data supplier mean six instead of five? At some point, data was read from one source and then input into Excel or a different spreadsheet program (unless a delimited text file was imported), so it's easy for an innocent typo to make its way through the vetting stage and into your hands.

You also need to check for context. You don't need to become an expert in the data's subject matter, but you should know where the original data came from, how it was collected, and what it's about. This can help you build a better graphic and tell a more complete story when you design your graphic. For example, say you're looking at poll results. When did the poll take place? Who conducted the poll? Who answered? Obviously, poll results from 1970 are going to take on a different meaning from poll results from the present day.

Finding Sources

If the data isn't directly sent to you, it's your job to go out and find it. The bad news is that, well, that's more work on your shoulders, but the good news is that's it's getting easier and easier to find data that's relevant and machine-readable (as in, you can easily load it into software). Here's where you can start your search.

SEARCH ENGINES

How do you find anything online nowadays? You Google it. This is a no-brainer, but you'd be surprised how many times people email me asking if I know where to find a particular dataset and a quick search provided relevant results. Personally, I turn to Google and occasionally look to Wolfram|Alpha, the computational search engine.

▶ See Wolfram|Alpha at `http://wolframalpha.com`. The search engine can be especially useful if you're looking for some basic statistics on a topic.

DIRECT FROM THE SOURCE

If a direct query for "data" doesn't provide anything of use, try searching for academics who specialize in the area you're interested in finding data for. Sometimes they post data on their personal sites. If not, scan their papers and studies for possible leads. You can also try emailing them, but make sure they've actually done related studies. Otherwise, you'll just be wasting everyone's time.

You can also spot sources in graphics published by news outlets such as *The New York Times*. Usually data sources are included in small print somewhere on the graphic. If it's not in the graphic, it should be mentioned in the related article. This is particularly useful when you see a graphic in the paper or online that uses data you're interested in exploring. Search for a site for the source, and the data might be available.

This won't always work because finding contacts seems to be a little easier when you email saying that you're a reporter for the so-and-so paper, but it's worth a shot.

UNIVERSITIES

As a graduate student, I frequently make use of the academic resources available to me, namely the library. Many libraries have amped up their technology resources and actually have some expansive data archives. A number of statistics departments also keep a list of data files, many of which are publicly accessible. Albeit, many of the datasets made available by these departments are intended for use with course labs and homework. I suggest visiting the following resources:

- Data and Story Library (DASL) (`http://lib.stat.cmu.edu/DASL/`)—An online library of data files and stories that illustrate the use of basic statistics methods, from Carnegie Mellon

- Berkeley Data Lab (http://sunsite3.berkeley.edu/wikis/datalab/)—Part of the University of California, Berkeley library system
- UCLA Statistics Data Sets (www.stat.ucla.edu/data/)—Some of the data that the UCLA Department of Statistics uses in their labs and assignments

GENERAL DATA APPLICATIONS

A growing number of general data-supplying applications are available. Some applications provide large data files that you can download for free or for a fee. Others are built with developers in mind with data accessible via Application Programming Interface (API). This lets you use data from a service, such as Twitter, and integrate the data with your own application. Following are a few suggested resources:

- Freebase (www.freebase.com)—A community effort that mostly provides data on people, places, and things. It's like Wikipedia for data but more structured. Download data dumps or use it as a backend for your application.
- Infochimps (http://infochimps.org)—A data marketplace with free and for-sale datasets. You can also access some datasets via their API.
- Numbrary (http://numbrary.com)—Serves as a catalog for (mostly government) data on the web.
- AggData (http://aggdata.com)—Another repository of for-sale datasets, mostly focused on comprehensive lists of retail locations.
- Amazon Public Data Sets (http://aws.amazon.com/publicdatasets)—There's not a lot of growth here, but it does host some large scientific datasets.
- Wikipedia (http://wikipedia.org)—A lot of smaller datasets in the form of HTML tables on this community-run encyclopedia.

TOPICAL DATA

Outside more general data suppliers, there's no shortage of subject-specific sites offering loads of free data.

Following is a small taste of what's available for the topic of your choice.

Geography

Do you have mapping software, but no geographic data? You're in luck. Plenty of shapefiles and other geographic file types are at your disposal.

- TIGER (`www.census.gov/geo/www/tiger/`)—From the Census Bureau, probably the most extensive detailed data about roads, railroads, rivers, and ZIP codes you can find
- OpenStreetMap (`www.openstreetmap.org/`)—One of the best examples of data and community effort
- Geocommons (`www.geocommons.com/`)—Both data and a mapmaker
- Flickr Shapefiles (`www.flickr.com/services/api/`)—Geographic boundaries as defined by Flickr users

Sports

People love sports statistics, and you can find decades' worth of sports data. You can find it on *Sports Illustrated* or team organizations' sites, but you can also find more on sites dedicated to the data specifically.

- Basketball Reference (`www.basketball-reference.com/`)—Provides data as specific as play-by-play for NBA games.
- Baseball DataBank (`http://baseball-databank.org/`)—Super basic site where you can download full datasets.
- databaseFootball (`www.databasefootball.com/`)—Browse data for NFL games by team, player, and season.

World

Several noteworthy international organizations keep data about the world, mainly health and development indicators. It does take some sifting though, because a lot of the datasets are quite sparse. It's not easy to get standardized data across countries with varied methods.

- Global Health Facts (`www.globalhealthfacts.org/`)—Health-related data about countries in the world.

- UNdata (http://data.un.org/)—Aggregator of world data from a variety of sources
- World Health Organization (www.who.int/research/en/)—Again, a variety of health-related datasets such as mortality and life expectancy
- OECD Statistics (http://stats.oecd.org/)—Major source for economic indicators
- World Bank (http://data.worldbank.org/)—Data for hundreds of indicators and developer-friendly

Government and Politics

There has been a fresh emphasis on data and transparency in recent years, so many government organizations supply data, and groups such as the Sunlight Foundation encourage developers and designers to make use of it. Government organizations have been doing this for awhile, but with the launch of data.gov, much of the data is available in one place. You can also find plenty of nongovernmental sites that aim to make politicians more accountable.

- Census Bureau (www.census.gov/)—Find extensive demographics here.
- Data.gov (http://data.gov/)—Catalog for data supplied by government organizations. Still relatively new, but has a lot of sources.
- Data.gov.uk (http://data.gov.uk/)—The Data.gov equivalent for the United Kingdom.
- DataSF (http://datasf.org/)—Data specific to San Francisco.
- NYC DataMine (http://nyc.gov/data/)—Just like the above, but for New York.
- Follow the Money (www.followthemoney.org/)—Big set of tools and datasets to investigate money in state politics.
- OpenSecrets (www.opensecrets.org/)—Also provides details on government spending and lobbying.

Data Scraping

Often you can find the exact data that you need, except there's one problem. It's not all in one place or in one file. Instead it's in a bunch of HTML pages or on multiple websites. What should you do?

The straightforward, but most time-consuming method would be to visit every page and manually enter your data point of interest in a spreadsheet. If you have only a few pages, sure, no problem.

What if you have a thousand pages? That would take too long—even a hundred pages would be tedious. It would be much easier if you could automate the process, which is what *data scraping* is for. You write some code to visit a bunch of pages automatically, grab some content from that page, and store it in a database or a text file.

