Application Development Framework
Application Express
Big Data
Business Intelligence
Cloud Computing
Communications
Database Performance & Availability
Data Warehousing
Database
.NET
Dynamic Scripting Languages
Embedded
Digital Experience
Enterprise Architecture
Enterprise Management
Identity & Security
Java
Linux
Mobile
Service-Oriented Architecture
Solaris
SQL & PL/SQL
Systems - All Articles
Virtualization

Mastering Oracle+Python, Part 6: Python Meets XML

by Przemyslaw Piotrowski

Leveraging Oracle XML DB and Python XML processing capabilities

Published November 2011

See series TOC

No one can argue that XML is now the de facto standard for information interchange in software.

For that reason, Oracle Database ships with a wide range of XML related enhancements and tools called collectively Oracle XML DB. XML DB encompasses a set of technologies embedded into the database for accessing and manipulating XML data at the SQL level, providing access to such technologies as XML Schema, XPath, or XQuery.

Then again, Python comes with a full-fledged library of structured markup parsers, that allow access and manipulation of XML in a clean and elegant manner. Besides modules that reside in the standard library, there's also a Python binding for libxml2, a popular parser that you might have already stumbled upon when coming from other languages.

In this tutorial we are going to digest various Python XML parsers and a way to communicate with Oracle Database using simple XML protocol.

Before diving into parsing let's generate two XML files directly from the HR schema using XML DB as shown in Listing 1.

Listing 1. Generating XML files through SQL*Plus script (hrxml.sql)
```
conn hr/hr

set timing off
set termout off
set heading off
set long 99999

spool emp.xml replace
select dbms_xmlgen.getxml('select * from employees') from dual;

spool dept.xml replace
select dbms_xmlgen.getxml('select * from departments') from dual;

exit
```

Running this script with `sqlplus -S /nolog @hrxml.sql` results in creation of two files in the current directory: dept.xml and emp.xml with full contents of tables DEPARTMENTS and EMPLOYEES respectively. XML structure generated by the dbms_xmlgen.getxml() function for DEPARTMENTS table is standardized into the following format:

```
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <DEPARTMENT_ID>10</DEPARTMENT_ID>
  <DEPARTMENT_NAME>Administration</DEPARTMENT_NAME>
  <MANAGER_ID>200</MANAGER_ID>
  <LOCATION_ID>1700</LOCATION_ID>
 </ROW>
...
</ROWSET>
```

To begin with, lets take a look at options we have in Python when it comes to processing XML.

Python XML Dossier

Among the XML parsers available through Python's standard library the major ones are:

xml.dom - Implements Document Object Model (DOM), standard representation of hierarchical XML tree. With this module you can access random XML data as defined in the W3C specification (DOM Level 2 recommendation). This API enables traversing DOM tree with a set of standard methods on XML nodes.
xml.dom.minidom - Simplified and lightweight implementation of DOM standard (DOM Level 1 recommendation with some extensions from Level 2). Function to_minidom in Listing 2. illustrates a way of creating Python minidom object out of a regular SQL query.

Listing 2. Turning SQL queries into minidom objects
```
import cx_Oracle
       from xml.dom.minidom import parseString

       def to_minidom(sql):
         with cx_Oracle.connect("hr/hr") as db:
           cursor = db.cursor()
           cursor.execute("select dbms_xmlgen.getxml('%s') from dual" % sql)
           return parseString(cursor.fetchone()[0].read())

       if __name__ == "__main__":
         md = to_minidom("select * from departments")
         rows = md.getElementsByTagName("ROW")
         print type(rows), len(rows)
```

Running this file directly would result in: `<class 'xml.dom.minicompat.NodeList'> 27`, and by taking length of getElementsByTagName results we get something equivalent to COUNT(*), only this time we do it the XML way.

