Data over the network

This lab covers

- [Fetching files via FTP/SFTP, SSH/SCP, and HTTPS]
- [Getting data via APIs]
- [Structured data file formats: JSON and XML]
- [Scraping data]

You've seen how to deal with text-based data files. In this lab, you use Python to move data files over the network. In some cases, those files might be text or spreadsheet files, as discussed in [chapter 21], but in other cases, they might be in more structured formats and served from REST or SOAP application programming interfaces (APIs). Sometimes, getting the data may mean scraping it from a website. This lab discusses all of these situations and shows some common use cases.

Fetching files

Before you can do anything with data files, you have to get them. Sometimes, this process is very easy, such as manually downloading a single zip archive, or maybe the files have been pushed to your machine from somewhere else. Quite often, however, the process is more involved. Maybe a large number of files needs to be retrieved from a remote server, files need to be retrieved regularly, or the retrieval process is sufficiently complex to be a pain to do manually. In any of those cases, you might well want to automate fetching the data files with Python.

First of all, I want to be clear that using a Python script isn't the only way, or always the best way, to retrieve files. The following sidebar offers more explanation of the factors I consider when deciding whether to use a Python script for file retrieval. Assuming that using Python does make sense for your particular use case, however, this section illustrates some common patterns you might employ.

Do I use Python?

Although using Python to retrieve files can work very well, it's not always the best choice. In making a decision, you might want to consider two things.

- [*Are simpler options available?* Depending on your operating system and your experience, you may find that simple shell scripts and command-line tools are simpler and easier to configure. If you don't have those tools available or aren't comfortable using them (or the people who will be maintaining them aren't comfortable with them), you may want to consider a Python script.]
- [*Is the retrieval process complex or tightly coupled with processing?* Although those situations are never desirable, they can occur. My rule these days is that if a shell script requires more than a few lines, or if I have to think hard about how to do something in a shell script, it's probably time to switch to Python.]

Using Python to fetch files from an FTP server

File Transfer Protocol (FTP) has been around for a very long time, but it's still a simple and easy way to share files when security isn't a huge concern. To access an FTP server in Python, you can use the [ftplib] module from the standard library. The steps to follow are straightforward: create an FTP object, connect to a server, and then log in with a username and password (or, quite commonly, with a username of "anonymous" and an empty password).

To continue working with weather data, you can connect to the National Oceanic and Atmospheric Administration (NOAA) FTP server, as shown here:

```
>>> import ftplib
>>> ftp = ftplib.FTP('tgftp.nws.noaa.gov')
>>> ftp.login()
'230 Login successful.'
```

When you're connected, you can use the [ftp] object to list and change directories:

```
>>> ftp.cwd('data')
'250 Directory successfully changed.'
>>> ftp.nlst()
['climate', 'fnmoc', 'forecasts', 'hurricane_products', 'ls_SS_services',
     'marine', 'nsd_bbsss.txt', 'nsd_cccc.txt', 'observations', 'products',
     'public_statement', 'raw', 'records', 'summaries', 'tampa',
     'watches_warnings', 'zonecatalog.curr', 'zonecatalog.curr.tar']
```

Then you can fetch, for example, the latest METAR report for Chicago O'Hare International Airport:

```
>>> x = ftp.retrbinary('RETR observations/metar/decoded/KORD.TXT',
     open('KORD.TXT', 'wb').write)
'226 Transfer complete.'
```

You pass the [ftp.retrbinary] method both the path to the file on the remote server and a method to handle that file's data on your end---in this case, the [write] method of a file you open for binary writing with the same name. When you look at KORD.TXT, you see that it contains the downloaded data:

```
CHICAGO O'HARE INTERNATIONAL, IL, United States (KORD) 41-59N 087-55W 200M
Jan 01, 2017 - 09:51 PM EST / 2017.01.02 0251 UTC
Wind: from the E (090 degrees) at 6 MPH (5 KT):0
Visibility: 10 mile(s):0
Sky conditions: mostly cloudy
Temperature: 33.1 F (0.6 C)
Windchill: 28 F (-2 C):1
Dew Point: 21.9 F (-5.6 C)
Relative Humidity: 63%
Pressure (altimeter): 30.14 in. Hg (1020 hPa)
Pressure tendency: 0.01 inches (0.2 hPa) lower than three hours ago
ob: KORD 020251Z 09005KT 10SM SCT150 BKN250 01/M06 A3014 RMK AO2 SLP214
     T00061056 58002
cycle: 3
```

You can also use [ftplib] to connect to servers using TLS encryption by using FTP_TLS instead of FTP:

```
ftp = ftplib.FTPTLS('tgftp.nws.noaa.gov')
```

Fetching files with SFTP

If the data requires more security, such as in a corporate context in which business data is being transferred over the network, it's fairly common to use SFTP. SFTP is a full-featured protocol that allows file access, transfer, and management over a Secure Shell (SSH) connection. Even though SFTP stands for SSH File Transfer Protocol and FTP stands for File Transfer Protocol, the two aren't related. SFTP isn't a reimplementation of FTP on SSH, but a fresh design specifically for SSH.

