

Application Development Framework

Application Express

Big Data

Business Intelligence

Cloud Computing

Communications

Database Performance &amp; Availability

Data Warehousing

Database

.NET

Dynamic Scripting Languages

Embedded

Digital Experience

Enterprise Architecture

Enterprise Management

Identity &amp; Security

Java

Linux

Mobile

Service-Oriented Architecture

Solaris

SQL &amp; PL/SQL

Systems - All Articles

Virtualization

# Mastering Oracle+Python, Part 4: Transactions and Large Objects

by Przemyslaw Piotrowski

## Managing database transactions and handling Large Objects from Python

Published March 2010

[See series TOC](#)

Transactions encompass a set of SQL statements that constitute a single logical operation inside the database, e.g. a money transfer or a credit card payment. The whole point of aggregating SQL statements into logical groups depends on the fact that the transaction either succeeds, committing the changes, or fails, withdrawing results of the inner SQL, as a whole. With Python you can take advantage of all the benefits of atomicity, consistency, isolation and durability which Oracle database has to offer.

Large Objects enable storing huge amounts of data in a single column (up to 128TB as of Oracle Database 11g) but such flexibility comes at a cost, as techniques for accessing and manipulating LOBs are different from regular query methods.

Note: Python's 2.x branch has been upgraded to 2.6 and cx\_Oracle module advanced to 5.0. These will be ones used in MO+P from now on. Moreover, this tutorial remains being based on the HR schema available in Oracle Database 10g Release 2 Express Edition.

### It's All About ACID

A database transaction is a logical group of statements characterized by four essential properties:

Atomicity: all of the operations succeed or fail as a whole

Consistency: committing transaction cannot result in data corruption

Isolation: other transactions remain unaware of transaction's execution

Durability: operations committed within transaction will prevail database crash.

The Python Oracle database API offers a natural way of handling transactions so that logical operations on data could be stored inside the database rollback segments waiting for a final decision to commit or roll back the whole group of statements.

A transaction is started with the first SQL statement reaching the database via the cursor.execute() method. A new transaction can be started explicitly using the db.begin() method, under the condition that no other transaction hasn't been started from the session. For maximum consistency cx\_Oracle rolls back all transactions by default when the connection object is closed or deleted.

The cx\_Oracle.Connection.autocommit attribute can still be set to 1 making Oracle commit every single statement issued through the cursor.execute\* family of methods. Another thing developers should be aware of is the fact that because Oracle's DDLs are not transactional, all DDL statements implicitly commit. And lastly, contrary to SQL\*Plus, closing the connection with db.close() doesn't commit the on-going transaction.

There are several Oracle transactional statements that are wrapped by cx\_Oracle like db.begin(), db.commit() and db.rollback() but you can use the rest of them by explicitly calling the SET TRANSACTION statement. A new transaction starts with the first DML in the session anyway, so there's no need to do this explicitly—unless you need to take advantage of specific transaction characteristics such as SET TRANSACTION [ READ ONLY | ISOLATION LEVEL SERIALIZABLE ].

Let's consider a class for performing HR operations of transactional characteristics:

```
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.__db.close()

    def swapDepartments(self, employee_id1, employee_id2):
        assert employee_id1!=employee_id2

        select_sql = """select employee_id, department_id from employees
        where employee_id in (:1, :2)"""
        update_sql = "update employees set department_id=:1 where employee_id=:2"

        self.__db.begin()
        self.__cursor.execute(select_sql, (employee_id1, employee_id2))
        D = dict(self.__cursor.fetchall())
        self.__cursor.execute(update_sql, (D[employee_id2], employee_id1))
        self.__cursor.execute(update_sql, (D[employee_id1], employee_id2))
        self.__db.commit()

    def raiseSalary(self, employee_ids, raise_pct):
        update_sql = "update employees set salary=salary*(:1) where employee_id=:2"

        self.__db.begin()

        for employee_id in employee_ids:
            try:
                self.__cursor.execute(update_sql, [1+raise_pct/100, employee_id])
                assert self.__cursor.rowcount==1
            except AssertionError:
                self.__db.rollback()
                raise Warning, "invalid employee_id (%s)" % employee_id
```

```

        self.__db.commit()

if __name__ == "__main__":
    with HR() as hr:
        hr.swapDepartments(106, 116)
        hr.raiseSalary([102, 106, 116], 20)

```

The two methods defined above enable swapping of departments between given employees and giving a raise to an arbitrary number of employees. Such operations, to be considered safe, can only be achieved through transactions. To make sure no other transaction takes place there's an explicit call to `self.__db.begin()` in both these methods (which would raise `DatabaseError ORA-01453` otherwise). Double underscore prefix has special meaning in Python since the language doesn't actually allow you to declare private variables (everything in a class is public), but it makes variables harder to access; they are exposed via `__Class__Member` syntax (`__HR__db` and `__HR__cursor` in the above example, respectively). At the same time, we declared `__enter__` and `__exit__` methods for the class so that we can use the [WITH statement](#) and the connection is closed automatically at the end of WITH block.

