# Mastering Oracle+Python, Part 1: Querying Best Practices

*by Przemyslaw Piotrowski*

**As a first step, get familiar with the basic concepts of Oracle-Python connectivity**

Published September 2007

See series TOC

Among the core principles of Python's way of doing things there is a rule about having high-level interfaces to APIs. The Database API (in this case the Oracle API) is one example. Using the cx_Oracle Python module from Computronix, you can take command over the Oracle query model while maintaining compatibility with Python Database API Specification v2.0.

The model of querying databases using DB API 2.0 remains consistent for all client libraries conforming to the specification. On top of this, Anthony Tuininga, the principal developer of cx_Oracle, has added a wide set of properties and methods that expose Oracle-specific features to developers. It is absolutely possible to use only the standard methods and forget about the "extra" ones, but in this installment you won't be doing that. The concept of universal database wrappers might work in some cases but at the same time, you lose all the optimizations that the RDBMS offers.

**Introducing DB API 2.0 and cx_Oracle**

The Python Database API Specification v2.0 is a community effort to unify the model of accessing different database systems. Having a relatively small set of methods and properties, it is easy to learn and remains consistent when switching database vendors. It doesn't map database objects to Python structures in any way. Users are still required to write SQL by hand. After changing to another database, this SQL would probably need to be rewritten. Nevertheless it solves Python-database connectivity issues in an elegant and clean manner.

The specification defines parts of the API such as the module interface, connection objects, cursor objects, type objects and constructors, optional extensions to the DB API and optional error handling mechanisms.

The gateway between the database and Python language is the Connection object. It contains all the ingredients for cooking database-driven applications, not only adhering to the DB API 2.0 but being a superset of the specification methods and attributes. In multi-threaded programs, modules as well as connections can be shared across threads; sharing cursors is not supported. This limitation is usually acceptable because shareable cursors can carry the risk of deadlocks.

Python makes extensive use of the exception model and the DB API defines several standard exceptions that could be very helpful in debugging problems in the application. Below are the standard exceptions with a short description of the types of causes:

**Warning**—Data was truncated during inserts, etc.
**Error**—Base class for all of the exceptions mentioned here except for Warning
**InterfaceError**—The database interface failed rather than the database itself (a cx_Oracle problem in this case)
**DatabaseError**—Strictly a database problem
**DataError**—Problems with the result data: division by zero, value out of range, etc.
**OperationalError**—Database error independent of the programmer: connection loss, memory allocation error, transaction processing error, etc.
**IntegrityError**—Database relational integrity has been affected, e.g. foreign key constraint fails
**InternalError**—Database has run into an internal error, e.g. invalid cursor, transaction out of synchronization
**ProgrammingError**—Table not found, syntax error in SQL statement, wrong number of parameters specified etc.
**NotSupportedError**—A non-existent part of API has been called
The connect process begins with the Connection object, which is the base for creating Cursor objects. Beside cursor operations, the Connection object also manages transactions with the commit() and rollback() methods. The process of executing SQL queries, issuing DML/DCL statements and fetching results are all controlled by cursors.

cx_Oracle extends the standard DB API 2.0 specification in its implementation of the Cursor and Connection classes at most. All such extensions will be clearly marked in the text if needed.

**Getting Started**
Before working with queries and cursors, a connection to the database needs to be established. The credentials and data source names can be supplied in one of several ways, with similar results. In the extract from the Python interactive session below, connection objects db, db1 and db2 are all equivalent. The makedsn() function creates a TNS entry based on the given parameter values. Here it is being assigned to the variable dsn_tns. When environment settings are properly set then you can use the shorter form cx_Oracle.connect('hr/hrpwd'), skipping even the Easy Connect string used for db and db1.

```
>>> import cx_Oracle
>>> db = cx_Oracle.connect('hr', 'hrpwd', 'localhost:1521/XE')
>>> db1 = cx_Oracle.connect('hr/hrpwd@localhost:1521/XE')
>>> dsn_tns = cx_Oracle.makedsn('localhost', 1521, 'XE')
>>> print dsn_tns
(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost)(PORT=1521)))(CONNECT_DATA=(SID=XE)))
>>> db2 = cx_Oracle.connect('hr', 'hrpwd', dsn_tns)
```

Within the scope of a Connection object (such as assigned to the db variable above) you can get the database version by querying the version attribute (an extension to DB API 2.0). This can be used to make Python programs Oracle-version dependent. Likewise, you can get the connect string for the connection by querying the dsn attribute.

```
>>> print db.version
10.2.0.1.0
>>> versioning = db.version.split('.')
>>> print versioning
['10', '2', '0', '1', '0']
>>> if versioning[0]=='10':
... print "Running 10g"
... elif versioning[0]=='9':
... print "Running 9i"
...
```

```
Running 10g
>>> print db.dsn
localhost:1521/XE
```

