11.2 The XML Data Format

May 9, 2019

1 11.2 The XML Data Format

XML, which stands for eXtensible Markup Language, is another way to represent hierarchical data. The basic building block of XML is the **tag**, denoted by angle brackets <>.

For example, a data set of movies might be represented using XML as follows:

```
<movies>
  <movie id="1" title="The Godfather">
    <director id="50" name="Coppola, Francis Ford">
    </director>
    <releasedate>1972-03-24</releasedate>
    <character id="100" name="Vito Corleone">
      <actor id="200" name="Brando, Marlon">
      </actor>
    </character>
    <character id="101" name="Michael Corleone">
      <actor id="201" name="Pacino, Al">
      </actor>
    </character>
    ...
  </movie>
  <movie id="2" title="The Godfather: Part II">
    <director id="50" name="Coppola, Francis Ford">
    </director>
    <releasedate>1974-10-20</releasedate>
    <character id="101" name="Michael Corleone">
      <actor id="201" name="Pacino, Al">
      </actor>
    </character>
    <character id="100" name="Vito Corleone">
      <actor id="250" name="De Niro, Robert">
      </actor>
    </character>
    ...
  </movie>
  ...
</movies>
```

Note the following features of XML:

- Every tag `<a>` has a corresponding closing tag `</a>`. You can always recognize a closing tag by the forward slash `/`.
- Additional tags and/or strings can be nested between the opening and closing tags. In the example above, `<actor>` is nested between `<character>` and `</character>`, and `<character>` is nested between `<movie>` and `</movie>`. The nesting is used to represent hierarchy.
- Indentation is used to make the code more readable (to make it easier to see the nesting structure). But it is optional.
- Attributes can be associated with each tag, like `id=` and `name=` with the `<character>` tag and `id=` and `title=` with the `<movie>` tag.

Each tag represents a variable in the data set. Unlike JSON, which uses lists to represent repeated fields, XML represents repeated fields by simply repeating tags where necessary. In the example above, there are multiple instances of `<movie>` within `<movies>` and multiple instances of `<character>` within `<movie>`, so `movie` and `character` are both repeated fields. (In fact, `director` is also a repeated field, but it is impossible to tell from the code above, since the movies shown above only have one director.)

You will learn XML by working with the same New York Philharmonic data as in the previous section, except that the data is now stored in XML format. Let's look at this file on disk.

```
In [1]: !ls -l /data301/data/nyphil/
```

```
total 78912
-rw-r--r-- 1 root root 35284693 Feb 25 08:35 complete.json
-rw-r--r-- 1 root root 45514892 Feb 25 08:36 complete.xml
```

Notice that this XML file is nearly twice as large as the JSON file. Although XML is more readable than JSON, it is a more expensive way to store hierarchical data, primarily because of the cost of storing both the opening and closing tags.

There are several libraries in Python for working with XML, including BeautifulSoup (which we will use in the next section to parse HTML), ElementTree, and `lxml`. We will use `lxml` to work with XML data because it is fastest for large data sets, provided that the data is well-formed. The `lxml` library provides a convenient API that replicates all of the functionality of ElementTree, plus implements a few additional features that are useful for data analysis.

```
In [2]: from lxml import etree
```

First, let's read in the data using `lxml`. The `.parse()` function of ElementTree reads in an XML document from a file or URL and returns a tree-like data structure called an ElementTree.

```
In [3]: tree = etree.parse("/data301/data/nyphil/complete.xml")
```

Every XML document has a single "root" tag that encloses all of the other tags. For the New York Philharmonic data, this root tag is `<programs>`.

```
In [4]: tree.getroot()
```

```
Out[4]: <Element programs at 0x7f4a50d5dc48>
```

If the XML data is already stored as a string in memory, then we instead use the `.fromstring()` method. Note that `.fromstring()` returns the root tag directly.

```
In [5]: with open("/data301/data/nyphil/complete.xml", "rb") as f:
            string = f.read()

        etree.fromstring(string)
```

```
Out[5]: <Element programs at 0x7f4a50060ac8>
```