NOTE

Although coding is the most flexible way to scrape the data you need, you can also try tools such as Needlebase and Able2Extract PDF converter. Use is straightforward, and they can save you time.

EXAMPLE: SCRAPE A WEBSITE

The best way to learn how to scrape data is to jump right into an example. Say you wanted to download temperature data for the past year, but you can't find a source that provides all the numbers for the right time frame or the correct city. Go to almost any weather website, and at the most, you'll usually see only temperatures for an extended 10-day forecast. That's not even close to what you want. You want actual temperatures from the past, not predictions about future weather.

▶ Visit Weather Underground at `http://wunderground.com`.

Fortunately, the Weather Underground site does provide historic temperatures; however, you can see only one day at a time.

To make things more concrete, look up temperature in Buffalo. Go to the Weather Underground site and search for **BUF** in the search box. This should take you to the weather page for Buffalo Niagara International, which is the airport in Buffalo (see Figure 2-1).

FIGURE 2-1 Temperature in Buffalo, New York, according to Weather Underground

The top of the page provides the current temperature, a 5-day forecast, and other details about the current day. Scroll down toward the middle of the page to the History & Almanac panel, as shown in Figure 2-2. Notice the drop-down menu where you can select a specific date.

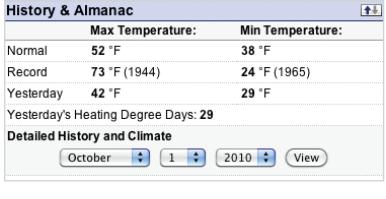
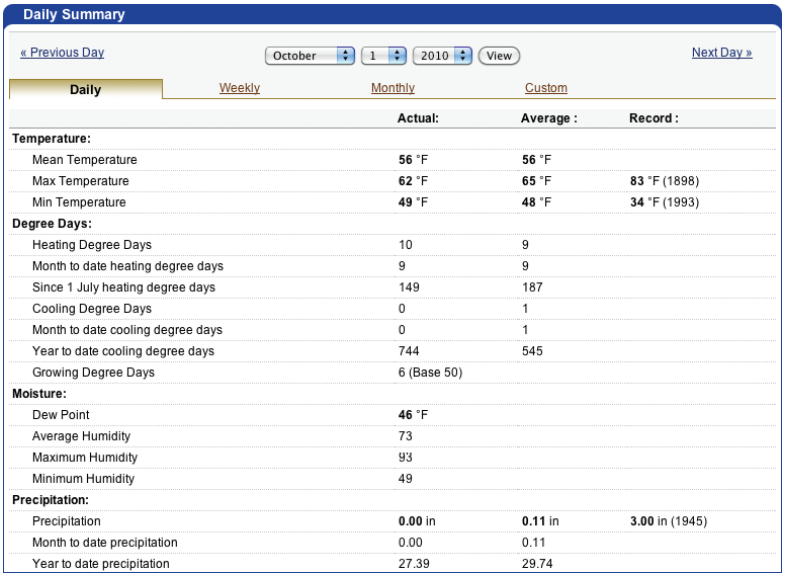


FIGURE 2-2 Drop-down menu to see historical data for a selected date

Adjust the menu to show October 1, 2010, and click the View button. This takes you to a different view that shows you details for your selected date (see Figure 2-3).



Daily Summary

« Previous Day | October 1 2010 View | Next Day »

Daily | Weekly | Monthly | Custom

	Actual:	Average :	Record :
Temperature:			
Mean Temperature	**56** °F	**56** °F	
Max Temperature	**62** °F	**65** °F	**83** °F (1898)
Min Temperature	**49** °F	**48** °F	**34** °F (1993)
Degree Days:			
Heating Degree Days	10	9	
Month to date heating degree days	9	9	
Since 1 July heating degree days	149	187	
Cooling Degree Days	0	1	
Month to date cooling degree days	0	1	
Year to date cooling degree days	744	545	
Growing Degree Days	6 (Base 50)		
Moisture:			
Dew Point	**46** °F		
Average Humidity	73		
Maximum Humidity	93		
Minimum Humidity	49		
Precipitation:			
Precipitation	**0.00** in	**0.11** in	**3.00** in (1945)
Month to date precipitation	0.00	0.11	
Year to date precipitation	27.39	29.74	

FIGURE 2-3 Temperature data for a single day

There's temperature, degree days, moisture, precipitation, and plenty of other data points, but for now, all you're interested in is maximum temperature per day, which you can find in the second column, second row down. On October 1, 2010, the maximum temperature in Buffalo was 62 degrees Fahrenheit.

Getting that single value was easy enough. Now how can you get that maximum temperature value every day, during the year 2009? The easy-and-straightforward way would be to keep changing the date in the drop-down. Do that 365 times and you're done.

Wouldn't that be fun? No. You can speed up the process with a little bit of code and some know-how, and for that, turn to the Python programming language and Leonard Richardson's Python library called Beautiful Soup.

You're about to get your first taste of code in the next few paragraphs. If you have programming experience, you can go through the following

relatively quickly. Don't worry if you don't have any programming experience though—I'll take you through it step-by-step. A lot of people like to keep everything within a safe click interface, but trust me. Pick up just a little bit of programming skills, and you can open up a whole bag of possibilities for what you can do with data. Ready? Here you go.

First, you need to make sure your computer has all the right software installed. If you work on Mac OS X, you should have Python installed already. Open the Terminal application and type **python** to start (see Figure 2-4).

▶ Visit `http://python.org` to download and install Python. Don't worry; it's not too hard.

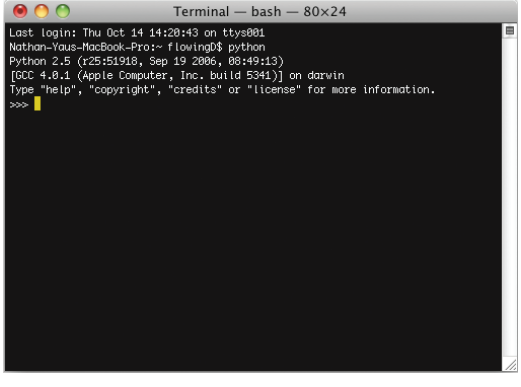


FIGURE 2-4 Starting Python in OS X

If you're on a Windows machine, you can visit the Python site and follow the directions on how to download and install.

▶ Visit `www.crummy.com/software/BeautifulSoup/` to download Beautiful Soup. Download the version that matches the version of Python that you use.

Next, you need to download Beautiful Soup, which can help you read web pages quickly and easily. Save the Beautiful Soup Python (`.py`) file in the directory that you plan to save your code in. If you know your way around Python, you can also put Beautiful Soup in your library path, but it'll work the same either way.

After you install Python and download Beautiful Soup, start a file in your favorite text or code editor, and save it as `get-weather-data.py`. Now you can code.

