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 3: Data Parsing

by Przemyslaw Piotrowski

Learn some basic and advanced techniques for parsing data in Python.

Published September 2007

See series TOC

There are countless reasons for parsing data, as well as tools and techniques to do it. But even the "right" tool may be insufficient when you need to do something new with the data. The same concerns exist for the integration of heterogeneous data sources. Sooner or later, the right tool for the right job happens to be a programming language.

Oracle offers some very powerful utilities for loading, processing, and unloading data. SQL*Loader, Data Pump, external tables, Oracle Text, regular expressions—it's all there. Yet there is often a need to do things outside the database (or, trivially, perhaps you just weren't granted the necessary database privileges).

Python delivers possibilities for efficient data parsing at a high level. The extensive standard library and many modules available for free on the Internet make it possible to work with data logic rather than dissecting bytes by hand.

String Theory
At the lowest level of text parsing are strings. Python doesn't differentiate characters as separate datatypes but distinguishes between regular and Unicode string types. They can be enclosed in single, double, or triple quotes and are one of Python's immutable objects—consequently you cannot change them once they are created. Each operation creates a new string object in-place. For programmers with statically-typed language experience this may sound really odd at first, but there are certain reasons for such an implementation, mostly concerning performance.

As Python fully supports Unicode, processing multi-language information is not a problem. When creating Unicode strings manually either use the `u` prefix directly before the string like (as in `u"Unicode text"`) or use the built-in unicode() function. Strings can be encoded in any supported character set using the unicode() or encode() methods. For a list of supported encodings please consult the Standard Encodings section of the *Python Library Reference* or use import encodings; print encodings._aliases.keys().

You can safely write Python programs in UTF-8, remembering that only variable names must be valid ASCII strings. Comments can be Greek, strings Chinese, or whatever. There is however a requirement that such a file should be either saved with an editor that prepends a Byte Order Mark (BOM), or alternatively you can make the very first code line:

```
# -*- coding: utf-8 -*-
```

Strings come with a set of methods for most useful text operations such as find(), split(), rjust() or upper(). They are implemented on the built-in str type which represents both regular and raw strings. (Raw strings interpret backslashes differently to regular strings.)

```
>>>  zen = "Explicit is better than implicit."
>>>  print zen.title()
'Explicit Is Better Than Implicit.'
>>> zen.split(' ')
['Explicit', 'is', 'better', 'than', 'implicit.']
>>> zen.rstrip('.').lower().replace('is', 'is always')
'explicit is always better than implicit'
```

One of the greatest features of Python iterable types is the method of indexing. Regular indexes starts with zero while negative indexes count backwards, so [-1] denotes the last character, [:5] the first five characters and [5:-5] means strip five leading and five trailing characters.

```
>>>  sample = "Oracle Database"
>>>  sample[0]
'O'
>>>  sample[0:6], sample[7:15]
('Oracle', 'Database')
>>>  sample[-8:]
'Database'
>>>  sample[sample.index('Data')+4:]
'base'
```

Regular Expressions

Regular expressions, of course, are supported in Python. In fact Python's regular expression re module supports Unicode, matching, searching, splitting, replacing and grouping. If you are familiar with Oracle's implementation of regular expressions you will be right at home with Python's functions.

When evaluating the Python and Oracle implementations of regular expressions head-to-head, the noticeable differences include:

re.search() can be used in place of Oracle's REGEXP_LIKE, REGEXP_INSTR and REGEXP_SUBSTR, the case is that relational design requires a different approach from programming language one.

re.sub() and REGEXP_REPLACE are equivalent to the point that Python syntax can be adapted to be used in the exact same manner as REGEXP_REPLACE. Notice however that Oracle's position parameter starts at 1 whereas Python indexes everything from 0.

Oracle's match_parameter represents a set of flags for a regular expression in the same manner Python uses the (?iLmsux) syntax inside search patterns or pattern object compilation attributes. For a list of valid flags please compare the 4.2.3 section of *Python Library Reference* with a list of valid values for the match_parameter in the *Oracle Database SQL Language Reference*.

Python's re.search() function is very flexible due to the fundamental concepts of regular expressions. At the lowest level of the re module there is an object that represents matched patterns in a manner that allows dissecting the source string in many different ways. The re.compile() function returns a compiled pattern object taking a pattern and optional flags such as re.I, which represents case-insensitive matching.

```
>>>  import re
>>>  p = re.compile("^a.*", re.I)
>>>  print p
<_sre.SRE_Pattern object at 0x011CA660>
```

You are not required to compile regular expressions explicitly. Functions in the re module do this transparently. It's good to have compiled patterns when they are to be used in several places in the code but those that are used only once don't require such coding overhead.

