CX$_{Oracle}$

Release 8.0.0

September 01, 2020

Contents

cx_Oracle is a module that enables access to Oracle Database and conforms to the Python database API specification. This module is currently tested against Oracle Client 19c, 18c, 12c, and 11.2, and Python 3.5, 3.6, 3.7 and 3.8.

cx_Oracle is distributed under an open-source license (the BSD license). A detailed description of cx_Oracle changes can be found in the release notes.

Contents:

CHAPTER 1

User Guide

1.1 Introduction to cx_Oracle

cx_Oracle is a Python extension module that enables Python access to Oracle Database. It conforms to the Python Database API v2.0 Specification with a considerable number of additions and a couple of exclusions.

1.1.1 Architecture

Python programs call cx_Oracle functions. Internally cx_Oracle dynamically loads Oracle Client libraries to access Oracle Database. The database can be on the same machine as Python, or it can be remote.

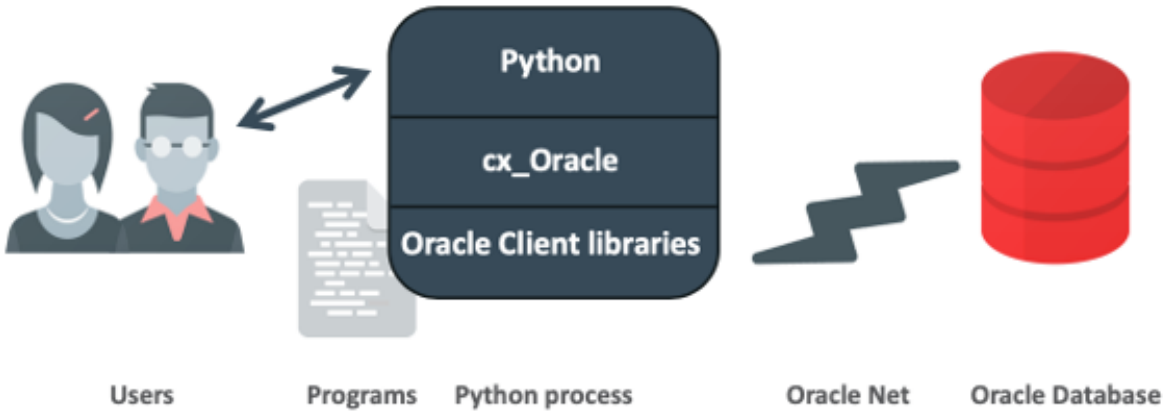

Fig. 1.1: cx_Oracle Architecture

cx_Oracle is typically installed from PyPI using pip. The Oracle Client libraries need to be installed separately. The libraries can be obtained from an installation of Oracle Instant Client, from a full Oracle Client installation, or even from an Oracle Database installation (if Python is running on the same machine as the database).

Some behaviors of the Oracle Client libraries can optionally be configured with an `oraaccess.xml` file, for example to enable auto-tuning of a statement cache. See *Optional Oracle Client Configuration Files*.

The Oracle Net layer can optionally be configured with files such as `tnsnames.ora` and `sqlnet.ora`, for example to enable *network encryption*. See *Optional Oracle Net Configuration Files*.

Oracle environment variables that are set before cx_Oracle first creates a database connection will affect cx_Oracle behavior. Optional variables include NLS_LANG, NLS_DATE_FORMAT and TNS_ADMIN. See *Oracle Environment Variables*.

1.1.2 Features

The cx_Oracle feature highlights are:

- Easily installed from PyPI
- Support for multiple Oracle Client and Database versions
- Execution of SQL and PL/SQL statements
- Extensive Oracle data type support, including large objects (CLOB and BLOB) and binding of SQL objects
- Connection management, including connection pooling
- Oracle Database High Availability features
- Full use of Oracle Network Service infrastructure, including encrypted network traffic and security features

A complete list of supported features can be seen here.

1.1.3 Getting Started

Install cx_Oracle using the *installation* steps.

Create a script `query.py` as shown below:

```
# query.py

import cx_Oracle

# Establish the database connection
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1")

# Obtain a cursor
cursor = connection.cursor()

# Data for binding
managerId = 145
firstName = "Peter"

# Execute the query
sql = """SELECT first_name, last_name
         FROM employees
         WHERE manager_id = :mid AND first_name = :fn"""
cursor.execute(sql, mid = managerId, fn = firstName)
```

(continues on next page)

(continued from previous page)

```
# Loop over the result set
for row in cursor:
    print(row)
```

This uses Oracle's sample HR schema.

Simple *connection* to the database requires a username, password and connection string. Locate your Oracle Database user name and password and the database *connection string*, and use them in `query.py`. For cx_Oracle the connection string is commonly of the format `hostname/servicename`, using the host name where the database is running and the Oracle Database service name of the database instance.

The *cursor* is the object that allows statements to be executed and results (if any) fetched.

The data values in `managerId` and `firstName` are 'bound' to the statement placeholder 'bind variables' `:mid` and `:fn` when the statement is executed. This separates the statement text from the data, which helps avoid SQL Injection security risks. *Binding* is also important for performance and scalability.

The cursor allows rows to be iterated over and displayed.

Run the script:

```
python query.py
```

The output is:

```
('Peter', 'Hall')
('Peter', 'Tucker')
```

1.1.4 Examples and Tutorials

Runnable examples are in the GitHub samples directory. A Python cx_Oracle tutorial is also available.

1.2 cx_Oracle 8 Installation

1.2.1 Overview

To use cx_Oracle 8 with Python and Oracle Database you need:

- Python 3.5 and higher. Older versions of cx_Oracle may work with older versions of Python, for example see *Installing cx_Oracle in Python 2*
- Oracle Client libraries. These can be from the free Oracle Instant Client, or those included in Oracle Database if Python is on the same machine as the database. Oracle client libraries versions 19, 18, 12, and 11.2 are supported on Linux, Windows and macOS. Users have also reported success with other platforms. Use the latest client possible: Oracle's standard client-server version interoperability allows connection to both older and newer databases.
- An Oracle Database, either local or remote.

The cx_Oracle module loads Oracle Client libraries which communicate over Oracle Net to an existing database. Oracle Net is not a separate product: it is how the Oracle Client and Oracle Database communicate.

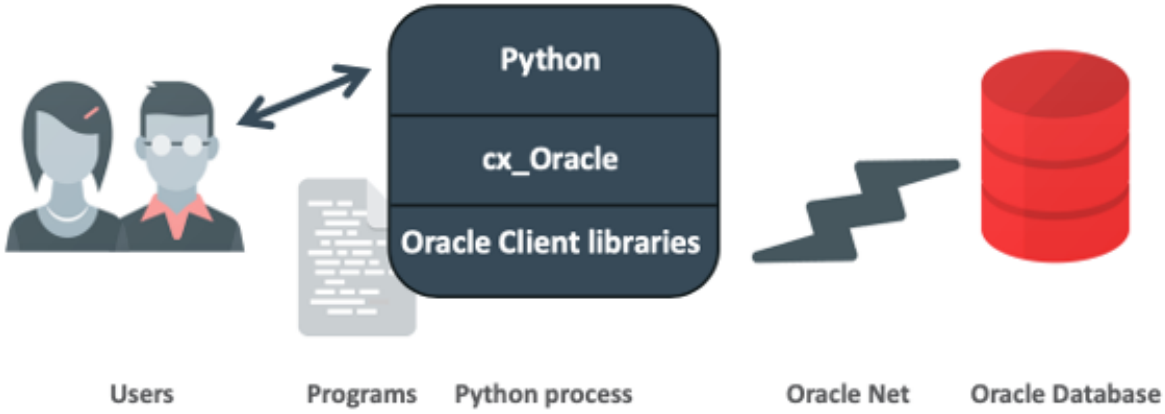

Fig. 1.2: cx_Oracle Architecture

1.2.2 Quick Start cx_Oracle Installation

- Install Python 3, if not already available. On macOS you must always install your own Python.

  Python 3.5 and higher are supported by cx_Oracle 8. For Python 2, see *Installing cx_Oracle in Python 2*.

- Install cx_Oracle from PyPI with:

```
python -m pip install cx_Oracle --upgrade
```

  Note: if a binary wheel package is not available for your platform, the source package will be downloaded instead. This will be compiled and the resulting binary installed.

  The `--user` option may be useful, if you don't have permission to write to system directories:

```
python -m pip install cx_Oracle --upgrade --user
```

  If you are behind a proxy, add a proxy server to the command, for example add `--proxy=http://proxy.example.com:80`

- Add Oracle 19, 18, 12 or 11.2 client libraries to your operating system library search path such as `PATH` on Windows or `LD_LIBRARY_PATH` on Linux. On macOS use *`init_oracle_client()`* in your application to pass the Oracle Client directory name, see *Locating the Oracle Client Libraries*. This is also usable on Windows.

  To get the libraries:

  – If your database is on a remote computer, then download and unzip the client libraries from the free Oracle Instant Client "Basic" or "Basic Light" package for your operating system architecture.

    Instant Client on Windows requires an appropriate Microsoft Windows Redistributables, see *Installing cx_Oracle on Windows*. On Linux, the `libaio` (sometimes called `libaio1`) package is needed. Oracle Linux 8 also needs the `libnsl` package.

  – Alternatively, use the client libraries already available in a locally installed database such as the free Oracle Database Express Edition ("XE") release.

  Version 19, 18 and 12.2 client libraries can connect to Oracle Database 11.2 or greater. Version 12.1 client libraries can connect to Oracle Database 10.2 or greater. Version 11.2 client libraries can connect to Oracle

Database 9.2 or greater.

- Create a script like the one below:

```
# myscript.py

import cx_Oracle

# Connect as user "hr" with password "welcome" to the "orclpdb1" service running␣
↪on this computer.
connection = cx_Oracle.connect("hr", "welcome", "localhost/orclpdb1")

cursor = connection.cursor()
cursor.execute("""
        SELECT first_name, last_name
        FROM employees
        WHERE department_id = :did AND employee_id > :eid""",
        did = 50,
        eid = 190)
for fname, lname in cursor:
    print("Values:", fname, lname)
```

Locate your Oracle Database username and password, and the database connection string. The connection string is commonly of the format `hostname/servicename`, using the hostname where the database is running, and using the service name of the Oracle Database instance.

Substitute your username, password and connection string in the code. Run the Python script, for example:

```
python myscript.py
```

You can learn how to use cx_Oracle from the *API documentation* and samples.

If you run into installation trouble, check out the section on *Troubleshooting*.

1.2.3 Oracle Client and Oracle Database Interoperability

cx_Oracle requires Oracle Client libraries. The libraries provide the necessary network connectivity to access an Oracle Database instance. They also provide basic and advanced connection management and data features to cx_Oracle.

The simplest way to get Oracle Client libraries is to install the free Oracle Instant Client "Basic" or "Basic Light" package. The libraries are also available in any Oracle Database installation or full Oracle Client installation.

Oracle's standard client-server network interoperability allows connections between different versions of Oracle Client libraries and Oracle Database. For certified configurations see Oracle Support's Doc ID 207303.1. In summary, Oracle Client 19, 18 and 12.2 can connect to Oracle Database 11.2 or greater. Oracle Client 12.1 can connect to Oracle Database 10.2 or greater. Oracle Client 11.2 can connect to Oracle Database 9.2 or greater. The technical restrictions on creating connections may be more flexible. For example Oracle Client 12.2 can successfully connect to Oracle Database 10.2.

cx_Oracle uses the shared library loading mechanism available on each supported platform to load the Oracle Client libraries at runtime. It does not need to be rebuilt for different versions of the libraries. Since a single cx_Oracle binary can use different client versions and also access multiple database versions, it is important your application is tested in your intended release environments. Newer Oracle clients support new features, such as the oraaccess.xml external configuration file available with 12.1 or later clients, session pool improvements, improved high availability features, call timeouts, and other enhancements.

The cx_Oracle function *clientversion()* can be used to determine which Oracle Client version is in use and the attribute *Connection.version* can be used to determine which Oracle Database version a connection is

accessing. These can then be used to adjust application behavior accordingly. Attempts to use some Oracle features that are not supported by a particular client/server combination may result in runtime errors. These include:

- when attempting to access attributes that are not supported by the current Oracle Client library you will get the error "ORA-24315: illegal attribute type"
- when attempting to use implicit results with Oracle Client 11.2 against Oracle Database 12c you will get the error "ORA-29481: Implicit results cannot be returned to client"
- when attempting to get array DML row counts with Oracle Client 11.2 you will get the error "DPI-1050: Oracle Client library must be at version 12.1 or higher"

1.2.4 Installing cx_Oracle on Linux

This section discusses the generic installation methods on Linux. To use Python and cx_Oracle RPM packages from yum on Oracle Linux, see *Installing cx_Oracle RPMs on Oracle Linux*.

Install cx_Oracle

The generic way to install cx_Oracle on Linux is to use Python's Pip package to install cx_Oracle from PyPI:

```
python -m pip install cx_Oracle --upgrade
```

The `--user` option may be useful, if you don't have permission to write to system directories:

```
python -m pip install cx_Oracle --upgrade --user
```

If you are behind a proxy, add a proxy server to the command, for example add `--proxy=http://proxy.example.com:80`

This will download and install a pre-compiled binary if one is available for your architecture. If a pre-compiled binary is not available, the source will be downloaded, compiled, and the resulting binary installed. Compiling cx_Oracle requires the `Python.h` header file. If you are using the default `python` package, this file is in the `python-devel` package or equivalent.

Install Oracle Client

Using cx_Oracle requires Oracle Client libraries to be installed. These provide the necessary network connectivity allowing cx_Oracle to access an Oracle Database instance.

- If your database is on a remote computer, then download the free Oracle Instant Client "Basic" or "Basic Light" package for your operating system architecture. Use the RPM or ZIP packages, based on your preferences.
- Alternatively, use the client libraries already available in a locally installed database such as the free Oracle Database Express Edition ("XE") release.

Oracle Instant Client Zip Files

To use cx_Oracle with Oracle Instant Client zip files:

1. Download an Oracle 19, 18, 12, or 11.2 "Basic" or "Basic Light" zip file: 64-bit or 32-bit, matching your Python architecture.

   The latest version is recommended. Oracle Instant Client 19 will connect to Oracle Database 11.2 or later.

2. Unzip the package into a single directory that is accessible to your application. For example:

```
mkdir -p /opt/oracle
cd /opt/oracle
unzip instantclient-basic-linux.x64-19.6.0.0.0dbru.zip
```

3. Install the `libaio` package with sudo or as the root user. For example:

```
sudo yum install libaio
```

   On some Linux distributions this package is called `libaio1` instead.

   On recent Linux versions, such as Oracle Linux 8, you may also need to install the `libnsl` package.

4. If there is no other Oracle software on the machine that will be impacted, permanently add Instant Client to the runtime link path. For example, with sudo or as the root user:

```
sudo sh -c "echo /opt/oracle/instantclient_19_6 > /etc/ld.so.conf.d/oracle-
↪instantclient.conf"
sudo ldconfig
```

   Alternatively, set the environment variable `LD_LIBRARY_PATH` to the appropriate directory for the Instant Client version. For example:

```
export LD_LIBRARY_PATH=/opt/oracle/instantclient_19_6:$LD_LIBRARY_PATH
```

5. If you use optional Oracle configuration files such as `tnsnames.ora`, `sqlnet.ora` or `oraaccess.xml` with Instant Client, then put the files in an accessible directory, for example in `/opt/oracle/your_config_dir`. Then use:

```
import cx_Oracle
cx_Oracle.init_oracle_client(config_dir="/home/your_username/oracle/your_config_
↪dir")
```

   Or set the environment variable `TNS_ADMIN` to that directory name.

   Alternatively, put the files in the `network/admin` subdirectory of Instant Client, for example in `/opt/oracle/instantclient_19_6/network/admin`. This is the default Oracle configuration directory for executables linked with this Instant Client.

Oracle Instant Client RPMs

To use cx_Oracle with Oracle Instant Client RPMs:

1. Download an Oracle 19, 18, 12, or 11.2 "Basic" or "Basic Light" RPM: 64-bit or 32-bit, matching your Python architecture.

   Oracle's yum server has Instant Client RPMs for Oracle Linux 7 and Instant Client RPMs for Oracle Linux 6 that can be downloaded without needing a click-through.

   The latest version is recommended. Oracle Instant Client 19 will connect to Oracle Database 11.2 or later.

2. Install the downloaded RPM with sudo or as the root user. For example:

```
sudo yum install oracle-instantclient19.6-basic-19.6.0.0.0-1.x86_64.rpm
```

   Yum will automatically install required dependencies, such as `libaio`.

   On recent Linux versions, such as Oracle Linux 8, you may need to manually install the `libnsl` package.

3. For Instant Client 19, the system library search path is automatically configured during installation.

   For older versions, if there is no other Oracle software on the machine that will be impacted, permanently add Instant Client to the runtime link path. For example, with sudo or as the root user:

   ```
   sudo sh -c "echo /usr/lib/oracle/18.3/client64/lib > /etc/ld.so.conf.d/oracle-
   ↪instantclient.conf"
   sudo ldconfig
   ```

   Alternatively, for version 18 and earlier, every shell running Python will need to have the environment variable `LD_LIBRARY_PATH` set to the appropriate directory for the Instant Client version. For example:

   ```
   export LD_LIBRARY_PATH=/usr/lib/oracle/18.3/client64/lib:$LD_LIBRARY_PATH
   ```

4. If you use optional Oracle configuration files such as `tnsnames.ora`, `sqlnet.ora` or `oraaccess.xml` with Instant Client, then put the files in an accessible directory, for example in `/opt/oracle/your_config_dir`. Then use:

   ```
   import cx_Oracle
   cx_Oracle.init_oracle_client(config_dir="/opt/oracle/your_config_dir")
   ```

   Or set the environment variable `TNS_ADMIN` to that directory name.

   Alternatively, put the files in the `network/admin` subdirectory of Instant Client, for example in `/usr/lib/oracle/19.6/client64/lib/network/admin`. This is the default Oracle configuration directory for executables linked with this Instant Client.

Local Database or Full Oracle Client

cx_Oracle applications can use Oracle Client 19, 18, 12, or 11.2 libraries from a local Oracle Database or full Oracle Client installation.

The libraries must be either 32-bit or 64-bit, matching your Python architecture.

1. Set required Oracle environment variables by running the Oracle environment script. For example:

   ```
   source /usr/local/bin/oraenv
   ```

   For Oracle Database Express Edition ("XE") 11.2, run:

   ```
   source /u01/app/oracle/product/11.2.0/xe/bin/oracle_env.sh
   ```

2. Optional Oracle configuration files such as `tnsnames.ora`, `sqlnet.ora` or `oraaccess.xml` can be placed in `$ORACLE_HOME/network/admin`.

   Alternatively, Oracle configuration files can be put in another, accessible directory. Then set the environment variable `TNS_ADMIN` to that directory name.

1.2.5 Installing cx_Oracle RPMs on Oracle Linux

Python and cx_Oracle RPM packages are available from the Oracle Linux yum server. Various versions of Python are easily installed. Using the yum server makes it easy to keep up to date.

Installation instructions are at Oracle Linux for Python Developers.

1.2.6 Installing cx_Oracle on Windows

Install cx_Oracle

Use Python's Pip package to install cx_Oracle from PyPI:

```
python -m pip install cx_Oracle --upgrade
```

If you are behind a proxy, specify your proxy server:

```
python -m pip install cx_Oracle --proxy=http://proxy.example.com:80 --upgrade
```

This will download and install a pre-compiled binary if one is available for your architecture. If a pre-compiled binary is not available, the source will be downloaded, compiled, and the resulting binary installed.

Install Oracle Client

Using cx_Oracle requires Oracle Client libraries to be installed. These provide the necessary network connectivity allowing cx_Oracle to access an Oracle Database instance. Oracle Client versions 19, 18, 12 and 11.2 are supported.

- If your database is on a remote computer, then download the free Oracle Instant Client "Basic" or "Basic Light" package for your operating system architecture.
- Alternatively, use the client libraries already available in a locally installed database such as the free Oracle Database Express Edition ("XE") release.

Oracle Instant Client Zip Files

To use cx_Oracle with Oracle Instant Client zip files:

1. Download an Oracle 19, 18, 12, or 11.2 "Basic" or "Basic Light" zip file: 64-bit or 32-bit, matching your Python architecture.

   The latest version is recommended. Oracle Instant Client 19 will connect to Oracle Database 11.2 or later.

   Windows 7 users: Note that Oracle 19c is not supported on Windows 7.

2. Unzip the package into a directory that is accessible to your application. For example unzip `instantclient-basic-windows.x64-19.6.0.0.0dbru.zip` to `C:\oracle\instantclient_19_6`.

3. There are several alternative ways to tell cx_Oracle where your Oracle Client libraries are, see *cx_Oracle 8 Initialization*.

   - With Oracle Instant Client you can use *`init_oracle_client()`* in your application, for example:

     ```
     import cx_Oracle
     cx_Oracle.init_oracle_client(lib_dir=r"C:\oracle\instantclient_19_6")
     ```

     Note a 'raw' string is used because backslashes occur in the path.

   - Alternatively, add the Oracle Instant Client directory to the `PATH` environment variable. The directory must occur in `PATH` before any other Oracle directories. Restart any open command prompt windows.

   - Another way to set `PATH` is to use a batch file that sets it before Python is executed, for example:

```
REM mypy.bat
SET PATH=C:\oracle\instantclient_19_6;%PATH%
python %*
```

Invoke this batch file every time you want to run Python.

4. Oracle Instant Client libraries require a Visual Studio redistributable with a 64-bit or 32-bit architecture to match Instant Client's architecture. Each Instant Client version requires a different redistributable version:
   - For Instant Client 19 install VS 2017.
   - For Instant Client 18 or 12.2 install VS 2013
   - For Instant Client 12.1 install VS 2010
   - For Instant Client 11.2 install VS 2005 64-bit or VS 2005 32-bit
5. If you use optional Oracle configuration files such as `tnsnames.ora`, `sqlnet.ora` or `oraaccess.xml` with Instant Client, then put the files in an accessible directory, for example in `C:\oracle\your_config_dir`. Then use:

```
import cx_Oracle
cx_Oracle.init_oracle_client(config_dir=r"C:\oracle\your_config_dir")
```

Or set the environment variable `TNS_ADMIN` to that directory name.

Alternatively, put the files in a `network\admin` subdirectory of Instant Client, for example in `C:\oracle\instantclient_19_6\network\admin`. This is the default Oracle configuration directory for executables linked with this Instant Client.

Local Database or Full Oracle Client

cx_Oracle applications can use Oracle Client 19, 18, 12, or 11.2 libraries libraries from a local Oracle Database or full Oracle Client.

The Oracle libraries must be either 32-bit or 64-bit, matching your Python architecture.

1. Set the environment variable `PATH` to include the path that contains `OCI.DLL`, if it is not already set.

   Restart any open command prompt windows.

2. Optional Oracle configuration files such as `tnsnames.ora`, `sqlnet.ora` or `oraaccess.xml` can be placed in the `network\admin` subdirectory of the Oracle Database software installation.

   Alternatively, pass `config_dir` to *`init_oracle_client()`* as shown in the previous section, or set `TNS_ADMIN` to the directory name.

1.2.7 Installing cx_Oracle on macOS

Install Python

Make sure you are not using a bundled Python. These have restricted entitlements and will fail to load Oracle client libraries. Instead use Homebrew or Python.org.

Note Instant Client 19 and earlier are not supported on macOS 10.15 Catalina. You will need to allow access to several Instant Client libraries from the Security & Privacy preference pane.

Install cx_Oracle

Use Python's Pip package to install cx_Oracle from PyPI:

```
python -m pip install cx_Oracle --upgrade
```

The `--user` option may be useful, if you don't have permission to write to system directories:

```
python -m pip install cx_Oracle --upgrade --user
```

If you are behind a proxy, add a proxy server to the command, for example add `--proxy=http://proxy.example.com:80`

The source will be downloaded, compiled, and the resulting binary installed.

Install Oracle Instant Client

cx_Oracle requires Oracle Client libraries, which are found in Oracle Instant Client for macOS. These provide the necessary network connectivity allowing cx_Oracle to access an Oracle Database instance.

To use cx_Oracle with Oracle Instant Client zip files:

1. Download the Oracle 19, 18, 12 or 11.2 "Basic" or "Basic Light" zip file from here. Choose either a 64-bit or 32-bit package, matching your Python architecture.

   The latest version is recommended. Oracle Instant Client 19 will connect to Oracle Database 11.2 or later.

2. Unzip the package into a single directory that is accessible to your application. For example, in Terminal you could unzip in your home directory:

   ```
   cd ~
   unzip instantclient-basic-macos.x64-19.3.0.0.0dbru.zip
   ```

   This will create a directory `/Users/your_username/instantclient_19_3`.

3. There are several alternative ways to tell cx_Oracle where your Oracle Instant Client libraries are, see *cx_Oracle 8 Initialization*.

   - You can use *`init_oracle_client()`* in your application:

     ```
     import cx_Oracle
     cx_Oracle.init_oracle_client(lib_dir="/Users/your_username/instantclient_19_3
     ↪")
     ```

   - Alternatively, locate the directory with the cx_Oracle module binary and link or copy Oracle Instant Client to that directory. For example, if you installed cx_Oracle with `--user` in Python 3.8, then `cx_Oracle.cpython-38-darwin.so` might be in `~/Library/Python/3.8/lib/python/site-packages`. You can then run `ln -s ~/instantclient_19_3/libclntsh.dylib ~/Library/Python/3.8/lib/python/site-packages` or copy the Instant Client libraries to that directory.

   - Alternatively, you can set `DYLD_LIBRARY_PATH` to the directory containing Oracle Instant Client, however this needs to be set in each terminal or process that invokes Python. The variable will not propagate to sub-shells.

   - Alternatively, on older versions of macOS, you could add a link to `$HOME/lib` or `/usr/local/lib` to enable applications to find Instant Client. If the `lib` sub-directory does not exist, you can create it. For example:

```
mkdir ~/lib
ln -s ~/instantclient_19_3/libclntsh.dylib ~/lib/
```

Instead of linking, you can copy the required OCI libraries. For example:

```
mkdir ~/lib
cp ~/instantclient_19_3/{libclntsh.dylib.19.1,libclntshcore.dylib.19.1,
↪libnnz19.dylib,libociei.dylib} ~/lib/
```

For Instant Client 11.2, the OCI libraries must be copied. For example:

```
mkdir ~/lib
cp ~/instantclient_11_2/{libclntsh.dylib.11.1,libnnz11.dylib,libociei.dylib} ~
↪/lib/
```

4. If you use optional Oracle configuration files such as `tnsnames.ora`, `sqlnet.ora` or `oraaccess.xml` with Oracle Instant Client, then put the files in an accessible directory, for example in `/Users/your_username/oracle/your_config_dir`. Then use:

```
import cx_Oracle
cx_Oracle.init_oracle_client(config_dir="/Users/your_username/oracle/your_config_
↪dir")
```

Or set the environment variable `TNS_ADMIN` to that directory name.

Alternatively, put the files in the `network/admin` subdirectory of Oracle Instant Client, for example in `/Users/your_username/instantclient_19_3/network/admin`. This is the default Oracle configuration directory for executables linked with this Instant Client.

1.2.8 Installing cx_Oracle without Internet Access

To install cx_Oracle on a computer that is not connected to the internet, download the appropriate cx_Oracle file from PyPI. Transfer this file to the offline computer and install it with:

```
python -m pip install "<file_name>"
```

Then follow the general cx_Oracle platform installation instructions to install Oracle client libraries.

1.2.9 Install Using GitHub

In order to install using the source on GitHub, use the following commands:

```
git clone https://github.com/oracle/python-cx_Oracle.git cx_Oracle
cd cx_Oracle
git submodule init
git submodule update
python setup.py install
```

Note that if you download a source zip file directly from GitHub then you will also need to download an ODPI-C source zip file and extract it inside the directory called "odpi".

cx_Oracle source code is also available from oss.oracle.com. This can be cloned with:

```
git clone git://oss.oracle.com/git/oracle/python-cx_Oracle.git cx_Oracle
cd cx_Oracle
git submodule init
git submodule update
```

1.2.10 Install Using Source from PyPI

The source package can be downloaded manually from PyPI and extracted, after which the following commands should be run:

```
python setup.py build
python setup.py install
```

1.2.11 Upgrading from Older Versions

Review the release notes for deprecations and modify any affected code.

If you are upgrading from cx_Oracle 7 note these changes:

- The default character set used by cx_Oracle 8 is now "UTF-8". Also, the character set component of the `NLS_LANG` environment variable is ignored. If you need to change the character set, then pass `encoding` and `nendcoding` parameters when creating a connection or connection pool. See *Character Sets and Globalization*.
- Any uses of `type(var)` need to be changed to `var.type`.
- Any uses of `var.type is not None` need to be changed to `isinstance(var.type, cx_Oracle.ObjectType)`
- Note that `TIMESTAMP WITH TIME ZONE` columns will now be reported as *`cx_Oracle.DB_TYPE_TIMESTAMP_TZ`* instead of *`cx_Oracle.TIMESTAMP`* in *`Cursor.description`*.
- Note that `TIMESTAMP WITH LOCAL TIME ZONE` columns will now be reported as *`cx_Oracle.DB_TYPE_TIMESTAMP_LTZ`* instead of *`cx_Oracle.TIMESTAMP`* in *`Cursor.description`*.
- Note that `BINARY_FLOAT` columns will now be reported as *`cx_Oracle.DB_TYPE_BINARY_FLOAT`* instead of `cx_Oracle.NATIVE_DOUBLE` in *`Cursor.description`*.

If you are upgrading from cx_Oracle 5 note these installation changes:

- When using Oracle Instant Client, you should not set `ORACLE_HOME`.
- On Linux, cx_Oracle 6 and higher no longer uses Instant Client RPMs automatically. You must set `LD_LIBRARY_PATH` or use `ldconfig` to locate the Oracle client library.
- PyPI no longer allows Windows installers or Linux RPMs to be hosted. Use the supplied cx_Oracle Wheels instead, or use RPMs from Oracle, see *Installing cx_Oracle RPMs on Oracle Linux*.

1.2.12 Installing cx_Oracle in Python 2

To install cx_Oracle in Python 2, use a command like:

```
python -m pip install cx_Oracle==7.3 --upgrade --user
```

cx_Oracle 7.3 was the last version with support for Python 2.

For other installation options such as installing through a proxy, see instructions above. Make sure the Oracle Client libraries are in the system library search path because cx_Oracle 7 does not support the *cx_Oracle.init_oracle_client()* method and does not support loading the Oracle Client libraries from the directory containing the cx_Oracle module binary.

1.2.13 Installing cx_Oracle 5.3

If you require cx_Oracle 5.3, download a Windows installer from PyPI or use `python -m pip install cx-oracle==5.3` to install from source.

Very old versions of cx_Oracle can be found in the files section at SourceForce.

1.2.14 Troubleshooting

If installation fails:

- Use option `-v` with pip. Review your output and logs. Try to install using a different method. **Google anything that looks like an error.** Try some potential solutions.
- Was there a network connection error? Do you need to set the environment variables `http_proxy` and/or `https_proxy`? Or try `pip install --proxy=http://proxy.example.com:80 cx_Oracle --upgrade`?
- If upgrading gave no errors but the old version is still installed, try `pip install cx_Oracle --upgrade --force-reinstall`
- If you do not have access to modify your system version of Python, can you use `pip install cx_Oracle --upgrade --user` or venv?
- Do you get the error "`No module named pip`"? The pip module is builtin to Python but is sometimes removed by the OS. Use the venv module (builtin to Python 3.x) or virtualenv module instead.
- Do you get the error "`fatal error: dpi.h: No such file or directory`" when building from source code? Ensure that your source installation has a subdirectory called "odpi" containing files. If missing, review the section on *Install Using GitHub*.

If using cx_Oracle fails:

- Do you get the error "`DPI-1047: Oracle Client library cannot be loaded`"?
  - Check that Python, cx_Oracle and your Oracle Client libraries are all 64-bit or all 32-bit. The `DPI-1047` message will tell you whether the 64-bit or 32-bit Oracle Client is needed for your Python.
  - On Windows, if you used *init_oracle_client()* and have a full database installation, make sure this database is the currently configured database.
  - On Windows, if you are not using *init_oracle_client()*, then restart your command prompt and use `set PATH` to check the environment variable has the correct Oracle Client listed before any other Oracle directories.
  - On Windows, use the `DIR` command to verify that `OCI.DLL` exists in the directory passed to `init_oracle_client()` or set in `PATH`.
  - On Windows, check that the correct Windows Redistributables have been installed.
  - On Linux, check the `LD_LIBRARY_PATH` environment variable contains the Oracle Client library directory. If you are using Oracle Instant Client, a preferred alternative is to ensure a file in the `/etc/ld.so.conf.d` directory contains the path to the Instant Client directory, and then run `ldconfig`.

  - On macOS, make sure you are not using the bundled Python (use Homebrew or Python.org instead). If you are not using *init_oracle_client()*, then put the Oracle Instant Client libraries in ~/lib or /usr/local/lib.

- If you got "DPI-1072: the Oracle Client library version is unsupported", then review the installation requirements. cx_Oracle needs Oracle client libraries 11.2 or later. Note that version 19 is not supported on Windows 7. Similar steps shown above for DPI-1047 may help.
- If you have multiple versions of Python installed, make sure you are using the correct python and pip (or python3 and pip3) executables.

1.3 cx_Oracle 8 Initialization

The cx_Oracle module loads Oracle Client libraries which communicate over Oracle Net to an existing database. The Oracle Client libraries need to be installed separately. See *cx_Oracle 8 Installation*. Oracle Net is not a separate product: it is how the Oracle Client and Oracle Database communicate.