Using SSH-based transfers is attractive both because SSH is already the de facto standard for accessing remote servers and because enabling support for SFTP on a server is fairly easy (and quite often on by default).

Python doesn't have an SFTP/SCP client module in its standard library, but a community-developed library called [paramiko] manages SFTP operations as well as SSH connections. To use [paramiko], the easiest thing is to install it via [pip]. If the NOAA site mentioned earlier in this lab were using SFTP (which it doesn't, so this code won't work!), the SFTP equivalent of the code above would be

```
>>> import paramiko
>>> t = paramiko.Transport((hostname, port))
>>> t.connect(username, password)
>>> sftp = paramiko.SFTPClient.from_transport(t)
```

It's also worth noting that although [paramiko] supports running commands on a remote server and receiving its outputs, just like a direct [ssh] session, it doesn't include an [scp] function. This function is rarely something you'll miss; if all you want to do is move a file or two over an [ssh] connection, a command-line [scp] utility usually makes the job easier and simpler.

Retrieving files over HTTP/HTTPS

The last common option for retrieving data files that I discuss in this chapter is getting files over an HTTP or HTTPS connection. This option is probably the easiest of all the options; you are in effect retrieving your data from a web server, and support for accessing web servers is very widespread. Again, in this case you may not need to use Python. Various command-line tools retrieve files via HTTP/HTTPS connections and have most of the capabilities you might need. The two most common of these tools are wget and curl. If you have a reason to do the retrieval in your Python code, however, that process isn't much harder. The [requests] library is by far the easiest and most reliable way to access HTTP/HTTPS servers from Python code. Again, [requests] is easiest to install with [pip install requests].

When you have requests installed, fetching a file is straightforward: import [requests] and use the correct HTTP verb (usually, GET) to connect to the server and return your data.

The following example code fetches the monthly temperature data for Heathrow Airport since 1948---a text file that's served via a web server. If you want to, you can put the URL in your browser, load the page, and then save it. If the page is large or you have a lot of pages to get, however, it's easier to use code like this:

```
>>> import requests
>>> response = requests.get("http://www.metoffice.gov.uk/pub/data/weather/uk/
     climate/stationdata/heathrowdata.txt")
```

The response will have a fair amount of information, including the header returned by the web server, which can be helpful in debugging if things aren't working. The part of the response object you'll most often be interested in, however, is data returned. To retrieve this data, you want to access the response's [text] property, which contains the response body as a string, or the [content] property, which contains the response body as bytes:

```
>>> print(response.text)
Heathrow (London Airport)
Location 507800E 176700N, Lat 51.479 Lon -0.449, 25m amsl
Estimated data is marked with a * after the value.
Missing data (more than 2 days missing in month) is marked by  ---.
Sunshine data taken from an automatic Kipp & Zonen sensor marked with a #,
     otherwise sunshine data taken from a Campbell Stokes recorder.
   yyyy  mm   tmax    tmin      af    rain     sun
              degC    degC    days      mm   hours
   1948   1    8.9     3.3    ---     85.0    ---
   1948   2    7.9     2.2    ---     26.0    ---
   1948   3   14.2     3.8    ---     14.0    ---
   1948   4   15.4     5.1    ---     35.0    ---
   1948   5   18.1     6.9    ---     57.0    ---
```

Typically, you'd write the response text to a file for later processing, but depending on your needs, you might first do some cleaning or even process directly.

Try this: Retrieving A file

If you're working with the example data file and want to break each line into separate fields, how might you do that? What other processing would you expect to do? Try writing some code to retrieve this file and calculate the average annual rainfall or (for more of a challenge) the average maximum and minimum temperature for each year.

Fetching data via an API

Serving data by way of an API is quite common, following a trend toward decoupling applications into services that communicate via APIs. APIs can work in several ways, but they commonly operate over regular HTTP/HTTPS protocols using the standard HTTP verbs, GET, POST, PUT, and DELETE. Fetching data this way is very similar to retrieving a file, as in [section 22.1.3], but the data isn't in a static file. Instead of the application serving static files that contain the data, it queries some other data source and then assembles and serves the data dynamically on request.

Although there's a lot of variation in the ways that an API can be set up, one of the most common is a RESTful (REpresentational State Transfer) interface that operates over the same HTTP/HTTPS protocols as the web. There are endless variations on how an API might work, but commonly, data is fetched by using a GET request, which is what your web browser uses to request a web page. When you're fetching via a GET request, the parameters to select the data you want are often appended to the URL in a query string.