The first thing you need to do is load the page that shows historical weather information. The URL for historical weather in Buffalo on October 1, 2010, follows:

```
www.wunderground.com/history/airport/KBUF/2010/10/1/DailyHistory
.html?req_city=NA&req_state=NA&req_statename=NA
```

If you remove everything after .html in the preceding URL, the same page still loads, so get rid of those. You don't care about those right now.

```
www.wunderground.com/history/airport/KBUF/2010/10/1/DailyHistory.html
```

The date is indicated in the URL with `/2010/10/1`. Using the drop-down menu, change the date to January 1, 2009, because you're going to scrape temperature for all of 2009. The URL is now this:

```
www.wunderground.com/history/airport/KBUF/2009/1/1/DailyHistory.html
```

Everything is the same as the URL for October 1, except the portion that indicates the date. It's `/2009/1/1` now. Interesting. Without using the drop-down menu, how can you load the page for January 2, 2009? Simply change the date parameter so that the URL looks like this:

```
www.wunderground.com/history/airport/KBUF/2009/1/2/DailyHistory.html
```

Load the preceding URL in your browser and you get the historical summary for January 2, 2009. So all you have to do to get the weather for a specific date is to modify the Weather Underground URL. Keep this in mind for later.

Now load a single page with Python, using the urllib2 library by importing it with the following line of code:

```
import urllib2
```

To load the January 1 page with Python, use the `urlopen` function.

```
page = urllib2.urlopen("www.wunderground.com/history/airport/
KBUF/2009/1/1/DailyHistory.html")
```

This loads all the HTML that the URL points to in the page variable. The next step is to extract the maximum temperature value you're interested

in from that HTML, and for that, Beautiful Soup makes your task much easier. After `urllib2`, import Beautiful Soup like so:

```
from BeautifulSoup import BeautifulSoup
```

At the end of your file, use Beautiful Soup to read (that is, parse) the page.

```
soup = BeautifulSoup(page)
```

NOTE

Beautiful Soup provides good documentation and straightforward examples, so if any of this is confusing, I strongly encourage you to check those out on the same Beautiful Soup site you used to download the library.

Without getting into nitty-gritty details, this line of code reads the HTML, which is essentially one long string, and then stores elements of the page, such as the header or images, in a way that is easier to work with.

For example, if you want to find all the images in the page, you can use this:

```
images = soup.findAll('img')
```

This gives you a list of all the images on the Weather Underground page displayed with the `<img />` HTML tag. Want the first image on the page? Do this:

```
first_image = images[0]
```

Want the second image? Change the zero to a one. If you want the `src` value in the first `<img />` tag, you would use this:

```
src = first_image['src']
```

Okay, you don't want images. You just want that one value: maximum temperature on January 1, 2009, in Buffalo, New York. It was 26 degrees Fahrenheit. It's a little trickier finding that value in your soup than it was finding images, but you still use the same method. You just need to figure out what to put in `findAll()`, so look at the HTML source.

You can easily do this in all the major browsers. In Firefox, go to the View menu, and select Page Source. A window with the HTML for your current page appears, as shown in Figure 2-5.

Scroll down to where it shows Mean Temperature, or just search for it, which is faster. Spot the `26`. That's what you want to extract.

The row is enclosed by a `<span>` tag with a `nobr` class. That's your key. You can find all the elements in the page with the `nobr` class.

```
nobrs = soup.findAll(attrs={"class":"nobr"})
```

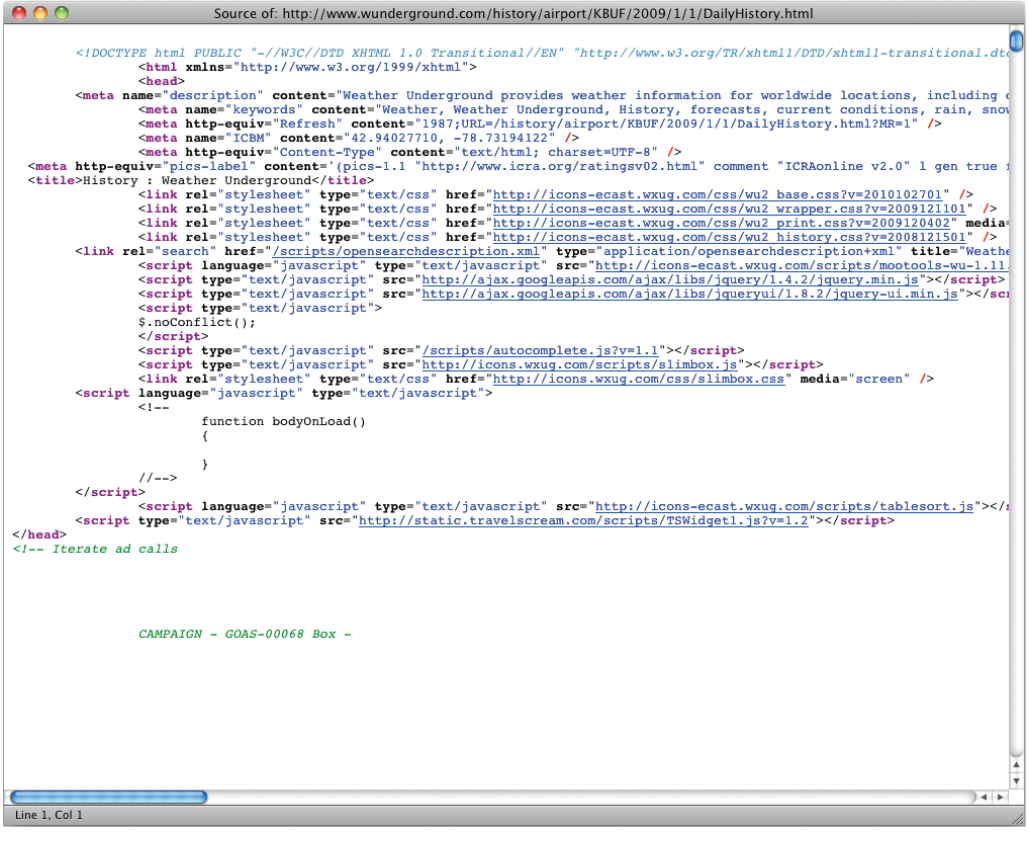


FIGURE 2-5 HTML source for a page on Weather Underground

As before, this gives you a list of all the occurrences of `nobr`. The one that you're interested in is the sixth occurrence, which you can find with the following:

```
print nobrs[5]
```

This gives you the whole element, but you just want the `26`. Inside the `<span>` tag with the `nobr` class is another `<span>` tag and *then* the `26`. So here's what you need to use:

```
dayTemp = nobrs[5].span.string
print dayTemp
```

Ta Da! You scraped your first value from an HTML web page. Next step: scrape all the pages for 2009. For that, return to the original URL.

```
www.wunderground.com/history/airport/KBUF/2009/1/1/DailyHistory.html
```

Remember that you changed the URL manually to get the weather data for the date you want. The preceding code is for January 1, 2009. If you want the page for January 2, 2009, simply change the date portion of the URL to match that. To get the data for every day of 2009, load every month (1 through 12) and then load every day of each month. Here's the script in full with comments. Save it to your `get-weather-data.py` file.

```
import urllib2
from BeautifulSoup import BeautifulSoup

# Create/open a file called wunder.txt (which will be a comma-delimited
file)
f = open('wunder-data.txt', 'w')

# Iterate through months and day
for m in range(1, 13):
    for d in range(1, 32):

      # Check if already gone through month
      if (m == 2 and d > 28):
        break
      elif (m in [4, 6, 9, 11] and d > 30):
        break

      # Open wunderground.com url
      timestamp = '2009' + str(m) + str(d)
      print "Getting data for " + timestamp
      url = "http://www.wunderground.com/history/airport/KBUF/2009/" +
str(m) + "/" + str(d) + "/DailyHistory.html"
      page = urllib2.urlopen(url)

      # Get temperature from page
      soup = BeautifulSoup(page)
      # dayTemp = soup.body.nobr.b.string
      dayTemp = soup.findAll(attrs={"class":"nobr"})[5].span.string

      # Format month for timestamp
      if len(str(m)) < 2:
        mStamp = '0' + str(m)
      else:
        mStamp = str(m)

      # Format day for timestamp
```

```
        if len(str(d)) < 2:
          dStamp = '0' + str(d)
        else:
          dStamp = str(d)

        # Build timestamp
        timestamp = '2009' + mStamp + dStamp

        # Write timestamp and temperature to file
        f.write(timestamp + ',' + dayTemp + '\n')

# Done getting data! Close file.
f.close()
```

