Mastering Oracle+Python, Part 8: Python for Oracle DBAs

by Przemyslaw Piotrowski

Achieve extreme database management productivity with rapid prototyping in Python.

Published December 2011

See series TOC

Traditionally, Bash or Perl are the tools of choice when operating systems need some scripting. Given their ease of use, they have become virtually ubiquitous and seeped into other software, including Oracle Database - which relies on them extensively for all kinds of administrative and management tasks.

Recently, however, this trend has shifted in favor of newer programming tools like Python, which offers intuitive development and a variety of flexible data structures and libraries. All modern Unix and Linux systems come with Python on board; for example, Oracle Linux 6.1 ships with Python 2.6.6.

This tutorial explores Python features especially useful to database administrators for either implementing one-off code snippets or fully reusable programs. In this installment, we'll delve into interaction with operating system and remote resources, and then look into various compression and filesystem traversal modules.

For the purposes of this tutorial, we'll be using Oracle Database 11*g* Express Edition (XE) on top of Oracle Linux 6.1 and Python 2.6.6.

Interacting with Filesystems

The core library for interacting with operating systems is the os module, with which you can handle system processes, recognize platforms, deal with OS pipes, and work with environment variables - in the form of over a hundred functions and variables.

Detecting the current platform is as easy as reaching to a predefined string in the os module. The following example illustrates the outcome on Oracle Linux 6.1 and also shows the default path separator for this OS.

```
>>> import os
>>> os.name
'posix'
>>> os.sep
'/'
```

A list of all Oracle's environment variables is accessible through os.environ. The following example makes use of an inline generator expression:

```
>>> import os
>>> oracle_vars = dict((a,b) for a,b in os.environ.items() if a.find('ORACLE')>=0)
>>> from pprint import pprint
>>> pprint(oracle_vars)
{'ORACLE_HOME': '/u01/app/oracle/product/11.2.0/xe', 'ORACLE_SID': 'XE'}
```

which would correspond to

```
SELECT key, value
FROM os.environ
WHERE key LIKE '%ORACLE%'
```

if written in SQL.

As we probe further, we begin checking the filesystem and looking at where we are. The table below lists the most common filesystem access functions and their descriptions.

Function	Role
os.getcwd()	Gets the current working directory from OS
os.chdir(*path*)	Change directory to the given *path*
os.chroot(*path*)	Change the root path for the current Python process to *path*
os.chown(*path*, *uid*, *gid*)	Same as *chmod* Linux command (*uid* and *gid* are numbers)
os.listdir(*path*)	List files and directories under the given *path*
os.mkdir(*path*, *mode*)	Creates directory under given *path* with octal permissions set to *mode* (0777 by default)
os.remove(*path*)	Deletes a file under *path*
os.rmdir(*path*)	Removes the directory under *path*
os.rename(*path*, *newpath*)	Renames *path* to *newpath*
os.stat(*path*)	Shows attributes of *path* using the OS *stat()* call
os.walk(*path*, *topdown*, *onerror*, *followlinks*)	Returns generator returning tuples of (path, directories, files) for the filesystem tree under *path*

Having grasped the basic functionality behind filesystem browsing, let's take a look how Python can be used to quickly go through a list of old trace files and "un-rotated" logs and show how much space they use. The program in Listing 1 takes two arguments: a path for Oracle logs (the directory

that DIAGNOSTIC_DEST points at) and the number of days for a file to be considered obsolete. This example is based on *os.walk*.

Listing 1. walk.py: old log and trace files under the Oracle diagnostic directory

```
import datetime
import os
import sys
import time
from pprint import pprint

def readable(size):
  si=('B','KB','MB','GB','TB', 'PB', 'EB', 'ZB', 'YB')
  div = [n for n, m in enumerate(si) if pow(1024, n+1)>size][0]
  return "%.1f%s"%(size/float(pow(1024, div)), si[div])

total = {"log":0, "trace":0}
for path, dirs, files in os.walk(sys.argv[1]):
  for f in files:
    filepath = path+os.sep+f
    if os.stat(filepath).st_mtime>time.time()-(3600*24*int(sys.argv[2])):
      size = readable(os.path.getsize(filepath))
      age = datetime.datetime.fromtimestamp(os.stat(filepath).st_mtime)
      if f in ("log.xml", "alert.log", "listener.log"):
        filetype = "log"
      elif f.endswith("trc") or f.endswith("trm"):
        filetype = "trace"
      else:
        filetype = None
      if filetype:
        total[filetype] += os.path.getsize(filepath)

for a, b in total.items():
  total[a] = readable(b)

pprint(total)
```

Running walk.py gives output similar to:

```
$ python walk.py /home/oracle/app 10
{'log': '132.0MB', 'trace': '0.0B'}
```

Within the os namespace there is another module that addresses path name manipulations called os.path. It contains platform-sensitive implementations for different systems, so importing os.path will always get the right version for your operating system.