If you want to get the current weather on Mars from the Curiosity rover, use http://mng.bz/g6UY as your URL.^[[1]]^ The [?format=json] is a query string parameter that specifies that the information be returned in JSON, which I discuss in [section 22.3.1]. If you want the Martian weather for a specific Martian day, or sol, of its mission---say, the 155th sol---use the URL http://mng.bz/4e0r. If you want to get the weather on Mars for a range of Earth dates, such as the month of October 2012, use http://mng.bz/83WO. Notice that the elements of the query string are separated by ampersands ([&]).

The site (ingenology.com) has been reliable in the past, but is down at the time of this writing and its future is uncertain.

When you know the URL to use, you can use the requests library to fetch data from an API and either process it on the fly or save it to a file for later processing. The simplest way to do this is exactly like retrieving a file:

```
>>> import requests
>>> response = requests.get("http://marsweather.ingenology.com/v1/latest/
     ?format=json")
>>> response.text
'{"report": {"terrestrial_date": "2017-01-08", "sol": 1573, "ls": 295.0,
     "min_temp": -74.0, "min_temp_fahrenheit": -101.2, "max_temp": -2.0,
     "max_temp_fahrenheit": 28.4, "pressure": 872.0, "pressure_string":
     "Higher", "abs_humidity": null, "wind_speed": null, "wind_direction":
     "--", "atmo_opacity": "Sunny", "season": "Month 10", "sunrise": "2017-01-
     08T12:29:00Z", "sunset": "2017-01-09T00:45:00Z"}}'
>>> response = requests.get("http://marsweather.ingenology.com/v1/archive/
     ?sol=155&format=json")
>>> response.text
'{"count": 1, "next": null, "previous": null, "results":
     [{"terrestrial_date": "2013-01-18", "sol": 155, "ls": 243.7, "min_temp":
     -64.45, "min_temp_fahrenheit": -84.01, "max_temp": 2.15,
     "max_temp_fahrenheit": 35.87, "pressure": 9.175, "pressure_string":
     "Higher", "abs_humidity": null, "wind_speed": 2.0, "wind_direction":
     null, "atmo_opacity": null, "season": "Month 9", "sunrise": null,
     "sunset": null}]}'
```

Keep in mind that you should escape spaces and most punctuation in your query parameters, because those elements aren't allowed in URLs even though many browsers automatically do the escaping on URLs.

For a final example, suppose that you want to grab the crime data for Chicago between noon and 1 PM on Jan. 10, 2017. The way that the API works, you specify a date range with the query string parameters of [$where date=between <start datetime>] and [<end datetime>], where the start and end datetimes are quoted in ISO format. So the URL for getting that one hour of Chicago crime data would be https://data.cityofchicago.org/resource/6zsd-86xi.json?$where=datebetween'2015-01-10T12:00:00'and'2015-01-10T13:00:00'.

In the example, several characters aren't welcome in URLs, such as the quote characters and the spaces. This is another situation in which the requests library makes good on its aim of making things easier for the user, because before it sends the URL, it takes care of quoting it properly. The URL that the request actually sends is https://data.cityofchicago.org/resource/6zsd-86xi.json?$where=date%20between%20%222015-01-10T12:00:00%22%20and%20%222015-01-10T14:00:00%22'.

Note that all of the single-quote characters have been quoted with %22 and all of the spaces with %20 without your even needing to think about it.

Try this: Accessing an API

Write some code to fetch some data from the city of Chicago website. Look at the fields mentioned in the results, and see whether you can select records based on another field in combination with the date range.

Structured data formats

Although APIs sometimes serve plain text, it's much more common for data served from APIs to be served in a structured file format. The two most common file formats are JSON and XML. Both of these formats are built on plain text but structure their contents so that they're more flexible and able to store more complex information.

JSON data

JSON, which stands for JavaScript Object Notation, dates to 1999. It consists of only two structures: key-value pairs, called *structures*, that are very similar to Python dictionaries; and ordered lists of values, called *arrays*, that are very much like Python lists.

Keys can be only strings in double quotes, and values can be strings in double quotes, numbers, true, false, null, arrays, or objects. These elements make JSON a lightweight way to represent most data in a way that's easily transmitted over the network and also fairly easy for humans to read. JSON is so common that most languages have features to translate JSON to and from native data types. In the case of Python, that feature is the [json] module, which became part of the standard library with version 2.6. The original externally maintained version of the module is available as [simplejson], which is still available. In Python 3, however, it's far more common to use the standard library version.