You should recognize the first two lines of code to import the necessary libraries, urllib2 and BeautifulSoup.

```
import urllib2
from BeautifulSoup import BeautifulSoup
```

Next, start a text file called `wunder-data-txt` with write permissions, using the `open()` method. All the data that you scrape will be stored in this text file, in the same directory that you saved this script in.

```
# Create/open a file called wunder.txt (which will be a comma-delimited
file)
f = open('wunder-data.txt', 'w')
```

With the next line of code, use a `for` loop, which tells the computer to visit each month. The month number is stored in the `m` variable. The loop that follows then tells the computer to visit each day of each month. The day number is stored in the `d` variable.

```
# Iterate through months and day
for m in range(1, 13):
    for d in range(1, 32):
```

Notice that you used `range (1, 32)` to iterate through the days. This means you can iterate through the numbers 1 to 31. However, not every month of the year has 31 days. February has 28 days; April, June, September, and November have 30 days. There's no temperature value for April 31 because it doesn't exist. So check what month it is and act accordingly. If the current month is February and the day is greater than 28, break and

▶ See Python documentation for more on how loops and iteration work: `http://docs.python.org/reference/compound_stmts.html`

move on to the next month. If you want to scrape multiple years, you need to use an additional `if` statement to handle leap years.

Similarly, if it's not February, but instead April, June, September, or November, move on to the next month if the current day is greater than 30.

```
# Check if already gone through month
if (m == 2 and d > 28):
  break
elif (m in [4, 6, 9, 11] and d > 30):
  break
```

Again, the next few lines of code should look familiar. You used them to scrape a single page from Weather Underground. The difference is in the month and day variable in the URL. Change that for each day instead of leaving it static; the rest is the same. Load the page with the urllib2 library, parse the contents with Beautiful Soup, and then extract the maximum temperature, but look for the sixth appearance of the `nobr` class.

```
# Open wunderground.com url
url = "http://www.wunderground.com/history/airport/KBUF/2009/" +
str(m) + "/" + str(d) + "/DailyHistory.html"
page = urllib2.urlopen(url)

# Get temperature from page
soup = BeautifulSoup(page)
# dayTemp = soup.body.nobr.b.string
dayTemp = soup.findAll(attrs={"class":"nobr"})[5].span.string
```

The next to last chunk of code puts together a timestamp based on the year, month, and day. Timestamps are put into this format: yyyymmdd. You can construct any format here, but keep it simple for now.

```
# Format day for timestamp
if len(str(d)) < 2:
  dStamp = '0' + str(d)
else:
  dStamp = str(d)

# Build timestamp
timestamp = '2009' + mStamp + dStamp
```

Finally, the temperature and timestamp are written to `'wunder-data.txt'` using the `write()` method.

```
    # Write timestamp and temperature to file
    f.write(timestamp + ',' + dayTemp + '\n')
```

Then use `close()`when you finish with all the months and days.

```
# Done getting data! Close file.
f.close()
```

The only thing left to do is run the code, which you do in your terminal with the following:

```
$ python get-weather-data.py
```

It takes a little while to run, so be patient. In the process of running, your computer is essentially loading 365 pages, one for each day of 2009. You should have a file named `wunder-data.txt` in your working directory when the script is done running. Open it up, and there's your data, as a comma-separated file. The first column is for the timestamps, and the second column is temperatures. It should look similar to Figure 2-6.

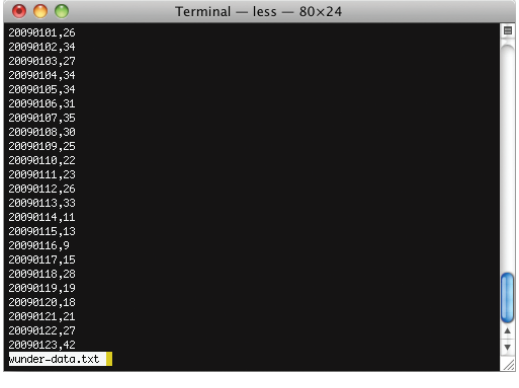


FIGURE 2-6 One year's worth of scraped temperature data

GENERALIZING THE EXAMPLE

Although you just scraped weather data from Weather Underground, you can generalize the process for use with other data sources. Data scraping typically involves three steps:

1. Identify the patterns.
2. Iterate.
3. Store the data.

In this example, you had to find two patterns. The first was in the URL, and the second was in the loaded web page to get the actual temperature value. To load the page for a different day in 2009, you changed the month and day portions of the URL. The temperature value was enclosed in the sixth occurrence of the `nobr` class in the HTML page. If there is no obvious pattern to the URL, try to figure out how you can get the URLs of all the pages you want to scrape. Maybe the site has a site map, or maybe you can go through the index via a search engine. In the end, you need to know all the URLs of the pages of data.

After you find the patterns, you iterate. That is, you visit all the pages programmatically, load them, and parse them. Here you did it with Beautiful Soup, which makes parsing XML and HTML easy in Python. There's probably a similar library if you choose a different programming language.

Lastly, you need to store it somewhere. The easiest solution is to store the data as a plain text file with comma-delimited values, but if you have a database set up, you can also store the values in there.

Things can get trickier as you run into web pages that use JavaScript to load all their data into view, but the process is still the same.

Formatting Data

Different visualization tools use different data formats, and the structure you use varies by the story you want to tell. So the more flexible you are with the structure of your data, the more possibilities you can gain. Make use of data formatting applications, and couple that with a little bit of programming know-how, and you can get your data in any format you want to fit your specific needs.

The easy way of course is to find a programmer who can format and parse all of your data, but you'll always be waiting on someone. This is especially evident during the early stages of any project where iteration and data exploration are key in designing a useful visualization. Honestly, if I were in a hiring position, I'd likely just get the person who knows how to work with data, over the one who needs help at the beginning of every project.

WHAT I LEARNED ABOUT FORMATTING

When I first learned statistics in high school, the data was always provided in a nice, rectangular format. All I had to do was plug some numbers into an Excel spreadsheet or my awesome graphing calculator (which was the best way to look like you were working in class, but actually playing Tetris). That's how it was all the way through my undergraduate education. Because I was learning about techniques and theorems for analyses, my teachers didn't spend any time on working with raw, preprocessed data. The data always seemed to be in just the right format.

This is perfectly understandable, given time constraints and such, but in graduate school, I realized that data in the real world never seems to be in the format that you need. There are missing values, inconsistent labels, typos, and values without any context. Often the data is spread across several tables, but you need everything in one, joined across a value, like a name or a unique id number.