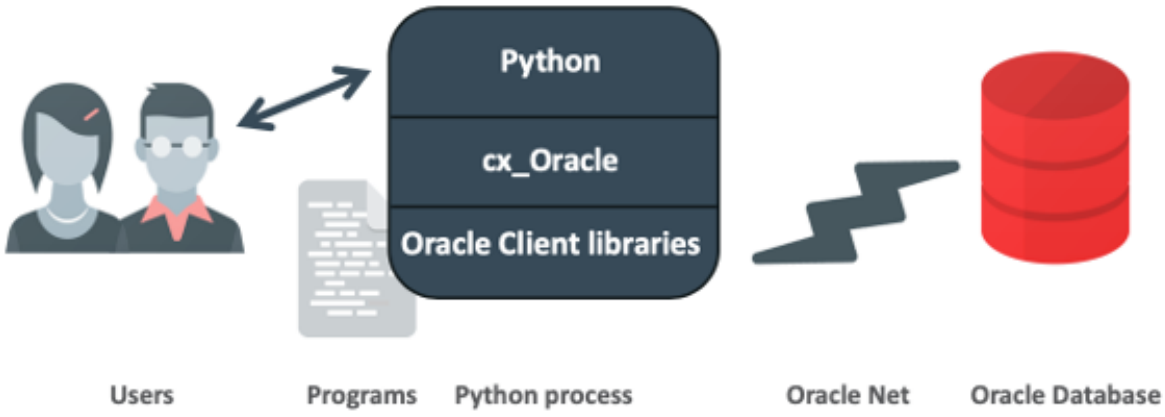

Fig. 1.3: cx_Oracle Architecture

1.3.1 Locating the Oracle Client Libraries

cx_Oracle dynamically loads the Oracle Client libraries using a search heuristic. Only the first set of libraries found are loaded. The libraries can be in an installation of Oracle Instant Client, in a full Oracle Client installation, or in an Oracle Database installation (if Python is running on the same machine as the database). The versions of Oracle Client and Oracle Database do not have to be the same. For certified configurations see Oracle Support's Doc ID 207303.1.

cx_Oracle looks for the Oracle Client libraries as follows:

- On Windows:
  - In the lib_dir directory specified in a call to *cx_Oracle.init_oracle_client()*. This directory should contain the libraries from an unzipped Instant Client 'Basic' or 'Basic Light' package. If you pass the library directory from a full client or database installation, such as Oracle Database "XE" Express Edition, then you will need to have previously set your environment to use that software installation, otherwise files such as message files will not be located. On Windows when the path contains backslashes, use

a 'raw' string like `lib_dir = r"C:\instantclient_19_6"`. If the Oracle Client libraries cannot be loaded from `lib_dir`, then an exception is raised.

  - If `lib_dir` was not specified, then Oracle Client libraries are looked for in the directory where the cx_Oracle binary module is installed. This directory should contain the libraries from an unzipped Instant Client 'Basic' or 'Basic Light' package. If the libraries are not found, no exception is raised and the search continues, see next bullet point.
  - In the directories on the system library search path, e.g. the `PATH` environment variable. If the Oracle Client libraries cannot be loaded, then an exception is raised.

- On macOS:
  - In the `lib_dir` directory specified in a call to *cx_Oracle.init_oracle_client()*. This directory should contain the libraries from an unzipped Instant Client 'Basic' or 'Basic Light' package. If the Oracle Client libraries cannot be loaded from `lib_dir`, then an exception is raised.
  - If `lib_dir` was not specified, then Oracle Client libraries are looked for in the directory where the cx_Oracle binary module is. This directory should contain the libraries from an unzipped Instant Client 'Basic' or 'Basic Light' package. For example if `/Users/your_username/Library/Python/3.8/lib/python/site-packages` contains `cx_Oracle.cpython-38-darwin.so`, then you could run `ln -s ~/instantclient_19_3/libclntsh.dylib ~/Library/Python/3.8/lib/python/site-packages`. If the libraries are not found, no exception is raised and the search continues, see next bullet point.
  - In the directories on the system library search path, e.g. `~/lib/` and `/usr/local/lib`, or in `$DYLD_LIBRARY_PATH`. These paths will vary with macOS version and Python version. Any value in `DYLD_LIBRARY_PATH` will not propagate to a sub-shell. If the Oracle Client libraries cannot be loaded, then an exception is raised.
- On Linux and related platforms:
  - In the `lib_dir` directory specified in a call to *cx_Oracle.init_oracle_client()*. Note on Linux this is only useful to force immediate loading of the libraries because the libraries must also be in the system library search path. This directory should contain the libraries from an unzipped Instant Client 'Basic' or 'Basic Light' package. If you pass the library directory from a full client or database installation, such as Oracle Database "XE" Express Edition then you will need to have previously set the `ORACLE_HOME` environment variable. If the Oracle Client libraries cannot be loaded from `lib_dir`, then an exception is raised.
  - If `lib_dir` was not specified, then Oracle Client libraries are looked for in the operating system library search path, such as configured with `ldconfig` or set in the environment variable `LD_LIBRARY_PATH`. On some UNIX platforms an OS specific equivalent, such as `LIBPATH` or `SHLIB_PATH` is used instead of `LD_LIBRARY_PATH`. If the libraries are not found, no exception is raised and the search continues, see next bullet point.
  - In `$ORACLE_HOME/lib`. Note the environment variable `ORACLE_HOME` should only ever be set when you have a full database installation or full client installation. It should not be set if you are using Oracle Instant Client. The `ORACLE_HOME` variable, and other necessary variables, should be set before starting Python. See *Oracle Environment Variables*. If the Oracle Client libraries cannot be loaded, then an exception is raised.

If you call *cx_Oracle.init_oracle_client()* with a `lib_dir` parameter, the Oracle Client libraries are loaded immediately from that directory. If you call *cx_Oracle.init_oracle_client()* but do *not* set the `lib_dir` parameter, the Oracle Client libraries are loaded immediately using the search heuristic above. If you do not call *cx_Oracle.init_oracle_client()*, then the libraries are loaded using the search heuristic when the first cx_Oracle function that depends on the libraries is called, for example when a connection pool is created. If there is a problem loading the libraries, then an exception is raised.

Make sure the Python process has directory and file access permissions for the Oracle Client libraries. On Linux ensure a `libclntsh.so` file exists. On macOS ensure a `libclntsh.dylib` file exists. cx_Oracle will not directly load `libclntsh.*.XX.1` files in `lib_dir` or from the directory where the cx_Oracle binary module is. Note other libraries used by `libclntsh*` are also required.

To trace the loading of Oracle Client libraries, the environment variable `DPI_DEBUG_LEVEL` can be set to 64 before starting Python. For example, on Linux, you might use:

```
$ export DPI_DEBUG_LEVEL=64
$ python myapp.py 2> log.txt
```

Using cx_Oracle.init_oracle_client() to set the Oracle Client directory

Applications can call the function *cx_Oracle.init_oracle_client()* to specify the directory containing Oracle Instant Client libraries. The Oracle Client Libraries are loaded when `init_oracle_client()` is called. For example, if the Oracle Instant Client Libraries are in `C:\oracle\instantclient_19_6` on Windows, then you can use:

```
import cx_Oracle
import sys

try:
    cx_Oracle.init_oracle_client(lib_dir=r"C:\oracle\instantclient_19_6")
except Exception as err:
    print("Whoops!")
    print(err);
    sys.exit(1);
```

The *init_oracle_client()* function should only be called once.

If you set `lib_dir` on Linux and related platforms, you must still have configured the system library search path to include that directory before starting Python.

On any operating system, if you set `lib_dir` to the library directory of a full database or full client installation, you will need to have previously set the Oracle environment, for example by setting the `ORACLE_HOME` environment variable. Otherwise you will get errors like ORA-1804. You should set this, and other Oracle environment variables, before starting Python, as shown in *Oracle Environment Variables*.

1.3.2 Optional Oracle Net Configuration Files

Optional Oracle Net configuration files are read when cx_Oracle is loaded. These files affect connections and applications. The common files are:

- `tnsnames.ora`: A configuration file that defines databases addresses for establishing connections. See *Net Service Name for Connection Strings*.
- `sqlnet.ora`: A profile configuration file that may contain information on features such as connection failover, network encryption, logging, and tracing. See Oracle Net Services Reference for more information.

The files should be in a directory accessible to Python, not on the database server host.

For example, if the file `/etc/my-oracle-config/tnsnames.ora` should be used, you can call *cx_Oracle.init_oracle_client()*:

```
import cx_Oracle
import sys
```

(continues on next page)

(continued from previous page)

```
try:
    cx_Oracle.init_oracle_client(config_dir="/etc/my-oracle-config")
except Exception as err:
    print("Whoops!")
    print(err);
    sys.exit(1);
```

This is equivalent to setting the environment variable TNS_ADMIN to `/etc/my-oracle-config`.

If *`init_oracle_client()`* is not called, or it is called but `config_dir` is not specified, then default directories searched for the configuration files. They include:

- `$TNS_ADMIN`
- `/opt/oracle/instantclient_19_6/network/admin` if Instant Client is in `/opt/oracle/instantclient_19_6`.
- `/usr/lib/oracle/19.6/client64/lib/network/admin` if Oracle 19.6 Instant Client RPMs are used on Linux.
- `$ORACLE_HOME/network/admin` if cx_Oracle is using libraries from a database installation.

A wallet configuration file `cwallet.sso` for secure connection can be located with, or separately from, the `tnsnames.ora` and `sqlnet.ora` files. It should be securely stored. The `sqlnet.ora` file's `WALLET_LOCATION` path should be set to the directory containing `cwallet.sso`. For Oracle Autonomous Database use of wallets, see *Connecting to Autononmous Databases*.

Note the *Easy Connect Syntax for Connection Strings* can set many common configuration options without needing `tnsnames.ora` or `sqlnet.ora` files.

The section *Network Configuration* has some discussion about Oracle Net configuration.

1.3.3 Optional Oracle Client Configuration Files

When cx_Oracle uses Oracle Client libraries version 12.1, or later, an optional client parameter file called `oraaccess.xml` can be used to configure some behviors of those libraries, such as statement caching and prefetching. This can be useful if the application cannot be altered. The file is read from the same directory as the *Optional Oracle Net Configuration Files*.

A sample `oraaccess.xml` file that sets the Oracle client 'prefetch' value to 1000 rows. This value affects every SQL query in the application:

```
<?xml version="1.0"?>
 <oraaccess xmlns="http://xmlns.oracle.com/oci/oraaccess"
  xmlns:oci="http://xmlns.oracle.com/oci/oraaccess"
  schemaLocation="http://xmlns.oracle.com/oci/oraaccess
  http://xmlns.oracle.com/oci/oraaccess.xsd">
  <default_parameters>
    <prefetch>
      <rows>1000</rows>
    </prefetch>
  </default_parameters>
</oraaccess>
```

Prefetching is the number of additional rows the underlying Oracle client library fetches whenever cx_Oracle requests query data from the database. Prefetching is a tuning option to maximize data transfer efficiency and minimize *round-trips* to the database. The prefetch size does not affect when, or how many, rows are returned by cx_Oracle to the

application. The cache management is transparently handled by the Oracle client libraries. Note, standard cx_Oracle fetch tuning is via *Cursor.arraysize*, but changing the prefetch value can be useful in some cases such as when modifying the application is not feasible.

The *oraaccess.xml* file has other uses including:

- Changing the value of Fast Application Notification *FAN* events which affects notifications and Runtime Load Balancing (RLB).
- Configuring Client Result Caching parameters
- Turning on Client Statement Cache Auto-tuning

Refer to the documentation on oraaccess.xml for more details.

1.3.4 Oracle Environment Variables

Some common environment variables that influence cx_Oracle are shown below. The variables that may be needed depend on how Python is installed, how you connect to the database, and what optional settings are desired. It is recommended to set Oracle variables in the environment before invoking Python, however they may also be set in the application with `os.putenv()` before the first connection is established. System environment variables like `LD_LIBRARY_PATH` must be set before Python starts.

Table 1.1: Common Oracle environment variables

Oracle Environment Variables	Purpose
LD_LIBRARY_PATH	The library search path for platforms like Linux should include the Oracle libraries, for example `$ORACLE_HOME/lib` or `/opt/instantclient_19_3`. This variable is not needed if the libraries are located by an alternative method, such as with `ldconfig`. On other UNIX platforms you may need to set an OS specific equivalent, such as `LIBPATH` or `SHLIB_PATH`.
PATH	The library search path for Windows should include the location where `OCI.DLL` is found. Not needed if you set `lib_dir` in a call to *cx_Oracle.init_oracle_client()*
TNS_ADMIN	The directory of optional Oracle Client configuration files such as `tnsnames.ora` and `sqlnet.ora`. Not needed if the configuration files are in a default location or if `config_dir` was not used in *cx_Oracle.init_oracle_client()*. See *Optional Oracle Net Configuration Files*.
ORA_SDTZ	The default session time zone.
ORA_TZFILE	The name of the Oracle time zone file to use. See below.
ORACLE_HOME	The directory containing the Oracle Database software. The directory and various configuration files must be readable by the Python process. This variable should not be set if you are using Oracle Instant Client.
NLS_LANG	Determines the 'national language support' globalization options for cx_Oracle. Note: from cx_Oracle 8, the character set component is ignored and only the language and territory components of `NLS_LANG` are used. The character set can instead be specified during connection or connection pool creation. See *Character Sets and Globalization*.
NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT	Often set in Python applications to force a consistent date format independent of the locale. The variables are ignored if the environment variable `NLS_LANG` is not set.

Oracle Instant Client includes a small and big time zone file, for example `timezone_32.dat` and `timezlrg_32.dat`. The versions can be shown by running the utility `genezi -v` located in the Instant Client directory. The small

file contains only the most commonly used time zones. By default the larger `timezlrg_n.dat` file is used. If you want to use the smaller `timezone_n.dat` file, then set the `ORA_TZFILE` environment variable to the name of the file without any directory prefix, for example `export ORA_TZFILE=timezone_32.dat`. With Oracle Instant Client 12.2 or later, you can also use an external time zone file. Create a subdirectory `oracore/zoneinfo` under the Instant Client directory, and move the file into it. Then set `ORA_TZFILE` to the file name, without any directory prefix. The `genezi -v` utility will show the time zone file in use.

If cx_Oracle is using Oracle Client libraries from an Oracle Database or full Oracle Client software installation, and you want to use a non-default time zone file, then set `ORA_TZFILE` to the file name with a directory prefix, for example: `export ORA_TZFILE=/opt/oracle/myconfig/timezone_31.dat`.

The Oracle Database documentation contains more information about time zone files, see Choosing a Time Zone File.

1.3.5 Other cx_Oracle Initialization

The *cx_Oracle.init_oracle_client()* function allows `driver_name` and `error_url` parameters to be set. These are useful for applications whose end-users are not aware cx_Oracle is being used. An example of setting the parameters is:

```
import cx_Oracle
import sys

try:
    cx_Oracle.init_oracle_client(driver_name = "My Great App : 3.1.4",
        error_url: "https://example.com/MyInstallInstructions.html")
except Exception as err:
    print("Whoops!")
    print(err);
    sys.exit(1);
```

The convention for `driver_name` is to separate the product name from the product version by a colon and single blank characters. The value will be shown in Oracle Database views like `V$SESSION_CONNECT_INFO`. If this parameter is not specified, then the value "cx_Oracle : *version*" is used.

The `error_url` string will be shown in the exception raised if `init_oracle_client()` cannot load the Oracle Client libraries. This allows applications that use node-oracledb to refer users to application-specific installation instructions. If this value is not specified, then the *cx_Oracle 8 Installation* URL is used.

1.4 Connecting to Oracle Database

Connections between cx_Oracle and Oracle Database are used for executing *SQL*, *PL/SQL*, and *SODA*.

1.4.1 Establishing Database Connections

There are two ways to connect to Oracle Database using cx_Oracle:

- **Standalone connections**

  These are useful when the application maintains a single user session to a database. Connections are created by *cx_Oracle.connect()* or its alias *cx_Oracle.Connection()*.

- **Pooled connections**

  Connection pooling is important for performance when applications frequently connect and disconnect from the database. Oracle high availability features in the pool implementation mean that small pools can also be useful

for applications that want a few connections available for infrequent use. Pools are created with *cx_Oracle.SessionPool()* and then *SessionPool.acquire()* can be called to obtain a connection from a pool.

Many connection behaviors can be controlled by cx_Oracle options. Other settings can be configured in *Optional Oracle Net Configuration Files* or in *Optional Oracle Client Configuration Files*. These include limiting the amount of time that opening a connection can take, or enabling *network encryption*.

Example: Standalone Connection to Oracle Database

```
import cx_Oracle

userpwd = ". . ." # Obtain password string from a user prompt or environment variable

connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1", encoding=
↪"UTF-8")
```

cx_Oracle also supports *external authentication* so passwords do not need to be in the application.

1.4.2 Closing Connections

Connections should be released when they are no longer needed by calling *Connection.close()*. Alternatively, you may prefer to let connections be automatically cleaned up when references to them go out of scope. This lets cx_Oracle close dependent resources in the correct order. One other approach is the use of a "with" block, which ensures that a connection is closed once the block is completed. For example:

```
with cx_Oracle.connect(userName, password, "dbhost.example.com/orclpdb1",
            encoding="UTF-8") as connection:
    cursor = connection.cursor()
    cursor.execute("insert into SomeTable values (:1, :2)",
            (1, "Some string"))
    connection.commit()
```

This code ensures that, once the block is completed, the connection is closed and resources have been reclaimed by the database. In addition, any attempt to use the variable `connection` outside of the block will simply fail.

1.4.3 Connection Strings

The data source name parameter `dsn` of *cx_Oracle.connect()* and *cx_Oracle.SessionPool()* is the Oracle Database connection string identifying which database service to connect to. The `dsn` string can be one of:

- An Oracle Easy Connect string
- An Oracle Net Connect Descriptor string
- A Net Service Name mapping to a connect descriptor

For more information about naming methods, see Oracle Net Service Reference.

Easy Connect Syntax for Connection Strings

An Easy Connect string is often the simplest connection string to use for the data source name parameter `dsn` of *cx_Oracle.connect()* and *cx_Oracle.SessionPool()*. This method does not need configuration files such as `tnsnames.ora`.

For example, to connect to the Oracle Database service `orclpdb1` that is running on the host `dbhost.example.com` with the default Oracle Database port 1521, use:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1",
        encoding="UTF-8")
```

If the database is using a non-default port, it must be specified:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com:1984/orclpdb1",
        encoding="UTF-8")
```

The Easy Connect syntax supports Oracle Database service names. It cannot be used with the older System Identifiers (SID).

The Easy Connect syntax has been extended in recent versions of Oracle Database client since its introduction in 10g. Check the Easy Connect Naming method in Oracle Net Service Administrator's Guide for the syntax to use in your version of the Oracle Client libraries.

If you are using Oracle Client 19c, the latest Easy Connect Plus syntax allows the use of multiple hosts or ports, along with optional entries for the wallet location, the distinguished name of the database server, and even lets some network configuration options be set. This means that a *sqlnet.ora* file is not needed for some common connection scenarios.

Oracle Net Connect Descriptor Strings

The *cx_Oracle.makedsn()* function can be used to construct a connect descriptor string for the data source name parameter `dsn` of *cx_Oracle.connect()* and *cx_Oracle.SessionPool()*. The *makedsn()* function accepts the database hostname, the port number, and the service name. It also supports *sharding* syntax.

For example, to connect to the Oracle Database service `orclpdb1` that is running on the host `dbhost.example.com` with the default Oracle Database port 1521, use:

```
dsn = cx_Oracle.makedsn("dbhost.example.com", 1521, service_name="orclpdb1")
connection = cx_Oracle.connect("hr", userpwd, dsn, encoding="UTF-8")
```

Note the use of the named argument `service_name`. By default, the third parameter of *makedsn()* is a database System Identifier (SID), not a service name. However, almost all current databases use service names.

The value of `dsn` in this example is the connect descriptor string:

```
(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=dbhost.example.com)(PORT=1521))(CONNECT_
↪DATA=(SERVICE_NAME=orclpdb1)))
```

You can manually create similar connect descriptor strings. This lets you extend the syntax, for example to support failover. These strings can be embedded directly in the application:

```
dsn = """(DESCRIPTION=
              (FAILOVER=on)
              (ADDRESS_LIST=
                (ADDRESS=(PROTOCOL=tcp)(HOST=sales1-svr)(PORT=1521))
                (ADDRESS=(PROTOCOL=tcp)(HOST=sales2-svr)(PORT=1521)))
              (CONNECT_DATA=(SERVICE_NAME=sales.example.com)))"""

connection = cx_Oracle.connect("hr", userpwd, dsn, encoding="UTF-8")
```

Net Service Names for Connection Strings

Connect Descriptor Strings are commonly stored in a *tnsnames.ora* file and associated with a Net Service Name. This name can be used directly for the data source name parameter `dsn` of *cx_Oracle.connect()* and *cx_Oracle.SessionPool()*. For example, given a `tnsnames.ora` file with the following contents:

```
ORCLPDB1 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = dbhost.example.com)(PORT = 1521))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = orclpdb1)
    )
  )
```

then you could connect using the following code:

```
connection = cx_Oracle.connect("hr", userpwd, "orclpdb1", encoding="UTF-8")
```

For more information about Net Service Names, see Database Net Services Reference.

JDBC and Oracle SQL Developer Connection Strings

The cx_Oracle connection string syntax is different to Java JDBC and the common Oracle SQL Developer syntax. If these JDBC connection strings reference a service name like:

```
jdbc:oracle:thin:@hostname:port/service_name
```

for example:

```
jdbc:oracle:thin:@dbhost.example.com:1521/orclpdb1
```

then use Oracle's Easy Connect syntax in cx_Oracle:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com:1521/orclpdb1",␣
↪encoding="UTF-8")
```

Alternatively, if a JDBC connection string uses an old-style Oracle SID "system identifier", and the database does not have a service name:

```
jdbc:oracle:thin:@hostname:port:sid
```

for example:

```
jdbc:oracle:thin:@dbhost.example.com:1521:orcl
```

then a connect descriptor string from `makedsn()` can be used in the application:

```
dsn = cx_Oracle.makedsn("dbhost.example.com", 1521, sid="orcl")
connection = cx_Oracle.connect("hr", userpwd, dsn, encoding="UTF-8")
```

Alternatively, create a `tnsnames.ora` (see *Optional Oracle Net Configuration Files*) entry, for example:

```
finance =
 (DESCRIPTION =
   (ADDRESS = (PROTOCOL = TCP)(HOST = dbhost.example.com)(PORT = 1521))
   (CONNECT_DATA =
     (SID = ORCL)
   )
 )
```

This can be referenced in cx_Oracle:

```
connection = cx_Oracle.connect("hr", userpwd, "finance", encoding="UTF-8")
```

1.4.4 Connection Pooling

cx_Oracle's connection pooling lets applications create and maintain a pool of connections to the database. The internal implementation uses Oracle's session pool technology. In general, each connection in a cx_Oracle connection pool corresponds to one Oracle session.

A connection pool is created by calling *SessionPool()*. This is generally called during application initialization. Connections can then be obtained from a pool by calling *acquire()*. The initial pool size and the maximum pool size are provided at the time of pool creation. When the pool needs to grow, new connections are created automatically. The pool can shrink back to the minimum size when connections are no longer in use.

Connections acquired from the pool should be released back to the pool using *SessionPool.release()* or *Connection.close()* when they are no longer required. Otherwise, they will be released back to the pool automatically when all of the variables referencing the connection go out of scope. The session pool can be completely closed using *SessionPool.close()*.

The example below shows how to connect to Oracle Database using a connection pool:

```
# Create the session pool
pool = cx_Oracle.SessionPool("hr", userpwd,
        "dbhost.example.com/orclpdb1", min=2, max=5, increment=1, encoding="UTF-8")

# Acquire a connection from the pool
connection = pool.acquire()

# Use the pooled connection
cursor = connection.cursor()
for result in cursor.execute("select * from mytab"):
    print(result)

# Release the connection to the pool
pool.release(connection)

# Close the pool
pool.close()
```

Applications that are using connections concurrently in multiple threads should set the `threaded` parameter to *True* when creating a connection pool:

```
# Create the session pool
pool = cx_Oracle.SessionPool("hr", userpwd, "dbhost.example.com/orclpdb1",
               min=2, max=5, increment=1, threaded=True, encoding="UTF-8")
```

See ConnectionPool.py for an example.

Before *SessionPool.acquire()* returns, cx_Oracle does a lightweight check to see if the network transport for the selected connection is still open. If it is not, then *acquire()* will clean up the connection and return a different one. This check will not detect cases such as where the database session has been killed by the DBA, or reached a database resource manager quota limit. To help in those cases, *acquire()* will also do a full *round-trip* ping to the database when it is about to return a connection that was unused in the pool for 60 seconds. If the ping fails, the connection will be discarded and another one obtained before *acquire()* returns to the application. Because this full ping is time based, it won't catch every failure. Also since network timeouts and session kills may occur after *acquire()* and before *Cursor.execute()*, applications need to check for errors after each *execute()* and make application-specific decisions about retrying work if there was a connection failure. Note both the lightweight

and full ping connection checks can mask configuration issues, for example firewalls killing connections, so monitor the connection rate in AWR for an unexpected value. You can explicitly initiate a full ping to check connection liveness with *Connection.ping()* but overuse will impact performance and scalability.

The Oracle Real-World Performance Group's recommendation is to use fixed size connection pools. The values of min and max should be the same (and the increment equal to zero). The *firewall*, resource manager or user profile IDLE_TIME should not expire idle sessions. This avoids connection storms which can decrease throughput. See Guideline for Preventing Connection Storms: Use Static Pools, which contains details about sizing of pools.

The Real-World Performance Group also recommends keeping pool sizes small, as they may perform better than larger pools. The pool attributes should be adjusted to handle the desired workload within the bounds of available resources in cx_Oracle and the database.

Session CallBacks for Setting Pooled Connection State

Applications can set "session" state in each connection. Examples of session state are NLS settings from `ALTER SESSION` statements. Pooled connections will retain their session state after they have been released back to the pool. However, because pools can grow, or connections in the pool can be recreated, there is no guarantee a subsequent *acquire()* call will return a database connection that has any particular state.

The *SessionPool()* parameter `sessionCallback` enables efficient setting of session state so that connections have a known session state, without requiring that state to be explicitly set after each *acquire()* call.

Connections can also be tagged when they are released back to the pool. The tag is a user-defined string that represents the session state of the connection. When acquiring connections, a particular tag can be requested. If a connection with that tag is available, it will be returned. If not, then another session will be returned. By comparing the actual and requested tags, applications can determine what exact state a session has, and make any necessary changes.

The session callback can be a Python function or a PL/SQL procedure.

There are three common scenarios for `sessionCallback`:

- When all connections in the pool should have the same state, use a Python callback without tagging.
- When connections in the pool require different state for different users, use a Python callback with tagging.
- When using *Database Resident Connection Pooling (DRCP)*: use a PL/SQL callback with tagging.

Python Callback

If the `sessionCallback` parameter is a Python procedure, it will be called whenever *acquire()* will return a newly created database connection that has not been used before. It is also called when connection tagging is being used and the requested tag is not identical to the tag in the connection returned by the pool.

An example is:

```
# Set the NLS_DATE_FORMAT for a session
def initSession(connection, requestedTag):
    cursor = connection.cursor()
    cursor.execute("ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI'")

# Create the pool with session callback defined
pool = cx_Oracle.SessionPool("hr", userpwd, "orclpdb1",
                     sessionCallback=initSession, encoding="UTF-8")

# Acquire a connection from the pool (will always have the new date format)
connection = pool.acquire()
```

If needed, the `initSession()` procedure is called internally before `acquire()` returns. It will not be called when previously used connections are returned from the pool. This means that the ALTER SESSION does not need to be executed after every `acquire()` call. This improves performance and scalability.

In this example tagging was not being used, so the `requestedTag` parameter is ignored.

Note: if you need to execute multiple SQL statements in the callback, use an anonymous PL/SQL block to save *round-trips* of repeated `execute()` calls. With ALTER SESSION, pass multiple settings in the one statement:

```
cursor.execute("""
        begin
            execute immediate
                    'alter session set nls_date_format = ''YYYY-MM-DD'' nls_language␣
↪= AMERICAN';
            -- other SQL statements could be put here
        end;""")
```

Connection Tagging

Connection tagging is used when connections in a pool should have differing session states. In order to retrieve a connection with a desired state, the `tag` attribute in *`acquire()`* needs to be set.

When cx_Oracle is using Oracle Client libraries 12.2 or later, then cx_Oracle uses 'multi-property tags' and the tag string must be of the form of one or more "name=value" pairs separated by a semi-colon, for example `"loc=uk; lang=cy"`.

When a connection is requested with a given tag, and a connection with that tag is not present in the pool, then a new connection, or an existing connection with cleaned session state, will be chosen by the pool and the session callback procedure will be invoked. The callback can then set desired session state and update the connection's tag. However if the `matchanytag` parameter of *`acquire()`* is *True*, then any other tagged connection may be chosen by the pool and the callback procedure should parse the actual and requested tags to determine which bits of session state should be reset.

The example below demonstrates connection tagging:

```
def initSession(connection, requestedTag):
    if requestedTag == "NLS_DATE_FORMAT=SIMPLE":
        sql = "ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD'"
    elif requestedTag == "NLS_DATE_FORMAT=FULL":
        sql = "ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI'"
    cursor = connection.cursor()
    cursor.execute(sql)
    connection.tag = requestedTag

pool = cx_Oracle.SessionPool("hr", userpwd, "orclpdb1",
                     sessionCallback=initSession, encoding="UTF-8")

# Two connections with different session state:
connection1 = pool.acquire(tag = "NLS_DATE_FORMAT=SIMPLE")
connection2 = pool.acquire(tag = "NLS_DATE_FORMAT=FULL")
```

See SessionCallback.py for an example.

PL/SQL Callback

When cx_Oracle uses Oracle Client 12.2 or later, the session callback can also be the name of a PL/SQL procedure. A PL/SQL callback will be initiated only when the tag currently associated with a connection does not match the tag that is requested. A PL/SQL callback is most useful when using *Database Resident Connection Pooling (DRCP)* because DRCP does not require a *round-trip* to invoke a PL/SQL session callback procedure.

The PL/SQL session callback should accept two VARCHAR2 arguments:

```
PROCEDURE myPlsqlCallback (
    requestedTag IN  VARCHAR2,
    actualTag    IN  VARCHAR2
);
```

The logic in this procedure can parse the actual tag in the session that has been selected by the pool and compare it with the tag requested by the application. The procedure can then change any state required before the connection is returned to the application from *acquire()*.

If the `matchanytag` attribute of *acquire()* is *True*, then a connection with any state may be chosen by the pool.

Oracle 'multi-property tags' must be used. The tag string must be of the form of one or more "name=value" pairs separated by a semi-colon, for example `"loc=uk;lang=cy"`.

In cx_Oracle set `sessionCallback` to the name of the PL/SQL procedure. For example:

```
pool = cx_Oracle.SessionPool("hr", userpwd, "dbhost.example.com/orclpdb1:pooled",
                             sessionCallback="myPlsqlCallback", encoding="UTF-8")

connection = pool.acquire(tag="NLS_DATE_FORMAT=SIMPLE",
        # DRCP options, if you are using DRCP
        cclass='MYCLASS', purity=cx_Oracle.ATTR_PURITY_SELF)
```

See SessionCallbackPLSQL.py for an example.

Heterogeneous and Homogeneous Connection Pools

By default, connection pools are 'homogeneous', meaning that all connections use the same database credentials. However, if the pool option `homogeneous` is False at the time of pool creation, then a 'heterogeneous' pool will be created. This allows different credentials to be used each time a connection is acquired from the pool with *acquire()*.

Heterogeneous Pools

When a heterogeneous pool is created by setting `homogeneous` to False and no credentials are supplied during pool creation, then a user name and password may be passed to *acquire()* as shown in this example:

```
pool = cx_Oracle.SessionPool(dsn="dbhost.example.com/orclpdb1", homogeneous=False,
                             encoding="UTF-8")
connection = pool.acquire(user="hr", password=userpwd)
```

1.4.5 Database Resident Connection Pooling (DRCP)

Database Resident Connection Pooling (DRCP) enables database resource sharing for applications that run in multiple client processes, or run on multiple middle-tier application servers. By default each connection from Python will use one database server process. DRCP allows pooling of these server processes. This reduces the amount of memory required on the database host. The DRCP pool can be shared by multiple applications.

DRCP is useful for applications which share the same database credentials, have similar session settings (for example date format settings or PL/SQL package state), and where the application gets a database connection, works on it for a relatively short duration, and then releases it.

Applications can choose whether or not to use pooled connections at runtime.

For efficiency, it is recommended that DRCP connections should be used in conjunction with cx_Oracle's local *connection pool*.

Using DRCP in Python

Using DRCP with cx_Oracle applications involves the following steps:

1. Configuring and enabling DRCP in the database
2. Configuring the application to use a DRCP connection
3. Deploying the application

Configuring and enabling DRCP

Every instance of Oracle Database uses a single, default connection pool. The pool can be configured and administered by a DBA using the `DBMS_CONNECTION_POOL` package:

```
EXECUTE DBMS_CONNECTION_POOL.CONFIGURE_POOL(
    pool_name => 'SYS_DEFAULT_CONNECTION_POOL',
    minsize => 4,
    maxsize => 40,
    incrsize => 2,
    session_cached_cursors => 20,
    inactivity_timeout => 300,
    max_think_time => 600,
    max_use_session => 500000,
    max_lifetime_session => 86400)
```

Alternatively the method `DBMS_CONNECTION_POOL.ALTER_PARAM()` can set a single parameter:

```
EXECUTE DBMS_CONNECTION_POOL.ALTER_PARAM(
    pool_name => 'SYS_DEFAULT_CONNECTION_POOL',
    param_name => 'MAX_THINK_TIME',
    param_value => '1200')
```

The `inactivity_timeout` setting terminates idle pooled servers, helping optimize database resources. To avoid pooled servers permanently being held onto by a selfish Python script, the `max_think_time` parameter can be set. The parameters `num_cbrok` and `maxconn_cbrok` can be used to distribute the persistent connections from the clients across multiple brokers. This may be needed in cases where the operating system per-process descriptor limit is small. Some customers have found that having several connection brokers improves performance. The `max_use_session` and `max_lifetime_session` parameters help protect against any unforeseen problems affecting server processes. The default values will be suitable for most users. See the Oracle DRCP documentation for details on parameters.

In general, if pool parameters are changed, the pool should be restarted, otherwise server processes will continue to use old settings.

There is a `DBMS_CONNECTION_POOL.RESTORE_DEFAULTS()` procedure to reset all values.

When DRCP is used with RAC, each database instance has its own connection broker and pool of servers. Each pool has the identical configuration. For example, all pools start with `minsize` server processes. A single DBMS_CONNECTION_POOL command will alter the pool of each instance at the same time. The pool needs to be started before connection requests begin. The command below does this by bringing up the broker, which registers itself with the database listener:

```
EXECUTE DBMS_CONNECTION_POOL.START_POOL()
```

Once enabled this way, the pool automatically restarts when the database instance restarts, unless explicitly stopped with the `DBMS_CONNECTION_POOL.STOP_POOL()` command:

```
EXECUTE DBMS_CONNECTION_POOL.STOP_POOL()
```

The pool cannot be stopped while connections are open.

Application Deployment for DRCP

In order to use DRCP, the `cclass` and `purity` parameters should be passed to *cx_Oracle.connect()* or *SessionPool.acquire()*. If `cclass` is not set, the pooled server sessions will not be reused optimally, and the DRCP statistic views will record large values for NUM_MISSES.

The DRCP `purity` can be one of `ATTR_PURITY_NEW`, `ATTR_PURITY_SELF`, or `ATTR_PURITY_DEFAULT`. The value `ATTR_PURITY_SELF` allows reuse of both the pooled server process and session memory, giving maximum benefit from DRCP. See the Oracle documentation on benefiting from scalability.

The connection string used for *connect()* or *acquire()* must request a pooled server by following one of the syntaxes shown below:

Using Oracle's Easy Connect syntax, the connection would look like:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orcl:pooled",
        encoding="UTF-8")
```

Or if you connect using a Net Service Name named `customerpool`:

```
connection = cx_Oracle.connect("hr", userpwd, "customerpool", encoding="UTF-8")
```

Then only the Oracle Network configuration file `tnsnames.ora` needs to be modified:

```
customerpool = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
          (HOST=dbhost.example.com)
          (PORT=1521))(CONNECT_DATA=(SERVICE_NAME=CUSTOMER)
          (SERVER=POOLED)))
```

If these changes are made and the database is not actually configured for DRCP, or the pool is not started, then connections will not succeed and an error will be returned to the Python application.