Each direct descendant, or **child**, of `<programs>` is a program. To find the direct descendants of a tag, we call the `.getchildren()` method.

```
In [6]: programs = tree.getroot()
        print(len(programs.getchildren()))
        programs.getchildren()[:10]
```

```
14009
```

```
Out[6]: [<Element program at 0x7f4a39c0c688>,
         <Element program at 0x7f4a39c0c708>,
         <Element program at 0x7f4a39c0c108>,
         <Element program at 0x7f4a39c0c908>,
         <Element program at 0x7f4a39c0c948>,
         <Element program at 0x7f4a39c0ca08>,
         <Element program at 0x7f4a39c0ca48>,
         <Element program at 0x7f4a39c0ca88>,
         <Element program at 0x7f4a39c0c988>,
         <Element program at 0x7f4a39c0c9c8>]
```

Let's print out the first of these programs. There are two ways to get the first program.

```
In [7]: # METHOD 1: Get it from the list above.
        program = programs.getchildren()[0]

        # METHOD 2: Use .find() to find the first instance of a tag.
        program = tree.find("program")
        program
```

```
Out[7]: <Element program at 0x7f4a39c0c688>
```

Now let's see how the data is represented by printing out the XML of this program. To do this, we use the `etree.tostring()` function.

```
In [8]: print(etree.tostring(program, encoding="unicode"))
```

```
<program>
        <id>00646b9f-fec7-4ffb-9fb1-faae410bd9dc-0.1</id>
        <programID>3853</programID>
        <orchestra>New York Philharmonic</orchestra>
        <season>1842-43</season>
        <concertInfo>
            <eventType>Subscription Season</eventType>
            <Location>Manhattan, NY</Location>
            <Venue>Apollo Rooms</Venue>
            <Date>1842-12-07T05:00:00Z</Date>
            <Time>8:00PM</Time>
        </concertInfo>
        <worksInfo>
            <work ID="52446*">
                <composerName>Beethoven,  Ludwig  van</composerName>
                <workTitle>SYMPHONY NO. 5 IN C MINOR, OP.67</workTitle>
                <conductorName>Hill, Ureli Corelli</conductorName>
            </work>
            <work ID="8834*4">
                <composerName>Weber,  Carl  Maria Von</composerName>
                <workTitle>OBERON</workTitle>
                <movement>"Ozean, du Ungeheuer" (Ocean, thou mighty monster), Reiza (Scene and
                <conductorName>Timm, Henry C.</conductorName>
                <soloists>
                    <soloist>
                        <soloistName>Otto, Antoinette</soloistName>
                        <soloistInstrument>Soprano</soloistInstrument>
                        <soloistRoles>S</soloistRoles>
                    </soloist>
                </soloists>
            </work>
            <work ID="3642*">
                <composerName>Hummel,  Johann</composerName>
                <workTitle>QUINTET, PIANO, D MINOR, OP. 74</workTitle>
                <soloists>
                    <soloist>
                        <soloistName>Scharfenberg, William</soloistName>
                        <soloistInstrument>Piano</soloistInstrument>
                        <soloistRoles>A</soloistRoles>
                    </soloist>
                    <soloist>
                        <soloistName>Hill, Ureli Corelli</soloistName>
                        <soloistInstrument>Violin</soloistInstrument>
                        <soloistRoles>A</soloistRoles>
                    </soloist>
                    <soloist>
                        <soloistName>Derwort, G. H.</soloistName>
                        <soloistInstrument>Viola</soloistInstrument>
            <soloistRoles>A</soloistRoles>
        </soloist>
        <soloist>
            <soloistName>Boucher, Alfred</soloistName>
            <soloistInstrument>Cello</soloistInstrument>
            <soloistRoles>A</soloistRoles>
        </soloist>
        <soloist>
            <soloistName>Rosier, F. W.</soloistName>
            <soloistInstrument>Double Bass</soloistInstrument>
            <soloistRoles>A</soloistRoles>
        </soloist>
    </soloists>
</work>
<work ID="0*">
    <interval>Intermission</interval>
</work>
<work ID="8834*3">
    <composerName>Weber,  Carl  Maria Von</composerName>
    <workTitle>OBERON</workTitle>
    <movement>Overture</movement>
    <conductorName>Etienne, Denis G.</conductorName>
</work>
<work ID="8835*1">
    <composerName>Rossini,  Gioachino</composerName>
    <workTitle>ARMIDA</workTitle>
    <movement>Duet</movement>
    <conductorName>Timm, Henry C.</conductorName>
    <soloists>
        <soloist>
            <soloistName>Otto, Antoinette</soloistName>
            <soloistInstrument>Soprano</soloistInstrument>
            <soloistRoles>S</soloistRoles>
        </soloist>
        <soloist>
            <soloistName>Horn, Charles Edward</soloistName>
            <soloistInstrument>Tenor</soloistInstrument>
            <soloistRoles>S</soloistRoles>
        </soloist>
    </soloists>
</work>
<work ID="8837*6">
    <composerName>Beethoven,  Ludwig  van</composerName>
    <workTitle>FIDELIO, OP. 72</workTitle>
    <movement>"In Des Lebens Fruhlingstagen...O spur ich nicht linde," Florestan (a
    <conductorName>Timm, Henry C.</conductorName>
    <soloists>
        <soloist>
```

```
                    <soloistName>Horn, Charles Edward</soloistName>
                    <soloistInstrument>Tenor</soloistInstrument>
                    <soloistRoles>S</soloistRoles>
                </soloist>
            </soloists>
        </work>
        <work ID="8336*4">
            <composerName>Mozart,  Wolfgang  Amadeus</composerName>
            <workTitle>ABDUCTION FROM THE SERAGLIO,THE, K.384</workTitle>
            <movement>"Ach Ich liebte," Konstanze (aria)</movement>
            <conductorName>Timm, Henry C.</conductorName>
            <soloists>
                <soloist>
                    <soloistName>Otto, Antoinette</soloistName>
                    <soloistInstrument>Soprano</soloistInstrument>
                    <soloistRoles>S</soloistRoles>
                </soloist>
            </soloists>
        </work>
        <work ID="5543*">
            <composerName>Kalliwoda,  Johann  W.</composerName>
            <workTitle>OVERTURE NO. 1, D MINOR, OP. 38</workTitle>
            <conductorName>Timm, Henry C.</conductorName>
        </work>
    </worksInfo>
</program>
```

