DCOracle2 User's Guide & Reference

Release 1.1
April 16, 2002

Matthew T. Kromer

Zope Corporation
Lafayette Technology Center
513 Prince Edward Street
Fredericksburg, VA 22408
(888) 344-4332 (sales)
http://www.zope.com

DCOracle2 User's Guide & Reference

This document describes the various objects and interfaces to the DCOracle2 binding of Oracle 8 OCI calls for Python. DCOracle2 is a Python wrapper around a C binding called ''dco2'' which provides Python DB API 2.0 functionality.

DB API

At the time of this writing, the Python DB 2.0 API was available on the web at http://www.python.org/topics/database/DatabaseAPI-2.0.html. The DB API forms the official interface of DCOracle2, although DCOracle2 has a number of extensions and alternatives.

Programming with DCOracle2

Writing DCOracle2 programs is fairly straightforward. Familiarity with the DB API specification is useful, but should not be required to use DCOracle2.

All DCOracle2 programs must import the DCOracle2 module (or have it made available to them.) The environment variable ORACLE_HOME should be set before importing DCOracle2, and the Oracle run time libraries should be within the runtime library search path (LD_LIBRARY_PATH on many systems.)

A program establishes one or more connections to Oracle. Each connection supports cursors, which allow statements to be executed. The results of an executed statement are retrievable with a fetch operation. For example:

```
import DCOracle2                                    # Import runtime
db = DCOracle2.connect('scott/tiger')               # Connect to database
c = db.cursor()                                     # Allocate a cursor
c.execute('select * from emp where empno=7902')     # Execute a statement
print c.fetchall()                                  # Print all results
```

Stored procedure objects are available through a special ''procedure'' name on cursors. Stored procedure objects simplify access to stored procedures. The docstring on a stored procedure object will describe its parameters in english, for convenience. Continuing with the previous example:

```
find = c.procedure.scott.emp_actions.find           # Reference the stored procedure
print find.__doc__                                  # Print its docstring
print find(7902)                                    # Invoke and print results
```

In the test directory of the source distribution, the ''procdef.sql'' file contains several sample stored procedures in the package ''emp_actions.''

Usage

```
import DCOracle2

connection = DCOracle2.connect(connectionstring)
traceback = DCOracle2.traceback([format={0|1}])
type = DCOracle2.TypeTable(type)
type = DCOracle2.Type(type)
version = DCOracle2.version
apilevel = DCOracle2.apilevel
threadsafety = DCOracle2.threadsafety
paramstyle = DCOracle2.paramstyle

connection.close()
connection.prepare([statement])
connection.commit([twophase={0|1}])
connection.rollback()
listoftuples = connection.objects([all={0|1}])
text = connection.getSource(procname)
description = connection.describe(name)
bao = connection.BindingArray(count, size, type)
versiondict = connection.version

cursor = connection.cursor()
cursor.arraysize = n
description = cursor.description
description = cursor.describe()
count = cursor.rowcount
cursor.close()
cursor.setPrefetch(records [, size])
cursor.execute(statement, [param1[, param2[, ...paramn]]]. [param=value, ...])
cursor.executemany(statment, paramsequence)
list = cursor.fetchone([skip=0])
listoflists = cursor.fetchmany([size=size][,skip=0])
listoflists = cursor.fetchall([skip=0])
listofparameters = cursor.callproc(name, parameters)
value = cursor.procedure.schema.package.name(param1, ... paramn)

length = LobLocator.length()
data = LobLocator.read([bytes, [offset, [csid, [csfrm]]]])
count = LobLocator.write(string, [offset, [csid, [csfrm]]])
LobLocator.trim(length)
```

Type Constructors:

```
OracleDate = DCOracle2.Date(year, month, day)
OracleDate = DCOracle2.Time(hour, minute, second)
OracleDate = DCOracle2.Timestamp(year, month, day, hour, minute, second)
OracleDate = DCOracle2.DateFromTicks(ticks)
OracleDate = DCOracle2.TimeFromTicks(ticks)
OracleDate = DCOracle2.TimestampFromTicks(ticks)
wrapper = DCOracle2.Binary(string)
wrapper = DCOracle2.TypeCoercion(value, type)
```

