
ORACLE®

Python and Oracle Database On the Table

TIP4076

Anthony Tuininga
Data Access Development
Oracle Database
October 23, 2018

ORACLE OPEN WORLD

ORACLE

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

About Me

Anthony Tuininga

- Creator and maintainer of cx_Oracle

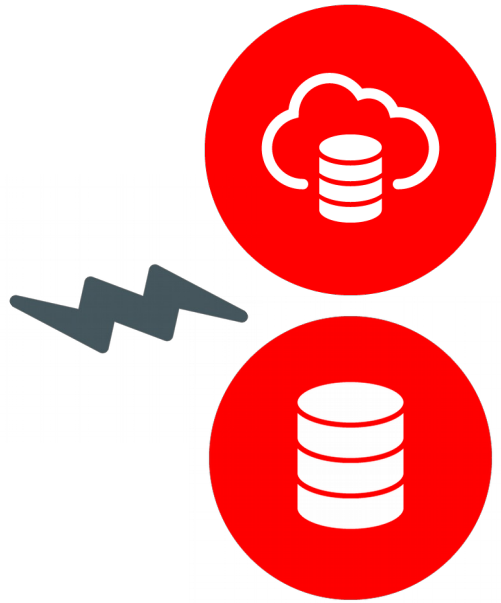
Contact information

- anthony.tuininga@oracle.com
- @AnthonyTuininga

Oracle Database for the Developer

LANGUAGE	API
C	OCI, ODBC, ODPI-C
C++	OCCI
Java	JDBC
.NET	ODP.NET
Node.js	node-oracledb
Python	cx_Oracle
PHP	OCI8, PDO_OCI
R	ROracle
Erlang	erloci
Perl	DBD::Oracle
Ruby	ruby-oci8, ruby-odpi
Rust	mimir, rust-oracle
Go	goracle, rana, mattn

- Oracle Proprietary Drivers
- Oracle Open Source Drivers
- Third Party Open Source Drivers



… and Pro*C, OLE DB, Pro*COBOL, Pro*Fortran, SQLJ, OLE DB

Program Agenda

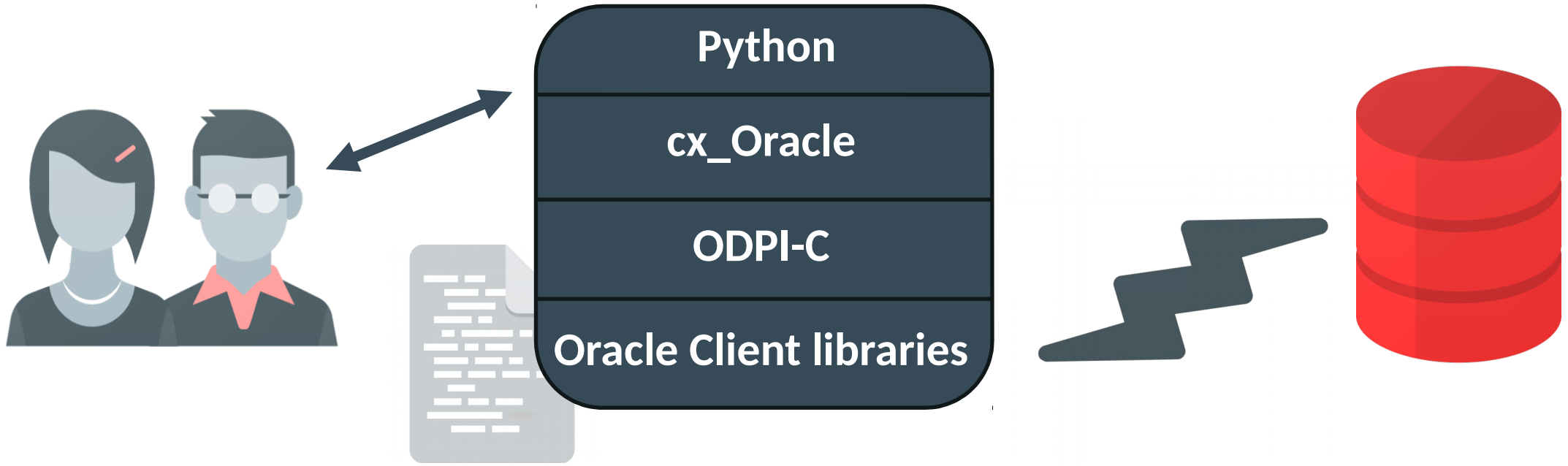
1. About cx_Oracle
2. SODA
3. Batch Execution
4. Named Object Types
5. Tips & Tricks

