Lab 12

Web Technologies 2: Data Serialization

Lab Objective: *Understand how serialization is used in web technologies. Practice using serialization to pack, transport, unpack, and navigate data structures in order to more easily utilize web based data sources.*

In order to more easily store and exchange data structures, standardized metalanguages have been developed to *serialize* data. Serialization is the process of packaging data with special properties in a form that can be easily unpacked and reconstructed as an identical copy on any computer. For example, if you wanted to share a k-d tree filled with data, you can easily send and replicate it on a colleague's computer by using serialization without having to send raw data and run the k-d sorting algorithm again. This can be used to transport exact data structures, or just to send data in an organized fashion, even between different programming languages. There are many different kinds of serialization methods for different languages and with different purposes, however, we will focus on serialization with the XML and JSON languages. These are two of the most prevalent serialization languages used for web communication and web design, but are commonly used to transport all programming languages. Their main application in web communication is in the transportation of organized data.

JSON

JSON stands for *JavaScript Object Notation*. This serialization method stores information about the objects as a specially formatted string that is easy for both humans and machines to read and write. When JSON is deserialized, this special string is parsed and the objects are recreated. Despite its name, it is a completely language independent format. JSON is built on top of two types of data structures: a collection of key/value pairs and an ordered list of values. In Python, these data structures are more familiarly called dictionaries and lists respectively.

The following is a very simple example of the characteristics of a family expressed in JSON:

```
{
    "lastname": "Smith"
```

```
    "father": "John",
    "mother": "Mary",
    "children": [
        {
            "name": "Timmy",
            "age": 8
        },
        {
            "name": "Missy",
            "age": 5
        }
    ]
}
```

> **Note**
>
> You have likely been working very closely with JSON without even knowing it! Jupyter Notebooks are actually stored as JSON. To see this, open one of your `.ipynb` files in a basic text editor.

In general, the JSON libraries of various languages have a fairly standard interface. Though the Python standard library has modules for JSON, if performance is critical, there are additional modules for JSON that are written in C such as `ujson` and `simplejson`.

Serialization using JSON

Let's begin by serializing some common Python data structures.

```
>>> import json
>>> ex1 = [0, 1, 2, 3, 4]
>>> json.dumps(ex1)
'[0, 1, 2, 3, 4]'
>>> ex2 = {'a': 34, 'b': 483, 'c':"Hello JSON"}
>>> json.dumps(ex2)
'{"a": 34, "c": "Hello JSON", "b": 483}'
```

The JSON representation of a Python list and dictionary are very similar to their respective string representations. Each of these generated strings is called a JSON *message*. Since JSON is based on a dictionary-like structure, you can nest multiple messages by distributing them appropriately within curly braces.

```
>>> aJSONstring = """{"car": {
    "make": "Ford",
    "model": "Focus",
    "year": 2010,
    "color": [255, 30, 30]
        }
    }"""
>>> t = json.loads(aJSONstring)
>>> print t
```

```
{u'car': {u'color': [255, 30, 30], u'make': u'Ford', u'model': u'Focus', u'year':↩
      2010}}
>>> print t['car']['color']
[255, 30, 30]
```

To generate a JSON message, use `dump()`. This method accepts a Python object, generate the message, and writes it to a file. The method `dumps()` does the same, but returns the string instead of writing it to a file. To perform the inverse operation, use `load()` or `loads()` to read a file or string, respectively.

The built-in JSON encoder/decoder only has support for the basic Python data structures such as lists and dictionaries. Trying to serialize an object which is not recognized will result in an error. Below is an example trying to serialize a set.

```
>>> a = set('abcdefg')
>>> json.dumps(a)
--------------------------------------------------------------------------
TypeError: set(['a', 'c', 'b', 'e', 'd', 'g', 'f']) is not JSON serializable
```

The serialization fails because the JSON encoder doesn't know how it should represent the set. However, we can extend the JSON encoder by subclassing it and adding support for sets. Since JSON has support for sequences and maps, one easy way would be to express the set as a map with one key that tells us the data structure type, and the other containing the data in a string. Now, we can encode our set.

```
class CustomEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, set):
            return {'dtype': 'set',
                    'data': list(obj)}
        return json.JSONEncoder.default(self, obj)

>>> a = set('abcdefg')
>>> json.dumps(a, cls=CustomEncoder)
'{"dtype": "set", "data": ["a", "c", "b", "e", "d", "g", "f"]}'
```