The data you retrieved from the Mars rover and the city of Chicago APIs in [section 22.2] is in JSON format. To send JSON across the network, the JSON object needs to be serialized---that is, transformed into a sequence of bytes. So although the batch of data you retrieved from the Mars rover and Chicago APIs looks like JSON, in fact it's just a byte string representation of a JSON object. To transform that byte string into a real JSON object and translate it into a Python dictionary, you need to use the JSON [loads()] function. If you want to get the Mars weather report, for example, you can do that just as you did previously, but this time you'll convert it to a Python dictionary:

```
>>> import json
>>> import requests
>>> response = requests.get("http://marsweather.ingenology.com/v1/latest/
     ?format=json")
>>> weather = json.loads(response.text)
```

```
>>> weather
{'report': {'terrestrial_date': '2017-01-10', 'sol': 1575, 'ls': 296.0,
     'min_temp': -58.0, 'min_temp_fahrenheit': -72.4, 'max_temp': 0.0,
     'max_temp_fahrenheit': None, 'pressure': 860.0, 'pressure_string':
     'Higher', 'abs_humidity': None, 'wind_speed': None, 'wind_direction': '-
     -', 'atmo_opacity': 'Sunny', 'season': 'Month 10', 'sunrise': '2017-01-
     10T12:30:00Z', 'sunset': '2017-01-11T00:46:00Z'}}
>>> weather['report']['sol']
1575
```

Note that the call to [json.loads()]is what takes the string representation of the JSON object and transforms, or loads, it into a Python dictionary. Also, a [json.load()] function will read from any filelike object that supports a read method.

If you look at a dictionary's representation as earlier, it can be very hard to make sense of what's going on. Improved formatting, also called *pretty printing*, can make data structures much easier to understand. Use the Python [prettyprint] module to see what's in the example dictionary:

```
>>> from pprint import pprint as pp
>>> pp(weather)
{'report': {'abs_humidity': None,
            'atmo_opacity': 'Sunny',
            'ls': 296.0,
            'max_temp': 0.0,
            'max_temp_fahrenheit': None,
            'min_temp': -58.0,
            'min_temp_fahrenheit': -72.4,
            'pressure': 860.0,
            'pressure_string': 'Higher',
            'season': 'Month 10',
            'sol': 1575,
            'sunrise': '2017-01-10T12:30:00Z',
            'sunset': '2017-01-11T00:46:00Z',
            'terrestrial_date': '2017-01-10',
            'wind_direction': '--',
            'wind_speed': None}}
```

Both load functions can be configured to control how to parse and decode the original JSON to Python objects, but the default translation is listed below.

Table 22.1. JSON to Python default decoding

JSON Python

---

object dict array list string str number (int) int number (real) float true True false False null None

Fetching JSON with the requests library

In this section, you used the requests library to retrieve the JSON formatted data and then used the [json.loads()] method to parse it into a Python object. This technique works fine, but because the requests library is used so often for exactly this purpose, the library provides a shortcut: The response object actually has a [json()] method that does that conversion for you. So in the example, instead of

```
>>> weather = json.loads(response.text)
```

you could have used

```
>>> weather = response.json()
```

The result is the same, but the code is simpler, more readable, and more Pythonic.

If you want to write JSON to a file or serialize it to a string, the reverse of [load()] and [loads()] is [dump()] and [dumps()]. [json.dump()] takes a file object with a [write()] method as a parameter, and [json.dumps()]returns a string. In both cases, the encoding to a JSON formatted string can be highly customized, but the default is still based on [table 22.1]. So if you want to write your Martian weather report to a JSON file, you could do this:

```
>>> outfile = open("mars_data_01.json", "w")
>>> json.dump(weather, outfile)
>>> outfile.close()
>>> json.dumps(weather)
'{"report": {"terrestrial_date": "2017-01-11", "sol": 1576, "ls": 296.0,
     "min_temp": -72.0, "min_temp_fahrenheit": -97.6, "max_temp": -1.0,
     "max_temp_fahrenheit": 30.2, "pressure": 869.0, "pressure_string":
     "Higher", "abs_humidity": null, "wind_speed": null, "wind_direction": "-
     -", "atmo_opacity": "Sunny", "season": "Month 10", "sunrise": "2017-01-
     11T12:31:00Z", "sunset": "2017-01-12T00:46:00Z"}}'
```

As you can see, the entire object has been encoded as a single string. Here again, it might be handy to format the string in a more readable way, just as you did by using the [pprint] module. To do so easily, use the [indent] parameter with the [dump] or [dumps] function:

```
>>> print(json.dumps(weather, indent=2))
{
  "report": {
    "terrestrial_date": "2017-01-10",
    "sol": 1575,
    "ls": 296.0,
    "min_temp": -58.0,
    "min_temp_fahrenheit": -72.4,
    "max_temp": 0.0,
    "max_temp_fahrenheit": null,
    "pressure": 860.0,
    "pressure_string": "Higher",
    "abs_humidity": null,
    "wind_speed": null,
    "wind_direction": "--",
    "atmo_opacity": "Sunny",
    "season": "Month 10",
    "sunrise": "2017-01-10T12:30:00Z",
    "sunset": "2017-01-11T00:46:00Z"
  }
}
```

You should be aware, however, that if you use repeated calls to [json.dump()] to write a series of objects to a file, the result is a *series* of legal JSON-formatted objects, but the contents of the file *as a whole* is *not* a legal JSON-formatted object, and attempting to read and parse the entire file by using a single call to [json.load()] will fail. If you have more than one object that you'd like to encode as a single JSON object, you need to put all those objects into a list (or, better still, an object) and then encode that item to the file.