About cx_Oracle

cx_Oracle Background

- Open Source package giving Python access to Oracle Database
- Covers all of Python Database API specification
  - many additions to support advanced Oracle features
- Begun by Anthony Tuininga in 1998 using Oracle 8i and Python 1.5
  - 30+ releases covering Oracle 8i through 18c
- Latest release is cx_Oracle 7.0 (September 2018)

cx_Oracle Architecture



SODA

Simple Oracle Document Access (SODA)

- NoSQL-style APIs to create and access documents
  - Documents are often JSON
  - SODA calls are mapped to managed tables, indexes and sequences
  - Query-by-example makes data operations easy
  - Preview in cx_Oracle 7 with Oracle Database 18.3
- Recommend use of autocommit
  - But like with SQL, avoid autocommit when inserting lots of documents
  - DDL-like operations occur in autonomous transactions
- SODA APIs also exist in Node.js, C and Java

Creating/Populating SODA Collections


```
SQL> grant SODA_APP to pythondemo;
```

```
 1 conn = cx_Oracle.connect("pythondemo/welcome@localhost/orclpdb")
 2 conn.autocommit = True
 3 soda = conn.getSodaDatabase()
 4 coll = soda.createCollection("mycollection")
 5
 6 content = {'name': 'Matilda', 'address': {'city': 'Melbourne'}}
 7 coll.insertOne(content)
 8
 9 content = {'name': 'Sally', 'address': {'city': 'Singapore'}}
10 doc = coll.insertOneAndGet(content)
11 keyForSally = doc.key
12 print("Key is", keyForSally)
13
14 content = {'name': 'Mary', 'address': {'city': 'Madrid'}}
15 coll.insertOne(content)
```

Querying Collections


```
 1 doc = coll.find().key(keyForSally).getOne()
 2 print("Document content:", doc.getContent())
 3 print()
 4
 5 docs = coll.find().filter({'name': {'$like': 'Ma%'}}).getDocuments()
 6 names = [d.getContent()["name"] for d in docs]
 7 print("Names matching 'Ma%':", names)
 8 print()
 9
10 count = coll.find().count()
11 print('Collection has', count, 'documents')
```

```
Document content: {'name': 'Sally', 'address': {'city': 'Singapore'}}

Names matching 'Ma%': ['Matilda', 'Mary']

Collection has 3 documents
```

Updating Collections


```
1 content = {'name': 'Sally', 'address': {'city': 'Sydney'}}
2 coll.find().key(keyForSally).replaceOne(content)
3
4 count = coll.find().filter({'address.city': {'$regex': '.*o.*'}}).remove()
5 print("Removed", count, "documents")
6 print()
7
8 count = coll.find().count()
9 print('Collection has', count, 'documents')
```

```
Removed 1 documents

Collection has 2 documents
```

Batch Execution

Batch Execution

Execute one DML or PL/SQL statement with many data values

```
 1 rows = [ (1, "First" ),
 2          (2, "Second" ),
 3          (3, "Third" ),
 4          (4, "Fourth" ),
 5          (5, "Fifth" ),
 6          (6, "Sixth" ),
 7          (7, "Seventh" ) ]
 8
 9 cursor = connection.cursor()
10 cursor.executemany("insert into mytab(id, data) values (:1, :2)", rows)
```

Saves “round-trips” between Python and Oracle Database

Array DML Row Counts

- Enables application to get a count of the number of rows affected when executing a DML statement multiple times using `cursor.executemany()`
- Enabled using `cursor.executemany(..., arraydmlrowcounts = True)`
- Row counts are returned using `cursor.getarraydmlrowcounts()`
- Available when both Oracle client and database are 12.1 or higher
- Added in cx_Oracle 5.2

Array DML Row Counts Example


```
 1 create table ParentTable (
 2     ParentId number(9) not null,
 3     Description varchar2(60) not null,
 4     constraint ParentTable_pk primary key (ParentId)
 5 );
 6
 7 create table ChildTable (
 8     ChildId number(9) not null,
 9     ParentId number(9) not null,
10     Description varchar2(60) not null,
11     constraint ChildTable_pk primary key (ChildId),
12     constraint ChildTable_fk foreign key (ParentId) references ParentTable
13 );
14 . . .
```