### Cursor Objects

You can define an arbitrary number of cursors using the cursor() method of the Connection object. Simple programs will do fine with just a single cursor, which can be used over and over again. Larger projects might however require several distinct cursors.

```
>>> cursor = db.cursor()
```

Application logic often requires clearly distinguishing the stages of processing a statement issued against the database. This will help understand performance bottlenecks better and allow writing faster, optimized code. The three stages of processing a statement are:

**Parse (optional)**

`cx_Oracle.Cursor.parse([statement])`

Not really required to be called because SQL statements are automatically parsed at the Execute stage. It can be used to validate statements before executing them. When an error is detected in such a statement, a DatabaseError exception is raised with a corresponding error message, most likely "ORA-00900: invalid SQL statement, ORA-01031: insufficient privileges or ORA-00921: unexpected end of SQL command."

**Execute**

`cx_Oracle.Cursor.execute(statement, [parameters], **keyword_parameters)`

This method can accept a single argument - a SQL statement - to be run directly against the database. Bind variables assigned through the parameters or keyword_parameters arguments can be specified as a dictionary, sequence, or a set of keyword arguments. If dictionary or keyword arguments are supplied then the values will be name-bound. If a sequence is given, the values will be resolved by their position. This method returns a list of variable objects if it is a query, and None when it's not.

`cx_Oracle.Cursor.executemany(statement, parameters)`

Especially useful for bulk inserts because it can limit the number of required Oracle execute operations to just a single one. For more information about how to use it please see the "Many at once" section below.

**Fetch (optional)**—Only used for queries (because DDL and DCL statements don't return results). On a cursor that didn't execute a query, these methods will raise an InterfaceError exception.

`cx_Oracle.Cursor.fetchall()`

Fetches all remaining rows of the result set as a list of tuples. If no more rows are available, it returns an empty list. Fetch actions can be fine-tuned by setting the arraysize attribute of the cursor which sets the number of rows to return from the database in each underlying request. The higher setting of arraysize, the fewer number of network round trips required. The default value for arraysize is 1.

`cx_Oracle.Cursor.fetchmany([rows_no])`

Fetches the next rows_no rows from the database. If the parameter isn't specified it fetches the arraysize number of rows. In situations where rows_no is greater than the number of fetched rows, it simply gets the remaining number of rows.

`cx_Oracle.Cursor.fetchone()`

Fetches a single tuple from the database or none if no more rows are available.

Before going forward with cursor examples please welcome the pprint function from the pprint module. It outputs Python data structures in a clean, readable form.

```
>>> from pprint import pprint
>>> cursor.execute('SELECT feed_id, feed_url, XMLType.GetClobVal(feed_xml) FROM rss_feeds')
>>> cursor.execute('SELECT * FROM jobs')
[<cx_Oracle.STRING with value None>, <cx_Oracle.STRING with value None>, <cx_Oracle.NUMBER with value None>,
<cx_Oracle.NUMBER with value None>]
>>> pprint(cursor.fetchall())
[('AD_PRES', 'President', 20000, 40000),
('AD_VP', 'Administration Vice President', 15000, 30000),
('AD_ASST', 'Administration Assistant', 3000, 6000),
('FI_MGR', 'Finance Manager', 8200, 16000),
('FI_ACCOUNT', 'Accountant', 4200, 9000),
⋮
('PR_REP', 'Public Relations Representative', 4500, 10500)]
```

cx_Oracle cursors are iterators. These powerful Python structures let you iterate over sequences in a natural way that fetches subsequent items on demand only. Costly database select operations naturally fit into this idea because the data only gets fetched when needed. Instead of creating or fetching the whole result set, you iterate until the desired value is found or another condition fulfilled.

```
>>> cursor = db.cursor()
>>> cursor.execute('SELECT * FROM jobs')
[<cx_Oracle.STRING with value None>, <cx_Oracle.STRING with value None>, <cx_Oracle.NUMBER with value None>,
<cx_Oracle.NUMBER with value None>]
>>> for row in cursor: ## notice that this is plain English!
... print row
...
('AD_VP', 'Administration Vice President', 15000, 30000)
('AD_ASST', 'Administration Assistant', 3000, 6000)
('FI_MGR', 'Finance Manager', 8200, 16000)
('FI_ACCOUNT', 'Accountant', 4200, 9000)
('AC_MGR', 'Accounting Manager', 8200, 16000)
⋮
('PR_REP', 'Public Relations Representative', 4500, 10500)
```

Just after an execute list(cursor) does the same job as cursor.fetchall(). This is because the built-in list() function iterates until the end of the given iterator.