Though this is helpful, decoding this string would give us all of our data in a list. To allow for this string to be decoded as a Python set, we must build a custom decoder. Notice that we don't need to subclass anything.

```
>>> accepted_dtypes = {'set': set}
>>> def CustomDecoder(item):
...     type = accepted_dtypes.get(item['dtype'], None)
...     if type is not None and 'data' in item:
...         return type(item['data'])
...     return item

>>> c = json.loads(s, object_hook=CustomDecoder)
{u'a', u'b', u'c', u'd', u'e', u'f', u'g'}
# The 'u' is a prefix that signifies that the string value is Unicode.
# You can test this with:
>>> print c[0]
a
```

Problem 1. Python has a module in the standard library that allows easy manipulation of times and dates. The functionality is built around a `datetime` object.

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> print now
2016-10-04 11:52:33.513885

# We can also extract the individual time units
>>> now.year
2016
>>> now.minute
33
>>> now.microsecond
513885
```

However, `datetime` objects are not JSON serializable. Determine how best to serialize and deserialize a datetime object, then write a custom encoder and decoder. The datetime object you serialize should be equal to the datetime object you get after deserializing.

APIs and JSON

Many websites and web APIs (Application Program Interface) make extensive use of JSON. For example, almost any programs that utilize Twitter, Facebook, Google Maps, or YouTube communicate with their APIs using JSON. This means that any website that uses an embedded version of Google Maps will receive JSON strings with data to display Google Maps interface on a portion of the webpage. It also allows developers to receive information and update the embedded portions of their page without needing to reload the webpage file. An example of this is available at `https://developers.google.com/maps/documentation/javascript/examples/map-simple`. There are also web APIs which allow developers to retrieve the website data in JSON strings. A list of APIs for public data collection can be found at `http://catalog.data.gov/dataset?q=-aapi+api+OR++res_format%3Aapi#topic=developers_navigation`

Achtung!

Each website has a policy about data usage and automated retrieval that requires certain behavior. If data is scraped without complying with these requirements, there can be legal consequences.

Problem 2. JSON files are often used by APIs to respond to data requests. To demonstrate an example of this, we will do a brief data examination of

water usage in Los Angeles, California. The City of Los Angeles publishes some water usage data for the public. The API *endpoint* at which this data is available is at the address `https://data.lacity.org/resource/v87k-wgde.json`.

We can use the `requests` library from the previous lab to `GET` the data from the page.

```
requests.get("https://data.lacity.org/resource/v87k-wgde.json").json()
```

This code with load the object from JSON format into a Python list. Once the list has been created, gather the water usage data from 2012 to 2013 and the longitude and latitude of each point.

Use Bokeh to plot these points on a map of Los Angeles (refer to Lab 10 for additional help on Bokeh).

To draw a map centered on Los Angeles, you may use the following code:

```
from bokeh.plotting import figure
from bokeh.models import WMTSTileSource

fig = figure(title="Los Angeles Water Usage 2012-2013", plot_width=600, ↩
     plot_height=600, tools=["wheel_zoom", "pan", "hover"],
            x_range=(-13209490, -13155375), y_range=(3992960, 4069860),
             webgl=True, active_scroll="wheel_zoom")
fig.axis.visible = False

STAMEN_TONER_BACKGROUND = WMTSTileSource(
    url='http://tile.stamen.com/toner-background/{Z}/{X}/{Y}.png',
    attribution=(
        'Map tiles by <a href="http://stamen.com">Stamen Design</a>, '
        'under <a href="http://creativecommons.org/licenses/by/3.0">CC BY ↩
            3.0</a>.'
        'Data by <a href="http://openstreetmap.org">OpenStreetMap</a>, '
        'under <a href="http://www.openstreetmap.org/copyright">ODbL</a>'
    )
)

background = fig.add_tile(STAMEN_TONER_BACKGROUND)
```

To convert the longitude and latitude locations to be compatible with the map, you may the following code:

```
from pyproj import Proj, transform

from_proj = Proj(init="epsg:4326")
to_proj = Proj(init="epsg:3857")

def convert(longitudes, latitudes):
    x_vals = []
    y_vals = []
    for lon, lat in zip(longitudes, latitudes):
        x, y = transform(from_proj, to_proj, lon, lat)
        x_vals.append(x)
        y_vals.append(y)
    return x_vals, y_vals
```

More information about this dataset is available at `http://catalog.data.gov/dataset/residential-water-usage-zip-code-on-top-cb2ac`.

XML

XML is another data interchange metalanguage. However, it is a markup language rather than a object notation language. This means that it utilizes tags to distinguish the formatting of data rather than just encoding them. So, broadly speaking, XML is somewhat more robust and versatile than JSON, but slightly less efficient. Due to this minor difference, it is utilized in different ways.

To understand XML, we need to further explore what tags are. A tag is a special command enclosed in angled brackets (< and >) that describes properties of the data enclosed. For example, we can represent a car's properties in the XML below. Notice that tags are used to both open and close a property.

```
<car>
    <make>Ford</make>
    <model>Focus</model>
    <year>2010</year>
    <color model='rgb'>255,30,30</color>
</car>
```

XML data can be read as a tree or as a stream.

Since XML is a hierarchical storage format, it is very easy to build a tree of the data. The advantage of a tree format is random access to any part of the document at any time. However, this requires all of the XML to be loaded into memory for construction of the tree.

Reading a document as a stream means reading each piece sequentially or only reading a small portion of the file at a time. Because memory usage is constant, there is no limit to size of XML document that we can process this way, however, we do not have the random access of a tree.

The following will explore two APIs that parse XML formatted files and strings.

DOM

The DOM (Document Object Model) API allows you to work with an XML document as a tree in a hierarchy of elements. In order to sort the tree, the XML tags are read from the file and distributed accordingly. The DOM tree of the car from the XML code above would have `<car>` at the root element. This root element would have four children, `<make>`, `<model>`, `<year>`, and `<color>`. After a DOM tree has been sorted, we can traverse it like any other tree structure or search it by tag. Python's XML module includes two version of DOM: `xml.dom` and `xml.minidom`. MiniDOM is a minimal, more simple implementation of the DOM API.

SAX

SAX, Simple API for XML, is a very fast and efficient way to read an XML file. The main advantage of this method is memory conservation. A SAX parser will read XML sequentially instead of all at once. As the SAX parser iterates through the file, it emits events at either the start or the end of tags. You can provide functions to handle these events.

ElementTree

ElementTree is Python's unification of DOM and SAX APIs into a single, high-level API for parsing and creating XML trees. ElementTree provides a SAX-like interface for reading XML files via its `iterparse()` method. This will have all the benefits of reading XML via SAX. In addition to stream processing the XML, it will build the DOM tree as it iterates through each line of the XML input. ElementTree provides a DOM-like interface for reading XML files via its `parse()` method. This will create the tag tree that DOM creates.

We will demonstrate ElementTree using the following XML file.

```
<?xml version="1.0"?>
<contacts>
    <person>
        <firstname>John</firstname>
        <lastname>Doe</lastname>
        <phone type="mobile">1234567890</phone>
        <phone type="home">5432229875</phone>
        <email type="home">doughboy@bakery.com</email>
        <address type="home">34 South Street, Jonesville</address>
        <groups>personal,work</groups>
    </person>
    <person>
        <firstname>Sally</firstname>
        <lastname>Sue</lastname>
        <phone type="mobile">8372289491</phone>
        <groups>personal</groups>
    </person>
    <person>
        <firstname>Thor</firstname>
        <lastname></lastname>
        <phone type="mobile"></phone>
        <email type="home"></email>
        <address type="home"></address>
        <groups>work</groups>
    </person>
</contacts>
```

contacts.xml

First, we will look at viewing an XML document as a tree similar to the DOM model described above.

```
import xml.etree.ElementTree as et

f = et.parse('contacts.xml')

# To manually traverse the tree:
```

```
# We iterate through the element directly
# Note: getchildren() is old and deprecated (not supported), so we instead use ↩
    list() to gather children.
root = f.getroot()
children = list(root) # root has three children
person0 = children[0]
fields = list(person0) # The children elements of person0
```

We can search the entire tree for specific elements by:

```
# Searching for all tags equal to firstname
for n in root.iter('firstname'):
    print n.text
```

We can also filter with multiple tags by:

```
seek = {'firstname', 'lastname', 'phone'}
fi = (x for x in root.iter() if x.tag in seek)
for n in fi:
    print n.text
```

We can even modify the document tree in place.