Array DML Row Counts Example


```
1 parentIdsToDelete = [20, 30, 50]
2
3 cur.executemany("delete from ChildTable where ParentId = :1",
4                 [(i,) for i in parentIdsToDelete],
5                 arraydmlrowcounts = True)
6
7 rowCounts = cur.getarraydmlrowcounts()
8 for parentId, count in zip(parentIdsToDelete, rowCounts):
9     print("Parent ID:", parentId, "deleted", count, "rows.")
```

```
Parent ID: 20 deleted 3 rows.
Parent ID: 30 deleted 2 rows.
Parent ID: 50 deleted 4 rows.
```

Array DML Returning

- Enables application to use batch execution and return data in one call
- Added in cx_Oracle 7
  - With cx_Oracle 6.3 can set cx_Oracle.__future__.dml_ret_array_val = True

Array DML Returning Example


```
 1 parentIdsToDelete = [20, 30, 50]
 2
 3 var = cur.var(int, arraysize = len(parentIdsToDelete))
 4 cur.setinputsizes(None, var)
 5
 6 cur.executemany("delete from ChildTable where ParentId = :1 returning ChildId into :2",
 7                 [(i,) for i in parentIdsToDelete])
 8
 9 for parentId, childIds in zip(parentIdsToDelete, var.values):
10     print("Parent ID:", parentId, "deleted children with ids", childIds)
```

```
Parent ID: 20 deleted children with ids [1002, 1003, 1004]
Parent ID: 30 deleted children with ids [1005, 1006]
Parent ID: 50 deleted children with ids [1012, 1013, 1014, 1015]
```

Batch Errors

- Enables application to determine which rows resulted in an error when calling `executemany()` without causing the entire batch to fail
- Enabled using `cursor.executemany(..., batcherrors = True)`
- Batch errors are returned using `cursor.getbatcherrors()`
  - Each element of the list is an error object
- Can be combined with Array DML Row Counts feature
- Available when both Oracle client and database are 12.1 or higher
- Added in cx_Oracle 5.2

Batch Errors Example


```
 1 dataToInsert = [
 2     (1016, 10, 'Red'),
 3     (1018, 20, 'Blue'),
 4     (1018, 30, 'Green’),  # duplicate key
 5     (1022, 40, 'Yellow'),
 6     (1021, 75, 'Orange'), # parent does not exist
 7 ]
 8
 9 cur.executemany("insert into ChildTable values (:1, :2, :3)", dataToInsert,
10         batcherrors = True)
11 for error in cur.getbatcherrors():
12     print("Error", error.message, "at row offset", error.offset)
```

```
Error ORA-00001: unique constraint (PYTHONDEMO.CHILDTABLE_PK) violated at row offset 2
Error ORA-02291: integrity constraint (PYTHONDEMO.CHILDTABLE_FK) violated - parent key not
found at row offset 4
```

Named Object Types

Binding Named Object Types

- Support added for binding named object types such as SDO (Spatial Data Objects) in cx_Oracle 5.3
  - cx_Oracle 4.3 had the ability to *query* named object types
- Object types can be looked up using **`connection.gettype()`**
- Objects are created by calling **`newobject()`** on the object type or calling the object type directly
- Attributes can be read and written directly
- Elements can be added to a collection using **`append()`** or **`extend()`**

Binding Named Object Types Example


```
 1 create table TestGeometry (
 2     IntCol number(9) not null,
 3     Geometry SDO_GEOMETRY not null
 4 );
 5
 6 describe SDO_GEOMETRY
 7 Name                                      Null?    Type
 8  ----------------------------------------- -------- ----------------------------
 9  SDO_GTYPE                                           NUMBER
10  . . .
11  SDO_ELEM_INFO                                       MDSYS.SDO_ELEM_INFO_ARRAY
12  SDO_ORDINATES                                       MDSYS.SDO_ORDINATE_ARRAY
13  . . .
```

Binding Named Object Types Example


```
 1 typeObj = conn.gettype("SDO_GEOMETRY")
 2 elementInfoTypeObj = conn.gettype("SDO_ELEM_INFO_ARRAY")
 3 ordinateTypeObj = conn.gettype("SDO_ORDINATE_ARRAY")
 4
 5 obj = typeObj.newobject()   # or typeObj()
 6 obj.SDO_GTYPE = 2003
 7 obj.SDO_ELEM_INFO = elementInfoTypeObj.newobject()
 8 obj.SDO_ELEM_INFO.extend([1, 1003, 3])
 9 obj.SDO_ORDINATES = ordinateTypeObj.newobject()
10 obj.SDO_ORDINATES.extend([1, 1, 5, 7])
11
12 cur = conn.cursor()
13 cur.execute("insert into TestGeometry values (1, :objbv)", objbv = obj)
```