Although applications can choose whether or not to use pooled connections at runtime, care must be taken to configure the database appropriately for the number of expected connections, and also to stop inadvertent use of non-DRCP connections leading to a database server resource shortage. Conversely, avoid using DRCP connections for long-running operations.

The example below shows how to connect to Oracle Database using Database Resident Connection Pooling:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orcl:pooled",
        cclass="MYCLASS", purity=cx_Oracle.ATTR_PURITY_SELF, encoding="UTF-8")
```

The example below shows connecting to Oracle Database using DRCP and cx_Oracle's connection pooling:

```
mypool = cx_Oracle.SessionPool("hr", userpwd, "dbhost.example.com/orcl:pooled",
                       encoding="UTF-8")
connection = mypool.acquire(cclass="MYCLASS", purity=cx_Oracle.ATTR_PURITY_SELF)
```

For more information about DRCP see Oracle Database Concepts Guide, and for DRCP Configuration see Oracle Database Administrator's Guide.

Closing Connections

Python scripts where cx_Oracle connections do not go out of scope quickly (which releases them), or do not currently use *Connection.close()*, should be examined to see if *close()* can be used, which then allows maximum use of DRCP pooled servers by the database:

```
 # Do some database operations
connection = mypool.acquire(cclass="MYCLASS", purity=cx_Oracle.ATTR_PURITY_SELF)
. . .
connection.close();

# Do lots of non-database work
. . .

# Do some more database operations
connection = mypool.acquire(cclass="MYCLASS", purity=cx_Oracle.ATTR_PURITY_SELF)
. . .
connection.close();
```

Monitoring DRCP

Data dictionary views are available to monitor the performance of DRCP. Database administrators can check statistics such as the number of busy and free servers, and the number of hits and misses in the pool against the total number of requests from clients. The views are:

- DBA_CPOOL_INFO
- V$PROCESS
- V$SESSION
- V$CPOOL_STATS
- V$CPOOL_CC_STATS
- V$CPOOL_CONN_INFO

DBA_CPOOL_INFO View

DBA_CPOOL_INFO displays configuration information about the DRCP pool. The columns are equivalent to the dbms_connection_pool.configure_pool() settings described in the table of DRCP configuration options, with the addition of a STATUS column. The status is ACTIVE if the pool has been started and INACTIVE otherwise. Note the pool name column is called CONNECTION_POOL. This example checks whether the pool has been started and finds the maximum number of pooled servers:

```
SQL> SELECT connection_pool, status, maxsize FROM dba_cpool_info;

CONNECTION_POOL              STATUS        MAXSIZE
---------------------------- ---------- ----------
SYS_DEFAULT_CONNECTION_POOL  ACTIVE             40
```

V$PROCESS and V$SESSION Views

The V$SESSION view shows information about the currently active DRCP sessions. It can also be joined with V$PROCESS via V$SESSION.PADDR = V$PROCESS.ADDR to correlate the views.

V$CPOOL_STATS View

The V$CPOOL_STATS view displays information about the DRCP statistics for an instance. The V$CPOOL_STATS view can be used to assess how efficient the pool settings are. T his example query shows an application using the pool effectively. The low number of misses indicates that servers and sessions were reused. The wait count shows just over 1% of requests had to wait for a pooled server to become available:

```
NUM_REQUESTS   NUM_HITS NUM_MISSES  NUM_WAITS
------------ ---------- ---------- ----------
       10031      99990         40       1055
```

If `cclass` was set (allowing pooled servers and sessions to be reused) then NUM_MISSES will be low. If the pool maxsize is too small for the connection load, then NUM_WAITS will be high.

V$CPOOL_CC_STATS View

The view `V$CPOOL_CC_STATS` displays information about the connection class level statistics for the pool per instance:

```
SQL> SELECT cclass_name, num_requests, num_hits, num_misses
     FROM v$cpool_cc_stats;

CCLASS_NAME                    NUM_REQUESTS NUM_HITS   NUM_MISSES
------------------------------ ------------ ---------- ----------
HR.MYCLASS                           100031      99993         38
```

V$CPOOL_CONN_INFO View

The `V$POOL_CONN_INFO` view gives insight into client processes that are connected to the connection broker, making it easier to monitor and trace applications that are currently using pooled servers or are idle. This view was introduced in Oracle 11gR2.

You can monitor the view `V$CPOOL_CONN_INFO` to, for example, identify misconfigured machines that do not have the connection class set correctly. This view maps the machine name to the class name:

```
SQL> SELECT cclass_name, machine FROM v$cpool_conn_info;

CCLASS_NAME                            MACHINE
-------------------------------------- ------------
CJ.OCI:SP:wshbIFDtb7rgQwMyuYvodA       cjlinux
. . .
```

In this example you would examine applications on `cjlinux` and make sure `cclass` is set.

1.4.6 Connecting Using Proxy Authentication

Proxy authentication allows a user (the "session user") to connect to Oracle Database using the credentials of a 'proxy user'. Statements will run as the session user. Proxy authentication is generally used in three-tier applications where one user owns the schema while multiple end-users access the data. For more information about proxy authentication, see the Oracle documentation.

An alternative to using proxy users is to set *`Connection.client_identifier`* after connecting and use its value in statements and in the database, for example for *monitoring*.

The following proxy examples use these schemas. The `mysessionuser` schema is granted access to use the password of `myproxyuser`:

```
CREATE USER myproxyuser IDENTIFIED BY myproxyuserpw;
GRANT CREATE SESSION TO myproxyuser;

CREATE USER mysessionuser IDENTIFIED BY itdoesntmatter;
GRANT CREATE SESSION TO mysessionuser;

ALTER USER mysessionuser GRANT CONNECT THROUGH myproxyuser;
```

After connecting to the database, the following query can be used to show the session and proxy users:

```
SELECT SYS_CONTEXT('USERENV', 'PROXY_USER'),
       SYS_CONTEXT('USERENV', 'SESSION_USER')
FROM DUAL;
```

Standalone connection examples:

```
# Basic Authentication without a proxy
connection = cx_Oracle.connect("myproxyuser", "myproxyuserpw", "dbhost.example.com/
↪orclpdb1",
        encoding="UTF-8")
# PROXY_USER:   None
# SESSION_USER: MYPROXYUSER

# Basic Authentication with a proxy
connection = cx_Oracle.connect(user="myproxyuser[mysessionuser]", "myproxyuserpw",
       "dbhost.example.com/orclpdb1", encoding="UTF-8")
# PROXY_USER:   MYPROXYUSER
# SESSION_USER: MYSESSIONUSER
```

Pooled connection examples:

```
# Basic Authentication without a proxy
pool = cx_Oracle.SessionPool("myproxyuser", "myproxyuser", "dbhost.example.com/
↪orclpdb1",
                     encoding="UTF-8")
connection = pool.acquire()
# PROXY_USER:   None
# SESSION_USER: MYPROXYUSER

# Basic Authentication with proxy
pool = cx_Oracle.SessionPool("myproxyuser[mysessionuser]", "myproxyuser",
                     "dbhost.example.com/orclpdb1", homogeneous=False, encoding="UTF-8
↪")
connection = pool.acquire()
# PROXY_USER:   MYPROXYUSER
# SESSION_USER: MYSESSIONUSER
```

Note the use of a *heterogeneous* pool in the example above. This is required in this scenario.

1.4.7 Connecting Using External Authentication

Instead of storing the database username and password in Python scripts or environment variables, database access can be authenticated by an outside system. External Authentication allows applications to validate user access by an external password store (such as an Oracle Wallet), by the operating system, or with an external authentication service.

Using an Oracle Wallet for External Authentication

The following steps give an overview of using an Oracle Wallet. Wallets should be kept securely. Wallets can be managed with Oracle Wallet Manager.

In this example the wallet is created for the `myuser` schema in the directory `/home/oracle/wallet_dir`. The `mkstore` command is available from a full Oracle client or Oracle Database installation. If you have been given wallet by your DBA, skip to step 3.

1. First create a new wallet as the `oracle` user:

```
mkstore -wrl "/home/oracle/wallet_dir" -create
```

This will prompt for a new password for the wallet.

2. Create the entry for the database user name and password that are currently hardcoded in your Python scripts. Use either of the methods shown below. They will prompt for the wallet password that was set in the first step.

   Method 1 - Using an Easy Connect string:

```
mkstore -wrl "/home/oracle/wallet_dir" -createCredential dbhost.example.com/
↪orclpdb1 myuser myuserpw
```

   Method 2 - Using a connect name identifier:

```
mkstore -wrl "/home/oracle/wallet_dir" -createCredential mynetalias myuser␣
↪myuserpw
```

   The alias key `mynetalias` immediately following the `-createCredential` option will be the connect name to be used in Python scripts. If your application connects with multiple different database users, you could create a wallet entry with different connect names for each.

   You can see the newly created credential with:

```
mkstore -wrl "/home/oracle/wallet_dir" -listCredential
```

3. Skip this step if the wallet was created using an Easy Connect String. Otherwise, add an entry in *tnsnames.ora* for the connect name as follows:

```
mynetalias =
    (DESCRIPTION =
        (ADDRESS = (PROTOCOL = TCP)(HOST = dbhost.example.com)(PORT = 1521))
        (CONNECT_DATA =
            (SERVER = DEDICATED)
            (SERVICE_NAME = orclpdb1)
        )
    )
```

   The file uses the description for your existing database and sets the connect name alias to `mynetalias`, which is the identifier used when adding the wallet entry.

4. Add the following wallet location entry in the *sqlnet.ora* file, using the `DIRECTORY` you created the wallet in:

```
WALLET_LOCATION =
    (SOURCE =
        (METHOD = FILE)
        (METHOD_DATA =
            (DIRECTORY = /home/oracle/wallet_dir)
        )
    )
SQLNET.WALLET_OVERRIDE = TRUE
```

   Examine the Oracle documentation for full settings and values.

5. Ensure the configuration files are in a default location or set TNS_ADMIN is set to the directory containing them. See *Optional Oracle Net Configuration Files*.

With an Oracle wallet configured, and readable by you, your scripts can connect using:

```
connection = cx_Oracle.connect(dsn="mynetalias", encoding="UTF-8")
```

or:

```
pool = cx_Oracle.SessionPool(externalauth=True, homogeneous=False, dsn="mynetalias",
                     encoding="UTF-8")
pool.acquire()
```

The `dsn` must match the one used in the wallet.

After connecting, the query:

```
SELECT SYS_CONTEXT('USERENV', 'SESSION_USER') FROM DUAL;
```

will show:

```
MYUSER
```

Note: Wallets are also used to configure TLS connections. If you are using a wallet like this, you may need a database username and password in *cx_Oracle.connect()* and *cx_Oracle.SessionPool()* calls.

External Authentication and Proxy Authentication

The following examples show external wallet authentication combined with *proxy authentication*. These examples use the wallet configuration from above, with the addition of a grant to another user:

```
ALTER USER mysessionuser GRANT CONNECT THROUGH myuser;
```

After connection, you can check who the session user is with:

```
SELECT SYS_CONTEXT('USERENV', 'PROXY_USER'),
       SYS_CONTEXT('USERENV', 'SESSION_USER')
FROM DUAL;
```

Standalone connection example:

```
# External Authentication with proxy
connection = cx_Oracle.connect(user="[mysessionuser]", dsn="mynetalias", encoding=
↪"UTF-8")
# PROXY_USER:   MYUSER
# SESSION_USER: MYSESSIONUSER
```

Pooled connection example:

```
# External Authentication with proxy
pool = cx_Oracle.SessionPool(externalauth=True, homogeneous=False, dsn="mynetalias",
                     encoding="UTF-8")
pool.acquire(user="[mysessionuser]")
# PROXY_USER:   MYUSER
# SESSION_USER: MYSESSIONUSER
```

The following usage is not supported:

```
pool = cx_Oracle.SessionPool("[mysessionuser]", externalauth=True, homogeneous=False,
                     dsn="mynetalias", encoding="UTF-8")
pool.acquire()
```

Operating System Authentication

With Operating System authentication, Oracle allows user authentication to be performed by the operating system. The following steps give an overview of how to implement OS Authentication on Linux.

1. Login to your computer. The commands used in these steps assume the operating system user name is "oracle".
2. Login to SQL*Plus as the SYSTEM user and verify the value for the `OS_AUTHENT_PREFIX` parameter:

```
SQL> SHOW PARAMETER os_authent_prefix

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
os_authent_prefix                    string      ops$
```

3. Create an Oracle database user using the `os_authent_prefix` determined in step 2, and the operating system user name:

```
CREATE USER ops$oracle IDENTIFIED EXTERNALLY;
GRANT CONNECT, RESOURCE TO ops$oracle;
```

In Python, connect using the following code:

```
connection = cx_Oracle.connect(dsn="mynetalias", encoding="UTF-8")
```

Your session user will be `OPS$ORACLE`.

If your database is not on the same computer as python, you can perform testing by setting the database configuration parameter `remote_os_authent=true`. Beware this is insecure.

See Oracle Database Security Guide for more information about Operating System Authentication.

1.4.8 Privileged Connections

The `mode` parameter of the function *`cx_Oracle.connect()`* specifies the database privilege that you want to associate with the user.

The example below shows how to connect to Oracle Database as SYSDBA:

```
connection = cx_Oracle.connect("sys", syspwd, "dbhost.example.com/orclpdb1",
        mode=cx_Oracle.SYSDBA, encoding="UTF-8")

cursor = con.cursor()
sql = "GRANT SYSOPER TO hr"
cursor.execute(sql)
```

This is equivalent to executing the following in SQL*Plus:

```
CONNECT sys/syspwd AS SYSDBA

GRANT SYSOPER TO hr;
```

1.4.9 Securely Encrypting Network Traffic to Oracle Database

You can encrypt data transferred between the Oracle Database and the Oracle Client libraries used by cx_Oracle so that unauthorized parties are not able to view plain text values as the data passes over the network. The easiest

configuration is Oracle's native network encryption. The standard SSL protocol can also be used if you have a PKI, but setup is necessarily more involved.

With native network encryption, the client and database server negotiate a key using Diffie-Hellman key exchange. This provides protection against man-in-the-middle attacks.

Native network encryption can be configured by editing Oracle Net's optional *sqlnet.ora* configuration file, on either the database server and/or on each cx_Oracle 'client' machine. Parameters control whether data integrity checking and encryption is required or just allowed, and which algorithms the client and server should consider for use.

As an example, to ensure all connections to the database are checked for integrity and are also encrypted, create or edit the Oracle Database `$ORACLE_HOME/network/admin/sqlnet.ora` file. Set the checksum negotiation to always validate a checksum and set the checksum type to your desired value. The network encryption settings can similarly be set. For example, to use the SHA512 checksum and AES256 encryption use:

```
SQLNET.CRYPTO_CHECKSUM_SERVER = required
SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER = (SHA512)
SQLNET.ENCRYPTION_SERVER = required
SQLNET.ENCRYPTION_TYPES_SERVER = (AES256)
```

If you definitely know that the database server enforces integrity and encryption, then you do not need to configure cx_Oracle separately. However you can also, or alternatively, do so depending on your business needs. Create a `sqlnet.ora` on your client machine and locate it with other *Optional Oracle Net Configuration Files*:

```
SQLNET.CRYPTO_CHECKSUM_CLIENT = required
SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT = (SHA512)
SQLNET.ENCRYPTION_CLIENT = required
SQLNET.ENCRYPTION_TYPES_CLIENT = (AES256)
```

The client and server sides can negotiate the protocols used if the settings indicate more than one value is accepted.

Note that these are example settings only. You must review your security requirements and read the documentation for your Oracle version. In particular review the available algorithms for security and performance.

The `NETWORK_SERVICE_BANNER` column of the database view V$SESSION_CONNECT_INFO can be used to verify the encryption status of a connection.

For more information on Oracle Data Network Encryption and Integrity, configuring SSL network encryption and Transparent Data Encryption of data-at-rest in the database, see Oracle Database Security Guide.

1.4.10 Resetting Passwords

After connecting, passwords can be changed by calling *Connection.changepassword()*:

```
# Get the passwords from somewhere, such as prompting the user
oldpwd = getpass.getpass("Old Password for %s: " % username)
newpwd = getpass.getpass("New Password for %s: " % username)

connection.changepassword(oldpwd, newpwd)
```

When a password has expired and you cannot connect directly, you can connect and change the password in one operation by using the `newpassword` parameter of the function *cx_Oracle.connect()* constructor:

```
# Get the passwords from somewhere, such as prompting the user
oldpwd = getpass.getpass("Old Password for %s: " % username)
newpwd = getpass.getpass("New Password for %s: " % username)

```

(continues on next page)

(continued from previous page)

```
connection = cx_Oracle.connect(username, oldpwd, "dbhost.example.com/orclpdb1",
        newpassword=newpwd, encoding="UTF-8")
```

1.4.11 Connecting to Autononmous Databases

To enable connection to Oracle Autonomous Database in Oracle Cloud, a wallet needs be downloaded from the cloud GUI, and cx_Oracle needs to be configured to use it. A database username and password is still required. The wallet only enables SSL/TLS.

Install the Wallet and Network Configuration Files

From the Oracle Cloud console for the database, download the wallet zip file. It contains the wallet and network configuration files. Note: keep wallet files in a secure location and share them only with authorized users.

Unzip the wallet zip file.

For cx_Oracle, only these files from the zip are needed:

- `tnsnames.ora` - Maps net service names used for application connection strings to your database services
- `sqlnet.ora` - Configures Oracle Network settings
- `cwallet.sso` - Enables SSL/TLS connections

The other files and the wallet password are not needed.

Place these files as shown in *Optional Oracle Net Configuration Files*.

Run Your Application

The `tnsnames.ora` file contains net service names for various levels of database service. For example, if you create a database called CJDB1 with the Always Free services from the Oracle Cloud Free Tier, then you might decide to use the connection string in `tnsnames.ora` called `cjdb1_high`.

Update your application to use your schema username, its database password, and a net service name, for example:

```
connection = cx_Oracle.connect("scott", userpwd, "cjdb1_high", encoding="UTF-8")
```

Once you have set Oracle environment variables required by your application, such as `TNS_ADMIN`, you can start your application.

If you need to create a new database schema so you do not login as the privileged ADMIN user, refer to the relevant Oracle Cloud documentation, for example see Create Database Users in the Oracle Autonomous Transaction Processing Dedicated Deployments manual.

Access Through a Proxy

If you are behind a firewall, you can tunnel TLS/SSL connections via a proxy using HTTPS_PROXY in the connect descriptor. Successful connection depends on specific proxy configurations. Oracle does not recommend doing this when performance is critical.

Edit `sqlnet.ora` and add a line:

SQLNET.USE_HTTPS_PROXY=on

Edit `tnsnames.ora` and add an `HTTPS_PROXY` proxy name and `HTTPS_PROXY_PORT` port to the connect descriptor address list of any service name you plan to use, for example:

cjdb1_high = (description= (address= (https_proxy=myproxy.example.com)(https_proxy_port=80) (protocol=tcps)(port=1522)(host= . . .)

1.4.12 Connecting to Sharded Databases

Oracle Sharding can be used to horizontally partition data across independent databases. A database table can be split so each shard contains a table with the same columns but a different subset of rows. These tables are known as sharded tables. Sharding is configured in Oracle Database, see the Oracle Sharding manual. Sharding requires Oracle Database and Oracle Client libraries 12.2, or later.

The *cx_Oracle.connect()* and *SessionPool.acquire()* functions accept `shardingkey` and `supershardingkey` parameters that are a sequence of values used to route the connection directly to a given shard. A sharding key is always required. A super sharding key is additionally required when using composite sharding, which is when data has been partitioned by a list or range (the super sharding key), and then further partitioned by a sharding key.

When creating a connection pool, the *cx_Oracle.SessionPool()* attribute `maxSessionsPerShard` can be set. This is used to balance connections in the pool equally across shards. It requires Oracle Client libraries 18.3, or later.

Shard key values may be of type string (mapping to VARCHAR2 shard keys), number (NUMBER), bytes (RAW), or date (DATE). Multiple types may be used in each array. Sharding keys of TIMESTAMP type are not supported.

When connected to a shard, queries will only return data from that shard. For queries that need to access data from multiple shards, connections can be established to the coordinator shard catalog database. In this case, no shard key or super shard key is used.

As an example of direct connection, if sharding had been configured on a single VARCHAR2 column like:

```
CREATE SHARDED TABLE customers (
  cust_id NUMBER,
  cust_name VARCHAR2(30),
  class VARCHAR2(10) NOT NULL,
  signup_date DATE,
  cust_code RAW(20),
  CONSTRAINT cust_name_pk PRIMARY KEY(cust_name))
  PARTITION BY CONSISTENT HASH (cust_name)
  PARTITIONS AUTO TABLESPACE SET ts1;
```

then direct connection to a shard can be made by passing a single sharding key:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1",
        encoding="UTF-8", shardingkey=["SCOTT"])
```

Numbers keys can be used in a similar way:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1",
        encoding="UTF-8", shardingkey=[110])
```

When sharding by DATE, you can connect like:

```
import datetime

d = datetime.datetime(2014, 7, 3)
```

(continues on next page)

(continued from previous page)

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1",
        encoding="UTF-8", shardingkey=[d])
```

When sharding by RAW, you can connect like:

```
b = b'\x01\x04\x08';

connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1",
        encoding="UTF-8", shardingkey=[b])
```

Multiple keys can be specified, for example:

```
keyArray = [70, "SCOTT", "gold", b'\x00\x01\x02']

connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1",
        encoding="UTF-8", shardingkey=keyArray)
```

A super sharding key example is:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1",
        encoding="UTF-8", supershardingkey=["goldclass"], shardingkey=["SCOTT"])
```

1.5 SQL Execution

Executing SQL statements is the primary way in which a Python application communicates with Oracle Database. Statements are executed using the methods *Cursor.execute()* or *Cursor.executemany()*. Statements include queries, Data Manipulation Language (DML), and Data Definition Language (DDL). A few other specialty statements can also be executed.

PL/SQL statements are discussed in *PL/SQL Execution*. Other chapters contain information on specific data types and features. See *Batch Statement Execution and Bulk Loading*, *Using CLOB and BLOB Data*, *Working with the JSON Data Type*, and *Working with XMLTYPE*.

cx_Oracle can be used to execute individual statements, one at a time. It does not read SQL*Plus ".sql" files. To read SQL files, use a technique like the one in `RunSqlScript()` in samples/SampleEnv.py

SQL statements should not contain a trailing semicolon (";") or forward slash ("/"). This will fail:

```
cur.execute("select * from MyTable;")
```

This is correct:

```
cur.execute("select * from MyTable")
```

1.5.1 SQL Queries

Queries (statements beginning with SELECT or WITH) can only be executed using the method *Cursor.execute()*. Rows can then be iterated over, or can be fetched using one of the methods *Cursor.fetchone()*, *Cursor.fetchmany()* or *Cursor.fetchall()*. There is a *default type mapping* to Python types that can be optionally *overridden*.

Important: Interpolating or concatenating user data with SQL statements, for example `cur.execute("SELECT * FROM mytab WHERE mycol = '" + myvar + "'")`, is a security risk and impacts performance. Use *bind variables* instead. For example, `cur.execute("SELECT * FROM mytab WHERE mycol = :mybv", mybv=myvar)`.

Fetch Methods

After *Cursor.execute()*, the cursor is returned as a convenience. This allows code to iterate over rows like:

```
cur = connection.cursor()
for row in cur.execute("select * from MyTable"):
    print(row)
```

Rows can also be fetched one at a time using the method *Cursor.fetchone()*:

```
cur = connection.cursor()
cur.execute("select * from MyTable")
while True:
    row = cur.fetchone()
    if row is None:
        break
    print(row)
```

If rows need to be processed in batches, the method *Cursor.fetchmany()* can be used. The size of the batch is controlled by the `numRows` parameter, which defaults to the value of *Cursor.arraysize*.

```
cur = connection.cursor()
cur.execute("select * from MyTable")
numRows = 10
while True:
    rows = cur.fetchmany(numRows)
    if not rows:
        break
    for row in rows:
        print(row)
```

If all of the rows need to be fetched, and can be contained in memory, the method *Cursor.fetchall()* can be used.

```
cur = connection.cursor()
cur.execute("select * from MyTable")
rows = cur.fetchall()
for row in rows:
    print(row)
```

The fetch methods return data as tuples. To return results as dictionaries, see *Changing Query Results with Rowfactories*.

Closing Cursors

A cursor may be used to execute multiple statements. Once it is no longer needed, it should be closed by calling *close()* in order to reclaim resources in the database. It will be closed automatically when the variable referencing it goes out of scope (and no further references are retained). One other way to control the lifetime of a cursor is to use a "with" block, which ensures that a cursor is closed once the block is completed. For example:

```
with connection.cursor() as cursor:
    for row in cursor.execute("select * from MyTable"):
        print(row)
```

This code ensures that, once the block is completed, the cursor is closed and resources have been reclaimed by the database. In addition, any attempt to use the variable `cursor` outside of the block will simply fail.

Query Column Metadata

After executing a query, the column metadata such as column names and data types can be obtained using *Cursor.description*:

```
cur = connection.cursor()
cur.execute("select * from MyTable")
for column in cur.description:
    print(column)
```

This could result in metadata like:

```
('ID', <class 'cx_Oracle.DB_TYPE_NUMBER'>, 39, None, 38, 0, 0)
('NAME', <class 'cx_Oracle.DB_TYPE_VARCHAR'>, 20, 20, None, None, 1)
```

Fetch Data Types

The following table provides a list of all of the data types that cx_Oracle knows how to fetch. The middle column gives the type that is returned in the *query metadata*. The last column gives the type of Python object that is returned by default. Python types can be changed with *Output Type Handlers*.

Oracle Database Type	cx_Oracle Database Type	Default Python type
BFILE	*cx_Oracle.DB_TYPE_BFILE*	*cx_Oracle.LOB*
BINARY_DOUBLE	*cx_Oracle.DB_TYPE_BINARY_DOUBLE*	float
BINARY_FLOAT	*cx_Oracle.DB_TYPE_BINARY_FLOAT*	float
BLOB	*cx_Oracle.DB_TYPE_BLOB*	*cx_Oracle.LOB*
CHAR	*cx_Oracle.DB_TYPE_CHAR*	str
CLOB	*cx_Oracle.DB_TYPE_CLOB*	*cx_Oracle.LOB*
CURSOR	*cx_Oracle.DB_TYPE_CURSOR*	*cx_Oracle.Cursor*
DATE	*cx_Oracle.DB_TYPE_DATE*	datetime.datetime
INTERVAL DAY TO SECOND	*cx_Oracle.DB_TYPE_INTERVAL_DS*	datetime.timedelta
LONG	*cx_Oracle.DB_TYPE_LONG*	str
LONG RAW	*cx_Oracle.DB_TYPE_LONG_RAW*	bytes
NCHAR	*cx_Oracle.DB_TYPE_NCHAR*	str
NCLOB	*cx_Oracle.DB_TYPE_NCLOB*	*cx_Oracle.LOB*
NUMBER	*cx_Oracle.DB_TYPE_NUMBER*	float or int[1]
NVARCHAR2	*cx_Oracle.DB_TYPE_NVARCHAR*	str
OBJECT[3]	*cx_Oracle.DB_TYPE_OBJECT*	*cx_Oracle.Object*
RAW	*cx_Oracle.DB_TYPE_RAW*	bytes
ROWID	*cx_Oracle.DB_TYPE_ROWID*	str
TIMESTAMP	*cx_Oracle.DB_TYPE_TIMESTAMP*	datetime.datetime
TIMESTAMP WITH LOCAL TIME ZONE	*cx_Oracle.DB_TYPE_TIMESTAMP_LTZ*	datetime.datetime[2]
TIMESTAMP WITH TIME ZONE	*cx_Oracle.DB_TYPE_TIMESTAMP_TZ*	datetime.datetime[2]
UROWID	*cx_Oracle.DB_TYPE_ROWID*	str
VARCHAR2	*cx_Oracle.DB_TYPE_VARCHAR*	str

Changing Fetched Data Types with Output Type Handlers

Sometimes the default conversion from an Oracle Database type to a Python type must be changed in order to prevent data loss or to fit the purposes of the Python application. In such cases, an output type handler can be specified for queries. Output type handlers do not affect values returned from *Cursor.callfunc()* or *Cursor.callproc()*.

Output type handlers can be specified on the *connection* or on the *cursor*. If specified on the cursor, fetch type handling is only changed on that particular cursor. If specified on the connection, all cursors created by that connection will have their fetch type handling changed.

The output type handler is expected to be a function with the following signature:

[1] If the precision and scale obtained from query column metadata indicate that the value can be expressed as an integer, the value will be returned as an int. If the column is unconstrained (no precision and scale specified), the value will be returned as a float or an int depending on whether the value itself is an integer. In all other cases the value is returned as a float.

[3] These include all user-defined types such as VARRAY, NESTED TABLE, etc.

[2] The timestamps returned are naive timestamps without any time zone information present.