### Datatypes

During the fetch stage, basic Oracle data types get mapped into their Python equivalents. cx_Oracle maintains a separate set of data types that helps in this transition. The Oracle - cx_Oracle - Python mappings are:

| Oracle | cx_Oracle | Python |
|---|---|---|
| VARCHAR2<br>NVARCHAR2<br>LONG | cx_Oracle.STRING | str |
| CHAR | cx_Oracle.FIXED_CHAR | |
| NUMBER | cx_Oracle.NUMBER | int |
| FLOAT | | float |

| | | |
|---|---|---|
| DATE | cx_Oracle.DATETIME | datetime.datetime |
| TIMESTAMP | cx_Oracle.TIMESTAMP | |
| CLOB | cx_Oracle.CLOB | cx_Oracle.LOB |
| BLOB | cx_Oracle.BLOB | |

The above data types are usually transparent to the user except for cases involving Large Objects. As of version 4.3, cx_Oracle still handles them itself and not wrapped with the built-in file type.

Other data types that are not yet handled by cx_Oracle include XMLTYPE and all complex types. All queries involving columns of unsupported types will currently fail with a NotSupportedError exception. You need to remove them from queries or cast to a supported data type.

For example, consider the following table for storing aggregated RSS feeds:

```
CREATE TABLE rss_feeds (
feed_id NUMBER PRIMARY KEY,
feed_url VARCHAR2(250) NOT NULL,
feed_xml XMLTYPE
);
```

When trying to query this table with Python, some additional steps need to be performed. In the example below XMLType.GetClobVal() is used to return XML from the table as CLOB values.

```
>>> cursor.execute('SELECT * FROM rss_feeds')

Traceback (most recent call last):
File "<pyshell#37>", line 1, in <module>
cursor.execute('SELECT * FROM rss_feeds')
NotSupportedError: Variable_TypeByOracleDataType: unhandled data type 108
>>> cursor.execute('SELECT feed_id, feed_url, XMLType.GetClobVal(feed_xml) FROM rss_feeds')
[<cx_Oracle.NUMBER with value None>, <cx_Oracle.STRING with value None>, <cx_Oracle.CLOB with value None>]
```

You might have already noticed the cx_Oracle.Cursor.execute* family of methods returns column data types for queries. These are lists of Variable objects (an extension to DB API 2.0), which get the value None before the fetch phase and proper data values after the fetch. Detailed information about data types is available through the description attribute of cursor objects. The description is a list of 7-item tuples where each tuple consists of a column name, column type, display size, internal size, precision, scale and whether null is possible. Note that column information is only accessible for SQL statements that are queries.

```
>>> column_data_types = cursor.execute('SELECT * FROM employees')
>>> print column_data_types
[<cx_Oracle.NUMBER with value None>, <cx_Oracle.STRING with value None>, <cx_Oracle.STRING with value None>,
<cx_Oracle.STRING with value None>, <cx_Oracle.STRING with value None>, <cx_Oracle.DATETIME with value None>,
<cx_Oracle.STRING with value None>, <cx_Oracle.NUMBER with value None>, <cx_Oracle.NUMBER with value None>,
<cx_Oracle.NUMBER with value None>, <cx_Oracle.NUMBER with value None>]
>>> pprint(cursor.description)
[('EMPLOYEE_ID', <type 'cx_Oracle.NUMBER'>, 7, 22, 6, 0, 0),
('FIRST_NAME', <type 'cx_Oracle.STRING'>, 20, 20, 0, 0, 1),
('LAST_NAME', <type 'cx_Oracle.STRING'>, 25, 25, 0, 0, 0),
('EMAIL', <type 'cx_Oracle.STRING'>, 25, 25, 0, 0, 0),
('PHONE_NUMBER', <type 'cx_Oracle.STRING'>, 20, 20, 0, 0, 1),
('HIRE_DATE', <type 'datetime.datetime'>, 23, 7, 0, 0, 0),
('JOB_ID', <type 'cx_Oracle.STRING'>, 10, 10, 0, 0, 0),
('SALARY', <type 'cx_Oracle.NUMBER'>, 12, 22, 8, 2, 1),
('COMMISSION_PCT', <type 'cx_Oracle.NUMBER'>, 6, 22, 2, 2, 1),
('MANAGER_ID', <type 'cx_Oracle.NUMBER'>, 7, 22, 6, 0, 1),
('DEPARTMENT_ID', <type 'cx_Oracle.NUMBER'>, 5, 22, 4, 0, 1)]
```