Commonly used function from the os.path module include:

basename(path,) for getting the leaf name of given path
dirname(path), for getting the directory part of a file path; it is supplemented by the split(path) function returning a tuple containing separated directory and file parts
exists(path), to check if a file under path exists, returning False for unresolvable symbolic links
getsize(path), for quickly checking the number of bytes under a path
isfile(path) and isdir(path) to resolve the path type

Even though we've seen some extensive filesystem browsing capabilities so far, we've only scratched the surface as there are multiple other modules available. For example, the filecmp module is capable of comparing both files and directories, tempfile provides easy temporary file management, glob resolves file paths matching a Unix-style pattern (as in ora_pmon_*.trc, log_*.xml, etc.), and the very useful shutil module implements high-level filesystem operations like copying and removing multiple files or whole file trees.

Talking Processes

The os module is not just restricted to file management. It can also be used to interact with and spawn system processes, and also to perform system kill and nice calls. The table below lists the most useful process management functions. These are only valid for Unix and Linux platforms, though there is some work under way to get them working on Windows in the Python 3.2 branch.

Function	Role
os.abort()	Sends SIGABRT to the current Python process
os.exec*(*path*, *arg1...argN*, *environ*)	Family of exec* functions to replace the current process with the one specified by *path*, optionally providing command line arguments and environment variables
os.kill(*pid*, *signal*)	Sends *signal* to a given *pid*
os.nice(*value*)	Changes the current process's *nice* value
os.popen(*command*, *mode*, *buffersize*)	Opens an unnamed pipe to a given *command*, effectively enabling further interaction with the process; *mode* denotes the pipe open handle attribute ('r' for read by default)
os.spawn*(*mode*, *path*, *environ*)	Runs the program under *path* in a new process (these are now obsoleted by the *subprocess* module)
os.system(*command*)	This runs a new process defined by the *command* using OS *system()* call; available on Unix and Windows

While many of these functions might come in handy on older Python releases, starting with version 2.4, there is a dedicated *subprocess* module created specifically with process management in mind. Initially submitted to the Python Enhancement Proposal Index (PEP) in 2003, the new module is now the preferred way of communicating with system processes.

Subprocess replaces the os.popen, *os.spawn**, and *os.system* functions with a usable, straightforward interface that is also quite versatile. Listing 2

shows the code for the ps.py program, which executes a `ps aux` command and moves the results into a Python dictionary. A pipe is used as the target for stdout to capture all information and suppresses output to the screen.

Listing 2. ps.py: moving the system process map into a Python dictionary

```
import re
import subprocess

args = ['ps', 'aux']
ps = subprocess.Popen(args, stdout=subprocess.PIPE)
processes = ps.stdout.readlines()
header = re.split('\s+', processes.pop(0))[:-1]
header.remove('COMMAND')

PS = {}
for process in processes:
  columns = re.split('\s+', process)
  if columns[0]!='oracle':
       continue
  PS[int(columns[1])] = {}
  for position, column in enumerate(columns[:9]):
       PS[int(columns[1])][header[position].lower()] = column
       PS[int(columns[1])]['command'] = ' '.join(columns[10:])

from pprint import pprint
pprint(PS)
```

The output is similar to:

```
...
 25892: {'%cpu': '0.0',
       '%mem': '3.9',
       'command': 'xe_w000_XE ',
       'pid': '25892',
       'rss': '23672',
       'start': '16:02',
       'stat': 'Ss',
       'tty': '?',
       'user': 'oracle',
       'vsz': '457240'},
 26142: {'%cpu': '2.0',
       '%mem': '0.9',
       'command': 'python proc.py ',
       'pid': '26142',
       'rss': '5732',
       'start': '16:36',
       'stat': 'S+',
       'tty': 'pts/2',
       'user': 'oracle',
       'vsz': '160776'},
 26143: {'%cpu': '0.0',
       '%mem': '0.1',
       'command': 'ps aux ',
       'pid': '26143',
       'rss': '1100',
       'start': '16:36',
       'stat': 'R+',
       'tty': 'pts/2',
       'user': 'oracle',
       'vsz': '108044'}}
```

The *popen* function accepts a number of keyword arguments like *stdin/stdout/stderr* descriptors, cwd for setting the working directory for a process, or *env* which sets the environment variables of the child process. To check the status of a command, you just peek at the *returncode* attribute. The process identifier is available under the *pid* property.