xml.sax - Implementation of Simple API for XML (SAX) which - opposite to DOM - enables serial access to XML nodes. With SAX it's impossible to access arbitrary XML element, yet SAX remains more efficient for processing large files than DOM, which needs to allocate memory for the whole XML tree while SAX only reads a part of XML stream. This module is accompanied by several helper ones in the same namespace: xml.sax.handler with interfaces to SAX handlers, xml.sax.saxutils with many useful utilities for dealing with SAX model, and xml.sax.xmlreader with implementation of XMLReader SAX interface.
xml.etree.ElementTree - Shipped with Python version 2.5 and already popular long before that, ElementTree module represents another breed of XML parsers. Module written by Fredrik Lundh quickly found its way into Python's standard library given its efficiency and `pythonic` nature. ElementTree (and its C-implemented counterpart cElementTree) is an object that maps XML trees into flexible data types, something - as the author says - "cross between a list and a dictionary". Due to its flexibility ElementTree is currently endorsed to be de facto standard module for parsing XML with Python. Some examples following this introduction should reveal that clearly.

Processing

Two files generated at the beginning of this tutorial will be used for demonstrating Python's native XML processing capabilities. Make sure the interpreter has access to these files by changing the current path with os.chdir() function (current path can be obtained using os.getcwd()). Since xml.dom and xml.dom.minidom cover basically the same functionality, only the latter one is going to be covered here.

Listing 3. Example of using xml.dom.minidom to parse departments into dictionary

```
from xml.dom.minidom import parse

class ParseDept(dict):
  def __init__(self, xmlfile):
    dict.__init__(self)
    self.dom = parse(xmlfile)

    dept = {}
    for i in self.dom.getElementsByTagName("ROW"):
      dept_id = i.getElementsByTagName("DEPARTMENT_ID").pop().firstChild.data
      self[dept_id] = {}
      for j in i.childNodes:
        if j.nodeType==self.dom.ELEMENT_NODE:
          self[dept_id][j.tagName.lower()] = j.firstChild.data

if __name__ == "__main__":
  dept = ParseDept("dept.xml")
```

Listing 4. Processing department data with xml.sax

```
from xml.sax import make_parser
from xml.sax.handler import ContentHandler

class DeptHandler(ContentHandler):
  dept = {}

  def startElement(self, name, attrs):
    self.text = ""
    return

  def characters(self, ch):
    self.text += ch
    return

  def endElement(self, name):
    if name=="DEPARTMENT_ID":
      self.curr = self.text
      self.dept[self.text] = {}
    elif name!="ROW":
      self.dept[self.curr][name.lower()] = self.text

  def __del__(self):
    print self.dept

if __name__ == "__main__":
  parser = make_parser()
  parser.setContentHandler(DeptHandler())
  parser.parse(open("dept.xml"))
```

Listing 5. Parsing XML with ElementTree module

```
from xml.etree.ElementTree import ElementTree

class DeptTree:
  def __init__(self, xmlfile):
    self.tree = ElementTree()
    self.tree.parse(xmlfile)

  def traverse(self):
    dept = {}
    rows = self.tree.findall("ROW")
    for row in rows:
      for elem in row.findall("*"):
        if elem.tag=="DEPARTMENT_ID":
          dept_id = elem.text
          dept[dept_id] = {}
        else:
          dept[dept_id][elem.tag.lower()] = elem.text
    return dept
```

```
if __name__ == "__main__":
  dt = DeptTree("dept.xml").traverse()
  print dt
```

Judging from these implementations, ElementTree is clearly a winner in areas of usability and development speed. It complements Python with powerful processing capabilities that are easy to use and just feel very natural. And then there's its C implementation called cElementTree (remember cPickle?), which makes it perform really efficiently.

Depending on the project needs, there would be times when DOM parser could be even more useful. Having this vast tool set at hand enables you to choose the right parser appropriately.

XML APIs

Speaking of information interchange, XML makes an excellent choice for transmitting structured data. All major standards such as WSDL, SOAP, RSS, XMPP, BPEL are built on XML and the number of assisting technologies like XSLT, XQuery, or XML Schema makes them flexible and compatible.