Hopefully, the basic structure of this data is already familiar to you from previous section. “Work”, “concertInfo”, and “soloist” are repeated fields inside “program”. One difference between the JSON and the XML is that “work” is not directly nested within “program”; the “work” tags are all nested inside an additional “worksInfo” tag.

Now suppose that we want to flatten the data at the level of soloists. To get all of the soloists, we can use the `.findall()` method. Let’s first try the obvious solution, which does not work:

```
In [9]: programs.findall("soloist")
```

```
Out[9]: []
```

Why did `lxml` fail to find any `<soloist>` tags? That’s because `.findall()` only searches among the direct descendants of a tag. We called `.findall()` on the `<programs>` tag, but all of its descendants are `<program>` tags.

To specify that `lxml` should look for `<soloist>` tags among all descendants, not just direct ones, we use the `.xpath()` command, which allows us to specify an XPath expression. XPath is a language used to select nodes from XML documents. The XPath expression to select all descendants named `<soloist>` of the current tag is `".//soloist"`. We pass this expression to the `.xpath()` method.

```
In [10]: soloists = programs.xpath(".//soloist")
         len(soloists)
```

```
Out[10]: 56931
```

Now, to flatten the data at the level of soloists, we just need to turn `soloists` into a `DataFrame` with as many rows. But what if we want to include information from parent levels, like the composer of the work the soloist played? There are two ways.

1.0.1 Method 1

Since `<composerName>` is a descendant of `<work>`, one way is to navigate up to the level of `<work>` by calling `.getparent()` repeatedly and then find `<composerName>` among its descendants:

```
In [11]: soloist = soloists[0]

         # The first .getparent() returns the <soloists> tag.
         # The second .getparent() returns the <work> tag.
         # You have to figure this out by inspecting the XML.
         work = soloist.getparent().getparent()
         work.xpath(".//composerName")
```

```
Out[11]: [<Element composerName at 0x7f4a50d5de88>]
```

This is a list with one tag, so we extract that tag and the text inside it.

```
In [12]: work.xpath(".//composerName")[0].text
```

```
Out[12]: 'Weber,  Carl  Maria Von'
```