Beside the standard way of issuing the DCL statements through the `db.commit()` and `db.rollback()` statements, raw SQL commands can be run with the `cursor.execute()` method e.g. to be able to use savepoints. There's no implicit difference between using `cursor.execute('ROLLBACK')` and `db.rollback()` except for the fact that the former may carry additional parameters, as in `cursor.execute('ROLLBACK TO SAVEPOINT some_savepoint')`. The SQL method also requires the command to be parsed and executed like a normal SQL statement, whereas the `db.commit()` and `db.rollback()` methods map to a single low level API call and allow the `cx_Oracle` driver to track the state of the transaction.

### LOB as in Large Objects

When it comes to data types available for table columns in the Oracle Database, `VARCHAR2` can only store values up to 4000 bytes. This is where Large Objects chime in with its ability to store large data such as text, images, videos and other multimedia formats. And with what you can hardly call a limit—a LOB's maximum size is as much as 128 terabytes (as of 11g Release 2).

There are several types of LOBs available in Oracle Database: Binary Large Object (BLOB), Character Large Object (CLOB), National Character Set Large Object (NCLOB) and External Binary File (BFILE). The latter one is used for accessing external operating system files in read-only mode while all the other ones enable storage of large data inside the database, in either persistent or temporary mode. Each LOB consists of an actual value and a small locator that points to the value. Passing a LOB around often simply means passing a LOB locator.

At any given time, a LOB can be in one of the three defined states: NULL, empty or populated. This resembles the behavior of regular `VARCHAR` columns in other RDBMS engines (not treating empty string as NULL). And lastly there are several restrictions with the major ones being that LOBs:

- Cannot be a primary key
- Cannot be part of a cluster
- Cannot be used in conjunction with `DISTINCT`, `ORDER BY` and `GROUP BY` clauses
- Extensive documentation of Large Objects is available in the [Oracle Database Application Developer's Guide](#).

Python's `cx_Oracle` module enables access to all types of LOBs:

```

>>> [i for i in dir(cx_Oracle) if i.endswith('LOB') or i=='BFILE']
['BFILE', 'BLOB', 'CLOB', 'LOB', 'NCLOB']

```

A special datatype exists in `cx_Oracle` for all those Oracle types whose length or type cannot be deduced automatically. It was designed particularly for dealing with IN/OUT parameters of stored procedures. Let's consider a Variable object, filled with 1 megabyte of data (repeating a hash character 2^20 times):

```

>>> db = cx_Oracle.Connection('hr/hrpwd@localhost:1521/XE')
>>> cursor = db.cursor()
>>> clob = cursor.var(cx_Oracle.CLOB)
>>> clob.setvalue(0, '#'*2**20)

```

To create a LOB object in Python we've passed a `cx_Oracle.CLOB` type to the constructor of the Variable object which exposes two basic methods (among others):

- `getvalue(pos=0)` for getting value at a given position (0 by default)
- `setvalue(pos, value)` for setting value at a given position

For some experiments with LOBs use the DDL below to create the objects listed below:

```

CREATE TABLE lobs (
    c CLOB,
    nc NCLOB,
    b BLOB,
    bf BFILE
);

CREATE VIEW v_lobs AS
SELECT
    ROWID id,
    c,
    nc,
    b,
    bf,
    dbms_lob.getlength(c) c_len,
    dbms_lob.getlength(nc) nc_len,
    dbms_lob.getlength(b) b_len,
    dbms_lob.getlength(bf) bf_len
FROM lobs;

```

For this example the BFILE locator will point to a file in the `/tmp` directory. As the SYSTEM user run:

```

create directory lob_dir AS '/tmp';
grant read on directory lob_dir to HR;

```

Finally, use an editor and create the file `/tmp/example.txt` containing any dummy text of your choice.

To get a closer insight into the default table creation options used for the LOBS table, try generating the full DDL with the `DBMS_METADATA.GET_DDL` procedure:

```

SET LONG 5000
SELECT
    DBMS_METADATA.GET_DDL('TABLE', TABLE_NAME)
FROM USER_TABLES
WHERE TABLE_NAME = 'LOBS';