There are six regular expression compilation flags in Python:

I (IGNORECASE) for case-insensitive matching

L (LOCALE) makes special sequences such as words and white-space locale-dependent

M (MULTILINE) means searching for the pattern in multiple lines so that ˆ matches the start of the string and after each newline, and $ matches before each newline and the end of the string

S (DOTALL) forces dot special character (.) to match any character, including newline

U (UNICODE) makes special sequences Unicode-aware

X (VERBOSE) lets you write regular expressions in a more readable form.

To use several flags at once, simply sum them—e.g. re.compile("Oracle", re.I+re.S+re.M). Another way to use flags is to prefix the search pattern with (?iLmsux) using the desired number of options. The previous expression can be rewritten as re.compile("(?ism)Oracle").

The best advice for using regular expressions is to avoid them if possible. Before embedding them in your code, please make sure that there are no string methods that do the same job, because string methods are faster and bring no extra overhead of the import and regular expression processing. Just use dir() on a string object to see what's available.

The following example illustrates the way to think about regular expressions in a such dynamic language as Python. Consider parsing a tnsnames.ora file to create Easy Connect strings for each network alias (point the file() function to the location of your tnsnames.ora file):

```
>>> import re
>>> tnsnames = file(r'tnsnames.ora').read()
>>> easy_connects = {}
>>> tns_re = "^(\w+?)\s?=.*?HOST\s?=\s?(.+?)\).*?PORT\s?=\s?(\d+?)\).


      *?SERVICE_NAME\s?=\s?(.+?)\)"
>>> for match in re.finditer(tns_re, tnsnames, re.M+re.S):
...  t = match.groups()
...  easy_connects[t[0]] = "%s:%s/%s" % t[1:]
>>> print easy_connects
```

The output of this program on Oracle Database XE's default tnsnames.ora file is:

```
{'XE': 'localhost:1521/XE'}
```

Please note that this regular expression is dumb enough to choke on IPC entries so they need to be placed at the end of the file. Parsing matching parenthesis is one of the NP-complete problems.

Python match objects are very powerful because of the exposed methods including span(), which returns the matched range; group(), which returns a matched group by a given index; and groupdict(), which returns matched groups as a dictionary when the pattern contains named groups.

Comma Separated Values

The CSV format is popular for exchanging information between organizations due to its simplicity and cross-platform design. Comma separated values can often be easily parsed with regular expressions but the task is made even easier with Python's csv module.

Working with the module requires developers to familiarize themselves with the logic behind it. The fundamental information about a CSV file is its "dialect," which contains information about delimiters, quote characters, line terminators, etc. The currently available dialects in Python 2.5 are excel and excel-tab. The built-in sniffer always tries to guess the right format. Reader and writer objects enable input and output of CSV data.

For this example I am using data from the JOBS_HISTORY table of the HR schema. It illustrates how to create a CSV file job_history.csv directly from a SQL query.

```
>>> import csv
>>> import cx_Oracle
>>> db = cx_Oracle.connect('hr/hrpwd@localhost:1521/XE')
>>> cursor = db.cursor()
>>> f = open("job_history.csv", "w")
>>> writer = csv.writer(f, lineterminator="\n", quoting=csv.QUOTE_NONNUMERIC)
>>> r = cursor.execute(" "SELECT * FROM job_history ORDER BY employee_id, start_date")
>>> for row in cursor:
...   writer.writerow(row)
...
>>> f.close()
```

The file contains:

```
101,"1989-09-21 00:00:00","1993-10-27 00:00:00","AC_ACCOUNT",110
101,"1993-10-28 00:00:00","1997-03-15 00:00:00","AC_MGR",110
102,"1993-01-13 00:00:00","1998-07-24 00:00:00","IT_PROG",60
114,"1998-03-24 00:00:00","1999-12-31 00:00:00","ST_CLERK",50
122,"1999-01-01 00:00:00","1999-12-31 00:00:00","ST_CLERK",50
176,"1998-03-24 00:00:00","1998-12-31 00:00:00","SA_REP",80
176,"1999-01-01 00:00:00","1999-12-31 00:00:00","SA_MAN",80
```

```
200,"1987-09-17 00:00:00","1993-06-17 00:00:00","AD_ASST",90
200,"1994-07-01 00:00:00","1998-12-31 00:00:00","AC_ACCOUNT",90
201,"1996-02-17 00:00:00","1999-12-19 00:00:00","MK_REP",20
```

Alternatively you could export the data in CSV format using Oracle SQL Developer.