1.0.2 Method 2

As the number of levels of nesting increases, it quickly becomes impractical to call `.getparent()` repeatedly. We want to be able to jump directly to the right ancestor. The easiest way to do this is to use the XPath expression for an ancestor. To search for all ancestors named "work", we can use the XPath expression `"ancestor::work"`.

```
In [13]: soloist.xpath("ancestor::work")
```

```
Out[13]: [<Element work at 0x7f4a398a0a88>]
```

Now, we can extract this single work tag and find its descendants named `<composerName>`. Or better yet, we can combine this step with the above step into a single XPath expression.

```
In [14]: soloist.xpath("ancestor::work//composerName")[0].text
```

```
Out[14]: 'Weber,  Carl  Maria Von'
```

Now let's put it all together. We will flatten the data to get a `DataFrame` with one soloist per row. We will keep track of the soloist's name, instrument, and role—as well as the composer of the work they performed. Unfortunately, it is much more manual to do this with XML than with JSON. There is no XML equivalent of the `json_normalize` function that will automatically produce a `DataFrame`, so we have to construct the `DataFrame` ourselves.

```
In [15]: import pandas as pd

         rows = []

         soloists = programs.xpath(".//soloist")
         for soloist in soloists:
             row = {}
             row["soloistName"] = soloist.find("soloistName").text
             row["soloistInstrument"] = soloist.find("soloistInstrument").text
             row["soloistRoles"] = soloist.find("soloistRoles").text
             row["composerName"] = soloist.xpath("ancestor::work//composerName")[0].text
             rows.append(row)

         soloistsdf = pd.DataFrame(rows)
         soloistsdf
```

```
Out[15]:                      composerName soloistInstrument  \
         0         Weber,  Carl  Maria Von           Soprano
         1                 Hummel,  Johann             Piano
         2                 Hummel,  Johann            Violin
         3                 Hummel,  Johann             Viola
         4                 Hummel,  Johann             Cello
         5                 Hummel,  Johann       Double Bass
         6             Rossini,  Gioachino           Soprano
         7             Rossini,  Gioachino             Tenor
         8         Beethoven,  Ludwig  van             Tenor
         9      Mozart,  Wolfgang  Amadeus           Soprano
         10             Bellini,  Vincenzo           Soprano
         11             Romberg,  Bernhard             Cello
         12            Rossini,  Gioachino           Soprano
         13                Hummel,  Johann             Piano
         14                Hummel,  Johann             Piano
         15        Weber,  Carl  Maria Von           Soprano
         16        Weber,  Carl  Maria Von             Piano
         17                Hummel,  Johann             Piano
         18              Pacini,  Giovanni           Soprano
         19              Pacini,  Giovanni             Piano
         20             Romberg,  Bernhard             Cello
         21                Onslow,  George             Piano
         22                Onslow,  George             Flute
         23                Onslow,  George          Clarinet
         24                Onslow,  George           Bassoon
         25                Onslow,  George       French Horn
         26                Onslow,  George       Double Bass
         27                Onslow,  George              None
         28                Onslow,  George             Piano
         29                Onslow,  George             Flute
         ...                           ...               ...
```

```
56901              Klein,  Gideon            Violin
56902              Klein,  Gideon             Viola
56903              Klein,  Gideon             Cello
56904     Beethoven,  Ludwig  van              Oboe
56905     Beethoven,  Ludwig  van              Oboe
56906     Beethoven,  Ludwig  van          Clarinet
56907     Beethoven,  Ludwig  van          Clarinet
56908     Beethoven,  Ludwig  van           Bassoon
56909     Beethoven,  Ludwig  van           Bassoon
56910     Beethoven,  Ludwig  van       French Horn
56911     Beethoven,  Ludwig  van       French Horn
56912       Shostakovich,  Dmitri            Violin