Methods on an already created process include *poll()* for checking whether it is still running, *wait()* for resuming upon program completion, send_signal() for sending a particular signal, and *terminate()* or *kill()* for sending SIGTERM or SIGKILL signals, respectively. Finally, to fully interact with the spawned child process, we use the communicate() function to send stdin input.

To illustrate this, let's create a simple SQL*Plus wrapper that thrives on a bequeathed SYSDBA connection.

Listing 3. sp.py: communicating with a SQL*Plus process from Python

```
import os
from subprocess import Popen, PIPE

sqlplus = Popen(["sqlplus", "-S", "/", "as", "sysdba"], stdout=PIPE, stdin=PIPE)
sqlplus.stdin.write("select sysdate from dual;"+os.linesep)
sqlplus.stdin.write("select count(*) from all_objects;"+os.linesep)
out, err = sqlplus.communicate()
print out
```

This return output similar to:

```
SYSDATE
--------------
02-DEC-11

COUNT(*)
--------------
76147
```

A Reporting Service

One of the most daunting tasks that involves stepping out of the database is sending alerts or pushing out recurring reports pulled from a data warehouse. The good news is that not only has Python been used to implement one of the world's popular mailing list systems - Mailman - but that it also offers a rich email handling library supporting MIME, attachments, message encoding, and literally every aspect related to processing electronic

mail. The email module separates the protocol intrinsics from presentation layer to focus purely on constructing messages, while the delivery work is handled via the smtplib module.

The Message class from email.message represents the core class for working with emails. Handlers from the email.mime namespace are used to deal with different attachment types. In this example however, we'll use the most generic one: MIMEBase from email.mime.base. There will be also some cheating on our part, cashing in on the fact that spreadsheet software will open HTML files in tabular format if they have an .xls extension. We will also take advantage of the help of the tempfile module.

Oracle Linux doesn't have the cx_Oracle module preinstalled, so you'll need to get it from cx-oracle.sourceforge.net. Also, to be able to import cx_Oracle and use network configuration files, ORACLE_HOME and LD_LIBRARY_PATH need to be set up before launching the Python interpreter.

```
[root@xe ~]# rpm -ivh cx_Oracle-5.1-11g-py26-1.x86_64.rpm
Preparing... ########################################### [100%]
1:cx_Oracle ########################################### [100%]
[root@xe ~]#
[root@xe ~]# su - oracle
[oracle@xe ~]$ export ORACLE_HOME=/u01/app/oracle/product/11.2.0/xe
[oracle@xe ~]$ export LD_LIBRARY_PATH=$ORACLE_HOME/lib
```

See Listing 4 for a complete program that connects to Oracle Database 11*g* XE, fetches employee data, and packages it as a spreadsheet attachment sent to an email group.

Listing 4. report.py: an email reporting service

```
import cx_Oracle
import datetime
import smtplib
import tempfile
from email.message import Message
from email.encoders import encode_base64
from email.mime.base import MIMEBase
from email.mime.multipart import MIMEMultipart

today = datetime.datetime.now()

msg = MIMEMultipart()
msg['From'] = 'Reports Service <reports@company.intranet>'
msg['To'] = 'receipients@company.intranet'
msg['Subject'] = 'Monthly employee report %d/%d ' % (today.month, today.year)

db = cx_Oracle.connect('hr', 'hrpwd', 'localhost/xe')
cursor = db.cursor()
cursor.execute("select * from employees order by 1")
report = tempfile.NamedTemporaryFile()
report.write("<table>")
for row in cursor:
  report.write("<tr>")
  for field in row:
    report.write("<td>%s</td>" % field)
  report.write("</tr>")
report.write("</table>")
report.flush()
cursor.close()
db.close()

attachment = MIMEBase('application', 'vnd.ms-excel')
report.file.seek(0)
attachment.set_payload(report.file.read())
encode_base64(attachment)
attachment.add_header('Content-Disposition', 'attachment;filename=emp_report_%d_%d.xls' % (today.month,
today.year))
msg.attach(attachment)

emailserver = smtplib.SMTP("localhost")
emailserver.sendmail(msg['From'], msg['To'], msg.as_string())
emailserver.quit()
```

If we were to take this example even further, we could use the Python Imaging Library (PIL) to grab statistic charts, attach thumbnails of BLOBs stored in a database, or generate PDF reports with ReportLab to be sent across groups of interest. The email module is powerful enough to handle every possible scenario.

Wrapping Up

Python's extensive library of crossplatform modules will definitely complement a DBA's portfolio of technologies used for watching over the entire database stack, offering rapid development speed with little upkeep overhead. The ubiquity of Python, shipping in every modern Linux platform, could further increase its adoption rate and over time, help it become the new language of choice for all database administration needs.

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.