Binding PL/SQL Collections and Records

- Prior to cx_Oracle 5.3 only able to bind contiguously populated PL/SQL index-by tables (also known as PL/SQL arrays)
- In cx_Oracle 5.3 now able to bind (both IN and OUT) sparsely populated PL/SQL index-by tables as well as records
- PL/SQL types (index-by tables and records) defined in packages can be looked up using `connection.gettype()`
- Available when both client and database are 12.1 or higher

Binding PL/SQL Collections Example


```
 1 create or replace package pkg_Demo as
 2
 3    type udt_StringList is table of varchar2(100) index by binary_integer;
 4    . . .
 5
 6 create or replace package body pkg_Demo as
 7
 8   procedure DemoCollectionOut (
 9         a_Value out nocopy udt_StringList
10     ) is
11     begin
12         a_Value(-1048576) := 'First element';
13         a_Value(-576)     := 'Second element';
14         a_Value(284)      := 'Third element';
15         a_Value(8388608)  := 'Fourth element';
16     end;
17     . . .
```

Binding PL/SQL Collections Example


```
 1 typeObj = con.gettype("PKG_DEMO.UDT_STRINGLIST")
 2 obj = typeObj.newobject()
 3 cur.callproc("pkg_Demo.DemoCollectionOut", (obj,))
 4
 5 ix = obj.first()
 6 while ix is not None:
 7    print(ix, "->", obj.getelement(ix))
 8    ix = obj.next(ix)
 9
10 print(obj.aslist())
11 print(obj.asdict())
```

```
-1048576 -> First element
-576 -> Second element
284 -> Third element
8388608 -> Fourth element
```

```
['First element', 'Second element', 'Third element', 'Fourth element']
```

```
{-1048576: 'First element', -576: 'Second element',
284: 'Third element', 8388608: 'Fourth element'}
```

Binding PL/SQL Records Example


```
 1 create or replace package pkg_Demo as
 2     type udt_DemoRecord is record (
 3         NumberValue          number,
 4         StringValue          varchar2(30),
 5         DateValue            date,
 6         BooleanValue         boolean
 7     );
 8    . . .
 9
10 create or replace package body pkg_Demo as
11     procedure DemoRecordsInOut (
12         a_Value in out nocopy udt_DemoRecord
13     ) is
14     begin
15         a_Value.NumberValue := a_Value.NumberValue * 2;
16         a_Value.StringValue := a_Value.StringValue || ' (Modified)';
17         a_Value.DateValue := a_Value.DateValue + 5;
18         a_Value.BooleanValue := not a_Value.BooleanValue;
19     end;
```

Binding PL/SQL Records Example


```
 1 typeObj = con.gettype("PKG_DEMO.UDT_DEMORECORD")
 2 obj = typeObj.newobject()
 3
 4 obj.NUMBERVALUE = 6
 5 obj.STRINGVALUE = "Test String"
 6 obj.DATEVALUE = datetime.datetime(2016, 5, 28)
 7 obj.BOOLEANVALUE = False
 8
 9 cur.callproc("pkg_Demo.DemoRecordsInOut", (obj,))
10 print("NUMBERVALUE ->", obj.NUMBERVALUE)
11 print("STRINGVALUE ->", obj.STRINGVALUE)
12 print("DATEVALUE ->", obj.DATEVALUE)
13 print("BOOLEANVALUE ->", obj.BOOLEANVALUE)
```

```
NUMBERVALUE -> 12.0
STRINGVALUE -> Test String (Modified)
DATEVALUE -> 2016-06-02 00:00:00
BOOLEANVALUE -> True
```

Tips and Tricks

General Tips

- For queries use cursor attribute "`arraysize`"
  - Increased size reduces network round trips but uses additional memory
- Use bind variables instead of literal values
  - Reduces overhead of parsing
  - Improves security (eliminates SQL injection)
- Use `executemany()` for Array DML
- Use Session Pooling (optionally with Database Resident Connection Pooling)
  - reduces overhead for establishing and tearing down connections

Subclassing Connections and Cursors

- Enables application to “hook” connection/cursor creation and execution of statements
  - Logging of statement execution, connection creation/destruction
  - Changing the number of parameters to connection creation or statement execution
- Base classes are **`cx_Oracle.Connection`** and **`cx_Oracle.Cursor`**