Database APIs are usually limited to atomic functions like ADD_EMPLOYEE, DROP_DEPARTMENT or so, and when there's a need to perform batch operations they are called in loop from the middleware. Such actions are often enforced by the client's inability to use complex types in database calls. This situation applies to cx_Oracle as well. You cannot define a complex object in Python and then use it for PL/SQL calls, besides the basic functionality of PL/SQL arrays demonstrated in Part 5 of this series.

With an advent of XML technologies, Oracle Database has been enriched with all kinds of them, including XQuery, XPath, XSLT, XML Schema, SQL/XML to name just a few. SQL/XML and XML/SQL extensions makes querying and constructing XML data as simple as working with relational data - literally, as the support for mappings between relational and XML models is really extensive.

With XML, Python and Oracle XML DB there's a possibility of calling APIs using complex structures.

Listing 6. Basic HR administration with the PKG_HR_XML package

```
CREATE OR REPLACE PACKAGE pkg_hr_xml AS

  FUNCTION department_get(
    p_deptid IN NUMBER
  ) RETURN CLOB;

  PROCEDURE department_merge(
    p_deptxml IN CLOB
  );

END pkg_hr_xml;
/

CREATE OR REPLACE PACKAGE BODY pkg_hr_xml AS

  FUNCTION department_get(
    p_deptid IN NUMBER
  ) RETURN CLOB AS
    l_deptxml CLOB;
  BEGIN
    l_deptxml := dbms_xmlgen.getxml('
      SELECT d.*, CURSOR(
        SELECT *
        FROM employees e
        WHERE e.department_id=d.department_id
      ) emp
      FROM departments d WHERE department_id='||p_deptid||'
    ');

    RETURN l_deptxml;
  END department_get;

  PROCEDURE department_merge(
    p_deptxml IN CLOB
  ) AS
    l_deptid INT;
    l_deptxml XMLType;
  BEGIN
    BEGIN
      l_deptxml := XMLType(p_deptxml);
      SELECT extractValue(l_deptxml, 'ROWSET/ROW/DEPARTMENT_ID[1]/text()')
      INTO l_deptid FROM dual;
    END;

    MERGE INTO departments d
    USING (
      WITH t1 AS (
      SELECT extract(l_deptxml, 'ROWSET/ROW/*[name(descendant::*)!="EMP_ROW"]') e
      FROM dual)
      SELECT
        extractValue(e, '/DEPARTMENT_ID') department_id,
        extractValue(e, '/DEPARTMENT_NAME') department_name,
        extractValue(e, '/MANAGER_ID') manager_id,
        extractValue(e, '/LOCATION_ID') location_id
      FROM t1
    ) t ON (t.department_id=d.department_id)
    WHEN MATCHED THEN
      UPDATE SET d.department_name=t.department_name, d.manager_id=t.manager_id,
        d.location_id=t.location_id
    WHEN NOT MATCHED THEN
```

```
      INSERT (department_id, department_name, manager_id, location_id)
      VALUES (departments_seq.nextval, t.department_name, t.manager_id, t.location_id);

    MERGE INTO employees e
    USING (
      SELECT
        extractValue(value(f), '/EMP_ROW/EMPLOYEE_ID') employee_id,
        extractValue(value(f), '/EMP_ROW/FIRST_NAME') first_name,
        extractValue(value(f), '/EMP_ROW/LAST_NAME') last_name,
        extractValue(value(f), '/EMP_ROW/EMAIL') email,
        extractValue(value(f), '/EMP_ROW/PHONE_NUMBER') phone_number,
        extractValue(value(f), '/EMP_ROW/HIRE_DATE') hire_date,
        extractValue(value(f), '/EMP_ROW/JOB_ID') job_id,
        extractValue(value(f), '/EMP_ROW/SALARY') salary,
        extractValue(value(f), '/EMP_ROW/COMMISSION_PCT') commission_pct,
        extractValue(value(f), '/EMP_ROW/MANAGER_ID') manager_id
      FROM TABLE(XMLSequence(extract(l_deptxml , 'ROWSET/ROW/EMP/EMP_ROW'))) f
    ) t ON (t.employee_id=e.employee_id)
    WHEN MATCHED THEN
      UPDATE SET e.first_name=t.first_name, e.last_name=t.last_name,
        e.email=t.email, e.phone_number=t.phone_number, e.hire_date=t.hire_date,
        e.job_id=t.job_id, e.salary=t.salary, e.commission_pct=t.commission_pct,
        e.manager_id=t.manager_id
    WHEN NOT MATCHED THEN
      INSERT (employee_id, first_name, last_name, email, phone_number, hire_date,
        job_id, salary, commission_pct, manager_id, department_id)
      VALUES (employees_seq.nextval, t.first_name, t.last_name, t.email,
        t.phone_number, t.hire_date, t.job_id, t.salary, t.commission_pct,
        t.manager_id, l_deptid);

  END department_merge;
END pkg_hr_xml;
/

ALTER TRIGGER update_job_history DISABLE;
/
```