This was also true when I started to work with visualization. It became increasingly important because I wanted to do more with the data I had. Nowadays, it's not out of the ordinary that I spend just as much time getting data in the format that I need as I do putting the visual part of a data graphic together. Sometimes I spend more time getting all my data in place. This might seem strange at first, but you'll find that the design of your data graphics comes much easier when you have your data neatly organized, just like it was back in that introductory statistics course in high school.

Various data formats, the tools available to deal with these formats, and finally, some programming, using the same logic you used to scrape data in the previous example are described next.

Data Formats

Most people are used to working with data in Excel. This is fine if you're going to do everything from analyses to visualization in the program, but if you want to step beyond that, you need to familiarize yourself with other data formats. The point of these formats is to make your data

machine-readable, or in other words, to structure your data in a way that a computer can understand. Which data format you use can change by visualization tool and purpose, but the three following formats can cover most of your bases: delimited text, JavaScript Object Notation, and Extensible Markup Language.

DELIMITED TEXT

Most people are familiar with delimited text. You did after all just make a comma-delimited text file in your data scraping example. If you think of a dataset in the context of rows and columns, a delimited text file splits columns by a delimiter. The delimiter is a comma in a comma-delimited file. The delimiter might also be a tab. It can be spaces, semicolons, colons, slashes, or whatever you want; although a comma and tab are the most common.

Delimited text is widely used and can be read into most spreadsheet programs such as Excel or Google Documents. You can also export spreadsheets as delimited text. If multiple sheets are in your workbook, you usually have multiple delimited files, unless you specify otherwise.

This format is also good for sharing data with others because it doesn't depend on any particular program.

JAVASCRIPT OBJECT NOTATION (JSON)

This is a common format offered by web APIs. It's designed to be both machine- and human-readable; although, if you have a lot of it in front of you, it'll probably make you cross-eyed if you stare at it too long. It's based on JavaScript notation, but it's not dependent on the language. There are a lot of specifications for JSON, but you can get by for the most part with just the basics.

JSON works with keywords and values, and treats items like objects. If you were to convert JSON data to comma-separated values (CSV), each object might be a row.

As you can see later in this book, a number of applications, languages, and libraries accept JSON as input. If you plan to design data graphics for the web, you're likely to run into this format.

► Visit `http://json.org` for the full specification of JSON. You don't need to know every detail of the format, but it can be handy at times when you don't understand a JSON data source.

EXTENSIBLE MARKUP LANGUAGE (XML)

XML is another popular format on the web, often used to transfer data via APIs. There are lots of different types and specifications for XML, but at the most basic level, it is a text document with values enclosed by tags. For example, the Really Simple Syndication (RSS) feed that people use to subscribe to blogs, such as FlowingData, is actually an XML file, as shown in Figure 2-7.

The RSS lists recently published items enclosed in the <item></item> tag, and each item has a title, description, author, and publish date, along with some other attributes.

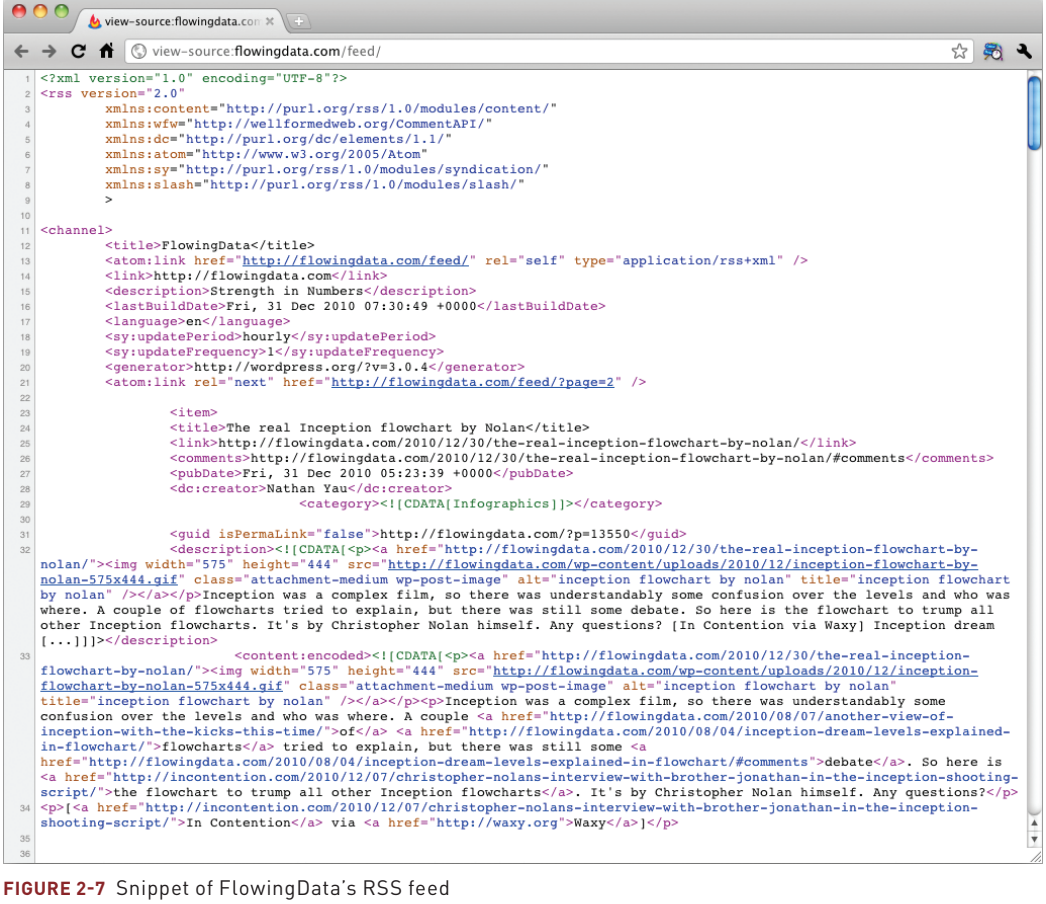


FIGURE 2-7 Snippet of FlowingData's RSS feed

XML is relatively easy to parse with libraries such as Beautiful Soup in Python. You can get a better feel for XML, along with CSV and JSON, in the sections that follow.

Formatting Tools

Just a couple of years ago, quick scripts were always written to handle and format data. After you've written a few scripts, you start to notice patterns in the logic, so it's not super hard to write new scripts for specific datasets, but it does take time. Luckily, with growing volumes of data, some tools have been developed to handle the boiler plate routines.

GOOGLE REFINE

Google Refine is the evolution of Freebase Gridworks. Gridworks was first developed as an in-house tool for an open data platform, Freebase; however, Freebase was acquired by Google, therefore the new name. Google Refine is essentially Gridworks 2.0 with an easier-to-use interface (Figure 2-8) with more features.

It runs on your desktop (but still through your browser), which is great, because you don't need to worry about uploading private data to Google's servers. All the processing happens on your computer. Refine is also open source, so if you feel ambitious, you can cater the tool to your own needs with extensions.

When you open Refine, you see a familiar spreadsheet interface with your rows and columns. You can easily sort by field and search for values. You can also find inconsistencies in your data and consolidate in a relatively easy way.

