

## Lab 1

# Serialization

**Lab Objective:** *Learn about JSON and XML.*

In order for computers to communicate one with another, they need standardized ways of storing structured data. For example, suppose you have a python list that you want to send to somebody else. How would you store it outside of the interpreter? However we choose to store our list, we need to be able to load it back into the Python interpreter and use it as a list. What if we wanted to store more complex objects? The process of serialization seeks to address this situation. Serialization is the process of storing an object and its properties in a form that can be saved or transmitted and later reconstructed back into an identical copy of the original object.

## JSON

JSON, pronounced “Jason”, stands for *JavaScript Object Notation*. This serialization method stores information about the objects as a specially formatted string. It is easy for both humans and machines to read and write the format. When JSON is deserialized, the 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. Python’s standard library has a module that can read and write JSON. In general, the JSON libraries of various languages have a fairly standard interface. If performance is critical, there are Python modules for JSON that are written in C such as `ujson` and `simplejson`.

Let’s begin with an example.

```
>>> import json
>>> json.dumps(range(5))
'[0, 1, 2, 3, 4]'
>>> json.dumps({'a': 34, 'b': 483, 'c': "Hello JSON"})
'{"a": 34, "c": "Hello JSON", "b": 483}'
```

As you can see, the JSON representation of a Python list and dictionary are very similar to their respective string representations. You can also see that each JSON message is enclosed in a pair of curly braces. We can even nest multiple messages.

```
>>> a = """{"car": {
    "make": "Ford",
    "model": "Focus",
    "year": 2010,
    "color": [255, 30, 30]
  }}"""
>>> t = json.loads(a)
>>> print t
{'car': {'color': [255, 30, 30], 'make': 'Ford', 'model': 'Focus', 'year': 2010}}
>>> print t['car']['color']
[255, 30, 30]
```

Most JSON libraries support the `dump[s]`/`load[s]` interface. To generate a JSON message, we use `dump` which will accept the Python object and generate the message and write it to a file. `dumps` does the same, but just returns the string rather than writing it to a file. To perform the inverse operation, we use `load` or `loads` for reading from 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 a set will result in an error

```
>>> 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 as a string. 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)

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

However, we want a Python set back when we decode. JSON will happily return our dictionary, but the data will be in a list. How do we tell it to convert our list back into a set? The answer is to build a custom decoder. Notice that we don't need to subclass anything.

```

accepted_dtypes = {'set': set}
def custom_decoder(dct):
    dt = accepted_dtypes.get(dct['dtype'], None)
    if dt is not None and 'data' in dct:
        return dt(dct['data'])
    return dct

>>> json.loads(s, object_hook=custom_decoder)
{'u'a', u'b', u'c', u'd', u'e', u'f', u'g'}

```

Many websites and web APIs make extensive use of JSON. Twitter, for example, return JSON messages for all queries.

**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

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.

## XML

XML is another data interchange format. It is a markup language rather than a object notation language. Broadly speaking, XML is somewhat more robust and versatile than JSON, but less efficient.

To understand XML, we need to understand what tags are. A tag is a special command enclosed in angled brackets (< and >) that describe properties of the data enclosed. For example, we can represent our car from above in the XML below.

```

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

```

We can read XML data 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 is random access to any part of the document at any time. However, all of the XML must be loaded into memory to build this tree.

To alleviate the burden of loading a large XML document into memory all at once, we can read the file sequentially. When streaming the XML data, we are only reading a small chunk of the file at a time. There is no limit to size of XML document that we can process this way as memory usage will be constant. However, we sacrifice the random access that the tree gives us.

## DOM

The DOM (Document Object Model) API allows you to work with an XML document as a tree. Python's XML module includes two versions of DOM: `xml.dom` and `xml.minidom`. MiniDOM is a minimal, more simple implementation of the DOM API. The motivation behind DOM is to represent an XML as a hierarchy of elements. This is accomplished by building a tree of the elements as the XML tags are read from the file. The DOM tree of the car above would have `<car>` at the root element. This root element would have four children, `<make>`, `<model>`, `<year>`, and `<color>`. We would traverse this DOM tree just like we would any other tree structure. DOM trees can be searched by tag as well.

## SAX

SAX, Simple API for XML, is a very fast, efficient way to read an XML file. The main advantage of this method for reading an XML file is memory conservation. A SAX parser reads 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 into a single, high-level API for parsing and creating XML. 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.

```
1 <?xml version="1.0"?>
2 <contacts>
3   <person>
4     <firstname>John</firstname>
5     <lastname>Doe</lastname>
6     <phone type="mobile">1234567890</phone>
7     <phone type="home">5432229875</phone>
8     <email type="home">doughboy@bakery.com</email>
9     <address type="home">34 South Street, Jonesville</address>
10    <groups>personal,work</groups>
11  </person>
12  <person>
13    <firstname>Sally</firstname>
14    <lastname>Sue</lastname>
15    <phone type="mobile">8372289491</phone>
16    <groups>personal</groups>
17  </person>
18  <person>
19    <firstname>Thor</firstname>
20    <lastname></lastname>
21    <phone type="mobile"></phone>
22    <email type="home"></email>
```

```

24     <address type="home"></address>
    <groups>work</groups>
    </person>
26 </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')

# manually traversing the tree
# we iterate through the element directly
# getchildren() is old and deprecated (not supported).
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
# searching for all tags equal to firstname
for n in root.iter('firstname'):
    print n.text

# we can also filter with multiple tags
# notice we use a set lookup in the conditional inside the generator expression
fields = {'firstname', 'lastname', 'phone'}
fi = (x for x in root.iter() if x.tag in fields)
for n in fi:
    print n.text

# we can even modify the document tree inplace
# let's remove Thor
# refer to the documentation of ElementTree for adding elements
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
for event, tag in f:
    print "{}: {}".format(tag.tag, tag.text)
    tag.clear()

# we can 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 guaranteed 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 2.** Using ElementTree to parse books.xml, 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 May 1, 2000?
- 3) Which books reference Microsoft in their descriptions?