Type Comparison Objects:

```
DCOracle2.STRING
DCOracle2.BINARY
DCOracle2.NUMBER
DCOracle2.DATETIME
DCOracle2.ROWID
```

Error Objects:

```
DCOracle2.Error
DCOracle2.Warning
DCOracle2.InterfaceError
DCOracle2.DatabaseError
DCOracle2.InternalError
DCOracle2.OperationalError
DCOracle2.ProgrammingError
DCOracle2.IntegrityError
DCOracle2.DataError
DCOracle2.NotSupportedError
```

DCOracle2.connect(*connectionstring)*

The connect method connects to the database with the provided credentials. The format of a connection string is `userid/password@service`, where the `@service` component may be omitted. If *service* is not specified, the current default database will be used. If *connectionstring* is null, the connection will be made using external credentials (`OCI_CRED_EXT`).

DCOracle2.traceback([*format=1*]) [NONAPI]

DCOracle can retrieve a traceback of the C layer events for debugging purposes. It is useful to save the output of the formatted traceback for problem reporting.

DCOracle2.TypeTable(*type*) [NONAPI]

The TypeTable method will convert numeric types to SQLT_ strings and back. For example, TypeTable(1) will return ''SQLT_CHR'' and TypeTable(''SQLT_CHR'') will return 1. The type table is a lookup table constructed from Oracle constants at compile time.

DCOracle2.Type(*type*) [NONAPI]

The Type method will convert a numeric or string type into the external type representation, e.g. Type(1) and Type(''SQLT_CHR'') will both return ''VARCHAR2.'' The type translation table is a dictionary inside the DCOracle2 module, and not managed by Oracle.

DCOracle2.apilevel

This is the DB API level of DCOracle2, the constant "2.0".

DCOracle2.threadsafety

This is the thread-safety-ness level of DCOracle2, the constant 3. This means that threads may share the module, connections, and cursors. Note that programming errors are still possible without a mutex to serialize resources between threads.

DCOracle2.paramstyle

The value ''numeric.'' Numeric positional parameters are the norm, but named keyword parameters are accepted by the execute() method. This means that bindings to SQL statements are normally specified as ‘‘:1’’ ‘‘:2’’ ‘‘:3’’ etc. e.g. ‘‘SELECT * FROM EMP WHERE EMPNO=:1.’’

***connection*.close()**

This closes the connection to Oracle. Any attempt to use a connection or cursor object to access the database after the connection has been closed will raise an `InterfaceError`. *Note*: Close the connection after all cursors have been closed; closing the cursor after the connection may result in a "stuck" cursor.

***connection*.prepare([*statement*])** [NONAPI]

With no statement, this prepares for a commit operation (for a two-phase commit). Under normal circumstances, applications use one-phase commit.

When a statement is provided, this creates a new cursor, with the statement bound to the cursor, and returns the cursor object.

***connection*.commit([twophase={0|1}])**

This commits all changes to the database on the connection. If the parameter `twophase` is specified as 1, a two-phase commit is performed. The default is to do a one-phase commit.

Two-phase commit is not very meaningful without spreading a transaction between multiple databases, and transaction-ID support is not provided by DCOracle2.

***connection*.rollback()**

This calls the dco2 *ServerContext*.rollback() method on the database connection.

***connection*.describe(*name*)** [NONAPI]

This calls the OCI function OCIDescribeAny upon the named schema object; the object can be of any type. The result is a dictionary which often contains lists of dictionaries based on the key found. The method *connection.*collapsedesc(*desc*) will collapse this dictionary somewhat into a more usable form.

***connection*.BindingArray(*count*, *size*, *type*)** [NONAPI]

BindingArrays are used to form array or OUT parameters for PL/SQL. Constructs a dco2 BindingArrayObject with *count* elements, each of size *size* and type *type.* The type may be numeric or symbolic e.g. ’SQLT_CHR’. This container class is capable of having its contents changed during the execution of a SQL statement, and forms the basis for stored procedure argument passing.

