Mastering Oracle+Python, Part 5: Stored Procedures, Programming Python
by Przemyslaw Piotrowski

Calling database stored procedures and other interesting aspects of advanced Python programming.

Published March 2010

See series TOC

There are two mainstream approaches when it comes to software development engaging the database: you can implement all the business logic on the application side (or in the middleware when speaking of 3-tier architectures) or inside the database. Bringing on pros and cons for both of these solutions is not the subject of this tutorial; nevertheless, going the Oracle Database path brings certain advantages to a database-oriented application.

Embedding all the business logic in PL/SQL minimizes the number of round trips between application and database so that processing takes place on the server side. PL/SQL is closely integrated with SQL and, similarly to Python, comes with an extensive standard library of packages ranging from scheduling database jobs (DBMS_SCHEDULER), through automatic query tuning (DBMS_SQLTUNE) and flashback (DBMS_FLASHBACK), up to linear algebra (UTL_NLA) and LDAP access (DBMS_LDAP).

This tutorial introduces methods for calling stored PL/SQL procedures and functions inside the Oracle Database from Python using the cx_Oracle module as well as some aspects of programming that are either unachievable or very complex to code in PL/SQL. At the end we are going to touch on Oracle Berkeley DB, which comes built-in into Python out-of-the-box.

The IN-OUT Game
Oracle procedures and functions are database objects that combine SQL capabilities with programming language functionality. Arguments of procedures (also referring to functions from now on) can be one of the three types:

IN: passed to procedures, cannot be written to inside the procedure
OUT: returned from procedures, writable from within the procedure body
IN OUT: passed to procedures and perfectly writable inside the procedure
By default, arguments are of IN type.
For illustrating the interaction between Python and Oracle procedures consider the package below to be installed in the HR schema of Oracle Database XE instance.

```
 CREATE OR REPLACE PACKAGE pkg_hr AS

  PROCEDURE add_department(
    p_department_id OUT NUMBER,
    p_department_name IN VARCHAR2,
    p_manager_id IN NUMBER,
    p_location_id IN NUMBER
  );

  FUNCTION get_employee_count(
    p_department_id IN NUMBER
  ) RETURN NUMBER;

  PROCEDURE find_employees(
    p_query IN VARCHAR2,
    p_results OUT SYS_REFCURSOR
  );

END pkg_hr;
/

CREATE OR REPLACE PACKAGE BODY pkg_hr AS

  PROCEDURE add_department(
    p_department_id OUT NUMBER,
    p_department_name IN VARCHAR2,
    p_manager_id IN NUMBER,
    p_location_id IN NUMBER
  ) AS
  BEGIN
    INSERT INTO departments(department_id, department_name, manager_id, location_id)
    VALUES (departments_seq.nextval, p_department_name, p_manager_id, p_location_id)
    RETURNING department_id
    INTO p_department_id;

    COMMIT;
  END add_department;

  FUNCTION get_employee_count(
    p_department_id IN NUMBER
  ) RETURN NUMBER AS
    l_count NUMBER;
  BEGIN
    SELECT COUNT(*)
    INTO l_count
    FROM employees
    WHERE department_id= p_department_id;

    RETURN l_count;
  END get_employee_count;

 PROCEDURE find_employees(
    p_query IN VARCHAR2,
```

```
    p_results OUT SYS_REFCURSOR
  ) AS
  BEGIN
    OPEN p_results FOR
      SELECT *
      FROM employees
      WHERE UPPER(first_name||' '||last_name||' '||email) LIKE '%'||UPPER(p_query)||'%';
  END find_employees;

END pkg_hr;
/
```

Three different access methods are introduced in the above example: a procedure with IN and OUT parameters, a function returning a number and a procedure with an OUT REF CURSOR parameter. Each one requires a different way of calling as shown below.

```
import cx_Oracle

class HR:
  def __enter__(self):
    self.__db = cx_Oracle.Connection("hr/hrpwd@//localhost:1521/XE")
    self.__cursor = self.__db.cursor()
    return self

  def __exit__(self, type, value, traceback):
    self.__cursor.close()
    self.__db.close()

  def add_department(self, p_department_name, p_manager_id, p_location_id):
    l_department_id = self.__cursor.var(cx_Oracle.NUMBER)
    self.__cursor.callproc("PKG_HR.ADD_DEPARTMENT",
      [l_department_id, p_department_name, p_manager_id, p_location_id])

    # there are no OUT parameters in Python, regular return here
    return l_department_id

  def get_employee_count(self, p_department_id):
    l_count = self.__cursor.callfunc("PKG_HR.GET_EMPLOYEE_COUNT",
      cx_Oracle.NUMBER, [p_department_id])
    return l_count

  def find_employees(self, p_query):
    # as it comes to all complex types we need to tell Oracle Client
    # what type to expect from an OUT parameter
    l_cur = self.__cursor.var(cx_Oracle.CURSOR)
    l_query, l_emp = self.__cursor.callproc("PKG_HR.FIND_EMPLOYEES", [p_query, l_cur])
    return list(l_emp)
```

Based upon the above example, calling stored procedures from Python is regulated through some basic rules:

Procedures are called with cx_Oracle.Cursor.callproc(proc, [params]) whereas functions with cx_Oracle.Cursor.callfunc(proc, returnType, [params]). Functions require their return type to be defined in advance - the get_employee_count() method declares the return type from PKG_HR.GET_EMPLOYEE_COUNT to be a cx_Oracle.NUMBER
Complex types such as REF CURSORs can be returned using cx_Oracle Variable objects as parameters in callproc/callfunc calls.