If you have two or more days' worth of Martian weather data that you want to store as JSON, you have to make a choice. You could use [json.dump()]once for each object, which would result in a file containing JSON-formatted objects. If you assume that [weather_list] is a list of weather-report objects, the code might look like this:

```
>>> outfile = open("mars_data.json", "w")
>>> for report in weather_list:
...     json.dump(weather, outfile)
>>> outfile.close()
```

If you do this, then you need to load each line as a separate JSON-formatted object:

```
>>> for line in open("mars_data.json"):
...     weather_list.append(json.loads(line))
```

As an alternative, you could put the list into a single JSON object. Because there's a possible vulnerability with top-level arrays in JSON, the recommended way is to put the array in a dictionary:

```
>>> outfile = open("mars_data.json", "w")
>>> weather_obj = {"reports": weather_list, "count": 2}
>>> json.dump(weather, outfile)
>>> outfile.close()
```

With this approach, you can use one operation to load the JSON-formatted object from the file:

```
>>> with open("mars_data.json") as infile:
>>> weather_obj = json.load(infile)
```

The second approach is fine if the size of your JSON files is manageable, but it may be less than ideal for very large files, because handling errors may be a bit harder and you may run out of memory.

Try this: Saving some JSON crime data

Modify the code you wrote in [section 22.2] to fetch the Chicago crime data. Then convert the fetched data from a JSON-formatted string to a Python object. Next, see whether you can save the crime events as a series of separate JSON objects in one file and as one JSON object in another file. Then see what code is needed to load each file.

XML data

XML (eXtensible Markup Language) has been around since the end of the 20th century. XML uses an angle-bracket tag notation similar to HTML, and elements are nested within other elements to form a tree structure. XML was intended to be readable by both machines and humans, but XML is often so verbose and complex that it's very difficult for people to understand. Nevertheless, because XML is an established standard, it's quite common to find data in XML format. And although XML is machine-readable, it's very likely that you'll want to translate it into something a bit easier to deal with.

Take a look at some XML data, in this case the XML version of weather data for Chicago:

```
<dwml xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
     www.w3.org/2001/XMLSchema-instance" version="1.0"
     xsi:noNamespaceSchemaLocation="http://www.nws.noaa.gov/forecasts/xml/
     DWMLgen/schema/DWML.xsd">
  <head>
    <product srsName="WGS 1984" concise-name="glance" operational-
     mode="official">
      <title>
```

```
NOAA's National Weather Service Forecast at a Glance
      </title>
      <field>meteorological</field>
      <category>forecast</category>
      <creation-date refresh-frequency="PT1H">2017-01-08T02:52:41Z</creation-
     date>
    </product>
    <source>
      <more-information>http://www.nws.noaa.gov/forecasts/xml/</more-
     information>
      <production-center>
Meteorological Development Laboratory
<sub-center>Product Generation Branch</sub-center>
      </production-center>
      <disclaimer>http://www.nws.noaa.gov/disclaimer.html</disclaimer>
      <credit>http://www.weather.gov/</credit>
      <credit-logo>http://www.weather.gov/images/xml_logo.gif</credit-logo>
      <feedback>http://www.weather.gov/feedback.php</feedback>
    </source>
  </head>
  <data>
    <location>
      <location-key>point1</location-key>
      <point latitude="41.78" longitude="-88.65"/>
    </location>
    ...
  </data>
</dwml>
```

This example is just the first section of the document, with most of the data omitted. Even so, it illustrates some of the issues you typically find in XML data. In particular, you can see the verbose nature of the protocol, with the tags in some cases taking more space than the value contained in them. This sample also shows the nested or tree structure common in XML, as well as the common use of a sizeable header of metadata before the actual data begins. On a spectrum from simple to complex for data files, you could think of CSV or delimited files as being at the simple end and XML at the complex end.

Finally, this file illustrates another feature of XML that makes pulling data a bit more of a challenge. XML supports the use of attributes to store data as well as the text values within the tags. So if you look at the point element at the bottom of this sample, you see that the [point] element doesn't have a text value. That element has just latitude and longitude values within the [<point>] tag itself:

```
<point latitude="41.78" longitude="-88.65"/>
```

This code is certainly legal XML, and it works for storing the data, but it would also be possible (likely, even) for the same data to be stored as

```
<point>
   <latitude>41.78</ latitude >
   <longitude>-88.65</longitude>
</point>
```

You really don't know which way any given bit of data will be handled without carefully inspecting the data or studying a specification document.