For example, say for some reason you have an inventory list for your kitchen. You can load the data in Refine and quickly find inconsistencies such as typos or differing classifications. Maybe a fork was misspelled as "frk," or you want to reclassify all the forks, spoons, and knives as utensils. You can easily find these things with Refine and make changes. If you don't like the changes you made or make a mistake, you can revert to the old dataset with a simple undo.

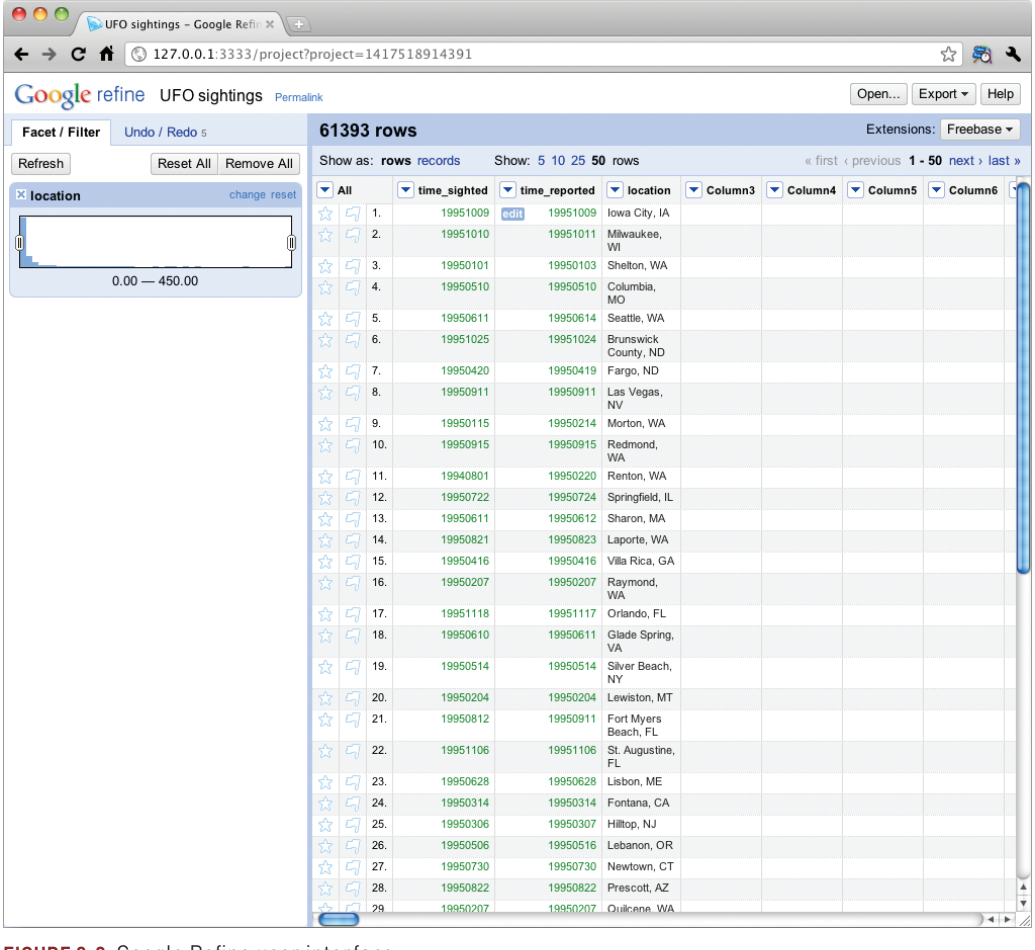


FIGURE 2-8 Google Refine user interface

Getting into the more advanced stuff, you can also incorporate data sources like your own with a dataset from Freebase to create a richer dataset.

If anything, Google Refine is a good tool to keep in your back pocket. It's powerful, and it's a free download, so I highly recommend you at least fiddle around with the tool.

▶ Download the open-source Google Refine and view tutorials on how to make the most out of the tool at `http://code.google.com/p/google-refine/`.

MR. DATA CONVERTER

Often, you might get all your data in Excel but then need to convert it to another format to fit your needs. This is almost always the case when you

create graphics for the web. You can already export Excel spreadsheets as CSV, but what if you need something other than that? Mr. Data Converter can help you.

Mr. Data Converter is a simple and free tool created by Shan Carter, who is a graphics editor for *The New York Times*. Carter spends most of his work time creating interactive graphics for the online version of the paper. He has to convert data often to fit the software that he uses, so it's not surprising he made a tool that streamlines the process.

It's easy to use, and Figure 2-9 shows that the interface is equally as simple. All you need to do is copy and paste data from Excel in the input section on the top and then select what output format you want in the bottom half of the screen. Choose from variants of XML, JSON, and a number of others.

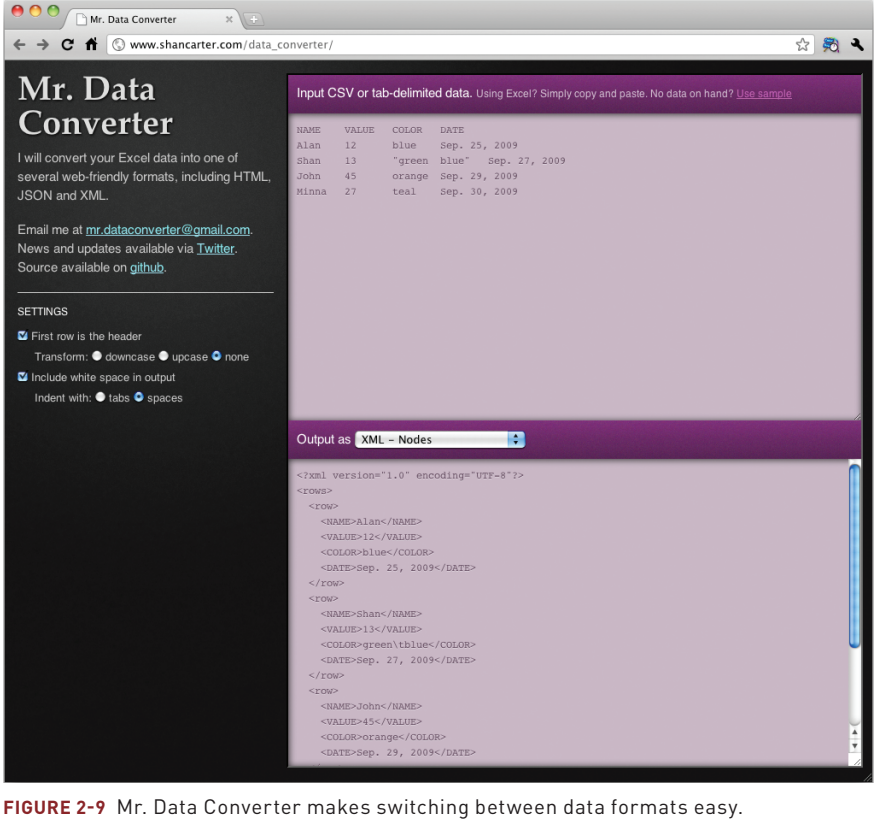


FIGURE 2-9 Mr. Data Converter makes switching between data formats easy.

The source code to Mr. Data Converter is also available if you want to make your own or extend.

> ▶ Try out Mr. Data Converter at `www.shancarter.com/data_converter/` or download the source on github at `https://github.com/shancarter/Mr-Data-Converter` to convert your Excel spreadsheets to a web-friendly format.