```
handler(cursor, name, defaultType, size, precision, scale)
```

The parameters are the same information as the query column metadata found in *Cursor.description*. The function is called once for each column that is going to be fetched. The function is expected to return a *variable object* (generally by a call to *Cursor.var()*) or the value None. The value None indicates that the default type should be used.

Examples of output handlers are shown in *Fetched Number Precision* and *Fetching LOBs as Strings and Bytes*. Also see samples such as samples/TypeHandlers.py

Fetched Number Precision

One reason for using an output type handler is to ensure that numeric precision is not lost when fetching certain numbers. Oracle Database uses decimal numbers and these cannot be converted seamlessly to binary number representations like Python floats. In addition, the range of Oracle numbers exceeds that of floating point numbers. Python has decimal objects which do not have these limitations and cx_Oracle knows how to perform the conversion between Oracle numbers and Python decimal values if directed to do so.

The following code sample demonstrates the issue:

```
cur = connection.cursor()
cur.execute("create table test_float (X number(5, 3))")
cur.execute("insert into test_float values (7.1)")
connection.commit()
cur.execute("select * from test_float")
val, = cur.fetchone()
print(val, "* 3 =", val * 3)
```

This displays `7.1 * 3 = 21.299999999999997`

Using Python decimal objects, however, there is no loss of precision:

```
import decimal

def NumberToDecimal(cursor, name, defaultType, size, precision, scale):
    if defaultType == cx_Oracle.DB_TYPE_NUMBER:
        return cursor.var(decimal.Decimal, arraysize=cursor.arraysize)

cur = connection.cursor()
cur.outputtypehandler = NumberToDecimal
cur.execute("select * from test_float")
val, = cur.fetchone()
print(val, "* 3 =", val * 3)
```

This displays `7.1 * 3 = 21.3`

The Python `decimal.Decimal` converter gets called with the string representation of the Oracle number. The output from `decimal.Decimal` is returned in the output tuple.

See samples/ReturnNumbersAsDecimals.py

Changing Query Results with Outconverters

cx_Oracle "outconverters" can be used with *output type handlers* to change returned data.

For example, to make queries return empty strings instead of NULLs:

```
def OutConverter(value):
    if value is None:
        return ''
    return value

def OutputTypeHandler(cursor, name, defaultType, size, precision, scale):
    if defaultType in (cx_Oracle.DB_TYPE_VARCHAR, cx_Oracle.DB_TYPE_CHAR):
        return cursor.var(str, size, cur.arraysize, outconverter=OutConverter)

connection.outputtypehandler = OutputTypeHandler
```

Changing Query Results with Rowfactories

cx_Oracle "rowfactories" are methods called for each row that is retrieved from the database. The *Cursor.rowfactory()* method is called with the tuple that would normally be returned from the database. The method can convert the tuple to a different value and return it to the application in place of the tuple.

For example, to fetch each row of a query as a dictionary:

```
cursor.execute("select * from locations where location_id = 1000")
columns = [col[0] for col in cursor.description]
cursor.rowfactory = lambda *args: dict(zip(columns, args))
data = cursor.fetchone()
print(data)
```

The output is:

```
{'LOCATION_ID': 1000, 'STREET_ADDRESS': '1297 Via Cola di Rie', 'POSTAL_CODE': '00989
↪', 'CITY': 'Roma', 'STATE_PROVINCE': None, 'COUNTRY_ID': 'IT'}
```

If you join tables where the same column name occurs in both tables with different meanings or values, then use a column alias in the query. Otherwise only one of the similarly named columns will be included in the dictionary:

```
select
    cat_name,
    cats.color as cat_color,
    dog_name,
    dogs.color
from cats, dogs
```

Scrollable Cursors

Scrollable cursors enable applications to move backwards, forwards, to skip rows, and to move to a particular row in a query result set. The result set is cached on the database server until the cursor is closed. In contrast, regular cursors are restricted to moving forward.

A scrollable cursor is created by setting the parameter `scrollable=True` when creating the cursor. The method *Cursor.scroll()* is used to move to different locations in the result set.

Examples are:

```
cursor = connection.cursor(scrollable=True)
cursor.execute("select * from ChildTable order by ChildId")

cursor.scroll(mode="last")
```

(continues on next page)

(continued from previous page)

```
print("LAST ROW:", cursor.fetchone())

cursor.scroll(mode="first")
print("FIRST ROW:", cursor.fetchone())

cursor.scroll(8, mode="absolute")
print("ROW 8:", cursor.fetchone())

cursor.scroll(6)
print("SKIP 6 ROWS:", cursor.fetchone())

cursor.scroll(-4)
print("SKIP BACK 4 ROWS:", cursor.fetchone())
```

Fetching Oracle Database Objects and Collections

Oracle Database named object types and user-defined types can be fetched directly in queries. Each item is represented as a *Python object* corresponding to the Oracle Database object. This Python object can be traversed to access its elements. Attributes including *ObjectType.name* and *ObjectType.iscollection*, and methods including *Object.aslist()* and *Object.asdict()* are available.

For example, if a table `mygeometrytab` contains a column `geometry` of Oracle's predefined Spatial object type SDO_GEOMETRY, then it can be queried and printed:

```
cur.execute("select geometry from mygeometrytab")
for obj, in cur:
    dumpobject(obj)
```

Where `dumpobject()` is defined as:

```
def dumpobject(obj, prefix = ""):
    if obj.type.iscollection:
        print(prefix, "[")
        for value in obj.aslist():
            if isinstance(value, cx_Oracle.Object):
                dumpobject(value, prefix + "  ")
            else:
                print(prefix + "  ", repr(value))
        print(prefix, "]")
    else:
        print(prefix, "{")
        for attr in obj.type.attributes:
            value = getattr(obj, attr.name)
            if isinstance(value, cx_Oracle.Object):
                print(prefix + "   " + attr.name + ":")
                dumpobject(value, prefix + "  ")
            else:
                print(prefix + "   " + attr.name + ":", repr(value))
        print(prefix, "}")
```

This might produce output like:

```
{
  SDO_GTYPE: 2003
  SDO_SRID: None
```

(continues on next page)

(continued from previous page)

```
  SDO_POINT:
  {
    X: 1
    Y: 2
    Z: 3
  }
  SDO_ELEM_INFO:
  [
    1
    1003
    3
  ]
  SDO_ORDINATES:
  [
    1
    1
    5
    7
  ]
}
```

Other information on using Oracle objects is in *Using Bind Variables*.

Performance-sensitive applications should consider using scalar types instead of objects. If you do use objects, avoid calling *Connection.gettype()* unnecessarily, and avoid objects with large numbers of attributes.

Limiting Rows

Query data is commonly broken into one or more sets:

- To give an upper bound on the number of rows that a query has to process, which can help improve database scalability.
- To perform 'Web pagination' that allows moving from one set of rows to a next, or previous, set on demand.
- For fetching of all data in consecutive small sets for batch processing. This happens because the number of records is too large for Python to handle at one time.

The latter can be handled by calling *Cursor.fetchmany()* with one execution of the SQL query.

'Web pagination' and limiting the maximum number of rows are discussed in this section. For each 'page' of results, a SQL query is executed to get the appropriate set of rows from a table. Since the query may be executed more than once, make sure to use *bind variables* for row numbers and row limits.

Oracle Database 12c SQL introduced an `OFFSET` / `FETCH` clause which is similar to the `LIMIT` keyword of MySQL. In Python you can fetch a set of rows using:

```
myoffset = 0       // do not skip any rows (start at row 1)
mymaxnumrows = 20  // get 20 rows

sql =
  """SELECT last_name
     FROM employees
     ORDER BY last_name
     OFFSET :offset ROWS FETCH NEXT :maxnumrows ROWS ONLY"""

cur = connection.cursor()
```

(continues on next page)

(continued from previous page)

```
for row in cur.execute(sql, offset=myoffset, maxnumrows=mymaxnumrows):
    print(row)
```

In applications where the SQL query is not known in advance, this method sometimes involves appending the `OFFSET` clause to the 'real' user query. Be very careful to avoid SQL injection security issues.

For Oracle Database 11g and earlier there are several alternative ways to limit the number of rows returned. The old, canonical paging query is:

```
SELECT *
FROM (SELECT a.*, ROWNUM AS rnum
      FROM (YOUR_QUERY_GOES_HERE -- including the order by) a
      WHERE ROWNUM <= MAX_ROW)
WHERE rnum >= MIN_ROW
```

Here, `MIN_ROW` is the row number of first row and `MAX_ROW` is the row number of the last row to return. For example:

```
SELECT *
FROM (SELECT a.*, ROWNUM AS rnum
      FROM (SELECT last_name FROM employees ORDER BY last_name) a
      WHERE ROWNUM <= 20)
WHERE rnum >= 1
```

This always has an 'extra' column, here called RNUM.

An alternative and preferred query syntax for Oracle Database 11g uses the analytic `ROW_NUMBER()` function. For example to get the 1st to 20th names the query is:

```
SELECT last_name FROM
(SELECT last_name,
        ROW_NUMBER() OVER (ORDER BY last_name) AS myr
        FROM employees)
WHERE myr BETWEEN 1 and 20
```

Make sure to use *bind variables* for the upper and lower limit values.

Client Result Cache

Python cx_Oracle applications can use Oracle Database's Client Result Cache The CRC enables client-side caching of SQL query (SELECT statement) results in client memory for immediate use when the same query is re-executed. This is useful for reducing the cost of queries for small, mostly static, lookup tables, such as for postal codes. CRC reduces network *round-trips*, and also reduces database server CPU usage.

The cache is at the application process level. Access and invalidation is managed by the Oracle Client libraries. This removes the need for extra application logic, or external utilities, to implement a cache.

CRC can be enabled by setting the database parameters `CLIENT_RESULT_CACHE_SIZE` and `CLIENT_RESULT_CACHE_LAG`, and then restarting the database. For example, to set the parameters:

```
SQL> ALTER SYSTEM SET CLIENT_RESULT_CACHE_LAG = 3000 SCOPE=SPFILE;
SQL> ALTER SYSTEM SET CLIENT_RESULT_CACHE_SIZE = 64K SCOPE=SPFILE;
```

CRC can alternatively be configured in an *oraaccess.xml* or *sqlnet.ora* file on the Python host, see Client Configuration Parameters.

Tables can then be created, or altered, so repeated queries use CRC. This allows existing applications to use CRC without needing modification. For example:

```
SQL> CREATE TABLE cities (id number, name varchar2(40)) RESULT_CACHE (MODE FORCE);
SQL> ALTER TABLE locations RESULT_CACHE (MODE FORCE);
```

Alternatively, hints can be used in SQL statements. For example:

```
SELECT /*+ result_cache */ postal_code FROM locations
```

Querying Corrupt Data

If queries fail with the error "codec can't decode byte" when you select data, then:

- Check your *character set* is correct. Review the *client and database character sets*. Consider using UTF-8, if this is appropriate:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1
↪",
        encoding="UTF-8", nencoding="UTF-8")
```

- Check for corrupt data in the database.

If data really is corrupt, you can pass options to the internal decode() used by cx_Oracle to allow it to be selected and prevent the whole query failing. Do this by creating an *outputtypehandler* and setting `encodingErrors`. For example to replace corrupt characters in character columns:

```
def OutputTypeHandler(cursor, name, defaultType, size, precision, scale):
    if defaultType == cx_Oracle.STRING:
        return cursor.var(defaultType, size, arraysize=cursor.arraysize,
                encodingErrors="replace")

cursor.outputtypehandler = OutputTypeHandler

cursor.execute("select column1, column2 from SomeTableWithBadData")
```

Other codec behaviors can be chosen for `encodingErrors`, see Error Handlers.

1.5.2 INSERT and UPDATE Statements

SQL Data Manipulation Language statements (DML) such as INSERT and UPDATE can easily be executed with cx_Oracle. For example:

```
cur = connection.cursor()
cur.execute("insert into MyTable values (:idbv, :nmbv)", [1, "Fredico"])
```

Do not concatenate or interpolate user data into SQL statements. See *Using Bind Variables* instead.

See *Transaction Management* for best practices on committing and rolling back data changes.

When handling multiple data values, use *executemany()* for performance. See *Batch Statement Execution and Bulk Loading*

Inserting NULLs

Oracle requires a type, even for null values. When you pass the value None, then cx_Oracle assumes the type is STRING. If this is not the desired type, you can explicitly set it. For example, to insert a null *Oracle Spatial SDO_GEOMETRY* object:

```
typeObj = connection.gettype("SDO_GEOMETRY")
cur = connection.cursor()
cur.setinputsizes(typeObj)
cur.execute("insert into sometable values (:1)", [None])
```

1.6 PL/SQL Execution

PL/SQL stored procedures, functions and anonymous blocks can be called from cx_Oracle.

1.6.1 PL/SQL Stored Procedures

The *Cursor.callproc()* method is used to call PL/SQL procedures.

If a procedure with the following definition exists:

```
create or replace procedure myproc (
    a_Value1                            number,
    a_Value2                            out number
) as
begin
    a_Value2 := a_Value1 * 2;
end;
```

then the following Python code can be used to call it:

```
outVal = cursor.var(int)
cursor.callproc('myproc', [123, outVal])
print(outVal.getvalue())        # will print 246
```

Calling *Cursor.callproc()* actually generates an anonymous PL/SQL block as shown below, which is then executed:

```
cursor.execute("begin myproc(:1,:2); end;", [123, outval])
```

See *Using Bind Variables* for information on binding.

1.6.2 PL/SQL Stored Functions

The *Cursor.callfunc()* method is used to call PL/SQL functions.

The `returnType` parameter for *callfunc()* is expected to be a Python type, one of the *cx_Oracle types* or an *Object Type*.

If a function with the following definition exists:

```
create or replace function myfunc (
    a_StrVal varchar2,
    a_NumVal number
) return number as
begin
    return length(a_StrVal) + a_NumVal * 2;
end;
```

then the following Python code can be used to call it:

```
returnVal = cursor.callfunc("myfunc", int, ["a string", 15])
print(returnVal)        # will print 38
```

A more complex example that returns a spatial (SDO) object can be seen below. First, the SQL statements necessary to set up the example:

```
create table MyPoints (
    id number(9) not null,
    point sdo_point_type not null
);

insert into MyPoints values (1, sdo_point_type(125, 375, 0));

create or replace function spatial_queryfn (
    a_Id     number
) return sdo_point_type is
    t_Result sdo_point_type;
begin
    select point
    into t_Result
    from MyPoints
    where Id = a_Id;

    return t_Result;
end;
/
```

The Python code that will call this procedure looks as follows:

```
objType = connection.gettype("SDO_POINT_TYPE")
cursor = connection.cursor()
returnVal = cursor.callfunc("spatial_queryfn", objType, [1])
print("(%d, %d, %d)" % (returnVal.X, returnVal.Y, returnVal.Z))
# will print (125, 375, 0)
```

See *Using Bind Variables* for information on binding.

1.6.3 Anonymous PL/SQL Blocks

An anonymous PL/SQL block can be called as shown:

```
var = cursor.var(int)
cursor.execute("""
        begin
            :outVal := length(:inVal);
        end;""", inVal="A sample string", outVal=var)
print(var.getvalue())        # will print 15
```

See *Using Bind Variables* for information on binding.

1.6.4 Using DBMS_OUTPUT

The standard way to print output from PL/SQL is with the package DBMS_OUTPUT. Note, PL/SQL code that uses `DBMS_OUTPUT` runs to completion before any output is available to the user. Also, other database connections cannot access the buffer.

To use DBMS_OUTPUT:

- Call the PL/SQL procedure `DBMS_OUTPUT.ENABLE()` to enable output to be buffered for the connection.
- Execute some PL/SQL that calls `DBMS_OUTPUT.PUT_LINE()` to put text in the buffer.
- Call `DBMS_OUTPUT.GET_LINE()` or `DBMS_OUTPUT.GET_LINES()` repeatedly to fetch the text from the buffer until there is no more output.

For example:

```
# enable DBMS_OUTPUT
cursor.callproc("dbms_output.enable")

# execute some PL/SQL that calls DBMS_OUTPUT.PUT_LINE
cursor.execute("""
        begin
            dbms_output.put_line('This is the cx_Oracle manual');
            dbms_output.put_line('Demonstrating how to use DBMS_OUTPUT');
        end;""")

# tune this size for your application
chunk_size = 100

# create variables to hold the output
lines_var = cursor.arrayvar(str, chunk_size)
num_lines_var = cursor.var(int)
num_lines_var.setvalue(0, chunk_size)

# fetch the text that was added by PL/SQL
while True:
    cursor.callproc("dbms_output.get_lines", (lines_var, num_lines_var))
    num_lines = num_lines_var.getvalue()
    lines = lines_var.getvalue()[:num_lines]
    for line in lines:
        print(line or "")
    if num_lines < chunk_size:
        break
```

This will produce the following output:

```
This is the cx_Oracle manual
Demonstrating use of DBMS_OUTPUT
```

An alternative is to call `DBMS_OUTPUT.GET_LINE()` once per output line, which may be much slower:

```
textVar = cursor.var(str)
statusVar = cursor.var(int)
while True:
    cursor.callproc("dbms_output.get_line", (textVar, statusVar))
    if statusVar.getvalue() != 0:
        break
    print(textVar.getvalue())
```

1.6.5 Implicit results

Implicit results permit a Python program to consume cursors returned by a PL/SQL block without the requirement to use OUT REF CURSOR parameters. The method *Cursor.getimplicitresults()* can be used for this

purpose. It requires both the Oracle Client and Oracle Database to be 12.1 or higher.

An example using implicit results is as shown:

```
cursor.execute("""
        declare
            cust_cur sys_refcursor;
            sales_cur sys_refcursor;
        begin
            open cust_cur for SELECT * FROM cust_table;
            dbms_sql.return_result(cust_cur);

            open sales_cur for SELECT * FROM sales_table;
            dbms_sql.return_result(sales_cur);
        end;""")

for implicitCursor in cursor.getimplicitresults():
    for row in implicitCursor:
        print(row)
```

Data from both the result sets are returned:

```
(1, 'Tom')
(2, 'Julia')
(1000, 1, 'BOOKS')
(2000, 2, 'FURNITURE')
```

1.6.6 Edition-Based Redefinition (EBR)

Oracle Database's Edition-Based Redefinition feature enables upgrading of the database component of an application while it is in use, thereby minimizing or eliminating down time. This feature allows multiple versions of views, synonyms, PL/SQL objects and SQL Translation profiles to be used concurrently. Different versions of the database objects are associated with an "edition".

The simplest way to set an edition is to pass the `edition` parameter to *`cx_Oracle.connect()`* or *`cx_Oracle.SessionPool()`*:

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1",
        edition="newsales", encoding="UTF-8")
```

The edition could also be set by setting the environment variable `ORA_EDITION` or by executing the SQL statement:

```
alter session set edition = <edition name>;
```

Regardless of which method is used to set the edition, the value that is in use can be seen by examining the attribute *`Connection.edition`*. If no value has been set, the value will be None. This corresponds to the database default edition `ORA$BASE`.

Consider an example where one version of a PL/SQL function `Discount` is defined in the database default edition `ORA$BASE` and the other version of the same function is defined in a user created edition `DEMO`.

```
connect <username>/<password>

-- create function using the database default edition
CREATE OR REPLACE FUNCTION Discount(price IN NUMBER) RETURN NUMBER IS
BEGIN
    return price * 0.9;
```

(continues on next page)

(continued from previous page)

```
END;
/
```

A new edition named 'DEMO' is created and the user given permission to use editions. The use of `FORCE` is required if the user already contains one or more objects whose type is editionable and that also have non-editioned dependent objects.

```
connect system/<password>

CREATE EDITION demo;
ALTER USER <username> ENABLE EDITIONS FORCE;
GRANT USE ON EDITION demo to <username>;
```

The `Discount` function for the demo edition is as follows:

```
connect <username>/<password>

alter session set edition = demo;

-- Function for the demo edition
CREATE OR REPLACE FUNCTION Discount(price IN NUMBER) RETURN NUMBER IS
BEGIN
    return price * 0.5;
END;
/
```

The Python application can then call the required version of the PL/SQL function as shown:

```
connection = cx_Oracle.connect(<username>, <password>, "dbhost.example.com/orclpdb1",
        encoding="UTF-8")
print("Edition is:", repr(connection.edition))

cursor = connection.cursor()
discountedPrice = cursor.callfunc("Discount", int, [100])
print("Price after discount is:", discountedPrice)

# Use the edition parameter for the connection
connection = cx_Oracle.connect(<username>, <password>, "dbhost.example.com/orclpdb1",
        edition = "demo", encoding="UTF-8")
print("Edition is:", repr(connection.edition))

cursor = connection.cursor()
discountedPrice = cursor.callfunc("Discount", int, [100])
print("Price after discount is:", discountedPrice)
```

The output of the function call for the default and demo edition is as shown:

```
Edition is: None
Price after discount is:  90
Edition is: 'DEMO'
Price after discount is:  50
```

1.7 Using Bind Variables

SQL and PL/SQL statements that pass data to and from Oracle Database should use placeholders in SQL and PL/SQL statements that mark where data is supplied or returned. These placeholders are referred to as bind variables or bind parameters A bind variable is a colon-prefixed identifier or numeral. For example, there are two bind variables (`dept_id` and `dept_name`) in this SQL statement:

```
sql = """insert into departments (department_id, department_name)
         values (:dept_id, :dept_name)"""
cursor.execute(sql, [280, "Facility"])
```

Using bind variables is important for scalability and security. They help avoid SQL Injection security problems because data is never treated as part of an executable statement. Never concatenate or interpolate user data into SQL statements:

```
did = 280
dnm = "Facility"

# !! Never do this !!
sql = f"""insert into departments (department_id, department_name)
         values ({did}, {dnm})"""
cursor.execute(sql)
```

Bind variables reduce parsing and execution costs when statements are executed more than once with different data values. If you do not use bind variables, Oracle must reparse and cache multiple statements. When using bind variables, Oracle Database may be able to reuse the statement execution plan and context.

Bind variables can be used to substitute data, but cannot be used to substitute the text of the statement. You cannot, for example, use a bind variable where a column name or a table name is required. Bind variables also cannot be used in Data Definition Language (DDL) statements, such as CREATE TABLE or ALTER statements.

1.7.1 Binding By Name or Position

Binding can be done by name or by position. A named bind is performed when the bind variables in a statement are associated with a name. For example:

```
cursor.execute("""
        insert into departments (department_id, department_name)
        values (:dept_id, :dept_name)""", dept_id=280,
        dept_name="Facility")

# alternatively, the parameters can be passed as a dictionary instead of as
# keyword parameters
data = { "dept_id": 280, "dept_name": "Facility" }
cursor.execute("""
        insert into departments (department_id, department_name)
        values (:dept_id, :dept_name)""", data)
```

In the above example, the keyword parameter names or the keys of the dictionary must match the bind variable names. The advantages of this approach are that the location of the bind variables in the statement is not important, the names can be meaningful and the names can be repeated while still only supplying the value once.

A positional bind is performed when a list of bind values are passed to the execute() call. For example:

```
cursor.execute("""
        insert into departments (department_id, department_name)
        values (:dept_id, :dept_name)""", [280, "Facility"])
```

Note that for SQL statements, the order of the bind values must exactly match the order of each bind variable and duplicated names must have their values repeated. For PL/SQL statements, however, the order of the bind values must exactly match the order of each **unique** bind variable found in the PL/SQL block and values should not be repeated. In order to avoid this difference, binding by name is recommended when bind variable names are repeated.

1.7.2 Bind Direction

The caller can supply data to the database (IN), the database can return data to the caller (OUT) or the caller can supply initial data to the database and the database can supply the modified data back to the caller (IN/OUT). This is known as the bind direction.

The examples shown above have all supplied data to the database and are therefore classified as IN bind variables. In order to have the database return data to the caller, a variable must be created. This is done by calling the method *Cursor.var()*, which identifies the type of data that will be found in that bind variable and its maximum size among other things.

Here is an example showing how to use OUT binds. It calculates the sum of the integers 8 and 7 and stores the result in an OUT bind variable of type integer:

```
outVal = cursor.var(int)
cursor.execute("""
        begin
            :outVal := :inBindVar1 + :inBindVar2;
        end;""", outVal=outVal, inBindVar1=8, inBindVar2=7)
print(outVal.getvalue())        # will print 15
```

If instead of simply getting data back you wish to supply an initial value to the database, you can set the variable's initial value. This example is the same as the previous one but it sets the initial value first:

```
inOutVal = cursor.var(int)
inOutVal.setvalue(0, 25)
cursor.execute("""
        begin
            :inOutBindVar := :inOutBindVar + :inBindVar1 + :inBindVar2;
        end;""", inOutBindVar=inOutVal, inBindVar1=8, inBindVar2=7)
print(inOutVal.getvalue())        # will print 40
```

When binding data to parameters of PL/SQL procedures that are declared as OUT parameters, it is worth noting that any value that is set in the bind variable will be ignored. In addition, any parameters declared as IN/OUT that do not have a value set will start out with a value of `null`.

1.7.3 Binding Null Values

In cx_Oracle, null values are represented by the Python singleton `None`.

For example:

```
cursor.execute("""
        insert into departments (department_id, department_name)
        values (:dept_id, :dept_name)""", dept_id=280, dept_name=None)
```

In this specific case, because the DEPARTMENT_NAME column is defined as a NOT NULL column, an error will occur:

```
cx_Oracle.IntegrityError: ORA-01400: cannot insert NULL into ("HR"."DEPARTMENTS".
↪"DEPARTMENT_NAME")
```

If this value is bound directly, cx_Oracle assumes it to be a string (equivalent to a VARCHAR2 column). If you need to use a different Oracle type you will need to make a call to *Cursor.setinputsizes()* or create a bind variable with the correct type by calling *Cursor.var()*.

1.7.4 Binding ROWID Values

The pseudo-column ROWID uniquely identifies a row within a table. In cx_Oracle, ROWID values are represented as strings. The example below shows fetching a row and then updating that row by binding its rowid:

```
# fetch the row
cursor.execute("""
        select rowid, manager_id
        from departments
        where department_id = :dept_id""", dept_id=280)
rowid, manager_id = cursor.fetchone()

# update the row by binding ROWID
cursor.execute("""
        update departments set
            manager_id = :manager_id
        where rowid = :rid""", manager_id=205, rid=rowid)
```

1.7.5 DML RETURNING Bind Variables

When a RETURNING clause is used with a DML statement like UPDATE, INSERT, or DELETE, the values are returned to the application through the use of OUT bind variables. Consider the following example:

```
# The RETURNING INTO bind variable is a string
dept_name = cursor.var(str)

cursor.execute("""
        update departments set
            location_id = :loc_id
        where department_id = :dept_id
        returning department_name into :dept_name""",
        loc_id=1700, dept_id=50, dept_name=dept_name)
print(dept_name.getvalue())     # will print ['Shipping']
```

In the above example, since the WHERE clause matches only one row, the output contains a single item in the list. If the WHERE clause matched multiple rows, however, the output would contain as many items as there were rows that were updated.

No duplicate binds are allowed in a DML statement with a RETURNING clause, and no duplication is allowed between bind variables in the DML section and the RETURNING section of the statement.

1.7.6 LOB Bind Variables

Database CLOBs, NCLOBS, BLOBs and BFILEs can be bound with types *cx_Oracle.DB_TYPE_CLOB*, *cx_Oracle.DB_TYPE_NCLOB*, *cx_Oracle.DB_TYPE_BLOB* and *cx_Oracle.DB_TYPE_BFILE* respectively. LOBs fetched from the database or created with *Connection.createlob()* can also be bound.

LOBs may represent Oracle Database persistent LOBs (those stored in tables) or temporary LOBs (such as those created with *Connection.createlob()* or returned by some SQL and PL/SQL operations).

LOBs can be used as IN, OUT or IN/OUT bind variables.

See *Using CLOB and BLOB Data* for examples.

1.7.7 REF CURSOR Bind Variables

cx_Oracle provides the ability to bind and define PL/SQL REF cursors. As an example, consider the PL/SQL procedure:

```
CREATE OR REPLACE PROCEDURE find_employees (
    p_query IN VARCHAR2,
    p_results OUT SYS_REFCURSOR
) AS
BEGIN
    OPEN p_results FOR
        SELECT employee_id, first_name, last_name
        FROM employees
        WHERE UPPER(first_name || ' ' || last_name || ' ' || email)
            LIKE '%' || UPPER(p_query) || '%';
END;
/
```

A newly opened cursor can be bound to the REF CURSOR parameter, as shown in the following Python code. After the PL/SQL procedure has been called with *Cursor.callproc()*, the cursor can then be fetched just like any other cursor which had executed a SQL query:

```
refCursor = connection.cursor()
cursor.callproc("find_employees", ['Smith', refCursor])
for row in refCursor:
    print(row)
```

With Oracle's sample HR schema there are two employees with the last name 'Smith' so the result is:

```
(159, 'Lindsey', 'Smith')
(171, 'William', 'Smith')
```

To return a REF CURSOR from a PL/SQL function, use `cx_Oracle.DB_TYPE_CURSOR` for the return type of *Cursor.callfunc()*:

```
refCursor = cursor.callfunc('example_package.f_get_cursor', cx_Oracle.DB_TYPE_CURSOR)
for row in refCursor:
    print(row)
```

1.7.8 Binding PL/SQL Collections

PL/SQL Collections like Associative Arrays can be bound as IN, OUT, and IN/OUT variables. When binding IN values, an array can be passed directly as shown in this example, which sums up the lengths of all of the strings in the

provided array. First the PL/SQL package definition:

```
create or replace package mypkg as

    type udt_StringList is table of varchar2(100) index by binary_integer;

    function DemoCollectionIn (
        a_Values            udt_StringList
    ) return number;

end;
/

create or replace package body mypkg as

    function DemoCollectionIn (
        a_Values            udt_StringList
    ) return number is
        t_ReturnValue       number := 0;
    begin
        for i in 1..a_Values.count loop
            t_ReturnValue := t_ReturnValue + length(a_Values(i));
        end loop;
        return t_ReturnValue;
    end;

end;
/
```

Then the Python code:

```
values = ["String One", "String Two", "String Three"]
returnVal = cursor.callfunc("mypkg.DemoCollectionIn", int, [values])
print(returnVal)        # will print 32
```

In order get values back from the database, a bind variable must be created using *Cursor.arrayvar()*. The first parameter to this method is a Python type that cx_Oracle knows how to handle or one of the cx_Oracle *DB API Types*. The second parameter is the maximum number of elements that the array can hold or an array providing the value (and indirectly the maximum length). The final parameter is optional and only used for strings and bytes. It identifies the maximum length of the strings and bytes that can be stored in the array. If not specified, the length defaults to 4000 bytes.

Consider the following PL/SQL package:

```
create or replace package mypkg as

    type udt_StringList is table of varchar2(100) index by binary_integer;

    procedure DemoCollectionOut (
        a_NumElements       number,
        a_Values            out nocopy udt_StringList
    );

    procedure DemoCollectionInOut (
        a_Values            in out nocopy udt_StringList
    );

end;
```

(continues on next page)

(continued from previous page)

```
/

create or replace package body mypkg as

    procedure DemoCollectionOut (
        a_NumElements       number,
        a_Values            out nocopy udt_StringList
    ) is
    begin
        for i in 1..a_NumElements loop
            a_Values(i) := 'Demo out element #' || to_char(i);
        end loop;
    end;

    procedure DemoCollectionInOut (
        a_Values            in out nocopy udt_StringList
    ) is
    begin
        for i in 1..a_Values.count loop
            a_Values(i) := 'Converted element #' || to_char(i) ||
                    ' originally had length ' || length(a_Values(i));
        end loop;
    end;

end;
/
```

The Python code to process an OUT collection would look as follows. Note the call to *Cursor.arrayvar()* which creates space for an array of strings. Each string would permit up to 100 bytes and only 10 strings would be permitted. If the PL/SQL block exceeds the maximum number of strings allowed the error `ORA-06513: PL/SQL: index for PL/SQL table out of range for host language array` would be raised.

```
outArrayVar = cursor.arrayvar(str, 10, 100)
cursor.callproc("mypkg.DemoCollectionOut", [5, outArrayVar])
for val in outArrayVar.getvalue():
    print(val)
```

This would produce the following output:

```
Demo out element #1
Demo out element #2
Demo out element #3
Demo out element #4
Demo out element #5
```

The Python code to process an IN/OUT collections is similar. Note the different call to *Cursor.arrayvar()* which creates space for an array of strings, but uses an array to determine both the maximum length of the array and its initial value.

```
inValues = ["String One", "String Two", "String Three", "String Four"]
inOutArrayVar = cursor.arrayvar(str, inValues)
cursor.callproc("mypkg.DemoCollectionInOut", [inOutArrayVar])
for val in inOutArrayVar.getvalue():
    print(val)
```

This would produce the following output:

```
Converted element #1 originally had length 10
Converted element #2 originally had length 10
Converted element #3 originally had length 12
Converted element #4 originally had length 11
```

If an array variable needs to have an initial value but also needs to allow for more elements than the initial value contains, the following code can be used instead:

```
inOutArrayVar = cursor.arrayvar(str, 10, 100)
inOutArrayVar.setvalue(0, ["String One", "String Two"])
```

All of the collections that have been bound in preceding examples have used contiguous array elements. If an associative array with sparse array elements is needed, a different approach is required. Consider the following PL/SQL code:

```
create or replace package mypkg as

    type udt_StringList is table of varchar2(100) index by binary_integer;

    procedure DemoCollectionOut (
        a_Value                         out nocopy udt_StringList
    );

end;
/

create or replace package body mypkg as

    procedure DemoCollectionOut (
        a_Value                         out nocopy udt_StringList
    ) is
    begin
        a_Value(-1048576) := 'First element';
        a_Value(-576) := 'Second element';
        a_Value(284) := 'Third element';
        a_Value(8388608) := 'Fourth element';
    end;

end;
/
```

Note that the collection element indices are separated by large values. The technique used above would fail with the exception `ORA-06513: PL/SQL: index for PL/SQL table out of range for host language array`. The code required to process this collection looks like this instead:

```
collectionType = connection.gettype("MYPKG.UDT_STRINGLIST")
collection = collectionType.newobject()
cursor.callproc("mypkg.DemoCollectionOut", [collection])
print(collection.aslist())
```

This produces the output:

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

Note the use of *`Object.aslist()`* which returns the collection element values in index order as a simple Python list. The indices themselves are lost in this approach. Starting from cx_Oracle 7.0, the associative array can be turned into a Python dictionary using *`Object.asdict()`*. If that value was printed in the previous example instead, the

output would be:

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

If the elements need to be traversed in index order, the methods *Object.first()* and *Object.next()* can be used. The method *Object.getelement()* can be used to acquire the element at a particular index. This is shown in the following code:

```
ix = collection.first()
while ix is not None:
    print(ix, "->", collection.getelement(ix))
    ix = collection.next(ix)
```

This produces the output:

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

Similarly, the elements can be traversed in reverse index order using the methods *Object.last()* and *Object.prev()* as shown in the following code:

```
ix = collection.last()
while ix is not None:
    print(ix, "->", collection.getelement(ix))
    ix = collection.prev(ix)
```

This produces the output:

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

1.7.9 Binding PL/SQL Records

PL/SQL record type objects can also be bound for IN, OUT and IN/OUT bind variables. For example:

```
create or replace package mypkg as

    type udt_DemoRecord is record (
        NumberValue                     number,
        StringValue                     varchar2(30),
        DateValue                       date,
        BooleanValue                    boolean
    );

    procedure DemoRecordsInOut (
        a_Value                         in out nocopy udt_DemoRecord
    );

end;
/
```

(continues on next page)

(continued from previous page)

```
create or replace package body mypkg as

    procedure DemoRecordsInOut (
        a_Value                         in out nocopy udt_DemoRecord
    ) is
    begin
        a_Value.NumberValue := a_Value.NumberValue * 2;
        a_Value.StringValue := a_Value.StringValue || ' (Modified)';
        a_Value.DateValue := a_Value.DateValue + 5;
        a_Value.BooleanValue := not a_Value.BooleanValue;
    end;

end;
/
```

Then this Python code can be used to call the stored procedure which will update the record:

```
# create and populate a record
recordType = connection.gettype("MYPKG.UDT_DEMORECORD")
record = recordType.newobject()
record.NUMBERVALUE = 6
record.STRINGVALUE = "Test String"
record.DATEVALUE = datetime.datetime(2016, 5, 28)
record.BOOLEANVALUE = False

# show the original values
print("NUMBERVALUE ->", record.NUMBERVALUE)
print("STRINGVALUE ->", record.STRINGVALUE)
print("DATEVALUE ->", record.DATEVALUE)
print("BOOLEANVALUE ->", record.BOOLEANVALUE)
print()

# call the stored procedure which will modify the record
cursor.callproc("mypkg.DemoRecordsInOut", [record])

# show the modified values
print("NUMBERVALUE ->", record.NUMBERVALUE)
print("STRINGVALUE ->", record.STRINGVALUE)
print("DATEVALUE ->", record.DATEVALUE)
print("BOOLEANVALUE ->", record.BOOLEANVALUE)
```

This will produce the following output:

```
NUMBERVALUE -> 6
STRINGVALUE -> Test String
DATEVALUE -> 2016-05-28 00:00:00
BOOLEANVALUE -> False

NUMBERVALUE -> 12
STRINGVALUE -> Test String (Modified)
DATEVALUE -> 2016-06-02 00:00:00
BOOLEANVALUE -> True
```

Note that when manipulating records, all of the attributes must be set by the Python program in order to avoid an Oracle Client bug which will result in unexpected values or the Python application segfaulting.

1.7.10 Binding Spatial Datatypes

Oracle Spatial datatypes objects can be represented by Python objects and their attribute values can be read and updated. The objects can further be bound and committed to database. This is similar to the examples above.

An example of fetching SDO_GEOMETRY is in *Oracle Database Objects and Collections*.

1.7.11 Changing Bind Data Types using an Input Type Handler

Input Type Handlers allow applications to change how data is bound to statements, or even to enable new types to be bound directly.

An input type handler is enabled by setting the attribute *Cursor.inputtypehandler* or *Connection.inputtypehandler*.

Input type handlers can be combined with variable converters to bind Python objects seamlessly:

```
# A standard Python object
class Building(object):
    def __init__(self, buildingId, description, numFloors, dateBuilt):
        self.buildingId = buildingId
        self.description = description
        self.numFloors = numFloors
        self.dateBuilt = dateBuilt

building = Building(1, "Skyscraper 1", 5, datetime.date(2001, 5, 24))