The following example shows the longhand format of invoking a stored procedure using a BindingArray. In this example, the `find` function accepts an employee ID and returns the employee name.

```
import DCOracle2

db = DCOracle2.connect("scott/tiger")
c = db.cursor()
bao = db.BindingArray(1, 64, 'SQLT_STR')


c.execute("begin :1 := emp_actions.find(7902); end;", bao)

print bao[0]
```

BindingArrays allow assignment into the array, or one past the end of the array. BindingArrays will grow as necessary to contain all values. Example:

```
import DCOracle2
db = DCOracle2.connect('scott/tiger')
bao = db.BindingArray(1, 64, 'SQLT_STR')

bao[1] = "eek!"   # fails

bao[0] = "this works"
bao[1] = "NOW this works"
```

In general, BindingArrays are not used directly; they are constructed automatically as part of the stored procedure invocation process and by *cursor*.executemany().

BindingArrays of size 1 form nonarray IN/OUT bindings to PL/SQL. BindingArrays of size > 1 form array IN/OUT bindings. Most stored procedures do not accept array input. BindingArrays of size > 1 are also used for multiple-row execution.

The default mode of invocation for a BindingArray is a dynamic bind; in some cases, this will not be appropriate, and the BindingArrays should be declared as static binds. The method setStatic(1) on the BindingArray will change it to be a static bind.

***connection*.version**

This returns a version dictionary of the relevant Oracle version strings. It is useful for diagnosing problems. Example:

```
{'CORE ': '8.1.3.0.0 (Production)',
 'DCOracle2': '1.84 (dco2: 1.106 -DORACLE8i -DUSEOBJECT -Dlinux -D_REENTRANT )',
 'NLSRTL ': '3.4.0.0.0 (Production)',
 'Oracle8i Enterprise Edition ': '8.1.5.0.2 (Production)',
 'PL/SQL ': '8.1.5.0.0 (Production)',
 'TNS for Linux: ': '8.1.5.0.0 (Production)'}
```

***connection*.cursor()**

This returns a cursor object suitable for executing SQL statements and retrieving results.

The *connection* internal cursor

Each *connection* object contains an internal cursor, and thus all *cursor* methods also apply to *connection* objects. It is advisable, however, to explicitly allocate a cursor for use rather than using the internal cursor.

***cursor.*arraysize**

This attribute controls the array size used to transfer SELECT results from Oracle. The default value is 20.

***cursor.*rowcount**

The row count affected by the last execute() operation. For SELECT statements, the rowcount is the number of rows retrieved so far, not the total available rows.

***cursor*.description**

This is a list of tuples of (`name, type, display_size, internal_size, precision, scale, nullok`) which has had the type field turned into text instead of the number. This value is not assignable, but is set after each *cursor*.execute() function. The description is `None` if the last statement executed did not return any results.

The type field is checked against the API neutral types by comparing it to the standard API type classes STRING, BINARY, NUMBER, DATETIME, and ROWID. The comparison will be equal if it is of that type. For an example, see the Type Comparison Objects section.

***cursor*.describe()** [NONAPI]

This returns a list of tuples of `(name, type, display_size, internal_size, precision, scale, nullok)`which is represents the result columns of the last *cursor*.execute. The `type` field returned is a numeric representation of the type.

cursor.close()

This closes the cursor. Using a closed cursor (or a stored procedure object bound to that cursor) to attempt to access the database will raise an `InterfaceError`. Cursors are implicitly closed when they are deleted by Python as their reference count goes to zero. Cursors should be closed before the underlying database object is closed.

cursor.execute(*statement*, [*parm1*[, *parm2*[, ... *parmn*]]], [*parm*=*value*, ...])

This executes an SQL *statement* with optional parameters being bound as input arguments by position. The statement is cached, and if the same statement is used successively the original statement handle is re-used. If the statement is `None`, the last statement to be executed will be used.

