Web Scraping

Part \#3: XML and Semi-Structured Data

Where do data scientists get their data?

A data scientist needs sources for data to do his or her work. While you don't need this level of detail on the AP test, here are is some information about the types of data available to data scientists:

- **Unstructured data:** Most data available on the web is unstructured data. Image files, sound files, video files, text files, and HTML files are all examples of unstructured data. These are some of the richest sources of data available, but they are also difficult to process (search, sort, classify, analyze, summarize etc). For example, while you were easily able to write algorithms in a previous chapter to apply image filters to the pixels of an image, it is very difficult to write functions to classify the objects featured in image files (people, trees, dogs, cats, mountains, etc). Similarly, while a web browser can effectively display an HTML file, it is not easy to write an algorithm to give a short summary of the content of the webpage. You can certainly extract useful data from an HTML file, but you just saw that it takes a great deal of effort using a module such as **Beautiful Soup**. In fact, the name of the module describes the World Wide Web: it is a beautiful soup of unstructured data. Data scientists often spend a great deal of time finding their data. To read more, here's a great blog post on the topic: https://www.dataquest.io/blog/web-scraping-beautifulsoup/
- **Semi-structured data:** You already have experience with semi-structured data: a *.csv* file is an example of semi-structured data. This is data that supports automated processing of its contents, such as we saw with Pandas during the our chapter on Open Data. In other courses, you may learn about JSON, and in this notebook you will learn about XML. As you will soon see, the beauty of XML is that you can work with data in an automated way.
- **Structured data:** This is data that is stored in a *database*. Organizations such as corporations, governments, and universities will have servers dedicated to their databases and database software. The data stored in a database is similar to what you have seen in *.csv* files, but has some additional structure. We will not work with databases (structured data) in this class, but they are a great source of information. But if you want to learn more, you may want look up the term *relational database* or *SQL*.

What is XML?

XML stands for eXtensible Markup Language. While HTML is meant to display webpages, XML is meant to store/transport/describe data. Humans can read and understand XML, and computers can also process XML in an automated way.

Task \#1: Read the following pages from W3Schools, then answer the questions below:

- XML Introduction: https://www.w3schools.com/xml/xml_whatis.asp
- XML Tutorial: https://www.w3schools.com/xml/default.asp

Question \#1: What does the "extensible" in "extensible markup language" mean?

Your Answer: Most XML applications will work as expected even if new data is added or removed.

Question \#2: Suppose you are trying to write an algorithm to process data in an automated way. Why would you prefer for your algorithm to work with *extensible* data?

Your Answer: You can easily add and remove data.

Chicago Weather Data

Before you read any further, visit the page http://w1.weather.gov/xml/current_obs/KORD.xml and look at the data on the page. This is a feed of current weather conditions at Chicago's O'Hare airport. You can use this XML document to create a webpage or app that always knows the most up-to-date weather in Chicago.

Let's take a look at the XML source for this feed:

In [1]:
```
from bs4 import BeautifulSoup       # Import BeaurtifulSoup
from urllib.request import urlopen  # Import urlopen

xml_page = urlopen("http://w1.weather.gov/xml/current_obs/KORD.xml")   # Opens whatever page we are requesting
bs_obj = BeautifulSoup(xml_page, 'xml')  # Extracts the xml data

print(bs_obj.prettify()[:1000])  # Makes it more easily readible or 'pretty'
```

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet href="latest_ob.xsl" type="text/xsl"?>
<current_observation version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://www.weather.gov/view/current_observation.xsd">
 <credit>
  NOAA's National Weather Service
 </credit>
 <credit_URL>
  https://weather.gov/
 </credit_URL>
 <image>
  <url>
   https://weather.gov/images/xml_logo.gif
  </url>
  <title>
   NOAA's National Weather Service
  </title>
  <link>
   https://www.weather.gov
  </link>
 </image>
 <suggested_pickup>
  15 minutes after the hour
 </suggested_pickup>
 <suggested_pickup_period>
  60
 </suggested_pickup_period>
 <location>
  Chicago, Chicago-O'Hare International Airport, IL
 </location>
 <station_id>
  KORD
 </station_id>
 <latitude>
  41.97972
 </latitude>
 <longitude>
  -87.90444
 </longitude>
 <observation_time>
  Last Updated on May 23 2022, 10:51 pm CDT
 </observation_ti
```

If you want to extract the current temperature from this data, just run the code below:

In [2]:
```
current_temp = bs_obj.find('temp_f').getText()  # Grabs the text found in the inner HTML of the <temp_f> tag
print(current_temp)
```