56913       Shostakovich,  Dmitri            Violin
56914       Shostakovich,  Dmitri             Viola
56915       Shostakovich,  Dmitri             Cello
56916       Shostakovich,  Dmitri             Piano
56917  Mozart,  Wolfgang  Amadeus              Oboe
56918  Mozart,  Wolfgang  Amadeus          Clarinet
56919  Mozart,  Wolfgang  Amadeus           Bassoon
56920  Mozart,  Wolfgang  Amadeus       French Horn
56921   Handel,  George  Frideric           Soprano
56922   Handel,  George  Frideric     Mezzo-Soprano
56923   Handel,  George  Frideric             Tenor
56924   Handel,  George  Frideric     Bass-Baritone
56925   Handel,  George  Frideric            Chorus
56926   Handel,  George  Frideric           Soprano
56927   Handel,  George  Frideric     Mezzo-Soprano
56928   Handel,  George  Frideric             Tenor
56929   Handel,  George  Frideric          Baritone
56930   Handel,  George  Frideric            Chorus

                                soloistName soloistRoles
0                          Otto, Antoinette            S
1                     Scharfenberg, William            A
2                       Hill, Ureli Corelli            A
3                            Derwort, G. H.            A
4                           Boucher, Alfred            A
5                             Rosier, F. W.            A
6                          Otto, Antoinette            S
7                      Horn, Charles Edward            S
8                      Horn, Charles Edward            S
9                          Otto, Antoinette            S
10                         Otto, Antoinette            S
11                          Boucher, Alfred            S
12                         Otto, Antoinette            S
13                           Timm, Henry C.            S
14                           Timm, Henry C.            S
15                         Otto, Antoinette            S
```

```
16                       Timm, Henry C.            A
17                Scharfenberg, William            S
18                     Otto, Antoinette            S
19                       Timm, Henry C.            A
20                      Boucher, Alfred            S
21                Scharfenberg, William            S
22                               Lehman            A
23              Groneveldt, Theodore W.            A
24                      Hegelund, H. W.            A
25                      Woehning, F. C.            A
26                        Rosier, F. W.            A
27                                 None         None
28                Scharfenberg, William            S
29                               Lehman            A
...                                 ...          ...
56901                          Ge, Quan            A
56902                    Young, Rebecca            A
56903         Gonzales, Alexei Yupanqui            A
56904                     Sylar, Sherry            A
56905                     Botti, Robert            A
56906  Martinez [Martínez] Forteza, Pascual            A
56907                       Zoloto, Amy            A
56908                    Laskowski, Kim            A
56909                       Fast, Arlen            A
56910                    Deane, Richard            A
56911                 Spanjer, R. Allen            A
56912                     Yao, Shanshan            A
56913                    Rossano, Marié            A
56914                     Kenote, Peter            A
56915                         Tu, Qiang            A
56916                  Wolfram, William            A
56917                       Wang, Liang            S
56918                   McGill, Anthony            S
56919                   LeClair, Judith            S
56920                    Deane, Richard            S
56921           Harvey, Joelle [Joélle]            S
56922            Johnson Cano, Jennifer            S
56923                        Bliss, Ben            S
56924            Foster-Williams, Andrew            S
56925       Westminster Symphonic Choir            S
56926           Harvey, Joelle [Joélle]            S
56927            Johnson Cano, Jennifer            S
56928                        Bliss, Ben            S
56929                     Duncan, Tyler            S
56930       Westminster Symphonic Choir            S

[56931 rows x 4 columns]
```