The Python DB 2.0 API specifies that execute takes only one parameter, `sequence_of_values`. DCOracle2 accepts this calling convention, but also allows multiple arguments to appear in the call (the outer list around the sequence of parameters is optional). DCOracle2 also allows named binding for execute parameters. The executemany method does not allow named parameters.

Examples:

```
cursor.execute("SELECT * FROM EMP WHERE EMPNO = :1", empno)             # Numeric form
cursor.execute("SELECT * FROM EMP WHERE EMPNO = :1", (empno,))          # DBAPI form
cursor.execute("SELECT * FROM EMP WHERE EMPNO = :empno', empno=empno)   # Named param
```

The underlying C module only has input binding translators for a restricted set of types, being OracleDates, Strings, Integers, Longs, and Floats. Any other type of object passed in as a parameter will generate an error. See the section ''Type Conversion Rules'' for more information on input type conversions.

Coercion example, where column "photo" is a LONG RAW:

```
cursor.execute("INSERT INTO PHOTOTABLE (id, photo) VALUES (:1, :2)", id, \
       Binary(photo))
```

When binding values, named binds may also be used with keyword arguments. For example:

```
cursor.execute("INSERT INTO TEST (name, id) VALUES (:name, :id)", name="Matt Kromer",
       id="1")
```

The underlying C implementation will accept BindingArrays as bound parameters; BindingArrays will cause iterative execution of non-select and non-PL/SQL statements (i.e. this is how *cursor*.executemany works).

cursor.executemany(*statement, paramsequence)*

This repeatedly executes an SQL *statement* with parameters being bound as input arguments by position. The statement handle is re-used for all inputs, and may also be shared with a successive execute or executemany. The parameter *paramsequence* is a sequence of lists of input values.

Example:

```
cursor.executemany("INSERT INTO TEST (NAME, ID) VALUES (:1, :2)", (("Matt", 1),
        ("Mike", 2)))
```

See the restriction in *cursor*.execute about the allowable object types to be bound. Type overriding is the same as for *cursor.*execute.

Named parameters to executemany are not implemented.

Using the executemany method can improve performance for INSERT operations.

Note: Calling executemany for a SELECT statement will not produce the desired results; only the results of the last SELECT will be available for fetching.

*cursor.*setPrefetch*(rows* [*, size*]*)* [NONAPI]

The setPrefetch method sets Oracle prefetching for this cursor; Oracle will instructed to prefetch up to `rows` rows, and if a size is specified, that is the maximum storage area to be used for prefetching. The default is to prefetch the lesser of 20 rows or one megabyte of storage.

cursor.fetchone([skip=0 [NONAPI]])

This retrieves one row of a result from a prior execute function. The values of the row are returned as a list. Fetch operations will internally buffer based on the value of the `arraysize` attribute of the cursor. If no data is available to be returned, the result will be `None`. If the *skip* argument is specified, that many results will be skipped before fetching a row. The skip argument is not a part of the DB 2.0 API.

cursor.fetchmany([size=*size*][,skip=0 [NONAPI]])

This retrieves at most *size* rows from a prior execute function. The values of the rows return is a list of lists (a list of rows each containing a list of values). If the *skip* argument is specified, that many results will be skipped before fetching the rows. The skip argument is not a part of the DB 2.0 API.

***cursor*.fetchall([skip=0** [NONAPI] **])**

This retrieves all remaining rows from a prior execute function. The values of the rows return is a list of lists (a list of rows each containing a list of values). If the *skip* argument is specified, that many results will be skipped before fetching the rows. The skip argument is not a part of the DB 2.0 API.