```
51.0
```

Question \#3: What will you probably notice about the output from the cell directly above if you were to re-run the code in this notebook at a later date?

Your Answer: The temp will change.

Question \#4: Where was this temperature measurement taken? To answer this question, you must write code that uses the latitude and longitude data in the XML above to create a tuple. Give your answer as a tuple in the form (latitude,longitude)

Location Tuple: ('41.97972', '-87.90444')

In [3]:
```
lat = float(bs_obj.find('latitude').getText())
long = float(bs_obj.find('longitude').getText())
lat, long
```

Out[3]:
```
(41.97972, -87.90444)
```

Task \#2 Use the Google Maps API to create a marker map with the location of the coordinates you found in the previous question.

- *Hint: Look back at Metadata Part 4 to see how we displayed GPS coordinates*

In [4]:
```
# Import the gmaps python module and load in your API Key:
import gmaps
gmaps.configure(api_key="AIzaSyCLla6Q7krE9xNg6SnNMoGNIzjCLddE9EU")
```

In [5]:
```
from ipywidgets.embed import embed_minimal_html # Allows us to create a separte file for the Google Maps

markers = gmaps.marker_layer([(lat, long)])    # Create markers for each tuple/coordinate
markermap = gmaps.Map()                         # Create a GMap variable
markermap.add_layer(markers)                    # Add the layer of markers to GMap

embed_minimal_html('output/MarkerMap2.html', views=[markermap])
print("*** If no map appears, uncomment the line above, re-run this cell, and check your 'Metadata Part 5' folder to find the new HTML file name \"MarkerMap1.html\". ***")

markermap
```

```
*** If no map appears, uncomment the line above, re-run this cell, and check your 'Metadata Part 5' folder to find the new HTML file name "MarkerMap1.html". ***
Map(configuration={'api_key': 'AIzaSyCLla6Q7krE9xNg6SnNMoGNIzjCLddE9EU'}, data_bounds=[(41.97971, -87.90445), …
```

Question \#5: What does the following function, *tag_extractor(url, tag)*, do? Some structure is provided below to help you answer this question:

Your Answers (Below):

- *What is the purpose of the function?*
  - Your answer: It finds the the specific tag in a xml file and return the inner html of the tag.
- *What purpose does the parameter **url** serve in the function?*
  - Your answer: It is the link to the xml file.
- *What purpose does the parameter **tag** serve in the function?*
  - Your answer: It looks for that specific tag in the xml file.
- *What information is being returned by the function?*
  - Your answer: It return the inner html of the specific tag in the xml file.

In [6]:
```
def tag_extractor(url, tag):
    from bs4 import BeautifulSoup
    from urllib.request import urlopen

    xml_page = urlopen(url)   #opens whatever page we are requesting
    bs_obj = BeautifulSoup(xml_page, 'xml')

    return bs_obj.find(tag).getText()

tag_extractor('http://w1.weather.gov/xml/current_obs/KORD.xml', 'temp_f')
```

Out[6]:
```
'51.0'
```

Question \#6: How can *tag_extractor(url, tag)* be considered an abstraction that helps to manage the complexity of a computer program?

Your Answer: Instead of repeatedly opening different xml files and writing the same code to get the same tags you can use an abstraction like above to make it much easier.

Task \#3: Use *tag_extractor(url, tag)* to determine the date/time of the most recent temperature measurement.

In [7]:
```
tag_extractor('http://w1.weather.gov/xml/current_obs/KORD.xml', 'observation_time')
```

Out[7]:
```
'Last Updated on May 23 2022, 10:51 pm CDT'
```

HTML as Output

Question \#7: What is the purpose of the function *html_output()*? What kind of data does html_output() produce as output?

Your Answer: It creates a html file that has the current weather from the xml file.

In [8]:
```
# define the function:
def html_output():
    output_string = """
    <html>
    <head>
        <style>
            body {
                background-color: #BBBBBB;
                text-align: center;
            }
        </style>
    </head>

    <body>
    <h1>Chicago Weather</h1>
    <p> The current temperature in Chicago is
    """

    output_string += tag_extractor('http://w1.weather.gov/xml/current_obs/KORD.xml', 'temp_f')

    output_string += """
    degrees Fahrenheit.
    </p>
    <br>

    </body>
    </html>
    """

    html_file= open("output/O'Hare Temperature.html","w")
    html_file.write(output_string)
    html_file.close()