```
# To remove Thor:
for n in root.findall("person"):
    if n.find("firstname").text == 'Thor':
        root.remove(n)

# Verify that Thor is really gone
for n in root.iter('firstname'):
    print n.text
```

Next, we will look at ElementTree's `iterparse()` method. This method is very similar to the SAX method for parsing XML. There is one important difference. ElementTree will still build the document tree in the background as it is parsing. We can prevent this by clearing each element by calling its `clear()` method when are finished processing it.

```
f = et.iterparse('contacts.xml') # This is an iterator that you can use to go ↩
    forward and backward in the tree
for event, tag in f:
    print "{}: {}".format(tag.tag, tag.text)
    tag.clear()
```

We can also get both start and end events, however, start events are mostly useful for looking at attributes or to trigger some other action on element starts. The element is not guarenteed to be complete until the end event.

```
for event, tag in et.iterparse('contacts.xml', events=('start', 'end')):
    print "{} {}<{}>: {}".format(event, tag.tag, tag.attrib, tag.text)
```

Problem 3. Using ElementTree to parse `books.xml`, which represents a small dataset of books, and answer the following questions. Include the code you used with your answers.

1) Who is the author of the most expensive book in the list?

2) How many of the books were published before Dec 1, 2000?

3) Which books reference Microsoft in their descriptions?

HINT: To answer these queries, it may be most convenient to populate a pandas DataFrame with all the XML data.

Problem 4. The City of New York makes publicly available some data concerning the location of publicly available recycling bins. The XML endpoint is located at `https://data.cityofnewyork.us/api/views/sxx4-xhzg/rows.xml?accessType=DOWNLOAD`. Using the `requests` library, GET the XML file from that address and build it into an Element Tree.

Then, using Monte Carlo approximation, estimate the average distance from a given point in New York City to the nearest recycling bin. One efficient way of solving this problem is to perform a nearest neighbor search using a KDTree. We recommend you use scipy's `cKDTree` and its `query` method to find the nearest neighbor for each point.

Here's a quick example of how to use `cKDTree`.

```
>>> import numpy as np
>>> from scipy.spatial import cKDTree

>>> pts = np.random.rand(10,2)
>>> kdtree = cKDTree(pts)
>>> query_pts = [[0.5, 0.5], [0.25, 0.25]]
# k=1 corresponds to finding the nearest neighbor
>>> distance, index = kdtree.query(query_pts, k=1)

# 'distance' returns the distance to the nearest neighbor
# 'index' returns the index in 'pts' that corresponds to the
#     nearest neighbor
```

The `random_newyork_locations.csv` file contains about 35,000 uniformly distributed points throughout New York City measured in longitude and latitude.

We could measure the distance between the longitude and latitude points, however the units would not have a very interpretable and relatable value. Therefore, use the `convert()` function in Problem 2 to transform your longitude and latitude points to epsg:3857. These units very closely approximately meters.

Return your final answer in miles.

More information on this dataset is available at `https://catalog.data.`

`gov/dataset/public-recycling-bins-eb48e`.
(Hint: If you get the file from requests, you may want to remove '<response>' and '</response>' from the beginning and end of your content string to make parsing a little easier.)

Additional Material

Pickle

Aside from the serialization methods we have demonstrated, Python has a serialization library called *pickle*. This library makes it very easy to serialize and share your python objects. The following code is a simple example of how to create a pickled object and then unpack it.

```
>>> import pickle

>>> listobject = [1, 2, 3, 4, 5]
>>> item = pickle.dumps(listobject)
>>> item
'(lp0\nI1\naI2\naI3\naI4\naI5\na.'

>>> pickle.loads(item)
[1, 2, 3, 4, 5]
```

You can also write these pickled objects to a text file as strings. The following code demonstrates this.

```
>>> a = open('list.txt', 'w')

>>> listobject = [1,2,3,4,5]
>>> item = pickle.dump(listobject, a)
>>> a.close()

>>> a = open('list.txt')
>>> pickle.load(a)
[1, 2, 3, 4, 5]
```

Pickle has many powerful applications such as these for communication between Python users. See `https://docs.python.org/2.7/library/pickle.html` for more documentation.