Passing Arrays with arrayvar
Another extension of the DB API 2.0 in cx_Oracle enables use of arrays as parameters in stored procedure calls. Currently supported are PL/SQL arrays with an INDEX BY clause. As an example of using arrayvar objects make sure the DDL below finds its way into the database.

```
CREATE OR REPLACE PACKAGE pkg_arrayvar AS
  TYPE num_array IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  FUNCTION sum(p_list IN NUM_ARRAY) RETURN NUMBER;
END pkg_arrayvar;
/

CREATE OR REPLACE PACKAGE BODY pkg_arrayvar AS

FUNCTION sum(p_list IN NUM_ARRAY) RETURN NUMBER AS
    l_sum NUMBER := 0;
  BEGIN
    FOR i IN 1..p_list.COUNT LOOP
      l_sum := l_sum+p_list(i);
    END LOOP i;

    RETURN l_sum;
  END sum;

END pkg_arrayvar;
/
```

Now the declaration for the Python object and the actual call to the function goes like this (followed by an assertion to verify the result):

```
>>> db = cx_Oracle.connect("hr/hrpwd@//localhost:1521/XE")
>>> cursor = db.cursor()
>>> L = cursor.arrayvar(cx_Oracle.NUMBER, [1, 2, 3])

>>> sum_result = cursor.callfunc("pkg_arrayvar.sum", cx_Oracle.NUMBER, [L])
>>> assert sum_result==6
```

As mentioned above, calling functions from Python requires the return type to be declared explicitly, this might be a little confusing since callproc() requires only two arguments but it's just the way it has to be.

Go Python
PL/SQL is a powerful language and combined with the possibilities of Oracle Database can dramatically reduce development efforts, enabling you to take advantage of the majority of database features. Yet some programming aspects just cannot be expressed or used with its in-database nature. And when there's a need to complement it with another programming language, Python is a great choice for short development time and fast results.

Multiprocessing

Parallel processing in Python, starting with version 2.6, is no longer limited by the GIL (Global Interpreter Lock). The threading module shipping with the standard library is limited to running one operation at a time. By substituting threads with operating system processes, it is now possible to leverage all the CPUs available to the application and therefore perform truly parallel computations. The Multiprocessing module lets applications spawn new processes, lock objects, share them in memory and what's more, all of this locally and remotely (yes, on different machines).

See below for an example of a simple database benchmark utility.

```
import cx_Oracle
import os
import time
from multiprocessing import Pool
from optparse import OptionParser

def benchmark(options):
  params = eval(options.bind) if options.bind else {}
  with cx_Oracle.connect(options.db) as db:
    try:
      cursor = db.cursor()
      before = time.clock()
      for i in xrange(options.requests):
        cursor.execute(options.sql, params)
      return (time.clock()-before)/options.requests
    except KeyboardInterrupt:
      pass
    finally:
      cursor.close()

class Orabench:
  def __init__(self, options):
    self.options = options
    print "Requests=%d, Concurrency=%d" % (self.options.requests,
                                           self.options.concurrency)

  def run(self):
    pool = Pool(processes=self.options.concurrency)
    result = pool.map_async(benchmark, [self.options]*self.options.concurrency)
    L = result.get()
    avg = sum(L)/len(L)
    print "Average=%.4f (%.4f requests per second)" % (avg, 1/avg)

if __name__ == "__main__":
  opt = OptionParser()
  opt.add_option("-d", "--database", help="EZCONNECT string", action="store",
                 type="string", dest="db")
  opt.add_option("-n", "--requests", help="number of requests", action="store",
                 type="int", dest="requests", default=10)
  opt.add_option("-c", "--concurrency", help="number of concurrent connections",
                 action="store", type="int", dest="concurrency", default=1)
  opt.add_option("-s", "--sql", help="SQL query or PL/SQL block",
                 action="store", type="string", dest="sql")
  opt.add_option("-b", "--bind", help="dictionary of bind parameters",
                 action="store", type="string", dest="bind")
  (options, args) = opt.parse_args()
  bench = Orabench(options)
  bench.run()
```

By taking advantage of the optparse module which makes an excellent job parsing the command line arguments, this tool automatically generates usage instructions when invoked with "--help" switch.

```
pp@oel:~$ python26 orabench.py --help
Usage: orabench.py [options]

Options:
  -h, --help            show this help message and exit
  -d DB, --database=DB  EZCONNECT string
  -n REQUESTS, --requests=REQUESTS
                        number of requests
  -c CONCURRENCY, --concurrency=CONCURRENCY
                        number of concurrent connections
  -s SQL, --sql=SQL     SQL query or PL/SQL block
  -b BIND, --bind=BIND  dictionary of bind parameters
```