# now call the function:

html_output()
print("*** Look in the 'Output' folder of 'Web Scraping Part 3' to find the new HTML file. ***")
```

```
*** Look in the 'Output' folder of 'Web Scraping Part 3' to find the new HTML file. ***
```

Task \#4: Create your own version of *html_output()* that includes the date/time of the most recent temperature measurement as well as two other measurements to your output *.html* document.

In [9]:
```
# define the function:
def html_output():
    output_string = """
    <html>
    <head>
        <style>
            body {
                background-color: #BBBBBB;
                text-align: center;
            }
        </style>
        <meta http-equiv="refresh" content="3600">
    </head>

    <body>
    <h1>Chicago Weather</h1>
    <p> The current temperature in Chicago is
    """

    output_string += tag_extractor('http://w1.weather.gov/xml/current_obs/KORD.xml', 'temp_f')
    output_string += """
    degrees Fahrenheit. <br>"""

    output_string += tag_extractor('http://w1.weather.gov/xml/current_obs/KORD.xml', 'observation_time') + '<br>'

    output_string += '(Lat, Long): ' + bs_obj.find('latitude').getText() + ',' + bs_obj.find('longitude').getText()

    output_string += """
    </p>
    <br>

    </body>
    </html>
    """

    html_file= open("output/O'Hare Temperature2.html","w")
    html_file.write(output_string)
    html_file.close()

# now call the function:

html_output()
print("*** Look in the 'Output' folder of 'Web Scraping Part 3' to find the new HTML file. ***")
```

```
*** Look in the 'Output' folder of 'Web Scraping Part 3' to find the new HTML file. ***
```

Task \#5: This is a multistep task:

1) Read this w3schools documentation

2) Add the HTML <meta http-equiv="refresh" content="3600"> between the head tags in your html_output() function

3) Read (but do not run) the code in the cell below:

In [10]:
```
running = False
import time