This kind of complexity can make simple data extraction from XML more of a challenge. You have several ways to handle XML. The Python standard library comes with modules that parse and handle XML data, but none of them is particularly convenient for simple data extraction.

For simple data extraction, the handiest utility I've found is a library called [xmltodict], which parses your XML data and returns a dictionary that reflects the tree. In fact, behind the scenes it uses the standard library's expat XML parser, parses your XML document into a tree, and uses that tree to create the dictionary. As a result, [xmltodict] can handle whatever the parser can, and it's also able to take a dictionary and "unparse" it to XML if necessary, making it a very handy tool. Over several years of use, I found this solution to be up to all my XML handling needs. To get [xmltodict], you can again use [pip install xmltodict].

To convert the XML to a dictionary, you can import [xmltodict] and use the [parse] method on an XML formatted string:

```
>>> import xmltodict
>>> data = xmltodict.parse(open("observations_01.xml").read())
```

In this case, for compactness, pass the contents of the file directly to the [parse] method. After being parsed, this data object is an ordered dictionary with the same values it would have if it had been loaded from this JSON:

```
{
    "dwml": {
        "@xmlns:xsd": "http://www.w3.org/2001/XMLSchema",
        "@xmlns:xsi": "http://www.w3.org/2001/XMLSchema-instance",
        "@version": "1.0",
        "@xsi:noNamespaceSchemaLocation": "http://www.nws.noaa.gov/forecasts/
     xml/DWMLgen/schema/DWML.xsd",
        "head": {
            "product": {
                "@srsName": "WGS 1984",
                "@concise-name": "glance",
                "@operational-mode": "official",
                "title": "NOAA's National Weather Service Forecast at a Glance",
                "field": "meteorological",
                "category": "forecast",
                "creation-date": {
                    "@refresh-frequency": "PT1H",
                    "#text": "2017-01-08T02:52:41Z"
                }
            },
            "source": {
                "more-information": "http://www.nws.noaa.gov/forecasts/xml/",
                "production-center": {
                    "sub-center": "Product Generation Branch",
                    "#text": "Meteorological Development Laboratory"
                },
                "disclaimer": "http://www.nws.noaa.gov/disclaimer.html",
                "credit": "http://www.weather.gov/",
                "credit-logo": "http://www.weather.gov/images/xml_logo.gif",
                "feedback": "http://www.weather.gov/feedback.php"
            }
        },
        "data": {
```

```
            "location": {
                "location-key": "point1",
                "point": {
                    "@latitude": "41.78",
                    "@longitude": "-88.65"
                }
            }
        }
    }
}
```

Notice that the attributes have been pulled out of the tags, but with an [@] prepended to indicate that they were originally attributes of their parent tag. If an XML node has both a text value and a nested element in it, notice that the key for the text value is ["#text"], as in the ["sub-center"] element under ["production-center"].

Earlier, I said that the result of parsing is an *ordered dictionary* (officially, an [OrderedDict]), so if you print it, the code looks like this:

```
OrderedDict([('dwml', OrderedDict([('@xmlns:xsd', 'http://www.w3.org/2001/
     XMLSchema'), ('@xmlns:xsi', 'http://www.w3.org/2001/XMLSchema-
     instance'), ('@version', '1.0'), ('@xsi:noNamespaceSchemaLocation',
     'http://www.nws.noaa.gov/forecasts/xml/DWMLgen/schema/DWML.xsd'),
     ('head', OrderedDict([('product', OrderedDict([('@srsName', 'WGS 1984'),
     ('@concise-name', 'glance'), ('@operational-mode', 'official'),
     ('title', "NOAA's National Weather Service Forecast at a Glance"),
     ('field', 'meteorological'), ('category', 'forecast'), ('creation-date',
     OrderedDict([('@refresh-frequency', 'PT1H'), ('#text', '2017-01-
     08T02:52:41Z')]))])), ('source', OrderedDict([('more-information',
     'http://www.nws.noaa.gov/forecasts/xml/'), ('production-center',
     OrderedDict([('sub-center', 'Product Generation Branch'), ('#text',
     'Meteorological Development Laboratory')])), ('disclaimer', 'http://
     www.nws.noaa.gov/disclaimer.html'), ('credit', 'http://www.weather.gov/
     '), ('credit-logo', 'http://www.weather.gov/images/xml_logo.gif'),
     ('feedback', 'http://www.weather.gov/feedback.php')]))])), ('data',
     OrderedDict([('location', OrderedDict([('location-key', 'point1'),
     ('point', OrderedDict([('@latitude', '41.78'), ('@longitude', '-
     88.65')]))])), ('#text', '...')]))])])])
```

Even though the representation of an [OrderedDict], with its lists of tuples, looks rather strange, it behaves exactly the same way as a normal [dict] except that it promises to maintain the order of elements, which is useful in this case.