Subclassing Connections and Cursors Example


```
 1 class Connection(cx_Oracle.Connection):
 2
 3   def __init__(self):
 4       print("CONNECT to a specific database")
 5       return super(Connection, self).__init__(dsn = "my_tns_entry")
 6
 7   def cursor(self):
 8       return Cursor(self)
 9
10 class Cursor(cx_Oracle.Cursor):
11
12   def execute(self, statement, args):
13       print("EXECUTE", statement)
14       print("ARGS:")
15       for argIndex, arg in enumerate(args):
16           print("   ", argIndex + 1, "=>", repr(arg))
17       return super(Cursor, self).execute(statement, args)
```

```
CONNECT to a specific database
EXECUTE select :1 from dual
ARGS:
    1 => 1234
```

Row Factory Functions

- Enables application to return an object other than a tuple when rows are returned from the various fetch routines
  - Can be used to provide names for the various columns
  - Can be used to return custom objects
- Enabled by setting cursor attribute "`rowfactory`" after the statement has been prepared or executed for the first time

Row Factory Functions Example


```
 1 import collections
 2
 3 cur.execute("select ParentId, Description from ParentTable")
 4 cur.rowfactory = collections.namedtuple("MyClass", ["newParentId", "newDescription"])
 5 for row in cursor:
 6    print(row.newParentId, "->", row.newDescription)
 7
 8 # Equivalent output from:
 9 # for i, d in cursor:
10 #    print(i, "->", d)
11 #
12 # for row in cursor:
13 #    print(row[0], "->", row[1])
```

```
1 -> Parent A
2 -> Parent B
. . .
```

Row Factory with Subclassing Example


```
 1 class Cursor(cx_Oracle.Cursor):
 2
 3     def execute(self, statement, args = None):
 4         prepareNeeded = (statement is not None and self.statement != statement)
 5         result = super(Cursor, self).execute(statement, args or [])
 6         if prepareNeeded and self.description:
 7             names = [d[0] for d in self.description]
 8             self.rowfactory = collections.namedtuple("GenericQuery", names)
 9         return result
10
11 for row in cur.execute("select ParentId, Description from ParentTable"):
12     print(row.PARENTID, "->", row.DESCRIPTION)
13
14 for row in cur.execute("""
15         select ParentId as "newParentId", Description as "newDescription"
16         from ParentTable"""):
17     print(row.newParentId, "->", row.newDescription)
```

```
1 -> Parent A
2 -> Parent B
. . .
```

Output Type Handlers

- Enables application to change how data is fetched from the database
  - Return numbers as strings or decimal objects
  - Return CLOB and NCLOB as strings or BLOB as bytes
  - In Python 2.x, return strings as Unicode objects
- Enabled by setting the attribute "`outputtypehandler`" on either the cursor or the connection
- Can be combined with variable converters to retrieve Python objects seamlessly

Output Type Handlers – Numbers as Strings


```
 1 for row in cur.execute("select * from ParentTable order by ParentId"):
 2     print(row)
 3
 4 def ReturnNumbersAsStrings(cursor, name, defaultType, size, precision, scale):
 5     if defaultType == cx_Oracle.NUMBER:
 6         return cursor.var(str, 9, cursor.arraysize)
 7
 8 cur.outputtypehandler = ReturnNumbersAsStrings
 9 for row in cur.execute("select * from ParentTable order by ParentId"):
10     print(row)
```

```
(1, 'Parent A')
(2, 'Parent B’)
. . .
```

```
('1', 'Parent A')
('2', 'Parent B’)
. . .
```

Output Type Handlers – Using a Decimal Converter


```
 1 for value, in cur.execute("select Data from DecimalTab"):
 2     print("Value:", value, "* 3 =", value * 3)
 3
 4 def ReturnNumbersAsDecimal(cursor, name, defaultType, size, precision, scale):
 5     if defaultType == cx_Oracle.NUMBER:
 6         return cursor.var(decimal.Decimal, arraysize = cursor.arraysize)
 7
 8 cur.outputtypehandler = ReturnNumbersAsDecimal
 9 for value, in cur.execute("select Data from DecimalTab"):
10     print("Value:", value, "* 3 =", value * 3)
```

Default output:

```
Value: 0.1 * 3 = 0.30000000000000004
Value: 3.1 * 3 = 9.3
Value: 7.100000000000005 * 3 = 21.3
```

Output Type Handler output:

```
Value: 0.1 * 3 = 0.3
Value: 3.1 * 3 = 9.3
Value: 7.1 * 3 = 21.3
```