```

Amongst the output there are two interesting parameters worth looking into:  
 ENABLE STORAGE IN ROW instructs Oracle to try putting the LOB data together with table data under condition it fits into 4000 bytes minus system metadata (the opposite is DISABLE STORAGE IN ROW)  
 CHUNK determines the number of bytes which are allocated while processing LOBs (this information is accessible in Python with the `cx_Oracle.LOB.getchunksize()` method)  
 The options can be used when creating the table to tune its behavior and performance.  
 For a more complete example of working with large objects consider the code below. It shows four different types of LOBs and four methods for inserting and selecting them into and from the database.

```
# -*- coding: utf8 -*-
import cx_Oracle
import operator
import os
from hashlib import md5
from random import randint

class LobExample:
    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):
        # calling close methods on cursor and connection -
        # this technique can be used to close arbitrary number of cursors
        map(operator.methodcaller("close"), (self.__cursor, self.__db))

    def clob(self):
        # populate the table with large data (1MB per insert) and then
        # select the data including dbms_lob.getlength for validating assertion
        self.__cursor.execute("INSERT INTO lobs(c) VALUES(:1)", ["~"*2**20])
        self.__cursor.execute("SELECT c, c_len FROM v_lobs WHERE c IS NOT NULL")
        c, c_len = self.__cursor.fetchone()
        clob_data = c.read()
        assert len(clob_data)==c_len
        self.__db.rollback()

    def nclob(self):
        unicode_data = u"€"*2**20
        # define variable object holding the nclob unicode data
        nclob_var = self.__cursor.var(cx_Oracle.NCLOB)
        nclob_var.setvalue(0, unicode_data)
        self.__cursor.execute("INSERT INTO lobs(nc) VALUES(:1)", [nclob_var])
        self.__cursor.execute("SELECT nc, nc_len FROM v_lobs WHERE nc IS NOT NULL")
        nc, nc_len = self.__cursor.fetchone()
        # reading only the first character just to check if encoding is right
        nclob_substr = nc.read(1, 1)
        assert nclob_substr==u"€"
        self.__db.rollback()

    def blob(self):
        # preparing the sample binary data with random 0-255 int and chr function
        binary_data = "".join(chr(randint(0, 255)) for c in xrange(2**2))
        binary_md5 = md5(binary_data).hexdigest()
        binary_var = self.__cursor.var(cx_Oracle.BLOB)
        binary_var.setvalue(0, binary_data)
        self.__cursor.execute("INSERT INTO lobs(b) VALUES(:1)", [binary_var])
        self.__cursor.execute("SELECT b FROM v_lobs WHERE b IS NOT NULL")
        b, = self.__cursor.fetchone()
        blob_data = b.read()
        blob_md5 = md5(blob_data).hexdigest()
        # data par is measured in hashes equality, what comes in must come out
        assert binary_md5==blob_md5
        self.__db.rollback()

    def bfile(self):
        # to insert bfile we need to use the bfilename function
        self.__cursor.execute("INSERT INTO lobs(bf) VALUES(BFILENAME(:1, :2))",
            ["LOB_DIR", "example.txt"])
        self.__cursor.execute("SELECT bf FROM v_lobs WHERE bf IS NOT NULL")
        # selecting is as simple as reading other types of large objects
        bf, = self.__cursor.fetchone()
        bfile_data = bf.read()
        assert bfile_data
        self.__db.rollback()

if __name__ == "__main__":
    with LobExample() as eg:
        eg.clob()
        eg.nclob()
        eg.blob()
        eg.bfile()
```

The file contains UTF-8 characters so the first line contains Python's source code encoding declaration (PEP-0263). To ensure the data is transmitted with that encoding to the database the environment variable `NLS_LANG` should be set to `_.AL32UTF8`. This instructs the Oracle Client libraries which character set should be used. This variable should be set before the Oracle Client initializes its internal data structures, which isn't guaranteed to occur at any specific point of the program. For safety, it is best to set it in the shell environment that invokes the program. Some other explanations and comments are included in the source code.



A couple of things to note here. When it comes to the NCLOB example, `unicode_data` cannot be used as a bind variable because if it's greater than 4000 characters the `"ValueError: unicode data too large"` exception is raised. The similar problem would occur in BLOB example. If we didn't use `binary_var`, then the `"DatabaseError: ORA-01465: invalid hex number"` exception would be raised because the content of the bind must be declared explicitly.

Calling stored procedures with LOB parameters (either IN or OUT) requires using cx\_Oracle's Variable objects as well but this is a subject for another part of the series.

Conclusion

In this tutorial you've been introduced to different aspects of transaction handling and processing of Large Objects from the Python environment. You should now be familiar with the characteristics of the cx\_Oracle module when it comes to dealing with transaction encapsulation and accessing all four kinds of LOBs, and also when confronted with UTF-8 encoding requirement.

**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.

 E-mail this page  Printer View

Contact Us

US Sales: +1.800.633.0738  
Global Contacts  
Support Directory  
Subscribe to Emails

About Oracle

Careers  
Company Information  
Social Responsibility  
Communities

Downloads and Trials

Java Runtime Download  
Java for Developers  
Software Downloads  
Try Oracle Cloud Free

News and Events

Acquisitions  
Blogs  
Events  
Newsroom