The CSV file can be read with:

```
>>>  reader = csv.reader(open("job_history.csv", "r"))
>>>  for employee_id, start_date, end_date, job_id, department_id in reader:
...   print job_id,
...
JOB_ID IT_PROG AC_ACCOUNT AC_MGR MK_REP ST_CLERK ST_CLERK


  AD_ASST SA_REP SA_MAN AC_ACCOUNT
```

Note that I didn't have to specify the dialect explicitly above; it was automatically deduced. I only printed the job_id column but what I really could do with such parsed file is insert it into the database. To make sure dates are handled correctly, NLS_DATE_FORMAT is set manually before bulk insertion.

```
SQL> CREATE TABLE job_his (
  2     employee_id   NUMBER(6)    NOT NULL,
  3     start_date    DATE         NOT NULL,
  4     end_date      DATE         NOT NULL,
  5     job_id        VARCHAR2(10) NOT NULL,
  6     department_id NUMBER(4)
  7 );

>>> reader = csv.reader(open("job_history.csv", "r"))
>>> lines = []
>>> for line in reader:
...   lines.append(line)
...
>>> cursor.execute("ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS'")
>>> cursor.executemany("INSERT INTO job_his VALUES(:1,:2,:3,:4,:5)", lines)
>>> db.commit()
```

If you used SQL Developer to create the CSV file, you may need to change the date format instead like:

```
>>> cursor.execute("ALTER SESSION SET NLS_DATE_FORMAT = 'YY/MM/DD'")
```

What takes the csv module a bit imperfect is the lack of native Unicode support. For a solution and more examples of working with CSV files, see the 9.1.5 Examples section of the *Python Library Reference*.

URLs

The urlparse module lets you break Uniform Resource Locator strings into components representing the URL scheme, network location, path, parameters, query string, fragment identifier, username, password, hostname and/or port. Python 2.5 supports as many as 24 of the most popular schemes, including svn+ssh, sftp and mms. This example shows some features of urlparse module:

```
>>>  from urlparse import urlparse
>>>  url = "http://www.oracle.com/technology/index.html?rssid=rss_otn_news#section5"
>>>  pr = urlparse(url)
>>>  print type(pr)
<class 'urlparse.ParseResult'>
>>>  print pr.hostname
www.oracle.com
>>>  print pr.query
rssid=rss_otn_news
>>>  print url==pr.geturl()
True
```

RSS Feeds

The concept underlying RSS is quite simple: You get the latest news as it happens, not by spotting it accidentally. Consolidating RSS feeds from many different sources is a popular trend in development, especially for news feeds aggregators and Web 2.0 mashups.

RSS is a dialect of XML so it could be easily processed with one of the XML parsers available for Python. The Python standard library doesn't offer a module for parsing feeds natively yet; however, there is a solid, widely-tested Universal Feed Parser available for free at feedparser.org. As it has no external dependencies, this is a good chance to quickly familiarize ourselves with module installation concepts.

After downloading the latest version (4.1 at the time of writing) of the feedparser module, unpack it and change the working directory to feedparser-4.1. At the console/command prompt run `python setup.py install`. This command will put the module into the Python folder and make it available for instant use. That's it.

How about checking what happened at Oracle lately?

```
>>> import feedparser
>>> import time
>>> rss_oracle = feedparser.parse("http://www.oracle.com/technology/syndication/rss_otn_news.xml")
>>> for e in rss_oracle.entries[:5]:
..    t = time.strftime("%Y/%m/%d", e.updated_parsed)
..    print t, e.title

2007/07/23 Integrating Oracle Spatial with Google Earth
2007/07/11 Oracle Database 11g Technical Product Information Now Available
2007/07/11 Explore the Oracle Secure Enterprise Search Training Center
```

```
2007/07/09 Implementing Row-Level Security in Java Applications
2007/06/29 Build Your Own Oracle RAC Cluster on Oracle Enterprise Linux and iSCSI
```

The feedparser module is smart enough to properly parse the date, sanitize HTML markup, normalize content so a consistent API for all supported RSS and ATOM variants can be used, resolve relative links, detect valid character encoding, and much more.

Parse What Next?

Equipped with a regular expression toolbox you can search for nearly any content as long as it is plain text. When it comes to parsing text data, Python has many other features including:

email.parse for parsing e-mail messages
ConfigParser for parsing INI configuration files known from Windows systems
robotparser module for parsing robots.txt of your Web sites
optparse module for powerful command line argument parsing
HTMLParse class in the HTMLParse module for parsing HTML and XHTML effectively (SAX-like)
Several XML parsers (xml.dom, xml.sax, xml.parsers.expat, xml.etree.ElementTree)

For binary data you can leverage the binascii module, which contains a set of functions for converting between binary- and ASCII-encoded data, accompanied by the base64 and uu modules for base64 and uuencode transformations, respectively.

Conclusion

This HowTo introduced some basic and more advanced techniques for parsing data in Python. You should now be aware of the power of the standard library that Python ships with. Before starting cooking your own parser, the first step is to check if the desired functionality is already available for import.

String operations are faster than regular expression operations and sufficient for many programming needs. But the decision whether to use Python or Oracle regular expression functions depends on your application logic and business requirements.

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.