***cursor*.callproc(*name,* [*param1,* [*param2*, ... *param*n]], [*name*=*value*, ...])**

Calls the stored procedure *name* with the given parameters. The complete list of parameters is returned as a result list. For example

```
import DCOracle2
db = DCOracle2.connect('scott/tiger')
c = db.cursor()
(empname, empid) = c.callproc(''emp_actions.findp'',7902)
```

***cursor.*procedure*.schema.package.name*(*args...*)** [NONAPI]

Invoke a stored procedure in the named schema and package. The schema and package names are optional; the default schema is used if the schema is omitted, and if a function or procedure is not in a package, the package name may be omitted. For example, user SCOTT may have a package named EMP_ACTIONS, which may contain a set of functions such as FIND, FINDP, and FINDY. The procedure FINDP has as an IN parameter an integer (the employee ID) and an OUT parameter (the employee name). The following example would invoke this procedure:

```
import DCOracle2
db = DCOracle2.connect('scott/tiger')
c = db.cursor()
empname = c.procedure.emp_actions.findp(7902)
```

The result of a procedure or function is a single item if there is only one OUT parameter, or a list of results if there are multiple OUT parameters (this differs from the *cursor.*callproc() method, which returns *all* parameters). Only IN parameters should be specified as arguments to the procedure/function; this includes IN OUT parameters. Because DCOracle2 knows the variable names used by the procedure, keyword arguments may also be passed, e.g.

```
empname = c.procedure.emp_actions.findp(empid=7902)
```

Overloaded procedure names are not currently handled by DCOracle2.

LOB Operations

Oracle can return Large OBjects (LOBs). DCOracle2 supports character LOBs and binary LOBs, CLOBs and BLOBs, respectively. When a LOB object is returned, it may be read and written to by its methods.

***LobLocator*.length()** [NONAPI]

Returns the length of the LOB referenced by *LobLocator*.

***LobLocator*.read([*size*, [*offset*, [*csid*, [*csfrm*]]]])** [NONAPI]

Returns the data from the *LobLocator* for *size* bytes, which defaults to the end of the LOB. Optionally, the starting LOB *offset* may be provided, as may be the character set ID *csid* and character set form *csfrm*. Both *csid* and *csfrm* default and can be safely omitted unless the default character set Oracle was installed with is not appropriate.

***LobLocator*.write(*string*, [*offset*, [*csid*, [*csfrm*]]])** [NONAPI]

Writes *string* to the selected LOB, optionally starting *offset* bytes into the LOB. The character set parameters *csid* and *csfrm* default to the Oracle installation defaults and can be safely omitted.

Note: Empty LOBs can be inserted by using the EMPTY_BLOB() or EMPTY_CLOB() Oracle functions. LOB objects must be selected for updating before they can be written, for example:

```
import DCOracle2

db = DCOracle2.connect('scott/tiger')
c = db.cursor()
c.execute('insert into blobtest values (:1, :2, EMPTY_BLOB())', 'Shane Hathaway', 4)
c.execute('select * from blobtest where id=4 for update')
r = c.fetchone()
lob = r[2]
lob.write(shanepic)
db.commit()
```

***LobLocator*.trim(*length*)** [NONAPI]

Trims the LOB to the specified length, which must be less than or equal to the current length.

Type Constructors

DCOracle2.Date(year, month, day)
DCOracle2.Time(hour, minute, second)

DCOracle2.Timestamp(year, month, day, hour, minute, second)
DCOracle2.DateFromTicks(ticks)
DCOracle2.TimeFromTicks(ticks)
DCOracle2.TimestampFromTicks(ticks)

Each of these constructors returns an `OracleDate` object, which represents a date suitable for passing by parameter into an execute() call. The object is GMT based.

DCOracle2.Binary(string)

The Binary constructor currently returns its input, coerced as a ''SQLT_LBI'' object (long binary). DCOracle2 deals with most Oracle types as strings, not Binary objects.

DCOracle2.TypeCoercion(*value*, *type*) [NONAPI]

The TypeCoercion constructor will return a coerced value, which will be represented to Oracle as the specified type. The type may be numeric or named, e.g. 5 or ''SQLT_STR''. Type coercion is normally not required unless the default type binding rules are insufficient, in which case the application must prepare the data and provide a type coercion to present it to Oracle.

Type Comparison Objects

The type field of the cursor description is only meaningful within the context of a particular database; the DB API provides type comparison objects which will compare equal to the value of the type field if the that is the meta-type represented.