Now, this is a DataFrame that we can analyze easily. For example, here is how many times Benny Goodman programmed a work by Mozart with the NY Phil:

```
In [16]: soloistsdf[soloistsdf["soloistName"] == "Goodman, Benny"].composerName.value_counts()
```

```
Out[16]: Mozart,  Wolfgang  Amadeus      3
         Weber,  Carl  Maria Von         3
         Gershwin,  George               2
         Sauter,  Eddie                  2
         Basie,  Count                   1
         Williams,  Mary Lou             1
         Prima,  Louis                   1
         Youmans,  Vincent               1
         Copland,  Aaron                 1
         Unspecified,                    1
         Green,  Johnny                  1
         Debussy,  Claude                1
         Baxter,  Phil                   1
         Cannon,  Hughie                 1
         Confrey,  Zez                   1
         Anthem,                         1
         Ellington,  Duke                1
         Sampson,  Edgar                 1
         Handy,  William  Christopher    1
         Name: composerName, dtype: int64
```

2 RESTful Web Services

Many RESTful web services return data in XML format. Like before, we use the requests library in Python to issue the HTTP request. For example, the website FloatRates provides exchange rates between world currencies in XML format.

```
In [17]: import requests
         resp = requests.get("http://www.floatrates.com/daily/usd.xml")
         resp
```

```
Out[17]: <Response [200]>
```

The XML is stored in the .content attribute of the response object. We can parse this string into an ElementTree using the .fromstring() function in the lxml library. Recall that this returns the root tag of the XML document.

```
In [18]: etree.fromstring(resp.content)
```

```
Out[18]: <Element channel at 0x7f4a3989f848>
```

3 Exercises

Exercises 1 and 2 deal with the New York Philharmonic data set from above. These exercises are the same as the ones in the previous section, except that now you have to do them with XML.

Exercise 1. What is the most frequent start time for New York Philharmonic concerts?

```
In [19]: rows = []


         concerts = programs.findall(".//concertInfo")
         for concert in concerts:
             row = {}
             row["Time"] = concert.find("Time").text
             row["Season"] = concert.xpath("ancestor::program//season")[0].text
             rows.append(row)

         concertsdf = pd.DataFrame(rows)
         concertsdf.Time.value_counts().head()

Out[19]: 8:30PM    4584
         8:00PM    4443
         3:00PM    2133
         7:30PM    2075
         2:30PM    1618
         Name: Time, dtype: int64
```

Exercise 2. How many total concerts did the New York Philharmonic perform in the 2014-15 season?

```
In [20]: len(concertsdf[concertsdf.Season == "2014-15"])

Out[20]: 217
```

In Exercises 3-4, you will work with APIXU, an weather API. This API returns data in both JSON and XML formats. In these exercises, you should request the data to be returned in XML format.

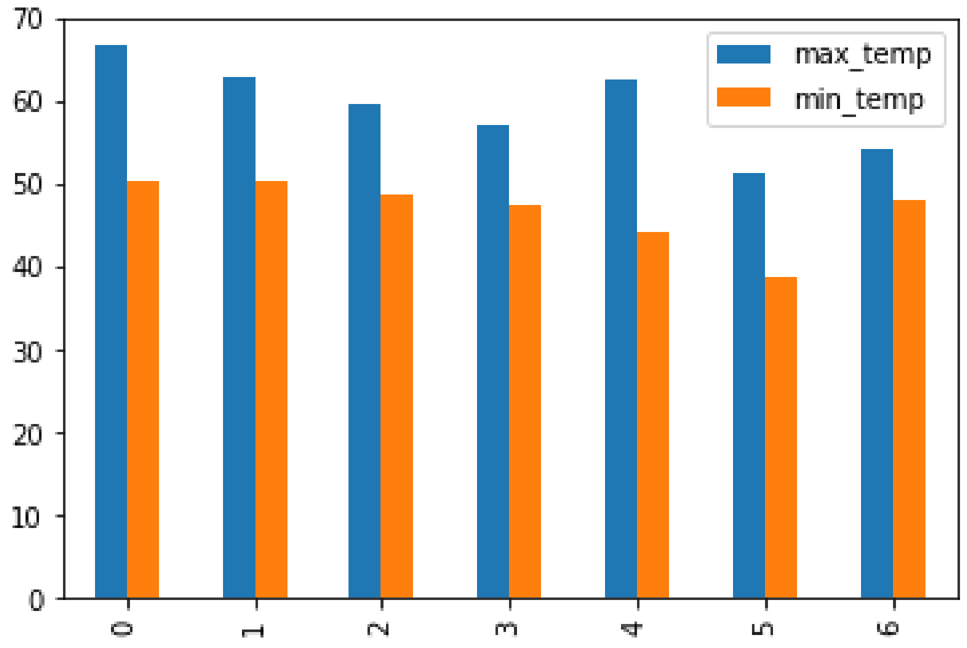
Register with the website to obtain an API key. You will likely need to refer to the API documentation here. If you run into unexpected errors, issue the HTTP request from your browser to make sure that the data is in the format you expect.