# Get Python representation of the Oracle user defined type UDT_BUILDING
objType = con.gettype("UDT_BUILDING")

# convert a Python Building object to the Oracle user defined type UDT_BUILDING
def BuildingInConverter(value):
    obj = objType.newobject()
    obj.BUILDINGID  = value.buildingId
    obj.DESCRIPTION = value.description
    obj.NUMFLOORS   = value.numFloors
    obj.DATEBUILT   = value.dateBuilt
    return obj

def InputTypeHandler(cursor, value, numElements):
    if isinstance(value, Building):
        return cursor.var(cx_Oracle.DB_TYPE_OBJECT, arraysize = numElements,
                inconverter = BuildingInConverter, typename = objType.name)


# With the input type handler, the bound Python object is converted
# to the required Oracle object before being inserted
cur.inputtypehandler = InputTypeHandler
cur.execute("insert into myTable values (:1, :2)", (1, building))
```

1.7.12 Binding Multiple Values to a SQL WHERE IN Clause

To use an IN clause with multiple values in a WHERE clause, you must define and bind multiple values. You cannot bind an array of values. For example:

```
cursor.execute("""
        select employee_id, first_name, last_name
        from employees
        where last_name in (:name1, :name2)""",
        name1="Smith", name2="Taylor")
for row in cursor:
    print(row)
```

This will produce the following output:

```
(159, 'Lindsey', 'Smith')
(171, 'William', 'Smith')
(176, 'Jonathon', 'Taylor')
(180, 'Winston', 'Taylor')
```

If this sort of query is executed multiple times with differing numbers of values, a bind variable should be included for each possible value up to the maximum number of values that can be provided. Missing values can be bound with the value `None`. For example, if the query above is used for up to 5 values, the code should be adjusted as follows:

```
cursor.execute("""
        select employee_id, first_name, last_name
        from employees
        where last_name in (:name1, :name2, :name3, :name4, :name5)""",
        name1="Smith", name2="Taylor", name3=None, name4=None, name5=None)
for row in cursor:
    print(row)
```

This will produce the same output as the original example.

If the number of values is only going to be known at runtime, then a SQL statement can be built up as follows:

```
bindValues = ["Gates", "Marvin", "Fay"]
bindNames = [":" + str(i + 1) for i in range(len(bindValues))]
sql = "select employee_id, first_name, last_name from employees " + \
        "where last_name in (%s)" % (",".join(bindNames))
cursor.execute(sql, bindValues)
for row in cursor:
    print(row)
```

Another solution for a larger number of values is to construct a SQL statement like:

```
SELECT ... WHERE col IN ( <something that returns a list of rows> )
```

The easiest way to do the '<something that returns a list of rows>' will depend on how the data is initially represented and the number of items. You might look at using CONNECT BY or nested tables. Or, for really large numbers of items, you might prefer to use a global temporary table.

1.7.13 Binding Column and Table Names

Column and table names cannot be bound in SQL queries. You can concatenate text to build up a SQL statement, but make sure you use an Allow List or other means to validate the data in order to avoid SQL Injection security issues:

```
tableAllowList = ['employees', 'departments']
tableName = getTableName() #  get the table name from user input
if tableName not in tableAllowList:
```

(continues on next page)

(continued from previous page)

```
    raise Exception('Invalid table name')
sql = 'select * from ' + tableName
```

Binding column names can be done either by using the above method or by using a CASE statement. The example below demonstrates binding a column name in an ORDER BY clause:

```
sql = """
        SELECT * FROM departments
        ORDER BY
            CASE :bindvar
                WHEN 'department_id' THEN DEPARTMENT_ID
                ELSE MANAGER_ID
            END"""

columnName = getColumnName() # Obtain a column name from the user
cursor.execute(sql, [colname])
```

Depending on the name provided by the user, the query results will be ordered either by the column `DEPARTMENT_ID` or the column `MANAGER_ID`.

1.8 Using CLOB and BLOB Data

Oracle Database uses *LOB Objects* to store large data such as text, images, videos and other multimedia formats. The maximum size of a LOB is limited to the size of the tablespace storing it.

There are four types of LOB (large object):

- BLOB - Binary Large Object, used for storing binary data. cx_Oracle uses the type *cx_Oracle.DB_TYPE_BLOB*.
- CLOB - Character Large Object, used for string strings in the database character set format. cx_Oracle uses the type *cx_Oracle.DB_TYPE_CLOB*.
- NCLOB - National Character Large Object, used for string strings in the national character set format. cx_Oracle uses the type *cx_Oracle.DB_TYPE_NCLOB*.
- BFILE - External Binary File, used for referencing a file stored on the host operating system outside of the database. cx_Oracle uses the type *cx_Oracle.DB_TYPE_BFILE*.

LOBs can be streamed to, and from, Oracle Database.

LOBs up to 1 GB in length can be also be handled directly as strings or bytes in cx_Oracle. This makes LOBs easy to work with, and has significant performance benefits over streaming. However it requires the entire LOB data to be present in Python memory, which may not be possible.

See GitHub for LOB examples.

1.8.1 Simple Insertion of LOBs

Consider a table with CLOB and BLOB columns:

```
CREATE TABLE lob_tbl (
    id NUMBER,
    c CLOB,
    b BLOB
);
```

With cx_Oracle, LOB data can be inserted in the table by binding strings or bytes as needed:

```
with open('example.txt', 'r') as f:
    textdata = f.read()

with open('image.png', 'rb') as f:
    imgdata = f.read()

cursor.execute("""
        insert into lob_tbl (id, c, b)
        values (:lobid, :clobdata, :blobdata)""",
        lobid=10, clobdata=textdata, blobdata=imgdata)
```

Note that with this approach, LOB data is limited to 1 GB in size.

1.8.2 Fetching LOBs as Strings and Bytes

CLOBs and BLOBs smaller than 1 GB can queried from the database directly as strings and bytes. This can be much faster than streaming.

A *Connection.outputtypehandler* or *Cursor.outputtypehandler* needs to be used as shown in this example:

```
def OutputTypeHandler(cursor, name, defaultType, size, precision, scale):
    if defaultType == cx_Oracle.DB_TYPE_CLOB:
        return cursor.var(cx_Oracle.DB_TYPE_LONG, arraysize=cursor.arraysize)
    if defaultType == cx_Oracle.DB_TYPE_BLOB:
        return cursor.var(cx_Oracle.DB_TYPE_LONG_RAW, arraysize=cursor.arraysize)

idVal = 1
textData = "The quick brown fox jumps over the lazy dog"
bytesData = b"Some binary data"
cursor.execute("insert into lob_tbl (id, c, b) values (:1, :2, :3)",
        [idVal, textData, bytesData])

connection.outputtypehandler = OutputTypeHandler
cursor.execute("select c, b from lob_tbl where id = :1", [idVal])
clobData, blobData = cursor.fetchone()
print("CLOB length:", len(clobData))
print("CLOB data:", clobData)
print("BLOB length:", len(blobData))
print("BLOB data:", blobData)
```

This displays:

```
CLOB length: 43
CLOB data: The quick brown fox jumps over the lazy dog
BLOB length: 16
BLOB data: b'Some binary data'
```

1.8.3 Streaming LOBs (Read)

Without the output type handler, the CLOB and BLOB values are fetched as *LOB objects*. The size of the LOB object can be obtained by calling *LOB.size()* and the data can be read by calling *LOB.read()*:

```
idVal = 1
textData = "The quick brown fox jumps over the lazy dog"
bytesData = b"Some binary data"
cursor.execute("insert into lob_tbl (id, c, b) values (:1, :2, :3)",
        [idVal, textData, bytesData])

cursor.execute("select b, c from lob_tbl where id = :1", [idVal])
b, c = cursor.fetchone()
print("CLOB length:", c.size())
print("CLOB data:", c.read())
print("BLOB length:", b.size())
print("BLOB data:", b.read())
```

This approach produces the same results as the previous example but it will perform more slowly because it requires more *round-trips* to Oracle Database and has higher overhead. It is needed, however, if the LOB data cannot be fetched as one block of data from the server.

To stream the BLOB column, the *LOB.read()* method can be called repeatedly until all of the data has been read, as shown below:

```
cursor.execute("select b from lob_tbl where id = :1", [10])
blob, = cursor.fetchone()
offset = 1
numBytesInChunk = 65536
with open("image.png", "wb") as f:
    while True:
        data = blob.read(offset, numBytesInChunk)
        if data:
            f.write(data)
        if len(data) < numBytesInChunk:
            break
        offset += len(data)
```

1.8.4 Streaming LOBs (Write)

If a row containing a LOB is being inserted or updated, and the quantity of data that is to be inserted or updated cannot fit in a single block of data, the data can be streamed using the method *LOB.write()* instead as shown in the following code:

```
idVal = 9
lobVar = cursor.var(cx_Oracle.DB_TYPE_BLOB)
cursor.execute("""
        insert into lob_tbl (id, b)
        values (:1, empty_blob())
        returning b into :2""", [idVal, lobVar])
blob, = lobVar.getvalue()
offset = 1
numBytesInChunk = 65536
with open("image.png", "rb") as f:
    while True:
        data = f.read(numBytesInChunk)
        if data:
            blob.write(data, offset)
        if len(data) < numBytesInChunk:
            break
```

(continues on next page)

(continued from previous page)

```
        offset += len(data)
connection.commit()
```

1.8.5 Temporary LOBs

All of the examples shown thus far have made use of permanent LOBs. These are LOBs that are stored in the database. Oracle also supports temporary LOBs that are not stored in the database but can be used to pass large quantities of data. These LOBs use space in the temporary tablespace until all variables referencing them go out of scope or the connection in which they are created is explicitly closed.

When calling PL/SQL procedures with data that exceeds 32,767 bytes in length, cx_Oracle automatically creates a temporary LOB internally and passes that value through to the procedure. If the data that is to be passed to the procedure exceeds that which can fit in a single block of data, however, you can use the method *Connection.createlob()* to create a temporary LOB. This LOB can then be read and written just like in the examples shown above for persistent LOBs.

1.9 Working with the JSON Data Type

Native support for JSON data was introduced in Oracle database 12c. You can use the relational database to store and query JSON data and benefit from the easy extensibility of JSON data while retaining the performance and structure of the relational database. JSON data is stored in the database in BLOB, CLOB or VARCHAR2 columns. For performance reasons, it is always a good idea to store JSON data in BLOB columns. To ensure that only JSON data is stored in that column, use a check constraint with the clause `is JSON` as shown in the following SQL to create a table containing JSON data:

```
create table customers (
    id integer not null primary key,
    json_data blob check (json_data is json)
);
```

The following Python code can then be used to insert some data into the database:

```
import json

customerData = dict(name="Rod", dept="Sales", location="Germany")
cursor.execute("insert into customers (id, json_data) values (:1, :2)",
        [1, json.dumps(customerData)])
```

The data can be retrieved in its entirety using the following code:

```
import json

for blob, in cursor.execute("select json_data from customers"):
    data = json.loads(blob.read())
    print(data["name"])      # will print Rod
```

If only the department needs to be read, the following code can be used instead:

```
for deptName, in cursor.execute("select c.json_data.dept from customers c"):
    print(deptName)          # will print Sales
```

You can convert the data stored in relational tables into JSON data by using the JSON_OBJECT SQL operator. For example:

```
import json
cursor.execute("""
        select json_object(
            'id' value employee_id,
            'name' value (first_name || ' ' || last_name))
        from employees where rownum <= 3""")
for value, in cursor:
    print(json.loads(value,))
```

The result is:

```
{'id': 100, 'name': 'Steven King'}
{'id': 101, 'name': 'Neena Kochhar'}
{'id': 102, 'name': 'Lex De Haan'}
```

See JSON Developer's Guide for more information about using JSON in Oracle Database.

1.10 Simple Oracle Document Access (SODA)

1.10.1 Overview

Oracle Database Simple Oracle Document Access (SODA) allows documents to be inserted, queried, and retrieved from Oracle Database using a set of NoSQL-style cx_Oracle methods. Documents are generally JSON data but they can be any data at all (including video, images, sounds, or other binary content). Documents can be fetched from the database by key lookup or by using query-by-example (QBE) pattern-matching.

SODA uses a SQL schema to store documents but you do not need to know SQL or how the documents are stored. However, access via SQL does allow use of advanced Oracle Database functionality such as analytics for reporting.

For general information on SODA, see the SODA home page and Oracle Database Introduction to SODA.

cx_Oracle uses the following objects for SODA:

- *SODA Database Object*: The top level object for cx_Oracle SODA operations. This is acquired from an Oracle Database connection. A 'SODA database' is an abstraction, allowing access to SODA collections in that 'SODA database', which then allow access to documents in those collections. A SODA database is analogous to an Oracle Database user or schema, a collection is analogous to a table, and a document is analogous to a table row with one column for a unique document key, a column for the document content, and other columns for various document attributes.
- *SODA Collection Object*: Represents a collection of SODA documents. By default, collections allow JSON documents to be stored. This is recommended for most SODA users. However optional metadata can set various details about a collection, such as its database storage, whether it should track version and time stamp document components, how such components are generated, and what document types are supported. By default, the name of the Oracle Database table storing a collection is the same as the collection name. Note: do not use SQL to drop the database table, since SODA metadata will not be correctly removed. Use the `SodaCollection.drop()` method instead.
- *SODA Document Object*: Represents a document. Typically the document content will be JSON. The document has properties including the content, a key, timestamps, and the media type. By default, document keys are automatically generated. See *SODA Document objects* for the forms of SodaDoc.
- *SODA Document Cursor*: A cursor object representing the result of the `SodaOperation.getCursor()` method from a `SodaCollection.find()` operation. It can be iterated over to access each SodaDoc.
- *SODA Operation Object*: An internal object used with `SodaCollection.find()` to perform read and write operations on documents. Chained methods set properties on a SodaOperation object which is then used by a

terminal method to find, count, replace, or remove documents. This is an internal object that should not be directly accessed.

1.10.2 SODA Example

Creating and adding documents to a collection can be done as follows:

```
soda = connection.getSodaDatabase()

# create a new SODA collection; this will open an existing collection, if
# the name is already in use
collection = soda.createCollection("mycollection")

# insert a document into the collection; for the common case of a JSON
# document, the content can be a simple Python dictionary which will
# internally be converted to a JSON document
content = {'name': 'Matilda', 'address': {'city': 'Melbourne'}}
returnedDoc = collection.insertOneAndGet(content)
key = returnedDoc.key
print('The key of the new SODA document is: ', key)
```

By default, a system generated key is created when documents are inserted. With a known key, you can retrieve a document:

```
# this will return a dictionary (as was inserted in the previous code)
content = collection.find().key(key).getOne().getContent()
print(content)
```

You can also search for documents using query-by-example syntax:

```
# Find all documents with names like 'Ma%'
print("Names matching 'Ma%'")
qbe = {'name': {'$like': 'Ma%'}}
for doc in collection.find().filter(qbe).getDocuments():
    content = doc.getContent()
    print(content["name"])
```

See the samples directory for runnable SODA examples.

1.11 Working with XMLTYPE

Oracle XMLType columns are fetched as strings by default. This is currently limited to the maximum length of a VARCHAR2 column. To return longer XML values, they must be queried as LOB values instead.

The examples below demonstrate using XMLType data with cx_Oracle. The following table will be used in these examples:

```
CREATE TABLE xml_table (
    id NUMBER,
    xml_data SYS.XMLTYPE
);
```

Inserting into the table can be done by simply binding a string as shown:

```
xmlData = """<?xml version="1.0"?>
        <customer>
            <name>John Smith</name>
            <Age>43</Age>
            <Designation>Professor</Designation>
            <Subject>Mathematics</Subject>
        </customer>"""
cursor.execute("insert into xml_table values (:id, :xml)",
        id=1, xml=xmlData)
```

This approach works with XML strings up to 1 GB in size. For longer strings, a temporary CLOB must be created using *Connection.createlob()* and bound as shown:

```
clob = connection.createlob(cx_Oracle.DB_TYPE_CLOB)
clob.write(xmlData)
cursor.execute("insert into xml_table values (:id, sys.xmltype(:xml))",
        id=2, xml=clob)
```

Fetching XML data can be done simply for values that are shorter than the length of a VARCHAR2 column, as shown:

```
cursor.execute("select xml_data from xml_table where id = :id", id=1)
xmlData, = cursor.fetchone()
print(xmlData)           # will print the string that was originally stored
```

For values that exceed the length of a VARCHAR2 column, a CLOB must be returned instead by using the function `XMLTYPE.GETCLOBVAL()` as shown:

```
cursor.execute("""
        select xmltype.getclobval(xml_data)
        from xml_table
        where id = :id""", id=1)
clob, = cursor.fetchone()
print(clob.read())
```

The LOB that is returned can be streamed or a string can be returned instead of a CLOB. See *Using CLOB and BLOB Data* for more information about processing LOBs.

1.12 Batch Statement Execution and Bulk Loading

Inserting or updating multiple rows can be performed efficiently with *Cursor.executemany()*, making it easy to work with large data sets with cx_Oracle. This method can significantly outperform repeated calls to *Cursor.execute()* by reducing network transfer costs and database overheads. The *executemany()* method can also be used to execute PL/SQL statements multiple times at once.

There are examples in the GitHub examples directory.

The following tables will be used in the samples that follow:

```
create table ParentTable (
    ParentId              number(9) not null,
    Description           varchar2(60) not null,
    constraint ParentTable_pk primary key (ParentId)
);

create table ChildTable (
```

(continues on next page)

(continued from previous page)

```
    ChildId                 number(9) not null,
    ParentId                number(9) not null,
    Description             varchar2(60) not null,
    constraint ChildTable_pk primary key (ChildId),
    constraint ChildTable_fk foreign key (ParentId)
            references ParentTable
);
```

1.12.1 Batch Execution of SQL

The following example inserts five rows into the table `ParentTable`:

```
dataToInsert = [
    (10, 'Parent 10'),
    (20, 'Parent 20'),
    (30, 'Parent 30'),
    (40, 'Parent 40'),
    (50, 'Parent 50')
]
cursor.executemany("insert into ParentTable values (:1, :2)", dataToInsert)
```

This code requires only one *round-trip* from the client to the database instead of the five round-trips that would be required for repeated calls to *execute()*. For very large data sets there may be an external buffer or network limits to how many rows can be processed, so repeated calls to `executemany()` may be required. The limits are based on both the number of rows being processed as well as the "size" of each row that is being processed. Repeated calls to *executemany()* are still better than repeated calls to *execute()*.

1.12.2 Batch Execution of PL/SQL

PL/SQL functions and procedures and anonymous PL/SQL blocks can also be called using *executemany()* in order to improve performance. For example:

```
dataToInsert = [
    (10, 'Parent 10'),
    (20, 'Parent 20'),
    (30, 'Parent 30'),
    (40, 'Parent 40'),
    (50, 'Parent 50')
]
cursor.executemany("begin mypkg.create_parent(:1, :2); end;", dataToInsert)
```

Note that the `batcherrors` parameter (discussed below) cannot be used with PL/SQL block execution.

1.12.3 Handling Data Errors

Large datasets may contain some invalid data. When using batch execution as discussed above, the entire batch will be discarded if a single error is detected, potentially eliminating the performance benefits of batch execution and increasing the complexity of the code required to handle those errors. If the parameter `batchErrors` is set to the value `True` when calling *executemany()*, however, processing will continue even if there are data errors in some rows, and the rows containing errors can be examined afterwards to determine what course the application should take. Note that if any errors are detected, a transaction will be started but not committed, even if *Connection.autocommit* is set to `True`. After examining the errors and deciding what to do with them, the application needs to

explicitly commit or roll back the transaction with *Connection.commit()* or *Connection.rollback()*, as needed.

This example shows how data errors can be identified:

```
dataToInsert = [
    (60, 'Parent 60'),
    (70, 'Parent 70'),
    (70, 'Parent 70 (duplicate)'),
    (80, 'Parent 80'),
    (80, 'Parent 80 (duplicate)'),
    (90, 'Parent 90')
]
cursor.executemany("insert into ParentTable values (:1, :2)", dataToInsert,
        batcherrors=True)
for error in cursor.getbatcherrors():
    print("Error", error.message, "at row offset", error.offset)
```

The output is:

```
Error ORA-00001: unique constraint (PYTHONDEMO.PARENTTABLE_PK) violated at row offset␣
↪2
Error ORA-00001: unique constraint (PYTHONDEMO.PARENTTABLE_PK) violated at row offset␣
↪4
```

The row offset is the index into the array of the data that could not be inserted due to errors. The application could choose to commit or rollback the other rows that were successfully inserted. Alternatively, it could correct the data for the two invalid rows and attempt to insert them again before committing.

1.12.4 Identifying Affected Rows

When executing a DML statement using *execute()*, the number of rows affected can be examined by looking at the attribute *rowcount*. When performing batch executing with *Cursor.executemany()*, however, the row count will return the *total* number of rows that were affected. If you want to know the total number of rows affected by each row of data that is bound you must set the parameter `arraydmlrowcounts` to `True`, as shown:

```
parentIdsToDelete = [20, 30, 50]
cursor.executemany("delete from ChildTable where ParentId = :1",
        [(i,) for i in parentIdsToDelete],
        arraydmlrowcounts=True)
rowCounts = cursor.getarraydmlrowcounts()
for parentId, count in zip(parentIdsToDelete, rowCounts):
    print("Parent ID:", parentId, "deleted", count, "rows.")
```

Using the data found in the GitHub samples the output is as follows:

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

1.12.5 DML RETURNING

DML statements like INSERT, UPDATE, DELETE and MERGE can return values by using the DML RETURNING syntax. A bind variable can be created to accept this data. See *Using Bind Variables* for more information.

If, instead of merely deleting the rows as shown in the previous example, you also wanted to know some information about each of the rows that were deleted, you could use the following code:

```
parentIdsToDelete = [20, 30, 50]
childIdVar = cursor.var(int, arraysize=len(parentIdsToDelete))
cursor.setinputsizes(None, childIdVar)
cursor.executemany("""
        delete from ChildTable
        where ParentId = :1
        returning ChildId into :2""",
        [(i,) for i in parentIdsToDelete])
for ix, parentId in enumerate(parentIdsToDelete):
    print("Child IDs deleted for parent ID", parentId, "are",
            childIdVar.getvalue(ix))
```

The output would then be:

```
Child IDs deleted for parent ID 20 are [1002, 1003, 1004]
Child IDs deleted for parent ID 30 are [1005, 1006]
Child IDs deleted for parent ID 50 are [1012, 1013, 1014, 1015]
```

Note that the bind variable created to accept the returned data must have an arraysize large enough to hold data for each row that is processed. Also, the call to *Cursor.setinputsizes()* binds this variable immediately so that it does not have to be passed in each row of data.

1.12.6 Predefining Memory Areas

When multiple rows of data are being processed there is the possibility that the data is not uniform in type and size. In such cases, cx_Oracle makes some effort to accommodate such differences. Type determination for each column is deferred until a value that is not `None` is found in the column's data. If all values in a particular column are `None`, then cx_Oracle assumes the type is a string and has a length of 1. cx_Oracle will also adjust the size of the buffers used to store strings and bytes when a longer value is encountered in the data. These sorts of operations incur overhead as memory has to be reallocated and data copied. To eliminate this overhead, using *setinputsizes()* tells cx_Oracle about the type and size of the data that is going to be used.

Consider the following code:

```
data = [
    (    110, "Parent 110"),
    (   2000, "Parent 2000"),
    (  30000, "Parent 30000"),
    ( 400000, "Parent 400000"),
    (5000000, "Parent 5000000")
]
cursor.setinputsizes(None, 20)
cursor.executemany("""
        insert into ParentTable (ParentId, Description)
        values (:1, :2)""", data)
```

In this example, without the call to *setinputsizes()*, cx_Oracle would perform five allocations of increasing size as it discovered each new, longer string. However `cursor.setinputsizes(None, 20)` tells cx_Oracle that the maximum size of the strings that will be processed is 20 characters. Since cx_Oracle allocates memory for each row based on this value, it is best not to oversize it. The first parameter of `None` tells cx_Oracle that its default processing will be sufficient.

1.12.7 Loading CSV Files into Oracle Database

The *Cursor.executemany()* method and csv module can be used to efficiently load CSV (Comma Separated Values) files. For example, consider the file `data.csv`:

```
101,Abel
154,Baker
132,Charlie
199,Delta
. . .
```

And the schema:

```
create table test (id number, name varchar2(25));
```

Instead of looping through each line of the CSV file and inserting it individually, you can insert batches of records using *Cursor.executemany()*:

```
import cx_Oracle
import csv

. . .

# Predefine the memory areas to match the table definition
cursor.setinputsizes(None, 25)

# Adjust the batch size to meet your memory and performance requirements
batch_size = 10000

with open('testsp.csv', 'r') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    sql = "insert into test (id,name) values (:1, :2)"
    data = []
    for line in csv_reader:
        data.append((line[0], line[1]))
        if len(data) % batch_size == 0:
            cursor.executemany(sql, data)
            data = []
    if data:
        cursor.executemany(sql, data)
    con.commit()
```

1.13 Exception Handling

All exceptions raised by cx_Oracle are inherited from *cx_Oracle.Error*. See *Exceptions* for more details on the various exceptions defined by cx_Oracle. See the exception handling section in the *API manual* for more details on the information available when an exception is raised.

Applications can catch exceptions as needed. For example, when trying to add a customer that already exists in the database, the following could could be used to catch the exception:

```
try:
    cursor.execute("insert into customer values (101, 'Customer A')")
except cx_Oracle.IntegrityError:
    print("Customer ID already exists")
```

(continues on next page)

(continued from previous page)

```
else:
    print("Customer added")
```

If information about the exception needs to be processed instead, the following code can be used:

```
try:
    cursor.execute("insert into customer values (101, 'Customer A')")
except cx_Oracle.IntegrityError as e:
    errorObj, = e.args
    print("Customer ID already exists")
    print("Error Code:", errorObj.code)
    print("Error Message:", errorObj.message)
else:
    print("Customer added")
```

1.14 Oracle Advanced Queuing (AQ)

Oracle Advanced Queuing is a highly configurable and scalable messaging feature of Oracle Database. It has interfaces in various languages, letting you integrate multiple tools in your architecture.

cx_Oracle 7.2 introduced an updated interface for Oracle Advanced Queuing.

There are Advanced Queuing examples in the GitHub examples directory.

1.14.1 Creating a Queue

Before being used, queues need to be created in the database, for example in SQL*Plus:

```
begin
    dbms_aqadm.create_queue_table('MY_QUEUE_TABLE', 'RAW');
    dbms_aqadm.create_queue('DEMO_RAW_QUEUE', 'MY_QUEUE_TABLE');
    dbms_aqadm.start_queue('DEMO_RAW_QUEUE');
end;
/
```

This examples creates a RAW queue suitable for sending string or raw bytes messages.

1.14.2 Enqueuing Messages

To send messages in Python you connect and get a *queue*. The queue can be used for enqueuing, dequeuing, or both as needed.

```
queue = connection.queue("DEMO_RAW_QUEUE")
```

Now messages can be queued using *Queue.enqOne()*. To send three messages:

```
PAYLOAD_DATA = [
    "The first message",
    "The second message",
    "The third message"
]
for data in PAYLOAD_DATA:
```

(continues on next page)

(continued from previous page)

```
    queue.enqOne(connection.msgproperties(payload=data))
connection.commit()
```

Since the queue sending the messages is a RAW queue, the strings in this example will be internally encoded to bytes using *Connection.encoding* before being enqueued.

1.14.3 Dequeuing Messages

Dequeuing is performed similarly. To dequeue a message call the method *Queue.deqOne()* as shown. Note that if the message is expected to be a string, the bytes must be decoded using *Connection.encoding*.

```
queue = connection.queue("DEMO_RAW_QUEUE")
msg = queue.deqOne()
connection.commit()
print(msg.payload.decode(connection.encoding))
```

1.14.4 Using Object Queues

Named Oracle objects can be enqueued and dequeued as well. Given an object type called UDT_BOOK:

```
CREATE OR REPLACE TYPE udt_book AS OBJECT (
    Title   VARCHAR2(100),
    Authors VARCHAR2(100),
    Price   NUMBER(5,2)
);
/
```

And a queue that accepts this type:

```
begin
    dbms_aqadm.create_queue_table('BOOK_QUEUE_TAB', 'UDT_BOOK');
    dbms_aqadm.create_queue('DEMO_BOOK_QUEUE', 'BOOK_QUEUE_TAB');
    dbms_aqadm.start_queue('DEMO_BOOK_QUEUE');
end;
/
```

You can queue messages:

```
booksType = connection.gettype("UDT_BOOK")
queue = connection.queue("DEMO_BOOK_QUEUE", booksType)

book = booksType.newobject()
book.TITLE = "Quick Brown Fox"
book.AUTHORS = "The Dog"
book.PRICE = 123

queue.enqOne(connection.msgproperties(payload=book))
connection.commit()
```

Dequeuing is done like this:

```
booksType = connection.gettype("UDT_BOOK")
queue = connection.queue("DEMO_BOOK_QUEUE", booksType)
```

(continues on next page)

(continued from previous page)

```
msg = queue.deqOne()
connection.commit()
print(msg.payload.TITLE)        # will print Quick Brown Fox
```

1.14.5 Changing Queue and Message Options

Refer to the *cx_Oracle AQ API* and Oracle Advanced Queuing documentation for details on all of the enqueue and dequeue options available.

Enqueue options can be set. For example, to make it so that an explicit call to *commit()* on the connection is not needed to commit messages:

```
queue = connection.queue("DEMO_RAW_QUEUE")
queue.enqOptions.visibility = cx_Oracle.ENQ_IMMEDIATE
```

Dequeue options can also be set. For example, to specify not to block on dequeuing if no messages are available:

```
queue = connection.queue("DEMO_RAW_QUEUE")
queue.deqOptions.wait = cx_Oracle.DEQ_NO_WAIT
```

Message properties can be set when enqueuing. For example, to set an expiration of 60 seconds on a message:

```
queue.enqOne(connection.msgproperties(payload="Message", expiration=60))
```

This means that if no dequeue operation occurs within 60 seconds that the message will be dropped from the queue.

1.14.6 Bulk Enqueue and Dequeue

The *Queue.enqMany()* and *Queue.deqMany()* methods can be used for efficient bulk message handling.

Queue.enqMany() is similar to *Queue.enqOne()* but accepts an array of messages:

```
messages = [
    "The first message",
    "The second message",
    "The third message",
]
queue = connection.queue("DEMO_RAW_QUEUE")
queue.enqMany(connection.msgproperties(payload=m) for m in messages)
connection.commit()
```

Warning: calling *Queue.enqMany()* in parallel on different connections acquired from the same pool may fail due to Oracle bug 29928074. Ensure that this function is not run in parallel, use standalone connections or connections from different pools, or make multiple calls to *Queue.enqOne()* instead. The function *Queue.deqMany()* call is not affected.

To dequeue multiple messages at one time, use *Queue.deqMany()*. This takes an argument specifying the maximum number of messages to dequeue at one time:

```
for m in queue.deqMany(maxMessages=10):
    print(m.payload.decode(connection.encoding))
```

Depending on the queue properties and the number of messages available to dequeue, this code will print out from zero to ten messages.

1.15 Continuous Query Notification (CQN)

Continuous Query Notification (CQN) allows applications to receive notifications when a table changes, such as when rows have been updated, regardless of the user or the application that made the change. This can be useful in many circumstances, such as near real-time monitoring, auditing applications, or for such purposes as mid-tier cache invalidation. A cache might hold some values that depend on data in a table. If the data in the table changes, the cached values must then be updated with the new information.

CQN notification behavior is widely configurable. Choices include specifying what types of SQL should trigger a notification, whether notifications should survive database loss, and control over unsubscription. You can also choose whether notification messages will include ROWIDs of affected rows.

By default, object-level (previously known as Database Change Notification) occurs and the Python notification method is invoked whenever a database transaction is committed that changes an object that a registered query references, regardless of whether the actual query result changed. However if the *subscription* option `qos` is *cx_Oracle.SUBSCR_QOS_QUERY* then query-level notification occurs. In this mode, the database notifies the application whenever a transaction changing the result of the registered query is committed.

CQN is best used to track infrequent data changes.

1.15.1 Requirements

Before using CQN, users must have appropriate permissions:

```
GRANT CHANGE NOTIFICATION TO <user-name>;
```

To use CQN, connections must have `events` mode set to `True`, for example:

```
connection = cx_Oracle.connect(userName, password, "dbhost.example.com/orclpdb1",␣
↪events=True)
```

The default CQN connection mode means the database must be able to connect back to the application using cx_Oracle in order to receive notification events. Alternatively, when using Oracle Database and Oracle client libraries 19.4, or later, subscriptions can set the optional `clientInitiated` parameter to `True`, see `Connection.subscribe()` below.

The default CQN connection mode typically means that the machine running cx_Oracle needs a fixed IP address. Note *Connection.subscribe()* does not verify that this reverse connection is possible. If there is any problem sending a notification, then the callback method will not be invoked. Configuration options can include an IP address and port on which cx_Oracle will listen for notifications; otherwise, the database chooses values.

1.15.2 Creating a Subscription

Subscriptions allow Python to receives notifications for events that take place in the database that match the given parameters.

For example, a basic CQN subscription might be created like:

```
connection.subscribe(namespace=cx_Oracle.SUBSCR_NAMESPACE_DBCHANGE,
        callback=MyCallback)
```

See *Connection.subscribe()* for details on all of the parameters.

See *Operation Codes* for the types of operations that are supported.

See *Subscription Quality of Service* for the quality of service values that are supported.

See *Subscription Namespaces* and *Subscription Protocols* for the namespaces and protocols that are supported.

See *Subscription Object* for more details on the subscription object that is created.

When using Oracle Database and Oracle client libraries 19.4, or later, the optional subscription parameter `clientInitiated` can be set:

```
connection.subscribe(namespace= . . ., callback=MyCallback, clientInitiated=True)
```

This enables CQN "client initiated" connections which internally use the same approach as normal cx_Oracle connections to the database, and do not require the database to be able to connect back to the application. Since client initiated connections do not need special network configuration they have ease-of-use and security advantages.

1.15.3 Registering Queries

Once a subscription has been created, one or more queries must be registered by calling *Subscription.registerquery()*. Registering a query behaves similarly to *Cursor.execute()*, but only queries are permitted and the `args` parameter must be a sequence or dictionary.

An example script to receive query notifications when the 'CUSTOMER' table data changes is:

```
def CQNCallback(message):
    print("Notification:")
    for query in message.queries:
        for tab in query.tables:
            print("Table:", tab.name)
            print("Operation:", tab.operation)
            for row in tab.rows:
                if row.operation & cx_Oracle.OPCODE_INSERT:
                    print("INSERT of rowid:", row.rowid)
                if row.operation & cx_Oracle.OPCODE_DELETE:
                    print("DELETE of rowid:", row.rowid)

subscr = connection.subscribe(namespace=cx_Oracle.SUBSCR_NAMESPACE_DBCHANGE,
        callback=CQNCallback,
        operations=cx_Oracle.OPCODE_INSERT | cx_Oracle.OPCODE_DELETE,
        qos = cx_Oracle.SUBSCR_QOS_QUERY | cx_Oracle.SUBSCR_QOS_ROWIDS)
subscr.registerquery("select * from regions")
input("Hit enter to stop CQN demo\n")
```

Running the above script, shows the initial output as:

```
Hit enter to stop CQN demo
```

Use SQL*Plus or another tool to commit a change to the table:

```
insert into regions values(120, 'L');
commit;
```

When the commit is executed, a notification will be received by the callback which should print something like the following:

```
Hit enter to stop CQN demo
Notification:
Table: HR.REGIONS
Operation: 2
INSERT of rowid: AAA7EsAAHAAAFS/AAA
```

See GitHub Samples for a runnable CQN example.

1.16 Transaction Management

A database transaction is a grouping of SQL statements that make a logical data change to the database.

When *Cursor.execute()* executes a SQL statement, a transaction is started or continued. By default, cx_Oracle does not commit this transaction to the database. The methods *Connection.commit()* and *Connection.rollback()* methods can be used to explicitly commit or rollback a transaction:

```
cursor.execute("INSERT INTO mytab (name) VALUES ('John')")
connection.commit()
```

When a database connection is closed, such as with *Connection.close()*, or when variables referencing the connection go out of scope, any uncommitted transaction will be rolled back.

1.16.1 Autocommitting

An alternative way to commit is to set the attribute *autocommit* of the connection to `True`. This ensures all *DML* statements (INSERT, UPDATE etc) are committed as they are executed. Unlike *Connection.commit()*, this does not require an additional *round-trip* to the database so it is more efficient when used appropriately.

Note that irrespective of the autocommit value, Oracle Database will always commit an open transaction when a DDL statement is executed.

When executing multiple DML statements that constitute a single transaction, it is recommended to use autocommit mode only for the last DML statement in the sequence of operations. Unnecessarily committing causes extra database load, and can destroy transactional consistency.

The example below shows a new customer being added to the table `CUST_TABLE`. The corresponding `SALES` table is updated with a purchase of 3000 pens from the customer. The final insert uses autocommit mode to commit both new records:

```
# Add a new customer
idVar = cursor.var(int)
connection.autocommit = False  # make sure any previous value is off
cursor.execute("""
        INSERT INTO cust_table (name) VALUES ('John')
        RETURNING id INTO :bvid""", bvid=idVar)

# Add sales data for the new customer and commit all new values
idVal = idVar.getvalue()[0]
connection.autocommit = True
cursor.execute("INSERT INTO sales_table VALUES (:bvid, 'pens', 3000)",
        bvid=idVal)
```

1.16.2 Explicit Transactions

The method *Connection.begin()* can be used to explicitly start a local or global transaction.

Without parameters, this explicitly begins a local transaction; otherwise, this explicitly begins a distributed (global) transaction with the given parameters. See the Oracle documentation for more details.

Note that in order to make use of global (distributed) transactions, the attributes *Connection.internal_name* and *Connection.external_name* attributes must be set.

1.17 Tuning cx_Oracle

Some general tuning tips are:

- Tune your application architecture.

  A general application goal is to reduce the number of *round-trips* between cx_Oracle and the database.

  For multi-user applications, make use of connection pooling. Create the pool once during application initialization. Do not oversize the pool, see *Connection Pooling* . Use a session callback function to set session state, see *Session CallBacks for Setting Pooled Connection State*.

  Make use of efficient cx_Oracle functions. For example, to insert multiple rows use `Cursor.executemany()` instead of `Cursor.execute()`.

- Tune your SQL statements. See the SQL Tuning Guide.

  Use *bind variables* to avoid statement reparsing.

  Tune `Cursor.arraysize` and `Cursor.prefetchrows` for each query, see *Tuning Fetch Performance*.

  Do simple optimizations like *limiting the number of rows* and avoiding selecting columns not used in the application.

  It may be faster to work with simple scalar relational values than to use Oracle Database object types.

  Make good use of PL/SQL to avoid executing many individual statements from cx_Oracle.

  Tune the *Statement Cache*.

  Enable *Client Result Caching* for small lookup tables.

- Tune your database. See the Database Performance Tuning Guide.

- Tune your network. For example, when inserting or retrieving a large number of rows (or for large data), or when using a slow network, then tune the Oracle Network Session Data Unit (SDU) and socket buffer sizes, see Oracle Net Services: Best Practices for Database Performance and High Availability.

- Do not commit or rollback unnecessarily. Use `Connection.autocommit` on the last of a sequence of DML statements.

1.17.1 Tuning Fetch Performance

To tune queries you can adjust cx_Oracle's internal buffer sizes to improve the speed of fetching rows across the network from the database, and to optimize memory usage. Regardless of which cx_Oracle method is used to get query results, internally all rows are fetched in batches from the database and buffered before being returned to the application. The internal buffer sizes can have a significant performance impact. The sizes do not affect how, or when, rows are returned to your application. They do not affect the minimum or maximum number of rows returned by a query.

For best performance, tune "array fetching" with `Cursor.arraysize` and "row prefetching" with `Cursor.prefetchrows` before calling `Cursor.execute()`. Queries that return LOBs and similar types will never prefetch rows, so the `prefetchrows` value is ignored in those cases.

The common query tuning scenario is for SELECT statements that return a large number of rows over a slow network. Increasing `arraysize` can improve performance by reducing the number of *round-trips* to the database. However increasing this value increases the amount of memory required. Adjusting `prefetchrows` will also affect performance and memory usage.

Row prefetching and array fetching are both internal buffering techniques to reduce *round-trips* to the database. The difference is the code layer that is doing the buffering, and when the buffering occurs. The Oracle Client libraries used

by cx_Oracle have separate "execute SQL statement" and "fetch data" calls. Prefetching allows query results to be returned to the application when the successful statement execution acknowledgment is returned from the database. This means that a subsequent internal "fetch data" operation does not always need to make a round-trip to the database because rows are already buffered in the Oracle Client libraries. Reducing round-trips helps performance and scalability. An overhead of prefetching is the need for an additional data copy from Oracle Client's prefetch buffers.

To tune queries that return an unknown number of rows, estimate the number of rows returned and start with an appropriate *Cursor.arraysize* value. The default is 100. Then set *Cursor.prefetchrows* to the `arraysize` value. Do not make the sizes unnecessarily large. Keep `arraysize` as big, or bigger than, `prefetchrows`. Adjust the values as needed for performance, memory and round-trip usage. An example is:

```
cur = connection.cursor()

cur.prefetchrows = 1000
cur.arraysize = 1000

for row in cur.execute("SELECT * FROM very_big_table"):
    print(row)
```

For a large quantity of rows or very "wide" rows on fast networks you may prefer to leave `prefetchrows` at its default value of 2. The documentation in *Database Round-trips* shows how to measure round-trips.

If you are fetching a fixed number of rows, start your tuning by setting `arraysize` to the number of expected rows, and set `prefetchrows` to one greater than this value. (Adding one removes the need for a round-trip to check for end-of-fetch). For example, if you are querying 20 rows, perhaps to *display a page* of data, set `prefetchrows` to 21 and `arraysize` to 20:

```
cur = connection.cursor()

cur.prefetchrows = 21
cur.arraysize = 20

for row in cur.execute("""
    SELECT last_name
       FROM employees
       ORDER BY last_name
       OFFSET 0 ROWS FETCH NEXT 20 ROWS ONLY"""):
    print(row)
```

This will return all rows for the query in one round-trip.

If you know that a query returns just one row then set *Cursor.arraysize* to 1 to minimize memory usage. The default prefetch value of 2 allows minimal round-trips for single-row queries:

```
cur = connection.cursor()
cur.arraysize = 1
cur.execute("select * from MyTable where id = 1"):
row = cur.fetchone()
print(row)
```

The best *Cursor.arraysize* and *Cursor.prefetchrows* values can be found by experimenting with your application under the expected load of normal application use. This is because the cost of the extra memory copy from the prefetch buffers when fetching a large quantity of rows or very "wide" rows may outweigh the cost of a round-trip for a single cx_Oracle user on a fast network. However under production application load, the reduction of round-trips may help performance and overall system scalability.

Prefetching can also be enabled in an external *oraaccess.xml* file, which may be useful for tuning an application when modifying its code is not feasible. Setting the size in `oraaccess.xml` will affect the whole application, so it should not be the first tuning choice.

One place where increasing `arraysize` is particularly useful is in copying data from one database to another:

```
# setup cursors
sourceCursor = sourceConnection.cursor()
sourceCursor.arraysize = 1000
targetCursor = targetConnection.cursor()

# perform fetch and bulk insertion
sourceCursor.execute("select * from MyTable")
while True:
    rows = sourceCursor.fetchmany()
    if not rows:
        break
    targetCursor.executemany("insert into MyTable values (:1, :2)", rows)
    targetConnection.commit()
```

In cx_Oracle, the `arraysize` and `prefetchrows` values are only examined when a statement is executed the first time. To change the values, create a new cursor. For example, to change `arraysize` for a repeated statement:

```
array_sizes = (10, 100, 1000)
for size in array_sizes:
    cursor = connection.cursor()
    cursor.arraysize = size
    start = time.time()
    cursor.execute(sql).fetchall()
    elapsed = time.time() - start
    print("Time for", size, elapsed, "seconds")
```

There are two cases that will benefit from setting *`Cursor.prefetchrows`* to 0:

- When passing REF CURSORS into PL/SQL packages. Setting `prefetchrows` to 0 can stop rows being prematurely (and silently) fetched into cx_Oracle's internal buffers, making them unavailable to the PL/SQL code that receives the REF CURSOR.
- When querying a PL/SQL function that uses PIPE ROW to emit rows at intermittent intervals. By default, several rows needs to be emitted by the function before cx_Oracle can return them to the application. Setting `prefetchrows` to 0 helps give a consistent flow of data to the application.

1.17.2 Database Round-trips

A round-trip is defined as the trip from the Oracle Client libraries (used by cx_Oracle) to the database and back. Calling each cx_Oracle function, or accessing each attribute, will require zero or more round-trips. Along with tuning an application's architecture and tuning its SQL statements, a general performance and scalability goal is to minimize round-trips.

Some general tips for reducing round-trips are:

- Tune *`Cursor.arraysize`* and *`Cursor.prefetchrows`* for each query.
- Use *`Cursor.executemany()`* for optimal DML execution.
- Only commit when necessary. Use *`Connection.autocommit`* on the last statement of a transaction.
- For connection pools, use a callback to set connection state, see *Session CallBacks for Setting Pooled Connection State*.
- Make use of PL/SQL procedures which execute multiple SQL statements instead of executing them individually from cx_Oracle.
- Use scalar types instead of Oracle Database object types.

- Avoid overuse of *Connection.ping()*.

Finding the Number of Round-Trips

Oracle's Automatic Workload Repository (AWR) reports show 'SQL*Net roundtrips to/from client' and are useful for finding the overall behavior of a system.

Sometimes you may wish to find the number of round-trips used for a specific application. Snapshots of the V$SESSTAT view taken before and after doing some work can be used for this:

```
SELECT ss.value, sn.display_name
FROM v$sesstat ss, v$statname sn
WHERE ss.sid = SYS_CONTEXT('USERENV','SID')
AND ss.statistic# = sn.statistic#
AND sn.name LIKE '%roundtrip%client%';
```

1.17.3 Statement Caching

cx_Oracle's *Cursor.execute()* and *Cursor.executemany()* functions use the Oracle Call Interface statement cache to make re-execution of statements efficient. Each standalone or pooled connection has its own cache of statements with a default size of 20. Statement caching lets cursors be used without re-parsing the statement. Statement caching also reduces metadata transfer costs between the cx_Oracle and the database. Performance and scalability are improved.

The statement cache size can be set with *Connection.stmtcachesize* or *SessionPool.stmtcachesize*. In general, set the statement cache size to the size of the working set of statements being executed by the application. To manually tune the cache, monitor the general application load and the Automatic Workload Repository (AWR) "bytes sent via SQL*Net to client" values. The latter statistic should benefit from not shipping statement metadata to cx_Oracle. Adjust the statement cache size to your satisfaction.

Statement caching can be disabled by setting the size to 0. Disabling the cache may be beneficial when the quantity or order of statements causes cache entries to be flushed before they get a chance to be reused. For example if there are more distinct statements than cache slots, and the order of statement execution causes older statements to be flushed from the cache before the statements are re-executed.

With Oracle Database 12c, or later, the statement cache size can be automatically tuned using the *oraaccess.xml* file.

When it is inconvenient to pass statement text through an application, the *Cursor.prepare()* call can be used to avoid statement re-parsing. Subsequent `execute()` calls use the value `None` instead of the SQL text:

```
cur.prepare("select * from dept where deptno = :id order by deptno")

cur.execute(None, id = 20)
res = cur.fetchall()
print(res)

cur.execute(None, id = 10)
res = cur.fetchall()
print(res)
```

Statements passed to *prepare()* are also stored in the statement cache.

1.17.4 Client Result Cache

cx_Oracle applications can use Oracle Database's Client Result Cache. The CRC enables client-side caching of SQL query (SELECT statement) results in client memory for immediate use when the same query is re-executed. This is useful for reducing the cost of queries for small, mostly static, lookup tables, such as for postal codes. CRC reduces network *round-trips*, and also reduces database server CPU usage.

The cache is at the application process level. Access and invalidation is managed by the Oracle Client libraries. This removes the need for extra application logic, or external utilities, to implement a cache.

CRC can be enabled by setting the database parameters `CLIENT_RESULT_CACHE_SIZE` and `CLIENT_RESULT_CACHE_LAG`, and then restarting the database, for example:

```
SQL> ALTER SYSTEM SET CLIENT_RESULT_CACHE_LAG = 3000 SCOPE=SPFILE;
SQL> ALTER SYSTEM SET CLIENT_RESULT_CACHE_SIZE = 64K SCOPE=SPFILE;
SQL> STARTUP FORCE
```

CRC can alternatively be configured in an *oraaccess.xml* or *sqlnet.ora* file on the Node.js host, see Client Configuration Parameters.

Tables can then be created, or altered, so repeated queries use CRC. This allows existing applications to use CRC without needing modification. For example:

```
SQL> CREATE TABLE cities (id number, name varchar2(40)) RESULT_CACHE (MODE FORCE);
SQL> ALTER TABLE locations RESULT_CACHE (MODE FORCE);
```

Alternatively, hints can be used in SQL statements. For example:

```
SELECT /*+ result_cache */ postal_code FROM locations
```

1.18 Character Sets and Globalization

Data fetched from, and sent to, Oracle Database will be mapped between the database character set and the "Oracle client" character set of the Oracle Client libraries used by cx_Oracle. If data cannot be correctly mapped between client and server character sets, then it may be corrupted or queries may fail with *"codec can't decode byte"*.

cx_Oracle uses Oracle's National Language Support (NLS) to assist in globalizing applications. As well as character set support, there are many other features that will be useful in applications. See the Database Globalization Support Guide.

1.18.1 Setting the Client Character Set

In cx_Oracle 8 the default encoding used for all character data changed to "UTF-8". This universal encoding is suitable for most applications. If you have a special need, you can pass the `encoding` and `nencoding` parameters to the *`cx_Oracle.connect()`* and *`cx_Oracle.SessionPool()`* methods to specify different Oracle Client character sets. For example:

```
import cx_Oracle
connection = cx_Oracle.connect(connectString, encoding="US-ASCII",
        nencoding="UTF-8")
```

The `encoding` parameter affects character data such as VARCHAR2 and CLOB columns. The `nencoding` parameter affects "National Character" data such as NVARCHAR2 and NCLOB. If you are not using national character types, then you can omit `nencoding`. Both the `encoding` and `nencoding` parameters are expected to be one of

the Python standard encodings such as UTF-8. Do not accidentally use UTF8, which Oracle uses to specify the older Unicode 3.0 Universal character set, CESU-8. Note that Oracle does not recognize all of the encodings that Python recognizes. You can see which encodings are usable in cx_Oracle by issuing this query:

```
select distinct utl_i18n.map_charset(value)
from v$nls_valid_values
where parameter = 'CHARACTERSET'
  and utl_i18n.map_charset(value) is not null
order by 1
```

Note: From cx_Oracle 8, it is no longer possible to change the character set using the NLS_LANG environment variable. The character set component of that variable is ignored. The language and territory components of NLS_LANG are still respected by the Oracle Client libraries.

Character Set Example

The script below tries to display data containing a Euro symbol from the database.

```
connection = cx_Oracle.connect(userName, password, "dbhost.example.com/orclpdb1",
        encoding="US-ASCII")
cursor = connection.cursor()
for row in cursor.execute("select nvarchar2_column from nchar_test"):
    print(row)
```

Because the '€' symbol is not supported by the US-ASCII character set, all '€' characters are replaced by '¿' in the cx_Oracle output:

```
('¿',)
```

When the encoding parameter is removed (or set to "UTF-8") during connection:

```
connection = cx_Oracle.connect(userName, password, "dbhost.example.com/orclpdb1")
```

Then the output displays the Euro symbol as desired:

```
('€',)
```

Finding the Database and Client Character Set

To find the database character set, execute the query:

```
SELECT value AS db_charset
FROM nls_database_parameters
WHERE parameter = 'NLS_CHARACTERSET';
```

To find the database 'national character set' used for NCHAR and related types, execute the query:

```
SELECT value AS db_ncharset
FROM nls_database_parameters
WHERE parameter = 'NLS_NCHAR_CHARACTERSET';
```

To find the current "client" character set used by cx_Oracle, execute the query:

```
SELECT DISTINCT client_charset AS client_charset
FROM v$session_connect_info
WHERE sid = SYS_CONTEXT('USERENV', 'SID');
```

If these character sets do not match, characters transferred over Oracle Net will be mapped from one character set to another. This may impact performance and may result in invalid data.

1.18.2 Setting the Oracle Client Locale

You can use the NLS_LANG environment variable to set the language and territory used by the Oracle Client libraries. For example, on Linux you could set:

```
export NLS_LANG=JAPANESE_JAPAN
```

The language ("JAPANESE" in this example) specifies conventions such as the language used for Oracle Database messages, sorting, day names, and month names. The territory ("JAPAN") specifies conventions such as the default date, monetary, and numeric formats. If the language is not specified, then the value defaults to AMERICAN. If the territory is not specified, then the value is derived from the language value. See Choosing a Locale with the NLS_LANG Environment Variable

If the NLS_LANG environment variable is set in the application with os.environ['NLS_LANG'], it must be set before any connection pool is created, or before any standalone connections are created.

Other Oracle globalization variables, such as NLS_DATE_FORMAT can also be set to change the behavior of cx_Oracle, see Setting NLS Parameters.

1.19 Starting and Stopping Oracle Database

This chapter covers how to start up and shutdown Oracle Database using cx_Oracle.

1.19.1 Starting Oracle Database Up

cx_Oracle can start up a database instance. A privileged connection is required. This example shows a script that could be run as the 'oracle' operating system user who administers a local database installation on Linux. It assumes that the environment variable ORACLE_SID has been set to the SID of the database that should be started:

```
# the connection must be in PRELIM_AUTH mode to perform startup
connection = cx_Oracle.connect("/",
        mode = cx_Oracle.SYSDBA | cx_Oracle.PRELIM_AUTH)
connection.startup()

# the following statements must be issued in normal SYSDBA mode
connection = cx_Oracle.connect("/", mode = cx_Oracle.SYSDBA, encoding="UTF-8")
cursor = connection.cursor()
cursor.execute("alter database mount")
cursor.execute("alter database open")
```

To start up a remote database, you may need to configure the Oracle Net listener to use static service registration by adding a SID_LIST_LISTENER entry to the database *listener.ora* file.

1.19.2 Shutting Oracle Database Down

cx_Oracle has the ability to shutdown the database using a privileged connection. This example also assumes that the environment variable ORACLE_SID has been set:

```
# need to connect as SYSDBA or SYSOPER
connection = cx_Oracle.connect("/", mode = cx_Oracle.SYSDBA)

# first shutdown() call must specify the mode, if DBSHUTDOWN_ABORT is used,
# there is no need for any of the other steps
connection.shutdown(mode = cx_Oracle.DBSHUTDOWN_IMMEDIATE)

# now close and dismount the database
cursor = connection.cursor()
cursor.execute("alter database close normal")
cursor.execute("alter database dismount")

# perform the final shutdown call
connection.shutdown(mode = cx_Oracle.DBSHUTDOWN_FINAL)
```

1.20 High Availability with cx_Oracle

Applications can utilize many features for high availability (HA) during planned and unplanned outages in order to:

- Reduce application downtime
- Eliminate compromises between high availability and performance
- Increase operational productivity

1.20.1 General HA Recommendations

General recommendations for creating highly available cx_Oracle programs are:

- Tune operating system and Oracle Network parameters to avoid long TCP timeouts, to prevent firewalls killing connections, and to avoid connection storms.
- Implement application error handling and recovery.
- Use the most recent version of the Oracle client libraries. New versions have improvements to features such as dead database server detection, and make it easier to set connection options.
- Use the most recent version of Oracle Database. New database versions introduce, and enhance, features such as Application Continuity (AC) and Transparent Application Continuity (TAC).
- Utilize Oracle Database technologies such as RAC or standby databases.
- Configure database services to emit *FAN* events.
- Use a *connection pool*, because pools can handle database events and take proactive and corrective action for draining, run time load balancing, and fail over. Set the minimum and maximum pool sizes to the same values to avoid connection storms. Remove resource manager or user profiles that prematurely close sessions.
- Test all scenarios thoroughly.

1.20.2 Network Configuration

The operating system TCP and *Oracle Net configuration* should be configured for performance and availability.

Options such as SQLNET.OUTBOUND_CONNECT_TIMEOUT, SQLNET.RECV_TIMEOUT and SQLNET.SEND_TIMEOUT can be explored.

Oracle Net Services options may also be useful for high availability and performance tuning. For example the database's *listener.ora* file can have RATE_LIMIT and QUEUESIZE parameters that can help handle connection storms.

With Oracle Client 19c, EXPIRE_TIME can be used in *tnsnames.ora* connect descriptors to prevent firewalls from terminating idle connections and to adjust keepalive timeouts. The general recommendation for `EXPIRE_TIME` is to use a value that is slightly less than half of the termination period. In older versions of Oracle Client, a `tnsnames.ora` connect descriptor option ENABLE=BROKEN can be used instead of `EXPIRE_TIME`. These settings can also aid detection of a terminated remote database server.

When cx_Oracle uses *Oracle Client libraries 19c*, then the *Easy Connect Plus syntax* syntax enables some options to be used without needing a `sqlnet.ora` file. For example, if your firewall times out every 4 minutes, and you cannot alter the firewall settings, then you may decide to use `EXPIRE_TIME` in your connect string to send a probe every 2 minutes to the database to keep connections 'alive':

```
connection = cx_Oracle.connect("hr", userpwd, "dbhost.example.com/orclpdb1?expire_
↪time=2")
```

1.20.3 Fast Application Notification (FAN)

Users of Oracle Database FAN must connect to a FAN-enabled database service. The application should have `events` set to True when connecting. This value can also be changed via *Oracle Client Configuration*.

FAN support is useful for planned and unplanned outages. It provides immediate notification to cx_Oracle following outages related to the database, computers, and networks. Without FAN, cx_Oracle can hang until a TCP timeout occurs and an error is returned, which might be several minutes.

FAN allows cx_Oracle to provide high availability features without the application being aware of an outage. Unused, idle connections in a *connection pool* will be automatically cleaned up. A future *`SessionPool.acquire()`* call will establish a fresh connection to a surviving database instance without the application being aware of any service disruption.

To handle errors that affect active connections, you can add application logic to re-connect (this will connect to a surviving database instance) and replay application logic without having to return an error to the application user.

FAN benefits users of Oracle Database's clustering technology Oracle RAC because connections to surviving database instances can be immediately made. Users of Oracle's Data Guard with a broker will get FAN events generated when the standby database goes online. Standalone databases will send FAN events when the database restarts.

For a more information on FAN see the white paper on Fast Application Notification.

1.20.4 Application Continuity (AC)

Oracle Application Continuity and Transparent Application Continuity are Oracle Database technologies that record application interaction with the database and, in the event of a database instance outage, attempt to replay the interaction on a surviving database instance. If successful, users will be unaware of any database issue.

When AC or TAC are configured on the database service, they are transparently available to cx_Oracle applications.

You must thoroughly test your application because not all lower level calls in the cx_Oracle implementation can be replayed.

See OCI and Application Continuity for more information.

1.20.5 Transaction Guard

cx_Oracle supports Transaction Guard which enables Python application to verify the success or failure of the last transaction in the event of an unplanned outage. This feature is available when both client and database are 12.1 or higher.

Using Transaction Guard helps to:

- Preserve the commit outcome
- Ensure a known outcome for every transaction

See Oracle Database Development Guide for more information about using Transaction Guard.

When an error occurs during commit, the Python application can acquire the logical transaction id (`ltxid`) from the connection and then call a procedure to determine the outcome of the commit for this logical transaction id.

Follow the steps below to use the Transaction Guard feature in Python:

1. Grant execute privileges to the database users who will be checking the outcome of the commit. Login as SYSDBA and run the following command:

```
GRANT EXECUTE ON DBMS_APP_CONT TO <username>;
```

2. Create a new service by executing the following PL/SQL block as SYSDBA. Replace the `<service-name>`, `<network-name>` and `<retention-value>` values with suitable values. It is important that the `COMMIT_OUTCOME` parameter be set to true for Transaction Guard to function properly.

```
DECLARE
    t_Params dbms_service.svc_parameter_array;
BEGIN
    t_Params('COMMIT_OUTCOME') := 'true';
    t_Params('RETENTION_TIMEOUT') := <retention-value>;
    DBMS_SERVICE.CREATE_SERVICE('<service-name>', '<network-name>', t_Params);
END;
/
```

3. Start the service by executing the following PL/SQL block as SYSDBA:

```
BEGIN
    DBMS_SERVICE.start_service('<service-name>');
END;
/
```

Ensure the service is running by examining the output of the following query:

```
SELECT name, network_name FROM V$ACTIVE_SERVICES ORDER BY 1;
```

Python Application code requirements to use Transaction Guard

In the Python application code:

- Use the connection attribute *ltxid* to determine the logical transaction id.
- Call the `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL procedure with the logical transaction id acquired from the connection attribute. This returns a boolean value indicating if the last transaction was committed and whether the last call was completed successfully or not.

See the Transaction Guard Sample for further details.

1.21 Tracing SQL and PL/SQL Statements

1.21.1 Subclass Connections

Subclassing enables applications to add "hooks" for connection and statement execution. This can be used to alter, or log, connection and execution parameters, and to extend cx_Oracle functionality.

The example below demonstrates subclassing a connection to log SQL execution to a file. This example also shows how connection credentials can be embedded in the custom subclass, so application code does not need to supply them.

```
class Connection(cx_Oracle.Connection):
    logFileName = "log.txt"

    def __init__(self):
        connectString = "hr/hr_password@dbhost.example.com/orclpdb1"
        self._log("Connect to the database")
        return super(Connection, self).__init__(connectString)

    def _log(self, message):
        with open(self.logFileName, "a") as f:
            print(message, file=f)

    def execute(self, sql, parameters):
        self._log(sql)
        cursor = self.cursor()
        try:
            return cursor.execute(sql, parameters)
        except cx_Oracle.Error as e:
            errorObj, = e.args
            self._log(errorObj.message)

connection = Connection()
connection.execute("""
        select department_name
        from departments
        where department_id = :id""", dict(id=270))
```

The messages logged in `log.txt` are:

```
Connect to the database

            select department_name
            from departments
            where department_id = :id
```

If an error occurs, perhaps due to a missing table, the log file would contain instead:

```
Connect to the database

            select department_name
            from departments
            where department_id = :id
ORA-00942: table or view does not exist
```

In production applications be careful not to log sensitive information.

See Subclassing.py for an example.

1.21.2 Oracle Database End-to-End Tracing

Oracle Database End-to-end application tracing simplifies diagnosing application code flow and performance problems in multi-tier or multi-user environments.

The connection attributes, *client_identifier*, *clientinfo*, *dbop*, *module* and *action*, set the metadata for end-to-end tracing. You can use data dictionary and V$ views to monitor tracing or use other application tracing utilities.

The attributes are sent to the database when the next *round-trip* to the database occurs, for example when the next SQL statement is executed.

The attribute values will remain set in connections released back to connection pools. When the application re-acquires a connection from the pool it should initialize the values to a desired state before using that connection.

The example below shows setting the action, module and client identifier attributes on the connection object:

```
# Set the tracing metadata
connection.client_identifier = "pythonuser"
connection.action = "Query Session tracing parameters"
connection.module = "End-to-end Demo"

for row in cursor.execute("""
        SELECT username, client_identifier, module, action
        FROM V$SESSION
        WHERE username = 'SYSTEM'"""):
    print(row)
```

The output will be:

```
('SYSTEM', 'pythonuser', 'End-to-end Demo', 'Query Session tracing parameters')
```

The values can also be manually set as shown by calling DBMS_APPLICATION_INFO procedures or DBMS_SESSION.SET_IDENTIFIER. These incur round-trips to the database, however, reducing scalability.