DCOracle2.STRING
DCOracle2.BINARY
DCOracle2.NUMBER
DCOracle2.DATETIME
DCOracle2.ROWID

The STRING type means the object uses string representaton to Python, it does not (in the case of DCOracle2) imply that the string does not contain nonprintable characters, only that it creates a Python string object.

The BINARY type means the object is a `LobLocator` , and has read(), write(), length(), and trim() methods. See the corresponding dco2 LobLocator documentation.

The NUMBER type means the object represents a number, either integer, long, or floating point.

The DATETIME type means the object represents an OracleDate. OracleDates return a string in the format "yyyy-mm-dd hh:mm:ss" when used in a string context, or an an integer (timestamp) when used in an integer context.

The ROWID type means the object represents a ROWID.

Example:

```
import DCOracle2
db = DCOracle2.connect('scott/tiger')
c = db.cursor()
c.execute('select * from emp')

if c.description[0][1] == DCOracle2.NUMBER:
        print "The first column is a number"
else:
        print "The first column is not a number"
```

Example Program

```
#

import DCOracle2
import common

names = (
    ("Matt Kromer", 1),
    ("Jens Vagelpohl", 2),
    ("Chris Petrilli", 3)
    )

db = DCOracle2.connect(common.getConnectionString())
c = db.cursor()


print "Dropping table"

try:
    c.execute("drop table test")
except:
    pass


print "Creating table"
c.execute("create table test ( name varchar2(64), id number(9) )")

print "Inserting into test"
c.executemany("insert into test (name, id) values (:1, :2)", names)

#print DCOracle2.traceback(format=1)

print "Selecting from test"
c.execute("select * from test")

print "Describing test"
print list(c.description)

c.arraysize = 20

print "Printing test"
r = c.fetchall()
print r

print "Commiting test"
db.commit()

print "Adding one more entry"
c.execute("insert into test (name, id) values (:1, :2)", "Fred Flintstone", 4)

print "Printing test"
c.execute("select * from test")
r = c.fetchall()
print r

print "Rolling back last entry"
db.rollback()

print "Printing test"
c.execute("select * from test")
r = c.fetchall()
print r
```

BUILDING and INSTALLATION

Generally, building the DCO2 module is as simple as setting up the Oracle build environment and typing `make`. Here's an example:

```
bane(1)$ . oraenv
ORACLE_SID = [ORAC] ? <enter>
bane(2)$ make
(lots of output)
```

That's it! The makefile will try to detect as much as it can and will properly copy the built dco2.so module to the DCOracle2 directory.

A number of problems can happen during building:

Oracle environment not set up correctly
If the environment variable `ORACLE_HOME` is not set, then the compilation will fail to find the proper Oracle include and library files.

Oracle development files not installed
Oracle installs the C headers to build OCI programs in `$ORACLE_HOME/rdbms/demo`, `$ORACLE_HOME/network/public`, and `$ORACLE_HOME/plsql/public`. If these directories do not have the proper include files, the build will not succeed. Oracle typically only installs these files when a complete server installation is performed.

Python development Makefile not available
If the make fails stating that `/usr/lib/python1.5/config/Makefile` (or similar) cannot be built, it is symptomatic of the problem that the Python development is not installed. On a RedHat Linux system this would be the *python-devel* package.

Python is not linked against libpthread (Linux)
Oracle on Linux is linked against libpthread. Some recent linux distributions may include versions of Python that are not linked against libpthread. When dco2 is loaded, it will load the Oracle shared libraries which are incompatible with the libpth libraries of the existing Python binary. This will cause random lockups to occur to the process. The only solution is to acquire a Python that is not linked against libpth, but is instead linked against libpthread. This may be accomplished by either obtaining a different Python binary package, or rebuilding Python from source.