If an element is repeated, it becomes a list. In a further section of the full version of the file shown previously the following element occurs (some elements omitted from this sample):

```
<time-layout >
    <start-valid-time period-name="Monday">2017-01-09T07:00:00-06:00</start-
     valid-time>
    <end-valid-time>2017-01-09T19:00:00-06:00</end-valid-time>
    <start-valid-time period-name="Tuesday">2017-01-10T07:00:00-06:00</start-
     valid-time>
    <end-valid-time>2017-01-10T19:00:00-06:00</end-valid-time>
    <start-valid-time period-name="Wednesday">2017-01-11T07:00:00-06:00</
     start-valid-time>
```

```
    <end-valid-time>2017-01-11T19:00:00-06:00</end-valid-time>
</time-layout>
```

Note that two elements---"[start-valid-time]" and "[end-valid-time]"---repeat in alternation. These two repeating elements are each translated to a list in the dictionary, keeping each set of elements in their proper order:

```
            "time-layout":
                {
                    "start-valid-time": [
                        {
                            "@period-name": "Monday",
                            "#text": "2017-01-09T07:00:00-06:00"
                        },
                        {
                            "@period-name": "Tuesday",
                            "#text": "2017-01-10T07:00:00-06:00"
                        },
                        {
                            "@period-name": "Wednesday",
                            "#text": "2017-01-11T07:00:00-06:00"
                        }
                    ],
                    "end-valid-time": [
                        "2017-01-09T19:00:00-06:00",
                        "2017-01-10T19:00:00-06:00",
                        "2017-01-11T19:00:00-06:00"
                    ]
                },
```

Because dictionaries and lists, even nested dictionaries and lists, are fairly easy to deal with in Python, using [xmltodict] is an effective way to handle most XML. In fact, I've used it for the past several years in production on a variety of XML documents and never had a problem.

Try this: Fetching and Parsing XML

Write the code to pull the Chicago XML weather forecast from http://mng.bz/103V. Then use [xmltodict] to parse the XML into a Python dictionary and extract tomorrow's forecast maximum temperature. Hint: To match up time layouts and values, compare the layout-key value of the first time-layout section and the time-layout attribute of the temperature element of the parameters element.

Scraping web data

In some cases, the data is on a website but for whatever reason isn't available anywhere else. In those situations, it may make sense to collect the data from the web pages themselves through a process called *crawling* or *scraping*.

Before saying anything more about scraping, let me make a disclaimer: Scraping or crawling websites that you don't own or control is at best a legal grey area, with a host of inconclusive and contradictory considerations involving things such as the terms of use of the site, the way in which the site is accessed, and the use to which the scraped data is put. Unless you have control of the site you want to scrape, the answer to the question "Is it legal for me to scrape this site?" usually is "It depends."

If you do decide to scrape a production website, you also need to be sensitive to the load you're putting on the site. Although an established, high-traffic site might well be able to handle anything you can throw at it, a smaller, less-active site might be brought to a standstill by a series of continuous requests. At the very least, you need to be careful that your scraping doesn't turn into an inadvertent denial-of-service attack.

Conversely, I've worked in situations in which it was actually easier to scrape our own website to get some needed data than it was to go through corporate channels. Although scraping web data has its place, it's too complex for full treatment here. In this section, I present a very simple example to give you a general idea of the basic method and follow up with suggestions to pursue in more complex cases.

Scraping a website consists of two parts: fetching the web page and extracting the data from it. Fetching the page can be done via requests and is fairly simple.

Consider the code of a very simple web page with only a little content and no CSS or JavaScript, as this one.

Listing 22.1. File test.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>Title</title>
</head>

<body>
<h1>Heading 1</h1>

This is plan text, and is boring
<span class="special">this is special</span>

Here is a <a href="http://bitbucket.dev.null">link</a>

<hr>
<address>Ann Address, Somewhere, AState 00000
</address>
</body> </html>
```

Suppose that you're interested in only a couple of kinds of data from this page: anything in an element with a class name of ["special"] and any links. You can process the file by searching for the strings ['class="special"'] and ["<a href"] and then write code to pick out the data from there, but even using regular expressions, this process will be tedious, bug-prone, and hard to maintain. It's much easier to use a library that knows how to parse HTML, such as Beautiful Soup. If you want to try the following code and experiment with parsing HTML pages, you can use [pip install bs4].

When you have Beautiful Soup installed, parsing a page of HTML is simple. For this example, assume that you've already retrieved the web page (presumably, using the requests library), so you'll just parse the HTML.

The first step is to load the text and create a Beautiful Soup parser:

```
>>> import bs4
>>> html = open("test.html").read()
>>> bs = bs4.BeautifulSoup(html, "html.parser")
```

This code is all it takes to parse the HTML into the parser object [bs]. A Beautiful Soup parser object has a lot of cool tricks, and if you're working with HTML at all, it's really worth your time to experiment a bit and get a feel for what it can do for you. For this example, you look at only two things: extracting content by HTML tag and getting data by CSS class.

First, find the link. The HTML tag for a link is [<a>] (Beautiful Soup by default converts all tags to lowercase), so to find all link tags, you can use the ["a"] as a parameter and call the [bs] object itself:

```
>>> a_list = bs("a")
>>> print(a_list)
[<a href="http://bitbucket.dev.null">link</a>]
```

Now you have a list of all (one in this case) of the HTML link tags. If that list is all you get, that's not so bad, but in fact, the elements returned in the list are also parser objects and can do the rest of the work of getting the link and text for you:

```
>>> a_item = a_list[0]
>>> a_item.text
'link'
>>> a_item["href"]
'http://bitbucket.dev.null'
```

The other feature you're looking for is anything with a CSS class of ["special"], which you can extract by using the parser's [select] method as follows:

```
>>> special_list = bs.select(".special")
>>> print(special_list)
[<span class="special">this is special</span>]
>>> special_item = special_list[0]
>>> special_item.text
'this is special'
>>> special_item["class"]
['special']
```

Because the items returned by the tag or by the [select] method are themselves parser objects, you can nest them, which allows you to extract just about anything from HTML or even XML.

Try this: Parsing HTML

Given the file forecast.html (which you can find in the code on this book's website), write a script using Beautiful Soup that extracts the data and saves it as a CSV file, shown here.

Listing 22.2. File forecast.html

```
<html>
  <body>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Tonight</b></div>
        <div class="grid col-75 forecast-text">A slight chance of showers and
     thunderstorms before 10pm. Mostly cloudy, with a low around 66. West
     southwest wind around 9 mph. Chance of precipitation is 20%. New
     rainfall amounts between a tenth and quarter of an inch possible.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Friday</b></div>
        <div class="grid col-75 forecast-text">Partly sunny. High near 77,
     with temperatures falling to around 75 in the afternoon. Northwest wind
     7 to 12 mph, with gusts as high as 18 mph.</div>
    </div>
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Friday Night</b></div>
        <div class="grid col-75 forecast-text">Mostly cloudy, with a low
```

```
 around 63. North wind 7 to 10 mph.</div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Saturday</b></div>
    <div class="grid col-75 forecast-text">Mostly sunny, with a high near
 73. North wind around 10 mph.</div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Saturday Night</b></div>
    <div class="grid col-75 forecast-text">Partly cloudy, with a low
 around 63. North wind 5 to 10 mph.</div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Sunday</b></div>
    <div class="grid col-75 forecast-text">Mostly sunny, with a high near
 73.</div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Sunday Night</b></div>
    <div class="grid col-75 forecast-text">Mostly cloudy, with a low
 around 64.</div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Monday</b></div>
    <div class="grid col-75 forecast-text">Mostly sunny, with a high near
 74.</div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Monday Night</b></div>
    <div class="grid col-75 forecast-text">Mostly clear, with a low
 around 65.</div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Tuesday</b></div>
    <div class="grid col-75 forecast-text">Sunny, with a high near 75.</
 div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Tuesday Night</b></div>
    <div class="grid col-75 forecast-text">Mostly clear, with a low
 around 65.</div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Wednesday</b></div>
    <div class="grid col-75 forecast-text">Sunny, with a high near 77.</
 div>
</div>
<div class="row row-forecast">
    <div class="grid col-25 forecast-label"><b>Wednesday Night</b></div>
    <div class="grid col-75 forecast-text">Mostly clear, with a low
 around 67.</div>
</div>
```

```
    <div class="row row-forecast">
        <div class="grid col-25 forecast-label"><b>Thursday</b></div>
        <div class="grid col-75 forecast-text">A chance of rain showers after
     1pm. Mostly sunny, with a high near 81. Chance of precipitation is
     30%.</div>
    </div>
  </body>
</html>
```

Lab 22: Track Curiosity's Weather

Use the API described in [section 22.2] to gather a weather history of *Curiosity*'s stay on Mars for a month. Hint: You can specify Martian days (sols) by adding ?sol=*sol_number* to the end of the archive query, like this:

http://marsweather.ingenology.com/v1/archive/?sol=155

Transform the data so that you can load it into a spreadsheet and graph it. For a version of this project, see the book's source code.

Summary

- [Using a Python script may not be the best choice for fetching files. Be sure to consider the options.]
- [Using the [requests] module is your best bet for fetching files by using HTTP/HTTPS and Python.]
- [Fetching files from an API is very similar to fetching static files.]
- [Parameters for API requests often need to be quoted and added as a query string to the request URL.]
- [JSON-formatted strings are quite common for data served from APIs, and XML is also used.]
- [Scraping sites that you don't control may not be legal or ethical and requires consideration not to overload the server.]