Exercise 3. Get the forecasted low (min) and high (max) temperatures (in Fahrenheit) for the next 7 days in San Luis Obispo. Make a graphic that displays this information.

```
In [21]: apikey = "be1c5704a83c459ea4f162211192802"
         resp = requests.get(
                 "https://api.apixu.com/v1/forecast.xml?q=San Luis Obispo&days=7&key=%s" %apik

In [22]: wea = etree.fromstring(resp.content)
         %matplotlib inline
```

```
In [23]: rows = []

         forecastdays = wea.xpath(".//forecastday")
         for forecastday in forecastdays:
             row = {}
             row["min_temp"] = pd.to_numeric(forecastday.xpath(".//mintemp_f")[0].text)
             row["max_temp"] = pd.to_numeric(forecastday.xpath(".//maxtemp_f")[0].text)
             rows.append(row)

         forecastdf = pd.DataFrame(rows)
         forecastdf.plot.bar()
```

```
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4a1ccad940>
```



Exercise 4. Get the hourly wind speed (in mph) for the past 7 days. (*Note:* This will require making 7 HTTP requests to the API. Try to do it programmatically.) You should end up with $24 \times 7 = 168$ rows in your `DataFrame`. Make a plot of the wind speed as a function of time. What do you notice?

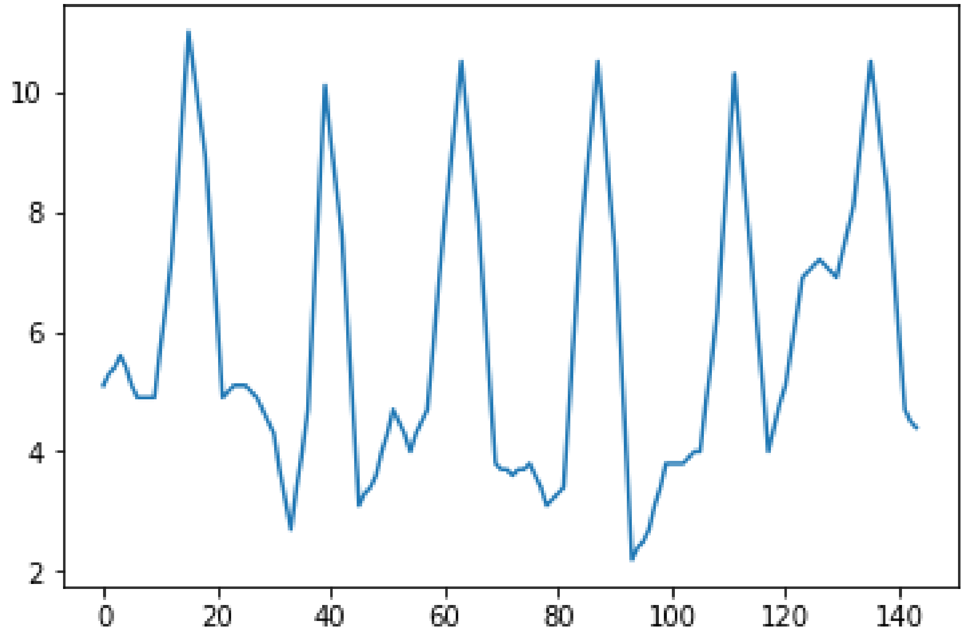
```
In [27]: apikey = "be1c5704a83c459ea4f162211192802"
         resp = requests.get(
             "https://api.apixu.com/v1/history.xml?q=San Luis Obispo&dt=2019-02-27&key=%s"
```

```
In [34]: root = etree.fromstring(resp.content)
         #print(etree.tostring(root, pretty_print=True).decode())
```

```
In [35]: import time
         len(root.xpath(".//hour"))
```

```
rows = []
for day in range(21,28):
    date = "2019-02-%d" % day

    resp = requests.get(
        "https://api.apixu.com/v1/history.xml?q=San Luis Obispo&dt=%s&key=%s" % (date

    root = etree.fromstring(resp.content)

    for hour in root.xpath(".//hour"):
        rows.append(float(hour.xpath(".//wind_mph")[0].text))

    time.sleep(0.1)

pd.Series(rows).plot.line()
```

```
Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4a1be1eb38>
```



```
In [ ]: hour = root.xpath(".//hour")
```