> ▶ Use Mr. People at `http://people.ericson.net/` or download the Ruby source on github to use the name parser in your own scripts: `http://github.com/mericson/people`.

MR. PEOPLE

Inspired by Carter's Mr. Data Converter, *The New York Times* graphics deputy director Matthew Ericson created Mr. People. Like Mr. Data Converter, Mr. People enables you to copy and paste data into a text field, and the tool parses and extracts for you. Mr. People, however, as you might guess, is specifically for parsing names.

Maybe you have a long list of names without a specific format, and you want to identify the first and last names, along with middle initial, prefix, and suffix. Maybe multiple people are listed on a single row. That's where Mr. People comes in. Copy and paste names, as shown in Figure 2-10, and you get a nice clean table that you can copy into your favorite spreadsheet software, as shown in Figure 2-11.

Like Mr. Data Converter, Mr. People is also available as open-source software on github.

SPREADSHEET SOFTWARE

Of course, if all you need is simple sorting, or you just need to make some small changes to individual data points, your favorite spreadsheet software is always available. Take this route if you're okay with manually editing data. Otherwise, try the preceding first (especially if you have a giganto dataset), or go with a custom coding solution.

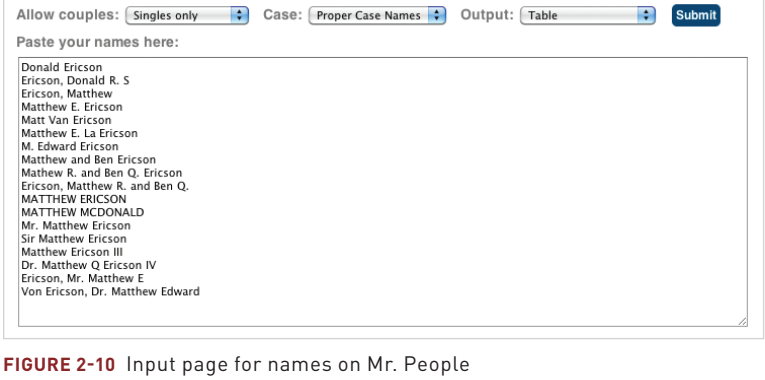
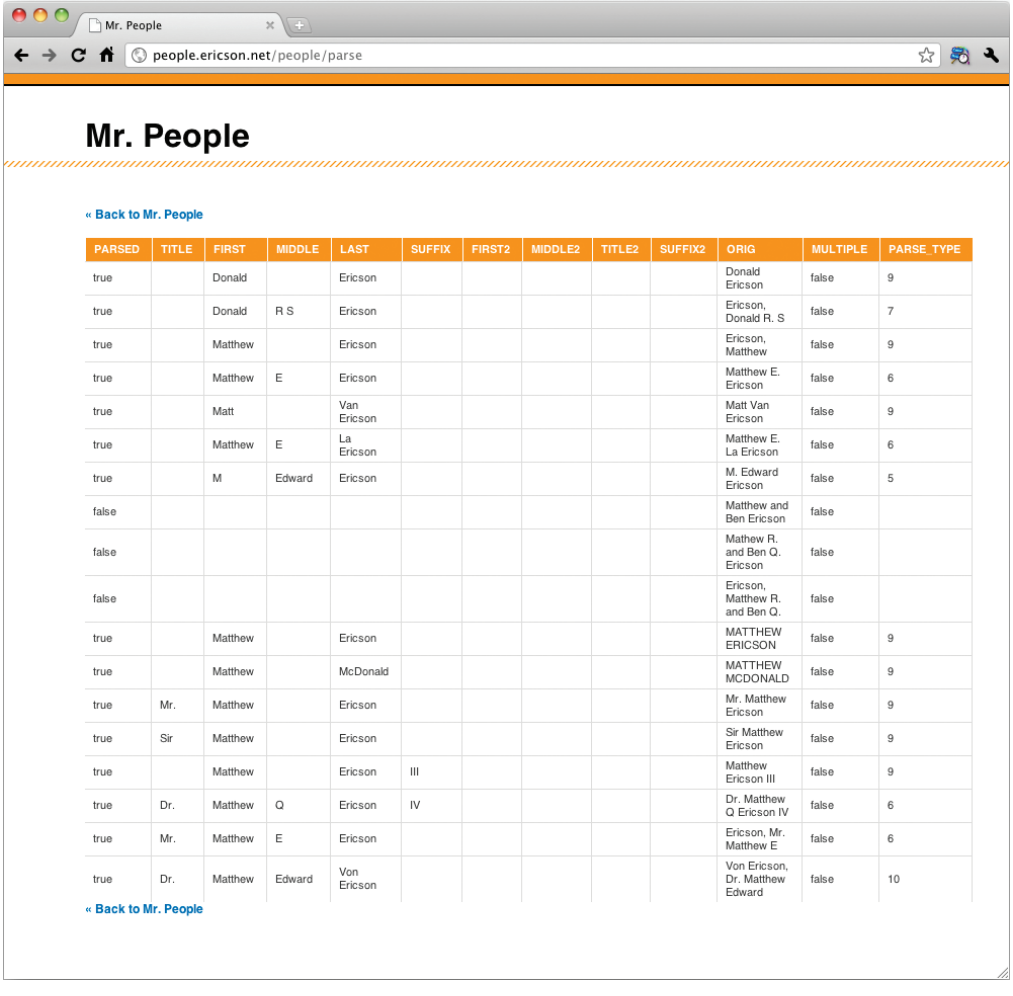


FIGURE 2-10 Input page for names on Mr. People



PARSED	TITLE	FIRST	MIDDLE	LAST	SUFFIX	FIRST2	MIDDLE2	TITLE2	SUFFIX2	ORIG	MULTIPLE	PARSE_TYPE
true		Donald		Ericson						Donald Ericson	false	9
true		Donald	R S	Ericson						Ericson, Donald R. S	false	7
true		Matthew		Ericson						Ericson, Matthew	false	9
true		Matthew	E	Ericson						Matthew E. Ericson	false	6
true		Matt		Van Ericson						Matt Van Ericson	false	9
true		Matthew	E	La Ericson						Matthew E. La Ericson	false	6
true		M	Edward	Ericson						M. Edward Ericson	false	5
false										Matthew and Ben Ericson	false	
false										Mathew R. and Ben Q. Ericson	false	
false										Ericson, Matthew R. and Ben Q.	false	
true		Matthew		Ericson						MATTHEW ERICSON	false	9
true		Matthew		McDonald						MATTHEW MCDONALD	false	9
true	Mr.	Matthew		Ericson						Mr. Matthew Ericson	false	9
true	Sir	Matthew		Ericson						Sir Matthew Ericson	false	9
true		Matthew		Ericson	III					Matthew Ericson III	false	9
true	Dr.	Matthew	Q	Ericson	IV					Dr. Matthew Q Ericson IV	false	6
true	Mr.	Matthew	E	Ericson						Ericson, Mr. Matthew E	false	6
true	Dr.	Matthew	Edward	Von Ericson						Von Ericson, Dr. Matthew Edward	false	10

FIGURE 2-11 Parsed names in table format with Mr. People

Formatting with Code

Although point-and-click software can be useful, sometimes the applications don't quite do what you want if you work with data long enough. Some software doesn't handle large data files well; they get slow or they crash.

