Networking With Python

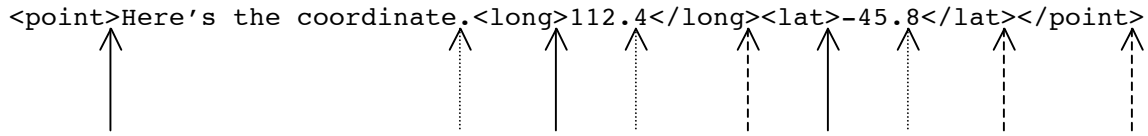
XML is an overwhelmingly popular data exchange format, because it's human-readable and easily digested by software.

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 either 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 a some web server's response to an http request:



In the above example, you'd expect events to be fired 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 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, tag): pass
   def endDocument(self, tag): 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.)

```
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 RSSNewsFeedTagHandler(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)
```

```
# Subclass of ContentHandler that, when planted within
# a stream-based parser, manages to print every single title
# within a RSS news feed to standard out.
class RSSNewsFeedTitleHandler(ContentHandler):

   def __init__(self):
      ContentHandler.__init__(self)
      self.__inItem = False
      self.__inTitle = False

   def startElement(self, tag, attributes):
      if tag == "item": self.__inItem = True
      if self.__inItem and tag == "title": self.__inTitle = True

   def endElement(self, tag):
      if tag == "item": self.__inItem = False
      if tag == "title":
         sys.stdout.write("\n")
         self.__inTitle = False

   def characters(self, data):
      if self.__inTitle:
         sys.stdout.write(data)

# 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 pullTitles(url):
   infile = urlopen(url)
   parser = make_parser()
   parser.setContentHandler(RSSNewsFeedTagHandler())
#  parser.setContentHandler(RSSNewsFeedTitleHandler())
   parser.parse(infile)

pullTitles("http://feeds.gawker.com/valleywag/full")
```

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 in-order 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>
```

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

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")
         assert len(titlesubtree) >= 1, \
               "Failed to find a title within a particular item subtree"
         assert len(titlesubtree) <= 1, \
               "Unexpected found more than one title within the item"
         titleNode = titlesubtree[0]
         titleTextNode = titleNode.firstChild
         title = titleTextNode.nodeValue
         sys.stdout.write("%s\n" % title)

feeds = ["http://feeds.gawker.com/wonkette/full",
         "http://feeds.gawker.com/gawker/full",
         "http://feeds.gawker.com/lifehacker/full",
         "http://feeds.gawker.com/gizmodo/full",
         "http://feeds.gawker.com/consumerist/full",
         "http://feeds.gawker.com/valleywag/full"]

listTitles(feeds)
```

Simple Sockets: Building a Time Server

```
from socket import socket, AF_INET, SOCK_STREAM
from time import gmtime, strftime
from sys import argv
import sys

def startServer(portNum):
   print "Starting timestamp server (listing to port %d)" % portNum
   s = socket(AF_INET, SOCK_STREAM)
   s.bind(("", portNum))
   s.listen(5)
   try:
      while True:
         client, address = s.accept()
         print "Accepted connection from remote client at " + str(address)
         infile = client.makefile()
         while True:
            line = infile.readline()
            if line == "\r\n": break
            sys.stdout.write(line)
         infile.close()
         client.send("Current time is: " + \
                     strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime()))
         client.close()
   except KeyboardInterrupt:
      print "Detected stop request: Killing server";
   s.close()

defaultPortNumber = 40000
def getPortNumber():
   port = defaultPortNumber
   if (len(argv) > 1):
      try:
         port = int(argv[1])
      except ValueError:
         print "Non-numeric argument detected... using default port"
   return port

startServer(getPortNumber())
```

Source: http://see.stanford.edu/materials/icsppcs107/37-Networking-With-Python.pdf