while running:
    html_output()
    time.sleep(3600)
```

Question \#9: If you were to change *running* to *True* in the code cell above, then running this cell would start an infinite loop. What would the purpose be of an infinite loop and the HTML above, <meta http-equiv="refresh" content="3600"> ? What would it allow you to do?

Your Answer: It would refresh the page with new updated data every 3600 seconds.

Experiment and Explore

Task \#6: With any extra time this period, go to http://w1.weather.gov/xml/current_obs/KORD.xml, but change the **KORD** portion of this URL to another ICAO airport code for an airport in the United States or its territories. Experiment. Show the results of your experimentation in an HTML output file, and produce a Google Maps Marker Map in this notebook that includes the location you chose to explore.

- Airport codes in the lower-48 states: https://en.wikipedia.org/wiki/List_of_airports_by_ICAO_code:_K
- All airport codes in the United States and territories: https://en.wikipedia.org/wiki/List_of_airports_in_the_United_States#Lists_by_ICAO_location_indicator

In [43]:
```
def get_name_of_every_airport():
    html_page = urlopen('https://en.wikipedia.org/wiki/List_of_airports_in_the_United_States#Lists_by_ICAO_location_indicator')                    # Opens whatever page we are requesting
    bs_obj = BeautifulSoup(html_page, 'html.parser')                              # Saves the html in a Beautiful Soup object
    try:
        url = bs_obj.findAll('tbody')[2].findAll("tr",{"valign":"top"})
        all_loc = []
        for i in url:
            td = i.findAll('td')[3].string
            all_loc.append(td[:-1])
    except AttributeError:
        all_loc = False                                                          # If no URL is available, store 'False' instead
    return all_loc
get_location_of_every_airport()[:10]
```

Out[43]:
```
['KBHM',
 'KDHN',
 'KHSV',
 'KMOB',
 'KMGM',
 'PALH',
 'PAMR',
 'PANC',
 'PANI',
 'PABE']
```

In [92]:
```
def get_loc_of_every_airport(how_many):
    loc = []
    all_loc = get_location_of_every_airport()
    j=0
    for i in all_loc:
        latitude = tag_extractor(f'http://w1.weather.gov/xml/current_obs/{i}.xml', 'latitude')
        longitude = tag_extractor(f'http://w1.weather.gov/xml/current_obs/{i}.xml', 'longitude')
        loc.append((float(latitude),float(longitude)))
        if j == how_many: return loc
        j += 1
    return loc
```

In [93]:
```
final_loc = get_loc_of_every_airport(100)
final_loc
```

Out[93]:
```
[(33.56556, -86.745),
 (31.32139, -85.44972),
 (34.64361, -86.78556),
 (30.68833, -88.24556),
 (32.30028, -86.40611),
 (61.18333, -149.96667),
 (61.21667, -149.85),
 (61.17444, -149.99611),
 (61.58139, -159.54278),
 (60.77972, -161.83778),
 (60.49167, -145.47778),
 (70.2, -148.46667),
 (59.05, -158.51667),
 (64.80389, -147.87611),
 (58.41667, -135.7),
 (59.65, -151.48333),
 (58.35472, -134.57611),
 (60.57306, -151.245),
 (55.35556, -131.71361),
 (58.67667, -156.64917),
 (55.5839, -133.067),
 (57.75, -152.5),
 (66.88576, -162.60624),
 (64.51194, -165.445),
 (56.8017, -132.9453),
 (62.05, -163.3),
 (57.048, -135.3647),
 (63.88333, -160.8),
 (53.9, -166.53333),
 (71.28528, -156.76583),
 (61.13333, -146.26667),
 (56.48333, -132.36667),
 (59.51667, -139.66667),
 (35.1575, -114.55944),
 (35.14433, -111.66637),
 (35.94582, -112.15538),
 (33.31667, -111.65),
 (36.92056, -111.44806),
 (33.427799, -112.003465),
 (34.64917, -112.42222),
 (32.13153, -110.95635),
 (32.65944, -114.59306),
 (36.28977, -94.31145),
 (35.3335, -94.36524),
 (34.72667, -92.23611),
 (33.45611, -93.9875),
 (40.97806, -124.10861),
 (35.43361, -119.05667),
 (34.19967, -118.36538),
 (37.99167, -122.05194),
 (36.78, -119.71944),
 (33.81167, -118.14639),
 (33.93806, -118.38889),
 (37.6241, -118.8423),
 (36.59047, -121.84875),
 (37.7178, -122.23294),
 (34.05316, -117.57685),
 (33.6798, -117.8674),
 (33.82219, -116.50431),
 (40.51461, -122.2977),
 (38.70071, -121.59479),
 (32.73361, -117.18306),
 (37.61961, -122.36558),
 (37.35917, -121.92417),
 (35.23611, -120.63611),
 (34.42611, -119.84361),
 (34.89408, -120.45212),
 (38.5, -122.81667),
 (37.88972, -121.22361),
 (39.22992, -106.87051),
 (38.80949, -104.68873),
 (39.84658, -104.65622),
 (37.14312, -107.76023),
 (39.65, -106.91667),
 (39.13389, -108.53861),
 (38.53333, -106.93333),
 (40.48333, -107.21667),
 (38.505, -107.8975),
 (38.28869, -104.5057),
 (41.93806, -72.6825),
 (41.26389, -72.88722),
 (29.17354, -81.07186),
 (26.07874, -80.1622),
 (26.52694, -81.76639),
 (30.48333, -86.51667),
 (29.69194, -82.27556),
 (30.49534, -81.6937),
 (24.55707, -81.75539),
 (28.09973, -80.6356),
 (25.79056, -80.31639),
 (28.41826, -81.32413),
 (30.3582, -85.7956),
 (30.47806, -87.18694),
 (26.91778, -81.99361),
 (28.78333, -81.25),
 (27.40139, -82.55861),
 (27.91259, -82.68554),
 (30.39675, -84.35087),
 (27.96139, -82.54028),
 (27.65556, -80.41806),
 (26.6851, -80.09919)]
```

In [95]:
```
from ipywidgets.embed import embed_minimal_html # Allows us to create a separte file for the Google Maps

markers = gmaps.marker_layer(final_loc)    # Create markers for each tuple/coordinate
markermap = gmaps.Map()                    # Create a GMap variable
markermap.add_layer(markers)               # Add the layer of markers to GMap

embed_minimal_html('output/MarkerMap2.html', views=[markermap])
print("*** If no map appears, uncomment the line above, re-run this cell, and check your 'Metadata Part 5' folder to find the new HTML file name \"MarkerMap1.html\". ***")

markermap
```

```
*** If no map appears, uncomment the line above, re-run this cell, and check your 'Metadata Part 5' folder to find the new HTML file name "MarkerMap1.html". ***
Map(configuration={'api_key': 'AIzaSyCLla6Q7krE9xNg6SnNMoGNIzjCLddE9EU'}, data_bounds=[(16.053299230651156, -1…
```

Location of a hundred airports across the us.