Using the PKG_HR_XML package we take advantage of two important Oracle XML DB features: mapping relational data to XML with DBMS_XMLGEN package and extract XML nodes into relational structure with XMLSequence() and extract(). (Note that we need to disable update_job_history trigger because of the unique constraint on employee_id, start_date columns in JOB_HISTORY table.)

Both department_get() and department_merge() work with the complex XML structure consisting of a single department information with a list of all employees in that department. This way we can update all department's employees in batch with one call.

On the client side, we set up a Python API based on ElementTree as shown in Listing 7.

Listing 7. Wrapping PKG_HR_XML PL/SQL package with cx_Oracle and ElementTree.

```
import cx_Oracle
from cStringIO import StringIO
from xml.etree.ElementTree import ElementTree

class Department:
  def __enter__(self):
    return self

  def __exit__(self, type, value, traceback):
    self.cursor.close()
    self.db.close()

  def __init__(self, deptid):
    self.db = cx_Oracle.Connection("hr/hr@xe")
    self.cursor = self.db.cursor()

    clob = self.cursor.var(cx_Oracle.CLOB)
    return_value = self.cursor.callfunc("pkg_hr_xml.department_get", clob, [deptid])
    try:
      self.tree = ElementTree(file=StringIO(return_value.read()))
    except AttributeError:
      raise Warning, "Department %s not found." % deptid

  def set(self):
    clob = self.cursor.var(cx_Oracle.CLOB)
    deptxml = StringIO()
    self.tree.write(deptxml)
    clob.setvalue(0, deptxml.getvalue())
    self.cursor.callproc("pkg_hr_xml.department_merge", [clob])

  def __str__(self):
    deptxml = StringIO()
    self.tree.write(deptxml)
    return deptxml.getvalue()

if __name__ == "__main__":
  with Department(60) as dept:
    print dept
    dept.set()
```

In the example in Listing 7, we are covering a PL/SQL package with Python XML API using ElementTree. The dept variable that gets Department(60) instantiated and assigned represents an XML tree with set() method that enables to save changes back to the database. In practice, changing the client XML translates into MERGE SQL statement which performs necessary INSERTs or UPDATEs.

For a detailed help on ElementTree as well as numerous examples visit Fredrik Lundh's effbot.org.

Conclusion

During this tutorial, several major Python XML approaches were introduced and demoed. Working in tandem with Oracle's XML DB there are excellent capabilities of tying these technologies together. Python's out-of-the-box support for DOM, SAX, and tree-like XML structures gives you plenty of options for processing XML data.

There's also another side of the coin, as Python can be a splendid tool for data integration, such that other serialization formats could be easily translated into XML and the other way. Given the flexibility of Python data types and its dynamic nature makes it ideal for transforming all kinds of file and data formats.

Przemyslaw Piotrowski is an information technology specialist working with emerging technologies and dynamic, agile development environments. Having a strong IT background that includes administration, development and design, he finds many paths of software interoperability.