CS106X Handout 43
Autumn 2012 December 5th, 2012

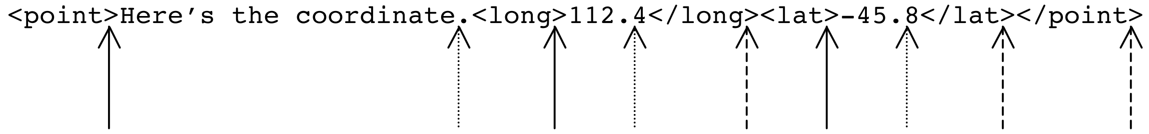
Python: XML, Sockets, Servers

XML is an overwhelmingly popular data exchange format, because it's human-readable and easily digested by software. Python has excellent support for XML, as it provides both SAX (Simple API for XML) and DOM (Document Object Model) parsers via **`xml.sax`**, **`xml.dom`**, and **`xml.dom.minidom`** modules. SAX parsers are event-driven parsers that prompt certain methods in a user-supplied object to be invoked every time some significant XML fragment is read. DOM parsers are object-based parsers than build in-memory representations of entire XML documents.

Python SAX: Event Driven Parsing

SAX parsers are popular when the XML documents are large, provided the program can get what it needs from the XML document in a single pass. SAX parsers are stream-based, and regardless of XML document length, SAX parsers keep only a constant amount of the XML in memory at any given time. Typically, SAX parsers read character by character, from beginning to end with no ability to rewind or backtrack. It accumulates the stream of characters building the next XML fragment, where the fragment is a start element tag, an end element tag, or character data content. As it reads the end of one fragment, it fires off the appropriate method in some handler class to handle the XML fragment.

For instance, consider the ordered stream of characters that might be some web server's response to an HTTP request:



```
<point>Here's the coordinate.<long>112.4</long><lat>-45.8</lat></point>
```

In the above example, you'd expect events to be fired (meaning, you'd expect functions or methods to be invoked) as the parser pulls in each of the characters are the tips of each arrow. Each of the solid arrows identifies the completion of an element start tag, each of the dotted arrows marks the end of a character data segment, and each of the dashed arrows addresses the end of an element end tag. (You can also configure the parser to fire start-of-document and end-of-document events as well.)

The SAX parser isn't an exposed class in the sense that you directly construct your own instance. Instead, you rely on a factory function (that fancy terminology for a function that creates objects for you) inside **`xml.sax`** called **`make_parser`** to construct one for you. The benefit of this factory-function pattern is that you're forced to respect the public API of the parser class, because the **`xml.sax`** module is free to change the implementation of the parser with any update to the Python runtime environment.

The other major player in the **`xml.sax`** module is the **`ContentHandler`** class, the implementation of which is little more than this:

```
class ContentHandler:
   def startDocument(self): pass
   def endDocument(self): pass
   def startElement(self, tag, attributes): pass
   def endElement(self, tag): pass
   def characters(self, data): pass
```

Normally, you'll subclass **`ContentHandler`** and override the implementation of those methods that really should be doing something. If you don't override a method, then you inherit the no-op implementation provided by **`ContentHandler`**. (Technically, you don't **need** to subclass **`ContentHandler`**, but whatever object you do install needs to respond to the same set of methods at the very minimum.)

Here's a little python program that prints out an XML document in human readable format that makes it clear what all the parent-tag/child-tag relationships are.

```
from urllib2 import urlopen
from xml.sax import make_parser, ContentHandler
import sys

# Subclass of ContentHandler that, when installed within
# a SAX parser, helps that parser print all of the start
# and end tags out to standard output.
class PrettyPrintingTagHandler(ContentHandler):

   def __init__(self):
      ContentHandler.__init__(self) # equivalent of super() in Java
      self.__indentLevel = 0        # initially at an indentation level of 0

   def startElement(self, tag, attributes):
      for i in xrange(self.__indentLevel):
         sys.stdout.write("    ")
      sys.stdout.write("<%s>\n" % tag)
      self.__indentLevel += 1

   def endElement(self, tag):
      self.__indentLevel -= 1
      for i in xrange(self.__indentLevel):
         sys.stdout.write("    ")
      sys.stdout.write("</%s>\n" % tag)



# Standard boilerplate associated with the parsing of any particular
# XML file, although the handlers in this example assume the XML file
# is actually an RSS file
def prettyPrint(url):
   infile = urlopen(url)
   parser = make_parser()
   parser.setContentHandler(PrettyPrintingTagHandler())
   parser.parse(infile)

prettyPrint("http://rss.nytimes.com/services/xml/rss/nyt/GlobalHome.xml")
```

DOM: Object-Based Parsing in Python

DOM parsers digest an entire XML document and build an ordered tree whose memory footprint is proportional to the size of the original document. DOM trees are memory burglars, but trees are dynamic data structure that are easily traversed, searched, pruned, updated, augmented, and otherwise manipulated. All web browsers store the entirely of an HTML document in DOM tree form, because doing so allows JavaScript and other web scripting languages to access and modify the tree, which in turn modifies the HTML presentation within the browser windows. DOM is the backbone of the dynamic Web experience, and SAX as a parsing methodology has little to contribute to dynamic, JavaScript-enabled web sites.

DOM parsers take the stance that all XML documents are really the inorder serialization of trees. Consider, for instance, the following XML fragment:

```
<entry>
   <address>
      <number>2210</number>
      <street>Hope Lane</street>
      <city>Cinnaminson</city>
      <state>New Jersey</state>
      <zipcode>08077</zipcode>
   </address>
   <phone type="home">856-786-06xx</phone>
   <phone type="cell">856-829-81xx</phone>
   <phone type="fax">856-829-72xx</phone>
</entry>
```

Look at the fragment as an alternate representation of a tree. The root of the tree is the **`entry`** node, and it has four children—one **`address`** node and the **`phone`** nodes. The address node itself has five children. The **`phone`**, **`number`**, **`street`**, **`city`**, **`state`**, and **`zipcode`** nodes all have single text nodes as children. Text nodes have are leaf nodes that store text on behalf of the overall document; all of the other nodes in the document exist to bear other children. If you think about it, an XML fragment like the one above is nothing more than a C++ **`struct`**.

```
from urllib2 import urlopen
from xml.dom.minidom import parse

def listTitles(feeds):
   for feed in feeds:
      instream = urlopen(feed)
      xmltree = parse(instream)
      items = xmltree.getElementsByTagName("item")
      for item in items:
         titlesubtree = item.getElementsByTagName("title")
         titleNode = titlesubtree[0]
         titleTextNode = titleNode.firstChild
         print titleTextNode.nodeValue

feeds = ["http://rss.nytimes.com/services/xml/rss/nyt/GlobalHome.xml",
         "http://feeds.chicagotribune.com/chicagotribune/news/",
         "http://www.philly.com/philly_news.rss"]

listTitles(feeds)
```