DIAGNOSIS

Trace Table Debugging

DCOracle2 keeps an internal trace table, controlled by four environment variables:

• DCO2TRACELOG	(trace file name)
• DCO2TRACEDUMP	(dump file name)
• DCO2TRACESIZE	(trace table size)
• DCO2TRACEFLAGS	(trace filter)

The variable DCO2TRACELOG, if set, names a file to record all trace events (as they occur). The variable DCO2TRACEDUMP names a file to dump the trace table to if an Oracle error occurs (but this is not the only reason DCOracle2 will raise an error).

DCOracle2 will sometimes trap errors within a try/except, but this will still cause a trace dump to take place if DCO2TRACEDUMP is set. Each new Oracle error will append the trace file to the dump file.

The variable DCO2TRACESIZE specifies the number of entries in the trace table (the default is 512). Each entry in the trace table is 10 words (40 bytes) long, so the default trace table consumes 20K of storage. Setting the size of the trace table to 0 will also set the DCO2TRACEFLAGS to 0. DCO2TRACEFLAGS is the binary OR (sum) of the following values:

1	Entry
2	Exit
4	Error
8	Thread
16	Program
32	Oracle
64	Info
128	Verbose

The typical codes used are:

17	Function Entry
18	Function Exit
33	Oracle call
34	Oracle return
36	Oracle error
65	Info arguments
66	Info result

161	Oracle handle call
162	Oracle handle return

Up to 7 arguments are also recorded in the trace table, with the first argument being a function name.

Problem Reporting

When submitting a problem report to Zope Corporation about DCOracle2, it is useful to include a trace dump when incorrect results are being returned. When reporting problems, please be sure and identify the following:

- Operating System
- Distribution (for Linux)
- Python Version
- Oracle Version
- DCO2TRACEDUMP generated file (where appropriate)
- Python tracebacks (where appropriate)

The problem collector for DCOracle2 is currently located at http://www.zope.org/Members/matt/dco2/Tracker.

Paid Support

Paid support is an option for users of DCOracle2. While DCOracle2 is a free product, it still requires the commitment of resources on the part of Zope Corporation to maintain it. Free support is handled on an ad-hoc basis when time permits; paid support contracts to Zope Corporation are handled on a priority basis.

Zope Corporation is willing to work with its customers to establish support programs which meet their needs. Persons interested in purchasing a support contract should contact sales@zope.com or visit http://www.zope.com/Services/SupportContracts.

Converting from DCOracle to DCOracle2

When converting from DCOracle, a few points are in order:

DCOracle uses the "Connect" method to connect to Oracle; DCOracle2 uses "connect" but allows "Connect" for compatibility.

Binding to execute() is different. DCOracle (and DB API 1.0) blurs the definition between execute and executemany() and will act as executemany() if it is passed lists of lists; DCOracle execute() takes list parameters in column major order -- DCOracle2 executemany() takes list parameters in row major order. DCOracle performs positional parameter binds by using the names ":p1" ":p2" etc. rather than ":1" ":2" which is the format supported by DCOracle2. DCOracle takes list parameters as a *single* parameter, whereas DCOracle2 takes each value as a separate parameter.

Example: single row insert

```
import DCOracle
db = DCOracle.Connect(’scott/tiger’)
c = db.cursor()
c.execute(’insert into test (name, id) values (:p1, :p2)’, (’Matt Kromer’, 1))
```

```
import DCOracle2
db = DCOracle2.connect(’scott/tiger’)
c = db.cursor()
c.execute(’insert into test (name, id) values (:1, :2)’, ’Matt Kromer’, 1)
```

Example: multiple row insert

```
import DCOracle
db = DCOracle.Connect(’scott/tiger’)
c = db.cursor()
c.execute(’insert into test (name, id) values (:p1, :p2)’, ([’Matt Kromer’, ’Jens
        Vagelpohl’], [1, 2]))
```

```
import DCOracle2
db = DCOracle2.connect(’scott/tiger’)
c = db.cursor()
c.executemany(’insert into test (name, id) values (:1, :2)’, ((’Matt Kromer’, 1), (’Jens
        Vagelpohl’, 2)))
```

DCOracle2 supports the list mode of aruments to execute() in a compatible way with DCOracle, but requires the parameter notation to be changed.