Output Type Handlers – LOBs as Strings/Bytes


```
 1 def OutputTypeHandler(cursor, name, defaultType, size, precision, scale):
 2     if defaultType == cx_Oracle.CLOB:
 3         return cursor.var(cx_Oracle.LONG_STRING, arraysize = cursor.arraysize)
 4     elif defaultType == cx_Oracle.BLOB:
 5         return cursor.var(cx_Oracle.LONG_BINARY, arraysize = cursor.arraysize)
 6
 7 cur.outputtypehandler = OutputTypeHandler
 8
 9 for value, in cur.execute("select myclob from mytable"):
10     print(value)
```

```
This is your CLOB data . . .
```

Output Type Handlers – Object


```
 1 def BuildingOutConverter(oracleObj):
 2     return Building(int(oracleObj.BUILDINGID), oracleObj.DESCRIPTION,
 3             int(oracleObj.NUMFLOORS), oracleObj.DATEBUILT)
 4
 5 def OutputTypeHandler(cursor, name, defaultType, size, precision, scale):
 6     if defaultType == cx_Oracle.OBJECT:
 7         return cursor.var(cx_Oracle.OBJECT, arraysize = cursor.arraysize,
 8                 outconverter = BuildingOutConverter, typename = "UDT_BUILDING")
 9
10 cur.outputtypehandler = OutputTypeHandler
11 for row in cur.execute("select * from Buildings order by BuildingId"):
12     print(row)
```

```
(1, <Building 1: The First Building>)
(2, <Building 2: The Second Building>)
(3, <Building 3: The Third Building>)
```

Input Type Handlers

- Enables application to change how data is bound to statements or to enable new types to be bound directly without having to be converted first
- Enabled by setting the attribute `inputtypehandler` on either the cursor or the connection
- Can be combined with variable converters to bind Python objects seamlessly

Input Type Handlers Example


```
 1 def BuildingInConverter(value):
 2     obj = objType.newobject()
 3     obj.BUILDINGID  = value.buildingId
 4     obj.DESCRIPTION = value.description
 5     obj.NUMFLOORS   = value.numFloors
 6     obj.DATEBUILT   = value.dateBuilt
 7     return obj
 8
 9 def InputTypeHandler(cursor, value, numElements):
10     if isinstance(value, Building):
11         return cursor.var(cx_Oracle.OBJECT, arraysize = numElements,
12                 inconverter = BuildingInConverter, typename = objType.name)
13
14 objType = con.gettype("UDT_BUILDING")
15 building = Building(1, ”Skyscraper 1", 5, datetime.date(2001, 5, 24))
16 cur.inputtypehandler = InputTypeHandler
17 cur.execute("insert into mytable values (:1, :2)", (1, building))
```

ORACLE Ask TOM

Feedback anthony.tuininga@oracle.com

Questions Office Hours Resources About

Python and Oracle Database Office Hours

Free tips and training every month! Subscribe for reminders and more from Office Hours.

Subscribe

November 20, 2018

20:00 - 20:30 UTC Start Times Around the World Add to Calendar All Sessions Your Experts Resources

Exploring Python and Oracle Database

Join this free AskTOM Q&A session to get your Python cx_Oracle questions answered each month by Oracle product managers, developers and evangelists.

Each session has a theme, but feel free to ask us about anything about cx_Oracle.

https://tinyurl.com/PythonHours

ORACLE

Other Sessions: https://tinyurl.com/AppDevOOW18

- Python and Oracle Database 18c: Scripting for the Future [HOL6329]
  - Tuesday, Oct 23, 2:15 pm - 3:15 pm | Marriott Marquis (Yerba Buena Level) - Salon 3/4
- A Database Proxy for Transparent High Availability, Performance, Routing, and Security [TRN4070]
  - Wednesday, Oct 24, 11:15 am - 12:00 pm | Moscone West - Room 3009
- Meet the Experts: Node.js, Python, PHP, and Go with Oracle Database [MTE6765]
  - Wednesday, Oct 24, 3:00 pm - 3:50 pm | Moscone West - The Hub - Lounge B
- Node.js: Async Data In and Data Out with Oracle Database [TIP4080]
  - Thursday, Oct 25, 11:00 am - 11:45 am | Moscone West - Room 3009
- Performance and Scalability Techniques for Oracle Database Applications [TIP4075]
  - Thursday, Oct 25, 12:00 pm - 12:45 pm | Moscone West - Room 3009
- Demo Booth: APD-A03 Moscone South


ORACLE®