Then, benchmarking a query 1000 times in 10 processes using the HR schemabecomes:

```
pp@oel:~$ python26 orabench.py -d hr/hrpwd@//localhost:1521/XE -n 1000 -c 10 -s "select count(*) from employees"
Requests=1000, Concurrency=10
Average=0.0006 (1667.7460 requests per second)
```

GROUP BY Outside the Database (Functional Programming)

For various aspects of functional programming there are very few better modules than the itertools. It comprises a number of traversing functions that yield custom, optimized iterators. Just as a reminder, iterators are objects where the __iter__() method returns the iterator itself and the next() method either steps to the subsequentelement or raisesg the StopIteration exception to complete the iteration. One can spot the difference in using iterators vs lists or tuples by looping through large data sets, because they basically avoid rendering the whole collection in memory.

```
import cx_Oracle
import itertools
from operator import itemgetter

with cx_Oracle.connect("hr/hrpwd@//localhost:1521/XE") as db:
  cursor = db.cursor()
  # fetch all employee data into local variable, no aggregation here
  employees = cursor.execute("select * from employees").fetchall()

D = {}
```

```
for dept, emp in itertools.groupby(employees, itemgetter(10)):
  D[dept] = len(list(emp))
```

The operator module includes all the core operators that native objects use, meaning that whenever you run 2+2, the operator.add() method handles the calculation. As the itertools.groupby() method accepts two parameters: iterable and key function, we need to extract the department_id from all rows using itemgetter(10) which simply returns the 10th element of a collection. Looping through results of itertools closely resembles the one you use for lists, tuples and dictionaries. For each department we generate its ID and a number of all employees in it (SELECT department_id, COUNT(*) FROM employees GROUP BY department_id).

Serializing Data

In Python, data serialization and de-serialization is handled by the pickle module and its C counterpart cPickle (up to 1000x faster than the native Python implementation of pickle). "Pickling" objects means converting them into a reversible byte representation:

```
>>> import pickle
>>> A = {'a':1, 'b':2, 'c':3}
>>> B = pickle.dumps(A)
>>> print B
"(dp0\nS'a'\np1\nI1\nsS'c'\np2\nI3\nsS'b'\np3\nI2\ns."
>>> C = pickle.loads(B)
>>> assert A==C
```

Pickling proves especially useful when storing complex structures next to relational data so that we can read and write regular Python objects as if they were natively supported by the database.

There are only few limits when it comes to types supported by pickle, since it can handle everything from dictionaries and tuples, through sets and functions, up to classes and instances. One of the objects that cannot be pickled is cx_Oracle.Connection objects, for obvious reasons.

Sleepy Cache

Oracle Berkeley DB is a transactional key-value storage solution with fine-grained locking, high availability, and replication. It fits in all those problems where extreme efficiency is required and the overhead of a full size relational database is too high. (Up to version 2.6, Python includes a built-in interface for Oracle Berkeley DB in form of bsddb module. New versions of Python, starting with 3.0, rely on an external module called PyBSDDB which needs to be installed separately.)

Next, we are going to make use of the built-in driver that comes with Python 2.6 to cache values from an Oracle database in Oracle Berkeley DB:

```
import bsddb
import cx_Oracle
import pickle

class Cache:
  def __init__(self, tab):
    self.__db = cx_Oracle.connect("hr/hrpwd@//localhost:1521/XE")
    self.__cursor = self.__db.cursor()
    self.__bdb = bsddb.hashopen(None)
    self.__cursor.execute("select * from employees")
    d = self.__cursor.description
    for row in self.__cursor:
      rowdict = dict((d[i][0].lower(), row[i]) for i in xrange(len(d)))
      self.__bdb[str(row[0])] = pickle.dumps(rowdict)

  def __del__(self):
    self.__cursor.close()
    self.__db.close()

  def __getitem__(self, name):
    try:
      return pickle.loads(self.__bdb[str(name)])
    except KeyError:
      raise Warning, "No such employee with ID %s" % name

if __name__ == "__main__":
  emp = Cache("EMPLOYEES")
```

Accessing employees is now as easy as using "emp[100]" which accesses the fast, in-memory hash table and un-pickles the serialized employee data. You can easily wrap such a cache with one of the built-in servers (SimpleHTTPServer, SimpleXMLRPCServer, wsgiref.simple_server) or use the Twisted Framework to make it even more robust.

Conclusion

This time we covered several core areas of the Oracle-Python bridge including PL/SQL stored procedure calls and handling PL/SQL function results. When it comes to Python, you should now be familiar with the essentials of the multiprocessing module which is one of the most important recent additions to the language. Finally, Oracle Berkeley DB was used in a proof-of-concept in-memory cache scenario.

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.