DCOracle uses the dbiRaw object to access LONG data; DCOracle2 stores LONG data as strings and uses type coercion. DCOracle2 provides a compatible DCOracle2.dbiRaw function which will return a DCOracle2.Binary object.

DCOracle2 uses different date handling than DCOracle. The DCOracle dbiDate constructor is not supported, use the API 2.0 date and time constructors instead.

Type Conversion Rules

The type conversion for the C layer of DCOracle2 uses the following rules (they are not necessarily the correct rules, but this the current implementation):

Input types are converted in the internal function bindObject and obey the following rules:

1. If the object is None, bind the input as NULL.
2. If the object is a BindingArray, set the column for array input/output.
3. If the object is a Cursor, bind the column as an SQL Result Set (SQLT_RSET).
4. If the object is an OracleDate, bind the column as a a Date (SQLT_DAT).
5. If the object is a String, bind the column as a string (SQLT_STR).
6. If the object is a Long, bind the column as an integer (SQLT_INT).
7. If the object is an Integer, bind the column as an integer (SQLT_INT).
8. If the object is a Float, bind the column as a float (SQLT_FLT).

However, the DCOracle2 Python layer will pass type conversion overrides to the C layer if the object is a TypeCoercion(value, type). The input type override is either a string constant or a numeric value representing the SQL data type, e.g. 'SQLT_STR' for string. Unsupported column types can be coerced from string data in this fashion.

Output types are converted by lookup in a type table. This table is:

Type Name	Descripton	Coercion	Internal Type	Result Object
SQLT_CHR	Char		Yes	String
SQLT_NUM	Number	SQLT_STR	Yes	Automatic[1]
SQLT_INT	Integer		No	Integer
SQLT_FLT	Float		No	Float
SQLT_STR	String		No	String
SQLT_VNU	Length Prefixed Number		No	--
SQLT_PDN	Packed Decimal Number		No	--
SQLT_LNG	Long		Yes	String
SQLT_VCS	Variable Character		No	--
SQLT_NON	Null/Empty PCC desc		No	--
SQLT_RID	Row ID		Yes	--
SQLT_DAT	Date		Yes	OracleDate
SQLT_VBI	Binary VCS format		No	--
SQLT_BIN	Binary data		Yes	String
SQLT_LBI	Long binary data		Yes	String
SQLT_UIN	Unsigned Integer		No	--
SQLT_SLS	Display sign leading		No	--
SQLT_LVC	Longer longs (char)		No	--
SQLT_LVB	Longer longs (binary)		No	--
SQLT_AFC	ANSI Fixed Char		Yes	String
SQLT_AVC	ANSI Var Char		No	--
SQLT_CUR	Cursor		No	--
SQLT_RDD	Rowid Descriptor	SQLT_RID	No	--
SQLT_LAB	Label		No	--
SQLT_OSL	OS Label		No	--

[1]Automatic format means conversion to Float if the scale is greater than zero or the precision and scale are zero; conversion to Integer if the scale is less than 10, or conversion to Long otherwise.

SQLT_NTY	Named Type		Yes	--
SQLT_REF	Reference Type		Yes	--
SQLT_CLOB	CLOB		Yes	LobLocator
SQLT_BLOB	BLOB		Yes	LobLocator
SQLT_BFILE	Binary File LOB		No	--
SQLT_CFILE	Character File LOB		No	--
SQLT_RSET	Result set		No	--
SQLT_NCO	Named collection		No	--
SQLT_VST	OCIString type		No	--
SQLT_ODT	OCIDate type		No	--
SQLT_DATE	ANSI date		No	--
SQLT_TIME	Time		No	--
SQLT_TIME_TZ	Time with zone		No	--
SQLT_TIMESTAMP	Timestamp		No	--
SQLT_TIMESTAMP_TZ	Timestamp with zone		No	--

The result object column, if ''--'' means that there is no type converter for this type, and that only the raw interface from dco2's ResultSet object can get at the value. When an output type is coerced, it is requested from Oracle as the type specified instead of the original format.