**Bind Variable Patterns**

As advertised by Oracle guru Tom Kyte, bind variables are core principles of database development. They do not only make programs run faster but also protect against SQL injection attacks. Consider the following queries:

```
SELECT * FROM emp_details_view WHERE department_id=50
SELECT * FROM emp_details_view WHERE department_id=60
SELECT * FROM emp_details_view WHERE department_id=90
SELECT * FROM emp_details_view WHERE department_id=110
```

When run one-by-one, each need to be parsed separately which adds extra overhead to your application. By using bind variables you can tell Oracle to parse a query only once. cx_Oracle supports binding variables by name or by position.

Passing bind variables by name requires the parameters argument of the execute method to be a dictionary or a set of keyword arguments. query1 and query2 below are equivalent:

```
>>> named_params = {'dept_id':50, 'sal':1000}
>>> query1 = cursor.execute('SELECT * FROM employees WHERE department_id=:dept_id AND salary>:sal',
named_params)
>>> query2 = cursor.execute('SELECT * FROM employees WHERE department_id=:dept_id AND salary>:sal',
dept_id=50, sal=1000)
```

When using named bind variables you can check the currently assigned ones using the bindnames() method of the cursor:

```
>>> print cursor.bindnames()
['DEPT_ID', 'SAL']
```

Passing by position is similar but you need to be careful about naming. Variable names are arbitrary so it's easy to mess up queries this way. In the example below, all three queries r1, r2, and r3 are equivalent. The parameters variable must be given as a sequence.

```
>>> r1 = cursor.execute('SELECT * FROM locations WHERE country_id=:1 AND city=:2', ('US', 'Seattle'))
```

```
>>> r2 = cursor.execute('SELECT * FROM locations WHERE country_id=:9 AND city=:4', ('US', 'Seattle'))
>>> r3 = cursor.execute('SELECT * FROM locations WHERE country_id=:m AND city=:0', ('US', 'Seattle'))
```

When binding, you can first prepare the statement and then execute None with changed parameters. Oracle will handle it as in the above case, governed by the rule that one prepare is enough when variables are bound. Any number of executions can be involved for prepared statements.

```
>>> cursor.prepare('SELECT * FROM jobs WHERE min_salary>:min')
>>> r = cursor.execute(None, {'min':1000})
>>> print len(cursor.fetchall())
19
```

You have already limited the number of parses. In the next paragraph we'll be eliminating unnecessary executions, especially expensive bulk inserts.

**Many at Once**

Large insert operations don't require many separate inserts because Python fully supports inserting many rows at once with the cx_Oracle.Cursor.executemany method. Limiting the number of execute operations improves program performance a lot and should be the first thing to think about when writing applications heavy on INSERTs.

Let's create a table for a Python module list, this time directly from Python. You will drop it later.

```
>>> create_table = """
CREATE TABLE python_modules (
module_name VARCHAR2(50) NOT NULL,
file_path VARCHAR2(300) NOT NULL
)
"""
>>> from sys import modules
>>> cursor.execute(create_table)
>>> M = []
>>> for m_name, m_info in modules.items():
... try:
... M.append((m_name, m_info.__file__))
... except AttributeError:
... pass
...
>>> len(M)
76
>>> cursor.prepare("INSERT INTO python_modules(module_name, file_path) VALUES (:1, :2)")
>>> cursor.executemany(None, M)
>>> db.commit()
>>> r = cursor.execute("SELECT COUNT(*) FROM python_modules")
>>> print cursor.fetchone()
(76,)
>>> cursor.execute("DROP TABLE python_modules PURGE")
```

Only one execute has been issued to the database to insert all 76 module names. This is huge performance boost for large insert operations. Notice two small quirks here: cursor.execute(create_tab) doesn't produce any output since it is a DDL statement and (76,) is a tuple with a single element. (76) without a comma would simply be equivalent to an integer 76.

**Conclusion**
After familiarizing yourself with basic concepts of Oracle-Python connectivity you are ready to start writing your own database-driven applications. I highly recommend playing with the Python interactive shell for some time as it really brings down the learning curve.

You have learned about three stages that SQL statements go through and how to minimize the number of steps the Oracle Database needs to perform. Bind variables are an inevitable part of database application development and Python enables binding them by name or by position.

You have also been introduced to the smooth transition between Oracle and Python datatypes and the natural way of handling database data in the context of handling cursors as iterators. All these features boost productivity and enable focusing on the data, which is what it's all about.

**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.