What do you do at this point? You can throw your hands in the air and give up; although, that wouldn't be productive. Instead, you can write some

code to get the job done. With code you become much more flexible, and you can tailor your scripts specifically for your data.

Now jump right into an example on how to easily switch between data formats with just a few lines of code.

EXAMPLE: SWITCH BETWEEN DATA FORMATS

This example uses Python, but you can of course use any language you want. The logic is the same, but the syntax will be different. (I like to develop applications in Python, so managing raw data with Python fits into my workflow.)

Going back to the previous example on scraping data, use the resulting `wunder-data.txt` file, which has dates and temperatures in Buffalo, New York, for 2009. The first rows look like this:

```
20090101,26
20090102,34
20090103,27
20090104,34
20090105,34
20090106,31
20090107,35
20090108,30
20090109,25
...
```

This is a CSV file, but say you want the data as XML in the following format:

```
<weather_data>
    <observation>
        <date>20090101</date>
        <max_temperature>26</max_temperature>
    </observation>
    <observation>
        <date>20090102</date>
        <max_temperature>34</max_temperature>
    </observation>
    <observation>
        <date>20090103</date>
        <max_temperature>27</max_temperature>
    </observation>
    <observation>
```

```
        <date>20090104</date>
        <max_temperature>34</max_temperature>
    </observation>
    ...
</weather_data>
```

Each day's temperature is enclosed in `<observation>` tags with a `<date>` and the `<max_temperature>`.

To convert the CSV into the preceding XML format, you can use the following code snippet:

```
import csv
reader = csv.reader(open('wunder-data.txt', 'r'), delimiter=",")
print '<weather_data>'

for row in reader:
    print '<observation>'
    print '<date>' + row[0] + '</date>'
    print '<max_temperature>' + row[1] + '</max_temperature>'
    print '</observation>'

print '</weather_data>'
```

As before, you import the necessary modules. You need only the `csv` module in this case to read in `wunder-data.txt`.

```
import csv
```

The second line of code opens `wunder-data.txt` to read using `open()` and then reads it with the `csv.reader()` method.

```
reader = csv.reader(open('wunder-data.txt', 'r'), delimiter=",")
```

Notice the delimiter is specified as a comma. If the file were a tab-delimited file, you could specify the delimiter as `'\t'`.

Then you can print the opening line of the XML file in line 3.

```
print '<weather_data>'
```

In the main chunk of the code, you can loop through each row of data and print in the format that you need the XML to be in. In this example, each row in the CSV header is equivalent to each observation in the XML.

```
for row in reader:
    print '<observation>'
```

```
    print '<date>' + row[0] + '</date>'
    print '<max_temperature>' + row[1] + '</max_temperature>'
    print '</observation>'
```

Each row has two values: the date and the maximum temperature.

End the XML conversion with its closing tag.

```
print '</weather_data>'
```

Two main things are at play here. First, you read the data in, and then you iterate over the data, changing each row in some way. It's the same logic if you were to convert the resulting XML back to CSV. As shown in the following snippet, the difference is that you use a different module to parse the XML file.

```
from BeautifulSoup import BeautifulStoneSoup

f = open('wunder-data.xml', 'r')
xml = f.read()

soup = BeautifulStoneSoup(xml)
observations = soup.findAll('observation')
for o in observations:
    print o.date.string + "," + o.max_temperature.string
```

The code looks different, but you're basically doing the same thing. Instead of importing the `csv` module, you import BeautifulStoneSoup from BeautifulSoup. Remember you used BeautifulSoup to parse the HTML from Weather Underground. BeautifulStoneSoup parses the more general XML.

You can open the XML file for reading with `open()` and then load the contents in the `xml` variable. At this point, the contents are stored as a string. To parse, pass the `xml` string to `BeautifulStoneSoup` to iterate through each `<observation>` in the `XML` file. Use `findAll()` to fetch all the observations, and finally, like you did with the CSV to XML conversion, loop through each observation, printing the values in your desired format.

This takes you back to where you began:

```
20090101,26
20090102,34
20090103,27
20090104,34
...
```

To drive the point home, here's the code to convert your CSV to JSON format.

```
import csv
reader = csv.reader(open('wunder-data.txt', 'r'), delimiter=",")

print "{ observations: ["
rows_so_far = 0
for row in reader:

    rows_so_far += 1

    print '{'
    print '"date": ' + '"' + row[0] + '", '
    print '"temperature": ' + row[1]

    if rows_so_far < 365:
        print " },"
    else:
        print " }"

print "] }"
```

Go through the lines to figure out what's going on, but again, it's the same logic with different output. Here's what the JSON looks like if you run the preceding code.

```
{
    "observations": [
        {
            "date": "20090101",
            "temperature": 26
        },
        {
            "date": "20090102",
            "temperature": 34
        },
        ...
    ]
}
```

This is still the same data, with date and temperature but in a different format. Computers just love variety.

Put Logic in the Loop

If you look at the code to convert your CSV file to JSON, you should notice the *if-else* statement in the *for* loop, after the three print lines. This checks if the current iteration is the last row of data. If it isn't, don't put a comma at the end of the observation. Otherwise, you do. This is part of the JSON specification. You can do more here.

You can check if the max temperature is more than a certain amount and create a new field that is 1 if a day is more than the threshold, or 0 if it is not. You can create categories or flag days with missing values.

Actually, it doesn't have to be just a check for a threshold. You can calculate a moving average or the difference between the current day and the previous. There are lots of things you can do within the loop to augment the raw data. Everything isn't covered here because you can do anything from trivial changes to advanced analyses, but now look at a simple example.

Going back to your original CSV file, `wunder-data.txt`, create a third column that indicates whether a day's maximum temperature was at or below freezing. A 0 indicates above freezing, and 1 indicates at or below freezing.

```
import csv
reader = csv.reader(open('wunder-data.txt', 'r'), delimiter=",")
for row in reader:
    if int(row[1]) <= 32:
        is_freezing = '1'
    else:
        is_freezing = '0'

    print row[0] + "," + row[1] + "," + is_freezing
```

Like before, read the data from the CSV file into Python, and then iterate over each row. Check each day and flag accordingly.

This is of course a simple example, but it should be easy to see how you can expand on this logic to format or augment your data to your liking. Remember the three steps of load, loop, and process, and expand from there.

Wrapping Up

This chapter covered where you can find the data you need and how to manage it after you have it. This is an important step, if not the most important, in the visualization process. A data graphic is only as interesting as its underlying data. You can dress up a graphic all you want, but the data (or the results from your analysis of the data) is still the substance; and now that you know where and how to get your data, you're already a step ahead of the pack.

You also got your first taste of programming. You scraped data from a website and then formatted and rearranged that data, which will be a useful trick in later chapters. The main takeaway, however, is the logic in the code. You used Python, but you easily could have used Ruby, Perl, or PHP. The logic is the same across languages. When you learn one programming language (and if you're a programmer already, you can attest to this), it's much easier to learn other languages later.

You don't always have to turn to code. Sometimes there are click-and-drag applications that make your job a lot easier, and you should take advantage of that when you can. In the end, the more tools you have in your toolbox, the less likely you're going to get stuck somewhere in the process.

Okay, you have your data. Now it's time to get visual.