```
BEGIN
    DBMS_SESSION.SET_IDENTIFIER('pythonuser');
    DBMS_APPLICATION_INFO.set_module('End-to-End Demo');
    DBMS_APPLICATION_INFO.set_action(action_name => 'Query Session tracing parameters
↪');
END;
```

1.21.3 Low Level SQL Tracing in cx_Oracle

cx_Oracle is implemented using the ODPI-C wrapper on top of the Oracle Client libraries. The ODPI-C tracing capability can be used to log executed cx_Oracle statements to the standard error stream. Before executing Python, set the environment variable DPI_DEBUG_LEVEL to 16.

At a Windows command prompt, this could be done with:

```
set DPI_DEBUG_LEVEL=16
```

On Linux, you might use:

```
export DPI_DEBUG_LEVEL=16
```

After setting the variable, run the Python Script, for example on Linux:

```
python end-to-endtracing.py 2> log.txt
```

For an application that does a single query, the log file might contain a tracing line consisting of the prefix 'ODPI', a thread identifier, a timestamp, and the SQL statement executed:

```
ODPI [26188] 2019-03-26 09:09:03.909: ODPI-C 3.1.1
ODPI [26188] 2019-03-26 09:09:03.909: debugging messages initialized at level 16
ODPI [26188] 2019-03-26 09:09:09.917: SQL SELECT * FROM jobss
Traceback (most recent call last):
File "end-to-endtracing.py", line 14, in <module>
  cursor.execute("select * from jobss")
cx_Oracle.DatabaseError: ORA-00942: table or view does not exist
```

See ODPI-C Debugging for documentation on `DPI_DEBUG_LEVEL`.

CHAPTER 2

API Manual

2.1 Module Interface

cx_Oracle.**__future__**
Special object which contains attributes which control the behavior of cx_Oracle, allowing for opting in for new features. No attributes are currently supported so all attributes will silently ignore being set and will always appear to have the value None.

Note: This method is an extension to the DB API definition.

New in version 6.2.

cx_Oracle.**Binary**(*string*)
Construct an object holding a binary (long) string value.

cx_Oracle.**clientversion**()
Return the version of the client library being used as a 5-tuple. The five values are the major version, minor version, update number, patch number and port update number.

Note: This method is an extension to the DB API definition.

cx_Oracle.**connect**(*user=None*, *password=None*, *dsn=None*, *mode=cx_Oracle.DEFAULT_AUTH*, *handle=0*, *pool=None*, *threaded=False*, *events=False*, *cclass=None*, *purity=cx_Oracle.ATTR_PURITY_DEFAULT*, *newpassword=None*, *encoding=None*, *nencoding=None*, *edition=None*, *appcontext=[]*, *tag=None*, *matchanytag=None*, *shardingkey=[]*, *supershardingkey=[]*)
cx_Oracle.**Connection**(*user=None*, *password=None*, *dsn=None*, *mode=cx_Oracle.DEFAULT_AUTH*, *handle=0*, *pool=None*, *threaded=False*, *events=False*, *cclass=None*, *purity=cx_Oracle.ATTR_PURITY_DEFAULT*, *newpassword=None*, *encoding=None*, *nencoding=None*, *edition=None*, *appcontext=[]*, *tag=None*, *matchanytag=False*, *shardingkey=[]*, *supershardingkey=[]*)
Constructor for creating a connection to the database. Return a *connection object*. All parameters are optional

and can be specified as keyword parameters. See *Connecting to Oracle Database* information about connections.

The dsn (data source name) is the TNS entry (from the Oracle names server or tnsnames.ora file) or is a string like the one returned from *makedsn()*. If the user parameter is passed and the password and dsn parameters are not passed, the user parameter is assumed to be a connect string in the format `user/password@dsn`, the same format accepted by Oracle applications such as SQL*Plus. See *Connection Strings* for more information.

If the mode is specified, it must be one of the *connection authorization modes* which are defined at the module level.

If the handle is specified, it must be of type OCISvcCtx* and is only of use when embedding Python in an application (like PowerBuilder) which has already made the connection.

The pool parameter is expected to be a *session pool object* and the use of this parameter is the equivalent of calling *SessionPool.acquire()*. Parameters not accepted by that method are ignored.

The threaded parameter is expected to be a boolean expression which indicates whether or not Oracle should wrap accesses to connections with a mutex. Doing so in single threaded applications imposes a performance penalty of about 10-15% which is why the default is False.

The events parameter is expected to be a boolean expression which indicates whether or not to initialize Oracle in events mode. This is required for continuous query notification and high availability event notifications.

The cclass parameter is expected to be a string and defines the connection class for database resident connection pooling (DRCP).

The purity parameter is expected to be one of *ATTR_PURITY_NEW*, *ATTR_PURITY_SELF*, or *ATTR_PURITY_DEFAULT*.

The newpassword parameter is expected to be a string if specified and sets the password for the logon during the connection process.

See the *globalization* section for details on the encoding and nencoding parameters. Note the default encoding and nencoding values changed to "UTF-8" in cx_Oracle 8, and any character set in NLS_LANG is ignored.

The edition parameter is expected to be a string if specified and sets the edition to use for the session. It is only relevant if both the client and the database are at least Oracle Database 11.2. If this parameter is used with the cclass parameter the exception "DPI-1058: edition not supported with connection class" will be raised.

The appcontext parameter is expected to be a list of 3-tuples, if specified, and sets the application context for the connection. Application context is available in the database by using the sys_context() PL/SQL method and can be used within a logon trigger as well as any other PL/SQL procedures. Each entry in the list is expected to contain three strings: the namespace, the name and the value.

The tag parameter, if specified, is expected to be a string and will limit the sessions that can be returned from a session pool unless the matchanytag parameter is set to True. In that case sessions with the specified tag will be preferred over others, but if no such sessions are available a session with a different tag may be returned instead. In any case, untagged sessions will always be returned if no sessions with the specified tag are available. Sessions are tagged when they are *released* back to the pool.

The shardingkey and supershardingkey parameters, if specified, are expected to be a sequence of values which will be used to identify the database shard to connect to. The key values can be strings, numbers, bytes or dates.

`cx_Oracle.`**`Cursor`**(*connection*)

Constructor for creating a cursor. Return a new *cursor object* using the connection.

Note: This method is an extension to the DB API definition.

`cx_Oracle.`**`Date`**(*year*, *month*, *day*)

Construct an object holding a date value.

cx_Oracle.**DateFromTicks**(*ticks*)

Construct an object holding a date value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

cx_Oracle.**init_oracle_client**(*lib_dir=None*, *config_dir=None*, *error_url=None*, *driver_name=None*)

Initialize the Oracle client library now, rather than when *cx_Oracle.clientversion()*, *cx_Oracle.connect()* or *cx_Oracle.SessionPool()* is called for the first time. If initialization has already taken place, an exception is raised.

If the parameter *lib_dir* is not *None* or the empty string, the specified directory is the only one searched for the Oracle Client libraries; otherwise, the standard way of locating the Oracle Client library is used.

If the parameter *config_dir* is not *None* or the empty string, the specified directory is used to find Oracle Client library configuration files. This is equivalent to setting the environment variable *TNS_ADMIN* and overrides any value already set in *TNS_ADMIN*. If this parameter is not set, the standard way of locating Oracle Client library configuration files is used.

If the parameter *error_url* is not *None* or the empty string, the specified value is included in the message of the exception raised when the Oracle Client library cannot be loaded; otherwise, the *cx_Oracle 8 Installation* URL is included.

If the parameter *driver_name* is not *None* or the empty string, the specified value can be found in database views that give information about connections. For example, it is in the CLIENT_DRIVER column of V$SESSION_CONNECT_INFO. The standard is to set this value to "<name> : version>", where <name> is the name of the driver and <version> is its version. Theere should be a single space character before and after the colon. If this value is not specified, then the default value of "cx_Oracle : <version>" is used.

See *cx_Oracle 8 Initialization* for more discussion.

Note: This method is an extension to the DB API definition.

---

cx_Oracle.**makedsn**(*host*, *port*, *sid=None*, *service_name=None*, *region=None*, *sharding_key=None*, *super_sharding_key=None*)

Return a string suitable for use as the dsn parameter for *connect()*. This string is identical to the strings that are defined by the Oracle names server or defined in the tnsnames.ora file.

Note: This method is an extension to the DB API definition.

---

cx_Oracle.**SessionPool**(*user=None*, *password=None*, *dsn=None*, *min=1*, *max=2*, *increment=1*, *connectiontype=cx_Oracle.Connection*, *threaded=False*, *getmode=cx_Oracle.SPOOL_ATTRVAL_NOWAIT*, *events=False*, *homogeneous=True*, *externalauth=False*, *encoding=None*, *nencoding=None*, *edition=None*, *timeout=0*, *waitTimeout=0*, *maxLifetimeSession=0*, *sessionCallback=None*, *maxSessionsPerShard=0*)

Create and return a *session pool object*. Session pooling (also known as connection pooling) creates a pool of available connections to the database, allowing applications to acquire a connection very quickly. It is of primary use in a server where connections are requested in rapid succession and used for a short period of time, for example in a web server. See *Connection Pooling* for more information.

Connection pooling in cx_Oracle is handled by Oracle's Session pooling technology. This allows cx_Oracle applications to support features like Application Continuity.

The user, password and dsn parameters are the same as for *cx_Oracle.connect()*

The min, max and increment parameters control pool growth behavior. A fixed pool size where min equals max is recommended to help prevent connection storms and to help overall system stability. The min parameter is the number of connections opened when the pool is created. The increment is the number of connections that are opened whenever a connection request exceeds the number of currently open connections. The max parameter is the maximum number of connections that can be open in the connection pool. Note that when *external authentication* or *heterogeneous pools* are used, the pool growth behavior is different.

If the connectiontype parameter is specified, all calls to `acquire()` will create connection objects of that type, rather than the base type defined at the module level.

The threaded parameter is expected to be a boolean expression which indicates whether Oracle should wrap accesses to connections with a mutex. Doing so in single threaded applications imposes a performance penalty of about 10-15% which is why the default is False.

The getmode parameter indicates whether or not future `SessionPool.acquire()` calls will wait for available connections. It can be one of the *Session Pool Get Modes* values.

The events parameter is expected to be a boolean expression which indicates whether or not to initialize Oracle in events mode. This is required for continuous query notification and high availability event notifications.

The homogeneous parameter is expected to be a boolean expression which indicates whether or not to create a homogeneous pool. A homogeneous pool requires that all connections in the pool use the same credentials. As such proxy authentication and external authentication is not possible with a homogeneous pool. See *Heterogeneous and Homogeneous Connection Pools*.

The externalauth parameter is expected to be a boolean expression which indicates whether or not external authentication should be used. External authentication implies that something other than the database is authenticating the user to the database. This includes the use of operating system authentication and Oracle wallets. See *Connecting Using External Authentication*.

The encoding and nencoding parameters set the encodings used for string values transferred between cx_Oracle and Oracle Database, see *Character Sets and Globalization*. Note the default encoding and nencoding values changed to "UTF-8" in cx_Oracle 8, and any character set in NLS_LANG is ignored.

The edition parameter is expected to be a string, if specified, and sets the edition to use for the sessions in the pool. It is only relevant if both the client and the server are at least Oracle Database 11.2. See *Edition-Based Redefinition (EBR)*.

The timeout parameter is expected to be an integer, if specified, and sets the length of time (in seconds) after which idle sessions in the pool are terminated. Note that termination only occurs when the pool is accessed. The default value of 0 means that no idle sessions are terminated.

The waitTimeout parameter is expected to be an integer, if specified, and sets the length of time (in milliseconds) that the caller should wait for a session to become available in the pool before returning with an error. This value is only used if the getmode parameter is set to the value `cx_Oracle.SPOOL_ATTRVAL_TIMEDWAIT`.

The maxLifetimeSession parameter is expected to be an integer, if specified, and sets the maximum length of time (in seconds) a pooled session may exist. Sessions that are in use will not be closed. They become candidates for termination only when they are released back to the pool and have existed for longer than maxLifetimeSession seconds. Note that termination only occurs when the pool is accessed. The default value is 0 which means that there is no maximum length of time that a pooled session may exist.

The sessionCallback parameter is expected to be either a string or a callable. If the sessionCallback parameter is a callable, it will be called when a newly created connection is returned from the pool, or when a tag is requested and that tag does not match the connection's actual tag. The callable will be invoked with the connection and the requested tag as its only parameters. If the parameter is a string, it should be the name of a PL/SQL procedure that will be called when `SessionPool.acquire()` requests a tag and that tag does not match the connection's actual tag. See *Session CallBacks for Setting Pooled Connection State*. Support for the PL/SQL procedure requires Oracle Client libraries 12.2 or later. See the OCI documentation for more information.

The maxSessionsPerShard parameter is expected to be an integer, if specified, and sets the maximum number of sessions in the pool that can be used for any given shard in a sharded database. This value is ignored if the Oracle client library version is less than 18.3.

Note: This method is an extension to the DB API definition.

cx_Oracle.**Time**(*hour*, *minute*, *second*)
Construct an object holding a time value.

Note: The time only data type is not supported by Oracle. Calling this function will raise a NotSupportedError exception.

cx_Oracle.**TimeFromTicks**(*ticks*)
Construct an object holding a time value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

Note: The time only data type is not supported by Oracle. Calling this function will raise a NotSupportedError exception.

cx_Oracle.**Timestamp**(*year*, *month*, *day*, *hour*, *minute*, *second*)
Construct an object holding a time stamp value.

cx_Oracle.**TimestampFromTicks**(*ticks*)
Construct an object holding a time stamp value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

2.1.1 Constants

General

cx_Oracle.**apilevel**
String constant stating the supported DB API level. Currently '2.0'.

cx_Oracle.**buildtime**
String constant stating the time when the binary was built.

Note: This constant is an extension to the DB API definition.

cx_Oracle.**paramstyle**
String constant stating the type of parameter marker formatting expected by the interface. Currently 'named' as in 'where name = :name'.

cx_Oracle.**threadsafety**
Integer constant stating the level of thread safety that the interface supports. Currently 2, which means that threads may share the module and connections, but not cursors. Sharing means that a thread may use a resource without wrapping it using a mutex semaphore to implement resource locking.

Note that in order to make use of multiple threads in a program which intends to connect and disconnect in different threads, the threaded parameter to *connect()* or *SessionPool()* must be true.

cx_Oracle.**version**

cx_Oracle.**__version__**
String constant stating the version of the module. Currently '8.0.0'.

Note: This attribute is an extension to the DB API definition.

Advanced Queuing: Delivery Modes

These constants are extensions to the DB API definition. They are possible values for the *deliverymode* attribute of the *dequeue options object* passed as the options parameter to the *Connection.deq()* method as well as the *deliverymode* attribute of the *enqueue options object* passed as the options parameter to the *Connection.enq()* method. They are also possible values for the *deliverymode* attribute of the *message properties object* passed as the msgproperties parameter to the *Connection.deq()* and *Connection.enq()* methods.

cx_Oracle.**MSG_BUFFERED**
This constant is used to specify that enqueue/dequeue operations should enqueue or dequeue buffered messages.

cx_Oracle.**MSG_PERSISTENT**
This constant is used to specify that enqueue/dequeue operations should enqueue or dequeue persistent messages. This is the default value.

cx_Oracle.**MSG_PERSISTENT_OR_BUFFERED**
This constant is used to specify that dequeue operations should dequeue either persistent or buffered messages.

Advanced Queuing: Dequeue Modes

These constants are extensions to the DB API definition. They are possible values for the *mode* attribute of the *dequeue options object*. This object is the options parameter for the *Connection.deq()* method.

cx_Oracle.**DEQ_BROWSE**
This constant is used to specify that dequeue should read the message without acquiring any lock on the message (equivalent to a select statement).

cx_Oracle.**DEQ_LOCKED**
This constant is used to specify that dequeue should read and obtain a write lock on the message for the duration of the transaction (equivalent to a select for update statement).

cx_Oracle.**DEQ_REMOVE**
This constant is used to specify that dequeue should read the message and update or delete it. This is the default value.

cx_Oracle.**DEQ_REMOVE_NODATA**
This constant is used to specify that dequeue should confirm receipt of the message but not deliver the actual message content.

Advanced Queuing: Dequeue Navigation Modes

These constants are extensions to the DB API definition. They are possible values for the *navigation* attribute of the *dequeue options object*. This object is the options parameter for the *Connection.deq()* method.

cx_Oracle.**DEQ_FIRST_MSG**
This constant is used to specify that dequeue should retrieve the first available message that matches the search criteria. This resets the position to the beginning of the queue.

cx_Oracle.**DEQ_NEXT_MSG**
: This constant is used to specify that dequeue should retrieve the next available message that matches the search criteria. If the previous message belongs to a message group, AQ retrieves the next available message that matches the search criteria and belongs to the message group. This is the default.

cx_Oracle.**DEQ_NEXT_TRANSACTION**
: This constant is used to specify that dequeue should skip the remainder of the transaction group and retrieve the first message of the next transaction group. This option can only be used if message grouping is enabled for the current queue.

Advanced Queuing: Dequeue Visibility Modes

These constants are extensions to the DB API definition. They are possible values for the *visibility* attribute of the *dequeue options object*. This object is the options parameter for the *Connection.deq()* method.

cx_Oracle.**DEQ_IMMEDIATE**
: This constant is used to specify that dequeue should perform its work as part of an independent transaction.

cx_Oracle.**DEQ_ON_COMMIT**
: This constant is used to specify that dequeue should be part of the current transaction. This is the default value.

Advanced Queuing: Dequeue Wait Modes

These constants are extensions to the DB API definition. They are possible values for the *wait* attribute of the *dequeue options object*. This object is the options parameter for the *Connection.deq()* method.

cx_Oracle.**DEQ_NO_WAIT**
: This constant is used to specify that dequeue not wait for messages to be available for dequeuing.

cx_Oracle.**DEQ_WAIT_FOREVER**
: This constant is used to specify that dequeue should wait forever for messages to be available for dequeuing. This is the default value.

Advanced Queuing: Enqueue Visibility Modes

These constants are extensions to the DB API definition. They are possible values for the *visibility* attribute of the *enqueue options object*. This object is the options parameter for the *Connection.enq()* method.

cx_Oracle.**ENQ_IMMEDIATE**
: This constant is used to specify that enqueue should perform its work as part of an independent transaction.

cx_Oracle.**ENQ_ON_COMMIT**
: This constant is used to specify that enqueue should be part of the current transaction. This is the default value.

Advanced Queuing: Message States

These constants are extensions to the DB API definition. They are possible values for the *state* attribute of the *message properties object*. This object is the msgproperties parameter for the *Connection.deq()* and *Connection.enq()* methods.

cx_Oracle.**MSG_EXPIRED**
: This constant is used to specify that the message has been moved to the exception queue.

cx_Oracle.**MSG_PROCESSED**
: This constant is used to specify that the message has been processed and has been retained.

cx_Oracle.**MSG_READY**
: This constant is used to specify that the message is ready to be processed.

cx_Oracle.**MSG_WAITING**
: This constant is used to specify that the message delay has not yet been reached.

Advanced Queuing: Other

These constants are extensions to the DB API definition. They are special constants used in advanced queuing.

cx_Oracle.**MSG_NO_DELAY**
: This constant is a possible value for the `delay` attribute of the *message properties object* passed as the msgproperties parameter to the `Connection.deq()` and `Connection.enq()` methods. It specifies that no delay should be imposed and the message should be immediately available for dequeuing. This is also the default value.

cx_Oracle.**MSG_NO_EXPIRATION**
: This constant is a possible value for the `expiration` attribute of the *message properties object* passed as the msgproperties parameter to the `Connection.deq()` and `Connection.enq()` methods. It specifies that the message never expires. This is also the default value.

Connection Authorization Modes

These constants are extensions to the DB API definition. They are possible values for the mode parameter of the `connect()` method.

cx_Oracle.**DEFAULT_AUTH**
: This constant is used to specify that default authentication is to take place. This is the default value if no mode is passed at all.

  New in version 7.2.

cx_Oracle.**PRELIM_AUTH**
: This constant is used to specify that preliminary authentication is to be used. This is needed for performing database startup and shutdown.

cx_Oracle.**SYSASM**
: This constant is used to specify that SYSASM access is to be acquired.

cx_Oracle.**SYSBKP**
: This constant is used to specify that SYSBACKUP access is to be acquired.

cx_Oracle.**SYSDBA**
: This constant is used to specify that SYSDBA access is to be acquired.

cx_Oracle.**SYSDGD**
: This constant is used to specify that SYSDG access is to be acquired.

cx_Oracle.**SYSKMT**
: This constant is used to specify that SYSKM access is to be acquired.

cx_Oracle.**SYSOPER**
: This constant is used to specify that SYSOPER access is to be acquired.

cx_Oracle.**SYSRAC**
: This constant is used to specify that SYSRAC access is to be acquired.

Database Shutdown Modes

These constants are extensions to the DB API definition. They are possible values for the mode parameter of the *Connection.shutdown()* method.

cx_Oracle.**DBSHUTDOWN_ABORT**
: This constant is used to specify that the caller should not wait for current processing to complete or for users to disconnect from the database. This should only be used in unusual circumstances since database recovery may be necessary upon next startup.

cx_Oracle.**DBSHUTDOWN_FINAL**
: This constant is used to specify that the instance can be truly halted. This should only be done after the database has been shutdown with one of the other modes (except abort) and the database has been closed and dismounted using the appropriate SQL commands.

cx_Oracle.**DBSHUTDOWN_IMMEDIATE**
: This constant is used to specify that all uncommitted transactions should be rolled back and any connected users should be disconnected.

cx_Oracle.**DBSHUTDOWN_TRANSACTIONAL**
: This constant is used to specify that further connections to the database should be prohibited and no new transactions should be allowed. It then waits for all active transactions to complete.

cx_Oracle.**DBSHUTDOWN_TRANSACTIONAL_LOCAL**
: This constant is used to specify that further connections to the database should be prohibited and no new transactions should be allowed. It then waits for only local active transactions to complete.

Event Types

These constants are extensions to the DB API definition. They are possible values for the *Message.type* attribute of the messages that are sent for subscriptions created by the *Connection.subscribe()* method.

cx_Oracle.**EVENT_AQ**
: This constant is used to specify that one or more messages are available for dequeuing on the queue specified when the subscription was created.

cx_Oracle.**EVENT_DEREG**
: This constant is used to specify that the subscription has been deregistered and no further notifications will be sent.

cx_Oracle.**EVENT_NONE**
: This constant is used to specify no information is available about the event.

cx_Oracle.**EVENT_OBJCHANGE**
: This constant is used to specify that a database change has taken place on a table registered with the *Subscription.registerquery()* method.

cx_Oracle.**EVENT_QUERYCHANGE**
: This constant is used to specify that the result set of a query registered with the *Subscription.registerquery()* method has been changed.

cx_Oracle.**EVENT_SHUTDOWN**
: This constant is used to specify that the instance is in the process of being shut down.

cx_Oracle.**EVENT_SHUTDOWN_ANY**
: This constant is used to specify that any instance (when running RAC) is in the process of being shut down.

cx_Oracle.**EVENT_STARTUP**
: This constant is used to specify that the instance is in the process of being started up.

Operation Codes

These constants are extensions to the DB API definition. They are possible values for the operations parameter for the *Connection.subscribe()* method. One or more of these values can be OR'ed together. These values are also used by the *MessageTable.operation* or *MessageQuery.operation* attributes of the messages that are sent.

cx_Oracle.**OPCODE_ALLOPS**
: This constant is used to specify that messages should be sent for all operations.

cx_Oracle.**OPCODE_ALLROWS**
: This constant is used to specify that the table or query has been completely invalidated.

cx_Oracle.**OPCODE_ALTER**
: This constant is used to specify that messages should be sent when a registered table has been altered in some fashion by DDL, or that the message identifies a table that has been altered.

cx_Oracle.**OPCODE_DELETE**
: This constant is used to specify that messages should be sent when data is deleted, or that the message identifies a row that has been deleted.

cx_Oracle.**OPCODE_DROP**
: This constant is used to specify that messages should be sent when a registered table has been dropped, or that the message identifies a table that has been dropped.

cx_Oracle.**OPCODE_INSERT**
: This constant is used to specify that messages should be sent when data is inserted, or that the message identifies a row that has been inserted.

cx_Oracle.**OPCODE_UPDATE**
: This constant is used to specify that messages should be sent when data is updated, or that the message identifies a row that has been updated.

Session Pool Get Modes

These constants are extensions to the DB API definition. They are possible values for the getmode parameter of the *SessionPool()* method.

cx_Oracle.**SPOOL_ATTRVAL_FORCEGET**
: This constant is used to specify that a new connection will be returned if there are no free sessions available in the pool.

cx_Oracle.**SPOOL_ATTRVAL_NOWAIT**
: This constant is used to specify that an exception should be raised if there are no free sessions available in the pool. This is the default value.

cx_Oracle.**SPOOL_ATTRVAL_WAIT**
: This constant is used to specify that the caller should wait until a session is available if there are no free sessions available in the pool.

cx_Oracle.**SPOOL_ATTRVAL_TIMEDWAIT**
: This constant is used to specify that the caller should wait for a period of time (defined by the waitTimeout parameter) for a session to become available before returning with an error.

Session Pool Purity

These constants are extensions to the DB API definition. They are possible values for the purity parameter of the *connect()* method, which is used in database resident connection pooling (DRCP).

cx_Oracle.**ATTR_PURITY_DEFAULT**
: This constant is used to specify that the purity of the session is the default value identified by Oracle (see Oracle's documentation for more information). This is the default value.

cx_Oracle.**ATTR_PURITY_NEW**
: This constant is used to specify that the session acquired from the pool should be new and not have any prior session state.

cx_Oracle.**ATTR_PURITY_SELF**
: This constant is used to specify that the session acquired from the pool need not be new and may have prior session state.

Subscription Grouping Classes

These constants are extensions to the DB API definition. They are possible values for the groupingClass parameter of the *Connection.subscribe()* method.

cx_Oracle.**SUBSCR_GROUPING_CLASS_TIME**
: This constant is used to specify that events are to be grouped by the period of time in which they are received.

Subscription Grouping Types

These constants are extensions to the DB API definition. They are possible values for the groupingType parameter of the *Connection.subscribe()* method.

cx_Oracle.**SUBSCR_GROUPING_TYPE_SUMMARY**
: This constant is used to specify that when events are grouped a summary of the events should be sent instead of the individual events. This is the default value.

cx_Oracle.**SUBSCR_GROUPING_TYPE_LAST**
: This constant is used to specify that when events are grouped the last event that makes up the group should be sent instead of the individual events.

Subscription Namespaces

These constants are extensions to the DB API definition. They are possible values for the namespace parameter of the *Connection.subscribe()* method.

cx_Oracle.**SUBSCR_NAMESPACE_AQ**
: This constant is used to specify that notifications should be sent when a queue has messages available to dequeue.

cx_Oracle.**SUBSCR_NAMESPACE_DBCHANGE**
: This constant is used to specify that database change notification or query change notification messages are to be sent. This is the default value.

Subscription Protocols

These constants are extensions to the DB API definition. They are possible values for the protocol parameter of the *Connection.subscribe()* method.

cx_Oracle.**SUBSCR_PROTO_HTTP**
: This constant is used to specify that notifications will be sent to an HTTP URL when a message is generated. This value is currently not supported.

cx_Oracle.**SUBSCR_PROTO_MAIL**
: This constant is used to specify that notifications will be sent to an e-mail address when a message is generated. This value is currently not supported.

cx_Oracle.**SUBSCR_PROTO_OCI**
: This constant is used to specify that notifications will be sent to the callback routine identified when the subscription was created. It is the default value and the only value currently supported.

cx_Oracle.**SUBSCR_PROTO_SERVER**
: This constant is used to specify that notifications will be sent to a PL/SQL procedure when a message is generated. This value is currently not supported.

Subscription Quality of Service

These constants are extensions to the DB API definition. They are possible values for the qos parameter of the *Connection.subscribe()* method. One or more of these values can be OR'ed together.

cx_Oracle.**SUBSCR_QOS_BEST_EFFORT**
: This constant is used to specify that best effort filtering for query result set changes is acceptable. False positive notifications may be received. This behaviour may be suitable for caching applications.

cx_Oracle.**SUBSCR_QOS_DEREG_NFY**
: This constant is used to specify that the subscription should be automatically unregistered after the first notification is received.

cx_Oracle.**SUBSCR_QOS_QUERY**
: This constant is used to specify that notifications should be sent if the result set of the registered query changes. By default no false positive notifications will be generated.

cx_Oracle.**SUBSCR_QOS_RELIABLE**
: This constant is used to specify that notifications should not be lost in the event of database failure.

cx_Oracle.**SUBSCR_QOS_ROWIDS**
: This constant is used to specify that the rowids of the inserted, updated or deleted rows should be included in the message objects that are sent.

DB API Types

cx_Oracle.**BINARY**
: This type object is used to describe columns in a database that contain binary data. The database types *DB_TYPE_RAW* and *DB_TYPE_LONG_RAW* will compare equal to this value. If a variable is created with this type, the database type *DB_TYPE_RAW* will be used.

cx_Oracle.**DATETIME**
: This type object is used to describe columns in a database that are dates. The database types *DB_TYPE_DATE*, *DB_TYPE_TIMESTAMP*, *DB_TYPE_TIMESTAMP_LTZ* and *DB_TYPE_TIMESTAMP_TZ* will all compare equal to this value. If a variable is created with this type, the database type *DB_TYPE_DATE* will be used.

cx_Oracle.**NUMBER**
: This type object is used to describe columns in a database that are numbers. The database types *DB_TYPE_BINARY_DOUBLE*, *DB_TYPE_BINARY_FLOAT*, *DB_TYPE_BINARY_INTEGER* and *DB_TYPE_NUMBER* will all compare equal to this value. If a variable is created with this type, the database type *DB_TYPE_NUMBER* will be used.

cx_Oracle.**ROWID**
: This type object is used to describe the pseudo column "rowid". The database type *DB_TYPE_ROWID* will compare equal to this value. If a variable is created with this type, the database type *DB_TYPE_VARCHAR* will be used.

cx_Oracle.**STRING**
: This type object is used to describe columns in a database that are strings. The database types *DB_TYPE_CHAR*, *DB_TYPE_LONG*, *DB_TYPE_NCHAR*, *DB_TYPE_NVARCHAR* and *DB_TYPE_VARCHAR* will all compare equal to this value. If a variable is created with this type, the database type *DB_TYPE_VARCHAR* will be used.

Database Types

All of these types are extensions to the DB API definition. They are found in query and object metadata. They can also be used to specify the database type when binding data.

cx_Oracle.**DB_TYPE_BFILE**
: Describes columns, attributes or array elements in a database that are of type BFILE. It will compare equal to the DB API type *BINARY*.

cx_Oracle.**DB_TYPE_BINARY_DOUBLE**
: Describes columns, attributes or array elements in a database that are of type BINARY_DOUBLE. It will compare equal to the DB API type *NUMBER*.

cx_Oracle.**DB_TYPE_BINARY_FLOAT**
: Describes columns, attributes or array elements in a database that are of type BINARY_FLOAT. It will compare equal to the DB API type *NUMBER*.

cx_Oracle.**DB_TYPE_BINARY_INTEGER**
: Describes attributes or array elements in a database that are of type BINARY_INTEGER. It will compare equal to the DB API type *NUMBER*.

cx_Oracle.**DB_TYPE_BLOB**
: Describes columns, attributes or array elements in a database that are of type BLOB. It will compare equal to the DB API type *BINARY*.

cx_Oracle.**DB_TYPE_BOOLEAN**
: Describes attributes or array elements in a database that are of type BOOLEAN. It is only available in Oracle 12.1 and higher and only within PL/SQL.

cx_Oracle.**DB_TYPE_CHAR**
: Describes columns, attributes or array elements in a database that are of type CHAR. It will compare equal to the DB API type *STRING*.

  Note that these are fixed length string values and behave differently from VARCHAR2.

cx_Oracle.**DB_TYPE_CLOB**
: Describes columns, attributes or array elements in a database that are of type CLOB. It will compare equal to the DB API type *STRING*.

cx_Oracle.**DB_TYPE_CURSOR**
: Describes columns in a database that are of type CURSOR. In PL/SQL these are knoown as REF CURSOR.

cx_Oracle.**DB_TYPE_DATE**
: Describes columns, attributes or array elements in a database that are of type DATE. It will compare equal to the DB API type *DATETIME*.

cx_Oracle.**DB_TYPE_INTERVAL_DS**
: Describes columns, attributes or array elements in a database that are of type INTERVAL DAY TO SECOND.

cx_Oracle.**DB_TYPE_INTERVAL_YM**
: Describes columns, attributes or array elements in a database that are of type INTERVAL YEAR TO MONTH. This database type is not currently supported by cx_Oracle.

cx_Oracle.**DB_TYPE_LONG**
: Describes columns, attributes or array elements in a database that are of type LONG. It will compare equal to the DB API type *STRING*.

cx_Oracle.**DB_TYPE_LONG_RAW**
: Describes columns, attributes or array elements in a database that are of type LONG RAW. It will compare equal to the DB API type *BINARY*.

cx_Oracle.**DB_TYPE_NCHAR**
: Describes columns, attributes or array elements in a database that are of type NCHAR. It will compare equal to the DB API type *STRING*.

  Note that these are fixed length string values and behave differently from NVARCHAR2.

cx_Oracle.**DB_TYPE_NCLOB**
: Describes columns, attributes or array elements in a database that are of type NCLOB. It will compare equal to the DB API type *STRING*.

cx_Oracle.**DB_TYPE_NUMBER**
: Describes columns, attributes or array elements in a database that are of type NUMBER. It will compare equal to the DB API type *NUMBER*.

cx_Oracle.**DB_TYPE_NVARCHAR**
: Describes columns, attributes or array elements in a database that are of type NVARCHAR2. It will compare equal to the DB API type *STRING*.

cx_Oracle.**DB_TYPE_OBJECT**
: Describes columns, attributes or array elements in a database that are an instance of a named SQL or PL/SQL type.

cx_Oracle.**DB_TYPE_RAW**
: Describes columns, attributes or array elements in a database that are of type RAW. It will compare equal to the DB API type *BINARY*.

cx_Oracle.**DB_TYPE_ROWID**
: Describes columns, attributes or array elements in a database that are of type ROWID or UROWID. It will compare equal to the DB API type *ROWID*.

cx_Oracle.**DB_TYPE_TIMESTAMP**
: Describes columns, attributes or array elements in a database that are of type TIMESTAMP. It will compare equal to the DB API type *DATETIME*.

cx_Oracle.**DB_TYPE_TIMESTAMP_LTZ**
: Describes columns, attributes or array elements in a database that are of type TIMESTAMP WITH LOCAL TIME ZONE. It will compare equal to the DB API type *DATETIME*.

cx_Oracle.**DB_TYPE_TIMESTAMP_TZ**
: Describes columns, attributes or array elements in a database that are of type TIMESTAMP WITH TIME ZONE. It will compare equal to the DB API type *DATETIME*.

cx_Oracle.**DB_TYPE_VARCHAR**
: Describes columns, attributes or array elements in a database that are of type VARCHAR2. It will compare equal to the DB API type *STRING*.

Database Type Synonyms

All of the following constants are deprecated and will be removed in a future version of cx_Oracle.

cx_Oracle.**BFILE**
: A synonym for *DB_TYPE_BFILE*.

Deprecated since version 8.0.

cx_Oracle.**BLOB**
A synonym for *DB_TYPE_BLOB*.

Deprecated since version 8.0.

cx_Oracle.**BOOLEAN**
A synonym for *DB_TYPE_BOOLEAN*.

Deprecated since version 8.0.

cx_Oracle.**CLOB**
A synonym for *DB_TYPE_CLOB*.

Deprecated since version 8.0.

cx_Oracle.**CURSOR**
A synonym for *DB_TYPE_CURSOR*.

Deprecated since version 8.0.

cx_Oracle.**FIXED_CHAR**
A synonym for *DB_TYPE_CHAR*.

Deprecated since version 8.0.

cx_Oracle.**FIXED_NCHAR**
A synonym for *DB_TYPE_NCHAR*.

Deprecated since version 8.0.

cx_Oracle.**INTERVAL**
A synonym for *DB_TYPE_INTERVAL_DS*.

Deprecated since version 8.0.

cx_Oracle.**LONG_BINARY**
A synonym for *DB_TYPE_LONG_RAW*.

Deprecated since version 8.0.

cx_Oracle.**LONG_STRING**
A synonym for *DB_TYPE_LONG*.

Deprecated since version 8.0.

cx_Oracle.**NATIVE_FLOAT**
A synonym for *DB_TYPE_BINARY_DOUBLE*.

Deprecated since version 8.0.

cx_Oracle.**NATIVE_INT**
A synonym for *DB_TYPE_BINARY_INTEGER*.

Deprecated since version 8.0.

cx_Oracle.**NCHAR**
A synonym for *DB_TYPE_NVARCHAR*.

Deprecated since version 8.0.

cx_Oracle.**NCLOB**
A synonym for *DB_TYPE_NCLOB*.

Deprecated since version 8.0.

cx_Oracle.**OBJECT**
A synonym for *DB_TYPE_OBJECT*.

Deprecated since version 8.0.

cx_Oracle.**TIMESTAMP**
A synonym for *DB_TYPE_TIMESTAMP*.

Deprecated since version 8.0.

Other Types

All of these types are extensions to the DB API definition.

cx_Oracle.**ApiType**
This type object is the Python type of the database API type constants *BINARY*, *DATETIME*, *NUMBER*, *ROWID* and *STRING*.

cx_Oracle.**DbType**
This type object is the Python type of the *database type constants*.

cx_Oracle.**LOB**
This type object is the Python type of *DB_TYPE_BLOB*, *DB_TYPE_BFILE*, *DB_TYPE_CLOB* and *DB_TYPE_NCLOB* data that is returned from cursors.

2.1.2 Exceptions

exception cx_Oracle.**Warning**
Exception raised for important warnings and defined by the DB API but not actually used by cx_Oracle.

exception cx_Oracle.**Error**
Exception that is the base class of all other exceptions defined by cx_Oracle and is a subclass of the Python StandardError exception (defined in the module exceptions).

exception cx_Oracle.**InterfaceError**
Exception raised for errors that are related to the database interface rather than the database itself. It is a subclass of Error.

exception cx_Oracle.**DatabaseError**
Exception raised for errors that are related to the database. It is a subclass of Error.

exception cx_Oracle.**DataError**
Exception raised for errors that are due to problems with the processed data. It is a subclass of DatabaseError.

exception cx_Oracle.**OperationalError**
Exception raised for errors that are related to the operation of the database but are not necessarily under the control of the programmer. It is a subclass of DatabaseError.

exception cx_Oracle.**IntegrityError**
Exception raised when the relational integrity of the database is affected. It is a subclass of DatabaseError.

exception cx_Oracle.**InternalError**
Exception raised when the database encounters an internal error. It is a subclass of DatabaseError.

exception cx_Oracle.**ProgrammingError**
Exception raised for programming errors. It is a subclass of DatabaseError.

exception cx_Oracle.**NotSupportedError**
Exception raised when a method or database API was used which is not supported by the database. It is a subclass of DatabaseError.

2.1.3 Exception handling

Note: PEP 249 (Python Database API Specification v2.0) says the following about exception values:

> [...] The values of these exceptions are not defined. They should give the user a fairly good idea of what went wrong, though. [...]

With cx_Oracle every exception object has exactly one argument in the `args` tuple. This argument is a `cx_Oracle._Error` object which has the following five read-only attributes.

`_Error.`**`code`**
: Integer attribute representing the Oracle error number (ORA-XXXXX).

`_Error.`**`offset`**
: Integer attribute representing the error offset when applicable.

`_Error.`**`message`**
: String attribute representing the Oracle message of the error. This message is localized by the environment of the Oracle connection.

`_Error.`**`context`**
: String attribute representing the context in which the exception was raised.

`_Error.`**`isrecoverable`**
: Boolean attribute representing whether the error is recoverable or not. This is False in all cases unless both Oracle Database 12.1 (or later) and Oracle Client 12.1 (or later) are being used.

  New in version 5.3.

This allows you to use the exceptions for example in the following way:

```
import cx_Oracle

connection = cx_Oracle.connect("cx_Oracle/dev@localhost/orclpdb1")
cursor = connection.cursor()

try:
    cursor.execute("select 1 / 0 from dual")
except cx_Oracle.DatabaseError as exc:
    error, = exc.args
    print("Oracle-Error-Code:", error.code)
    print("Oracle-Error-Message:", error.message)
```

2.2 Connection Object

Note: Any outstanding changes will be rolled back when the connection object is destroyed or closed.

`Connection.`**`__enter__`**`()`
: The entry point for the connection as a context manager. It returns itself.

  Note: This method is an extension to the DB API definition.

Connection.**__exit__**()
The exit point for the connection as a context manager. This will close the connection and roll back any uncommitted transaction.

Note: This method is an extension to the DB API definition.

Connection.**action**
This write-only attribute sets the action column in the v$session table. It is a string attribute and cannot be set to None – use the empty string instead.

Note: This attribute is an extension to the DB API definition.

Connection.**autocommit**
This read-write attribute determines whether autocommit mode is on or off. When autocommit mode is on, all statements are committed as soon as they have completed executing.

Note: This attribute is an extension to the DB API definition.

Connection.**begin**([*formatId*, *transactionId*, *branchId*])
Explicitly begin a new transaction. Without parameters, this explicitly begins a local transaction; otherwise, this explicitly begins a distributed (global) transaction with the given parameters. See the Oracle documentation for more details.

Note that in order to make use of global (distributed) transactions, the *internal_name* and *external_name* attributes must be set.

Note: This method is an extension to the DB API definition.

Connection.**callTimeout**
This read-write attribute specifies the amount of time (in milliseconds) that a single round-trip to the database may take before a timeout will occur. A value of 0 means that no timeout will take place.

New in version 7.0.

Note: This attribute is an extension to the DB API definition and is only available in Oracle Client 18c and higher.

Connection.**cancel**()
Cancel a long-running transaction.

Note: This method is an extension to the DB API definition.

Connection.**changepassword**(*oldpassword*, *newpassword*)
Change the password of the logon.

Note: This method is an extension to the DB API definition.

Connection.**client_identifier**
This write-only attribute sets the client_identifier column in the v$session table.

Note: This attribute is an extension to the DB API definition.

Connection.**clientinfo**
This write-only attribute sets the client_info column in the v$session table.

Note: This attribute is an extension to the DB API definition.

Connection.**close**()
Close the connection now, rather than whenever __del__ is called. The connection will be unusable from this point forward; an Error exception will be raised if any operation is attempted with the connection.

All open cursors and LOBs created by the connection will be closed and will also no longer be usable.

Internally, references to the connection are held by cursor objects, LOB objects, subscription objects, etc. Once all of these references are released, the connection itself will be closed automatically. Either control references to these related objects carefully or explicitly close connections in order to ensure sufficient resources are available.

Connection.**commit**()
Commit any pending transactions to the database.

Connection.**createlob**(*lobType*)
Create and return a new temporary *LOB object* of the specified type. The lobType parameter should be one of *cx_Oracle.CLOB*, *cx_Oracle.BLOB* or *cx_Oracle.NCLOB*.

New in version 6.2.

Note: This method is an extension to the DB API definition.

Connection.**current_schema**
This read-write attribute sets the current schema attribute for the session. Setting this value is the same as executing the SQL statement "ALTER SESSION SET CURRENT_SCHEMA". The attribute is set (and verified) on the next call that does a round trip to the server. The value is placed before unqualified database objects in SQL statements you then execute.

Note: This attribute is an extension to the DB API definition.

Connection.**cursor**()
Return a new *cursor object* using the connection.

Connection.**dbop**
This write-only attribute sets the database operation that is to be monitored. This can be viewed in the DBOP_NAME column of the V$SQL_MONITOR table.

Note: This attribute is an extension to the DB API definition.

Connection.**deq**(*name*, *options*, *msgproperties*, *payload*)
Returns a message id after successfully dequeuing a message. The options object can be created using *deqoptions()* and the msgproperties object can be created using *msgproperties()*. The payload must be an object created using *ObjectType.newobject()*.

New in version 5.3.

Deprecated since version 7.2: Use the methods *Queue.deqOne()* or *Queue.deqMany()* instead.

Note: This method is an extension to the DB API definition.

Connection.**deqoptions**()
Returns an object specifying the options to use when dequeuing messages. See *Dequeue Options* for more information.

New in version 5.3.

Deprecated since version 7.2: Use the attribute *Queue.deqOptions* instead.

Note: This method is an extension to the DB API definition.

Connection.**dsn**
This read-only attribute returns the TNS entry of the database to which a connection has been established.

Note: This attribute is an extension to the DB API definition.

Connection.**edition**
This read-only attribute gets the session edition and is only available in Oracle Database 11.2 (both client and server must be at this level or higher for this to work).

New in version 5.3.

Note: This attribute is an extension to the DB API definition.

Connection.**encoding**
This read-only attribute returns the IANA character set name of the character set in use by the Oracle client for regular strings.

Note: This attribute is an extension to the DB API definition.

Connection.**enq**(*name*, *options*, *msgproperties*, *payload*)
Returns a message id after successfully enqueuing a message. The options object can be created using *enqoptions()* and the msgproperties object can be created using *msgproperties()*. The payload must be an object created using *ObjectType.newobject()*.

New in version 5.3.

Deprecated since version 7.2: Use the methods *Queue.enqOne()* or *Queue.enqMany()* instead.

Note: This method is an extension to the DB API definition.

Connection.**enqoptions**()
Returns an object specifying the options to use when enqueuing messages. See *Enqueue Options* for more information.

New in version 5.3.

Deprecated since version 7.2: Use the attribute *Queue.enqOptions* instead.

Note: This method is an extension to the DB API definition.

Connection.**external_name**
This read-write attribute specifies the external name that is used by the connection when logging distributed transactions.

New in version 5.3.

Note: This attribute is an extension to the DB API definition.

Connection.**getSodaDatabase**()
Return a *SodaDatabase* object for Simple Oracle Document Access (SODA). All SODA operations are performed either on the returned SodaDatabase object or from objects created by the returned SodaDatabase object. See here for additional information on SODA.

New in version 7.0.

Note: This method is an extension to the DB API definition.

Connection.**gettype**(*name*)
Return a *type object* given its name. This can then be used to create objects which can be bound to cursors created by this connection.

New in version 5.3.

Note: This method is an extension to the DB API definition.

Connection.**handle**
This read-only attribute returns the OCI service context handle for the connection. It is primarily provided to facilitate testing the creation of a connection using the OCI service context handle.

Note: This attribute is an extension to the DB API definition.

Connection.**inputtypehandler**
This read-write attribute specifies a method called for each value that is bound to a statement executed on any cursor associated with this connection. The method signature is handler(cursor, value, arraysize) and the return value is expected to be a variable object or None in which case a default variable object will be created. If this attribute is None, the default behavior will take place for all values bound to statements.

Note: This attribute is an extension to the DB API definition.

Connection.**internal_name**
This read-write attribute specifies the internal name that is used by the connection when logging distributed transactions.

New in version 5.3.

Note: This attribute is an extension to the DB API definition.

Connection.**ltxid**
This read-only attribute returns the logical transaction id for the connection. It is used within Oracle Transaction Guard as a means of ensuring that transactions are not duplicated. See the Oracle documentation and the provided sample for more information.

New in version 5.3.

Connection.**maxBytesPerCharacter**
This read-only attribute returns the maximum number of bytes each character can use for the client character set.

Note: This attribute is an extension to the DB API definition.

Connection.**module**
This write-only attribute sets the module column in the v$session table. The maximum length for this string is 48 and if you exceed this length you will get ORA-24960.

Connection.**msgproperties**(*payload*, *correlation*, *delay*, *exceptionq*, *expiration*, *priority*)
Returns an object specifying the properties of messages used in advanced queuing. See *Message Properties* for more information.

Each of the parameters are optional. If specified, they act as a shortcut for setting each of the equivalently named properties.

New in version 5.3.

Changed in version 7.2: Added parameters

Note: This method is an extension to the DB API definition.

Connection.**nencoding**
This read-only attribute returns the IANA character set name of the national character set in use by the Oracle client.

Note: This attribute is an extension to the DB API definition.

Connection.**outputtypehandler**
This read-write attribute specifies a method called for each column that is going to be fetched from any cursor associated with this connection. The method signature is handler(cursor, name, defaultType, length, precision, scale) and the return value is expected to be a variable object or None in which case a default variable object will be created. If this attribute is None, the default behavior will take place for all columns fetched from cursors.

See *Changing Fetched Data Types with Output Type Handlers*.

Note: This attribute is an extension to the DB API definition.

Connection.**ping**()
Ping the server which can be used to test if the connection is still active.

Note: This method is an extension to the DB API definition.

Connection.**prepare**()
Prepare the distributed (global) transaction for commit. Return a boolean indicating if a transaction was actually prepared in order to avoid the error ORA-24756 (transaction does not exist).

Note: This method is an extension to the DB API definition.

Connection.**queue**(*name*, *payloadType=None*)
Creates a *queue* which is used to enqueue and dequeue messages in Advanced Queueing.

The name parameter is expected to be a string identifying the queue in which messages are to be enqueued or dequeued.

The payloadType parameter, if specified, is expected to be an *object type* that identifies the type of payload the queue expects. If not specified, RAW data is enqueued and dequeued.

New in version 7.2.

Note: This method is an extension to the DB API definition.

Connection.**rollback**()
Rollback any pending transactions.

Connection.**shutdown**([*mode*])
Shutdown the database. In order to do this the connection must be connected as *SYSDBA* or *SYSOPER*. Two calls must be made unless the mode specified is *DBSHUTDOWN_ABORT*. An example is shown below:

```
import cx_Oracle

connection = cx_Oracle.connect(mode = cx_Oracle.SYSDBA)
connection.shutdown(mode = cx_Oracle.DBSHUTDOWN_IMMEDIATE)
cursor = connection.cursor()
cursor.execute("alter database close normal")
cursor.execute("alter database dismount")
connection.shutdown(mode = cx_Oracle.DBSHUTDOWN_FINAL)
```

Note: This method is an extension to the DB API definition.

Connection.**startup**(*force=False*, *restrict=False*, *pfile=None*)
Startup the database. This is equivalent to the SQL*Plus command "startup nomount". The connection must be connected as *SYSDBA* or *SYSOPER* with the *PRELIM_AUTH* option specified for this to work.

The pfile parameter, if specified, is expected to be a string identifying the location of the parameter file (PFILE) which will be used instead of the stored parameter file (SPFILE).

An example is shown below:

```
import cx_Oracle

connection = cx_Oracle.connect(
        mode=cx_Oracle.SYSDBA | cx_Oracle.PRELIM_AUTH)
connection.startup()
```

(continues on next page)

(continued from previous page)

```
connection = cx_Oracle.connect(mode=cx_Oracle.SYSDBA)
cursor = connection.cursor()
cursor.execute("alter database mount")
cursor.execute("alter database open")
```

Note: This method is an extension to the DB API definition.

Connection.**stmtcachesize**

This read-write attribute specifies the size of the statement cache. This value can make a significant difference in performance (up to 100x) if you have a small number of statements that you execute repeatedly.

The default value is 20.

See *Statement Caching* for more information.

Note: This attribute is an extension to the DB API definition.

Connection.**subscribe**(*namespace=cx_Oracle.SUBSCR_NAMESPACE_DBCHANGE*, *protocol=cx_Oracle.SUBSCR_PROTO_OCI*, *callback=None*, *timeout=0*, *operations=OPCODE_ALLOPS*, *port=0*, *qos=0*, *ipAddress=None*, *groupingClass=0*, *groupingValue=0*, *groupingType=cx_Oracle.SUBSCR_GROUPING_TYPE_SUMMARY*, *name=None*, *clientInitiated=False*)

Return a new *subscription object* that receives notifications for events that take place in the database that match the given parameters.

The namespace parameter specifies the namespace the subscription uses. It can be one of *cx_Oracle.SUBSCR_NAMESPACE_DBCHANGE* or *cx_Oracle.SUBSCR_NAMESPACE_AQ*.

The protocol parameter specifies the protocol to use when notifications are sent. Currently the only valid value is *cx_Oracle.SUBSCR_PROTO_OCI*.

The callback is expected to be a callable that accepts a single parameter. A *message object* is passed to this callback whenever a notification is received.

The timeout value specifies that the subscription expires after the given time in seconds. The default value of 0 indicates that the subscription never expires.

The operations parameter enables filtering of the messages that are sent (insert, update, delete). The default value will send notifications for all operations. This parameter is only used when the namespace is set to *cx_Oracle.SUBSCR_NAMESPACE_DBCHANGE*.

The port parameter specifies the listening port for callback notifications from the database server. If not specified, an unused port will be selected by the Oracle Client libraries.

The qos parameter specifies quality of service options. It should be one or more of the following flags, OR'ed together: *cx_Oracle.SUBSCR_QOS_RELIABLE*, *cx_Oracle.SUBSCR_QOS_DEREG_NFY*, *cx_Oracle.SUBSCR_QOS_ROWIDS*, *cx_Oracle.SUBSCR_QOS_QUERY*, *cx_Oracle.SUBSCR_QOS_BEST_EFFORT*.

The ipAddress parameter specifies the IP address (IPv4 or IPv6) in standard string notation to bind for callback notifications from the database server. If not specified, the client IP address will be determined by the Oracle Client libraries.

The groupingClass parameter specifies what type of grouping of notifications should take place. Currently, if set, this value can only be set to the value *cx_Oracle.SUBSCR_GROUPING_CLASS_TIME*, which will

group notifications by the number of seconds specified in the groupingValue parameter. The groupingType parameter should be one of the values *cx_Oracle.SUBSCR_GROUPING_TYPE_SUMMARY* (the default) or *cx_Oracle.SUBSCR_GROUPING_TYPE_LAST*.

The name parameter is used to identify the subscription and is specific to the selected namespace. If the namespace parameter is *cx_Oracle.SUBSCR_NAMESPACE_DBCHANGE* then the name is optional and can be any value. If the namespace parameter is *cx_Oracle.SUBSCR_NAMESPACE_AQ*, however, the name must be in the format '<QUEUE_NAME>' for single consumer queues and '<QUEUE_NAME>:<CONSUMER_NAME>' for multiple consumer queues, and identifies the queue that will be monitored for messages. The queue name may include the schema, if needed.

The clientInitiated parameter is used to determine if client initiated connections or server initiated connections (the default) will be established. Client initiated connections are only available in Oracle Client 19.4 and Oracle Database 19.4 and higher.

New in version 6.4: The parameters ipAddress, groupingClass, groupingValue, groupingType and name were added.

New in version 7.3: The parameter clientInitiated was added.

Note: This method is an extension to the DB API definition.

Note: The subscription can be deregistered in the database by calling the function *unsubscribe()*. If this method is not called and the connection that was used to create the subscription is explicitly closed using the function *close()*, the subscription will not be deregistered in the database.

Connection.**tag**

This read-write attribute initially contains the actual tag of the session that was acquired from a pool by *SessionPool.acquire()*. If the connection was not acquired from a pool or no tagging parameters were specified (tag and matchanytag) when the connection was acquired from the pool, this value will be None. If the value is changed, it must be a string containing name=value pairs like "k1=v1;k2=v2".

If this value is not None when the connection is released back to the pool it will be used to retag the session. This value can be overridden in the call to *SessionPool.release()*.

Note: This attribute is an extension to the DB API definition.

New in version 7.1.

Connection.**tnsentry**

This read-only attribute returns the TNS entry of the database to which a connection has been established.

Note: This attribute is an extension to the DB API definition.

Connection.**unsubscribe**(*subscr*)

Unsubscribe from events in the database that were originally subscribed to using *subscribe()*. The connection used to unsubscribe should be the same one used to create the subscription, or should access the same database and be connected as the same user name.

New in version 6.4.

Connection.**username**

This read-only attribute returns the name of the user which established the connection to the database.

Note: This attribute is an extension to the DB API definition.

`Connection.`**`version`**
This read-only attribute returns the version of the database to which a connection has been established.

Note: This attribute is an extension to the DB API definition.

Note: If you connect to Oracle Database 18 or higher with client libraries 12.2 or lower that you will only receive the base version (such as 18.0.0.0.0) instead of the full version (18.3.0.0.0).

2.3 Cursor Object

`Cursor.`**`__enter__`**`()`
The entry point for the cursor as a context manager. It returns itself.

Note: This method is an extension to the DB API definition.

`Cursor.`**`__exit__`**`()`
The exit point for the cursor as a context manager. It closes the cursor.

Note: This method is an extension to the DB API definition.

`Cursor.`**`arraysize`**
This read-write attribute can be used to tune the number of rows internally fetched and buffered by internal calls to the database. The value can drastically affect the performance of a query since it directly affects the number of network round trips between Python and the database. For methods like *`fetchone()`* and *`fetchall()`* it does not change how many rows are returned to the application. For *`fetchmany()`* it is the default number of rows to fetch.

Due to the performance benefits, the default `Cursor.arraysize` is 100 instead of the 1 that the DB API recommends. This value means that 100 rows are fetched by each internal call to the database.

See *Tuning Fetch Performance* for more information.

`Cursor.`**`bindarraysize`**
This read-write attribute specifies the number of rows to bind at a time and is used when creating variables via *`setinputsizes()`* or *`var()`*. It defaults to 1 meaning to bind a single row at a time.

Note: The DB API definition does not define this attribute.

`Cursor.`**`arrayvar`**`(`*dataType*, *value*[, *size*]`)`
Create an array variable associated with the cursor of the given type and size and return a *variable object*. The value is either an integer specifying the number of elements to allocate or it is a list and the number of elements allocated is drawn from the size of the list. If the value is a list, the variable is also set with the contents of the list. If the size is not specified and the type is a string or binary, 4000 bytes is allocated. This is needed for passing

arrays to PL/SQL (in cases where the list might be empty and the type cannot be determined automatically) or returning arrays from PL/SQL.

Array variables can only be used for PL/SQL associative arrays with contiguous keys. For PL/SQL associative arrays with sparsely populated keys or for varrays and nested tables, the approach shown in this example needs to be used.

Note: The DB API definition does not define this method.

Cursor.**bindnames**()

Return the list of bind variable names bound to the statement. Note that a statement must have been prepared first.

Note: The DB API definition does not define this method.

Cursor.**bindvars**

This read-only attribute provides the bind variables used for the last execute. The value will be either a list or a dictionary depending on whether binding was done by position or name. Care should be taken when referencing this attribute. In particular, elements should not be removed or replaced.

Note: The DB API definition does not define this attribute.

Cursor.**callfunc**(*name*, *returnType*, *parameters=[]*, *keywordParameters={}*)

Call a function with the given name. The return type is specified in the same notation as is required by *setinputsizes()*. The sequence of parameters must contain one entry for each parameter that the function expects. Any keyword parameters will be included after the positional parameters. The result of the call is the return value of the function.

See *PL/SQL Stored Functions* for an example.

Note: The DB API definition does not define this method.

Note: If you intend to call *Cursor.setinputsizes()* on the cursor prior to making this call, then note that the first item in the parameter list refers to the return value of the function.

Cursor.**callproc**(*name*, *parameters=[]*, *keywordParameters={}*)

Call a procedure with the given name. The sequence of parameters must contain one entry for each parameter that the procedure expects. The result of the call is a modified copy of the input sequence. Input parameters are left untouched; output and input/output parameters are replaced with possibly new values. Keyword parameters will be included after the positional parameters and are not returned as part of the output sequence.

See *PL/SQL Stored Procedures* for an example.

Note: The DB API definition does not allow for keyword parameters.

Cursor.**close**()

Close the cursor now, rather than whenever __del__ is called. The cursor will be unusable from this point forward; an Error exception will be raised if any operation is attempted with the cursor.

`Cursor.`**`connection`**
This read-only attribute returns a reference to the connection object on which the cursor was created.

Note: This attribute is an extension to the DB API definition but it is mentioned in PEP 249 as an optional extension.

`Cursor.`**`description`**
This read-only attribute is a sequence of 7-item sequences. Each of these sequences contains information describing one result column: (name, type, display_size, internal_size, precision, scale, null_ok). This attribute will be None for operations that do not return rows or if the cursor has not had an operation invoked via the *`execute()`* method yet.

The type will be one of the *database type constants* defined at the module level.

`Cursor.`**`execute`**(*statement*[, *parameters*], ***keywordParameters*)
Execute a statement against the database. See *SQL Execution*.

Parameters may be passed as a dictionary or sequence or as keyword parameters. If the parameters are a dictionary, the values will be bound by name and if the parameters are a sequence the values will be bound by position. Note that if the values are bound by position, the order of the variables is from left to right as they are encountered in the statement and SQL statements are processed differently than PL/SQL statements. For this reason, it is generally recommended to bind parameters by name instead of by position.

Parameters passed as a dictionary are name and value pairs. The name maps to the bind variable name used by the statement and the value maps to the Python value you wish bound to that bind variable.

A reference to the statement will be retained by the cursor. If None or the same string object is passed in again, the cursor will execute that statement again without performing a prepare or rebinding and redefining. This is most effective for algorithms where the same statement is used, but different parameters are bound to it (many times). Note that parameters that are not passed in during subsequent executions will retain the value passed in during the last execution that contained them.

For maximum efficiency when reusing an statement, it is best to use the *`setinputsizes()`* method to specify the parameter types and sizes ahead of time; in particular, None is assumed to be a string of length 1 so any values that are later bound as numbers or dates will raise a TypeError exception.

If the statement is a query, the cursor is returned as a convenience to the caller (so it can be used directly as an iterator over the rows in the cursor); otherwise, `None` is returned.

Note: The DB API definition does not define the return value of this method.

`Cursor.`**`executemany`**(*statement*, *parameters*, *batcherrors=False*, *arraydmlrowcounts=False*)
Prepare a statement for execution against a database and then execute it against all parameter mappings or sequences found in the sequence parameters. See *Batch Statement Execution and Bulk Loading*.

The statement is managed in the same way as the *`execute()`* method manages it. If the size of the buffers allocated for any of the parameters exceeds 2 GB, you will receive the error "DPI-1015: array size of <n> is too large", where <n> varies with the size of each element being allocated in the buffer. If you receive this error, decrease the number of elements in the sequence parameters.

If there are no parameters, or parameters have previously been bound, the number of iterations can be specified as an integer instead of needing to provide a list of empty mappings or sequences.

When true, the batcherrors parameter enables batch error support within Oracle and ensures that the call succeeds even if an exception takes place in one or more of the sequence of parameters. The errors can then be retrieved using *`getbatcherrors()`*.

When true, the arraydmlrowcounts parameter enables DML row counts to be retrieved from Oracle after the method has completed. The row counts can then be retrieved using `getarraydmlrowcounts()`.

Both the batcherrors parameter and the arraydmlrowcounts parameter can only be true when executing an insert, update, delete or merge statement; in all other cases an error will be raised.

For maximum efficiency, it is best to use the `setinputsizes()` method to specify the parameter types and sizes ahead of time; in particular, None is assumed to be a string of length 1 so any values that are later bound as numbers or dates will raise a TypeError exception.

Cursor.**executemanyprepared**(*numIters*)

Execute the previously prepared and bound statement the given number of times. The variables that are bound must have already been set to their desired value before this call is made. This method was designed for the case where optimal performance is required as it comes at the expense of compatibility with the DB API.

Note: The DB API definition does not define this method.

Deprecated since version 6.4: Use `executemany()` instead with None for the statement argument and an integer for the parameters argument.

Cursor.**fetchall**()

Fetch all (remaining) rows of a query result, returning them as a list of tuples. An empty list is returned if no more rows are available. Note that the cursor's arraysize attribute can affect the performance of this operation, as internally reads from the database are done in batches corresponding to the arraysize.

An exception is raised if the previous call to `execute()` did not produce any result set or no call was issued yet.

See *Fetch Methods* for an example.

Cursor.**fetchmany**([*numRows=cursor.arraysize*])

Fetch the next set of rows of a query result, returning a list of tuples. An empty list is returned if no more rows are available. Note that the cursor's arraysize attribute can affect the performance of this operation.

The number of rows to fetch is specified by the parameter. If it is not given, the cursor's arraysize attribute determines the number of rows to be fetched. If the number of rows available to be fetched is fewer than the amount requested, fewer rows will be returned.

An exception is raised if the previous call to `execute()` did not produce any result set or no call was issued yet.

See *Fetch Methods* for an example.

Cursor.**fetchone**()

Fetch the next row of a query result set, returning a single tuple or None when no more data is available.

An exception is raised if the previous call to `execute()` did not produce any result set or no call was issued yet.

See *Fetch Methods* for an example.

Cursor.**fetchraw**([*numRows=cursor.arraysize*])

Fetch the next set of rows of a query result into the internal buffers of the defined variables for the cursor. The number of rows actually fetched is returned. This method was designed for the case where optimal performance is required as it comes at the expense of compatibility with the DB API.

An exception is raised if the previous call to `execute()` did not produce any result set or no call was issued yet.

Note: The DB API definition does not define this method.

Cursor.**fetchvars**
This read-only attribute specifies the list of variables created for the last query that was executed on the cursor. Care should be taken when referencing this attribute. In particular, elements should not be removed or replaced.

Note: The DB API definition does not define this attribute.

Cursor.**getarraydmlrowcounts**()
Retrieve the DML row counts after a call to *executemany()* with arraydmlrowcounts enabled. This will return a list of integers corresponding to the number of rows affected by the DML statement for each element of the array passed to *executemany()*.

Note: The DB API definition does not define this method and it is only available for Oracle 12.1 and higher.

Cursor.**getbatcherrors**()
Retrieve the exceptions that took place after a call to *executemany()* with batcherrors enabled. This will return a list of Error objects, one error for each iteration that failed. The offset can be determined by looking at the offset attribute of the error object.

Note: The DB API definition does not define this method.

Cursor.**getimplicitresults**()
Return a list of cursors which correspond to implicit results made available from a PL/SQL block or procedure without the use of OUT ref cursor parameters. The PL/SQL block or procedure opens the cursors and marks them for return to the client using the procedure dbms_sql.return_result. Cursors returned in this fashion should not be closed. They will be closed automatically by the parent cursor when it is closed. Closing the parent cursor will invalidate the cursors returned by this method.

New in version 5.3.

Note: The DB API definition does not define this method and it is only available for Oracle Database 12.1 (both client and server must be at this level or higher). It is most like the DB API method nextset(), but unlike that method (which requires that the next result set overwrite the current result set), this method returns cursors which can be fetched independently of each other.

Cursor.**inputtypehandler**
This read-write attribute specifies a method called for each value that is bound to a statement executed on the cursor and overrides the attribute with the same name on the connection if specified. The method signature is handler(cursor, value, arraysize) and the return value is expected to be a variable object or None in which case a default variable object will be created. If this attribute is None, the value of the attribute with the same name on the connection is used.

Note: This attribute is an extension to the DB API definition.

Cursor.**__iter__**()
Returns the cursor itself to be used as an iterator.

Note: This method is an extension to the DB API definition but it is mentioned in PEP 249 as an optional extension.

Cursor.**lastrowid**

This read-only attribute returns the rowid of the last row modified by the cursor. If no row was modified by the last operation performed on the cursor, the value None is returned.

New in version 7.3.

Cursor.**outputtypehandler**

This read-write attribute specifies a method called for each column that is to be fetched from this cursor. The method signature is handler(cursor, name, defaultType, length, precision, scale) and the return value is expected to be a variable object or None in which case a default variable object will be created. If this attribute is None, the value of the attribute with the same name on the connection is used instead.

See *Changing Fetched Data Types with Output Type Handlers*.

Note: This attribute is an extension to the DB API definition.

Cursor.**parse**(*statement*)

This can be used to parse a statement without actually executing it (this step is done automatically by Oracle when a statement is executed).

Note: The DB API definition does not define this method.

Note: You can parse any DML or DDL statement. DDL statements are executed immediately and an implied commit takes place.

Cursor.**prefetchrows**

This read-write attribute can be used to tune the number of rows that the Oracle Client library fetches when a query is executed. This value can reduce the number of round-trips to the database that are required to fetch rows but at the cost of additional memory. Setting this value to 0 can be useful when the timing of fetches must be explicitly controlled.

See *Tuning Fetch Performance* for more information.

Note: The DB API definition does not define this method.

Cursor.**prepare**(*statement*[, *tag*])

This can be used before a call to `execute()` to define the statement that will be executed. When this is done, the prepare phase will not be performed when the call to `execute()` is made with None or the same string object as the statement. If specified the statement will be returned to the statement cache with the given tag. See the Oracle documentation for more information about the statement cache.

See *Statement Caching* for more information.

Note: The DB API definition does not define this method.

Cursor.**rowcount**
: This read-only attribute specifies the number of rows that have currently been fetched from the cursor (for select statements), that have been affected by the operation (for insert, update, delete and merge statements), or the number of successful executions of the statement (for PL/SQL statements).

Cursor.**rowfactory**
: This read-write attribute specifies a method to call for each row that is retrieved from the database. Ordinarily a tuple is returned for each row but if this attribute is set, the method is called with the tuple that would normally be returned, and the result of the method is returned instead.

  See *Changing Query Results with Rowfactories*.

  Note: The DB API definition does not define this attribute.

Cursor.**scroll**(*value=0*, *mode="relative"*)
: Scroll the cursor in the result set to a new position according to the mode.

  If mode is "relative" (the default value), the value is taken as an offset to the current position in the result set. If set to "absolute", value states an absolute target position. If set to "first", the cursor is positioned at the first row and if set to "last", the cursor is set to the last row in the result set.

  An error is raised if the mode is "relative" or "absolute" and the scroll operation would position the cursor outside of the result set.

  New in version 5.3.

  Note: This method is an extension to the DB API definition but it is mentioned in PEP 249 as an optional extension.

Cursor.**scrollable**
: This read-write boolean attribute specifies whether the cursor can be scrolled or not. By default, cursors are not scrollable, as the server resources and response times are greater than nonscrollable cursors. This attribute is checked and the corresponding mode set in Oracle when calling the method `execute()`.

  New in version 5.3.

  Note: The DB API definition does not define this attribute.

Cursor.**setinputsizes**(**args*, ***keywordArgs*)
: This can be used before a call to `execute()`, `callfunc()` or `callproc()` to predefine memory areas for the operation's parameters. Each parameter should be a type object corresponding to the input that will be used or it should be an integer specifying the maximum length of a string parameter. Use keyword parameters when binding by name and positional parameters when binding by position. The singleton None can be used as a parameter when using positional parameters to indicate that no space should be reserved for that position.

  Note: If you plan to use `callfunc()` then be aware that the first parameter in the list refers to the return value of the function.

Cursor.**setoutputsize**(*size*[, *column*])
: This method does nothing and is retained solely for compatibility with the DB API. The module automatically allocates as much space as needed to fetch LONG and LONG RAW columns (or CLOB as string and BLOB as bytes).

Cursor.**statement**
This read-only attribute provides the string object that was previously prepared with *prepare()* or executed with *execute()*.

Note: The DB API definition does not define this attribute.

Cursor.**var**(*dataType*[, *size*, *arraysize*, *inconverter*, *outconverter*, *typename*, *encodingErrors*])
Create a variable with the specified characteristics. This method was designed for use with PL/SQL in/out variables where the length or type cannot be determined automatically from the Python object passed in or for use in input and output type handlers defined on cursors or connections.

The dataType parameter specifies the type of data that should be stored in the variable. This should be one of the *database type constants*, *DB API constants*, an object type returned from the method *Connection.gettype()* or one of the following Python types:

Python Type	Database Type
bool	*cx_Oracle.DB_TYPE_BOOLEAN*
bytes	*cx_Oracle.DB_TYPE_RAW*
datetime.date	*cx_Oracle.DB_TYPE_DATE*
datetime.datetime	*cx_Oracle.DB_TYPE_DATE*
datetime.timedelta	*cx_Oracle.DB_TYPE_INTERVAL_DS*
decimal.Decimal	*cx_Oracle.DB_TYPE_NUMBER*
float	*cx_Oracle.DB_TYPE_NUMBER*
int	*cx_Oracle.DB_TYPE_NUMBER*
str	*cx_Oracle.DB_TYPE_VARCHAR*

The size parameter specifies the length of string and raw variables and is ignored in all other cases. If not specified for string and raw variables, the value 4000 is used.

The arraysize parameter specifies the number of elements the variable will have. If not specified the bind array size (usually 1) is used. When a variable is created in an output type handler this parameter should be set to the cursor's array size.

The inconverter and outconverter parameters specify methods used for converting values to/from the database. More information can be found in the section on *variable objects*.

The typename parameter specifies the name of a SQL object type and must be specified when using type *cx_Oracle.OBJECT* unless the type object was passed directly as the first parameter.

The encodingErrors parameter specifies what should happen when decoding byte strings fetched from the database into strings. It should be one of the values noted in the builtin decode function.

Note: The DB API definition does not define this method.

2.4 Variable Objects

Note: The DB API definition does not define this object.

Variable.**actualElements**
This read-only attribute returns the actual number of elements in the variable. This corresponds to the number of

elements in a PL/SQL index-by table for variables that are created using the method *Cursor.arrayvar()*. For all other variables this value will be identical to the attribute *numElements*.

Variable.**bufferSize**
This read-only attribute returns the size of the buffer allocated for each element in bytes.

Variable.**getvalue**([*pos=0*])
Return the value at the given position in the variable. For variables created using the method *Cursor.arrayvar()* the value returned will be a list of each of the values in the PL/SQL index-by table. For variables bound to DML returning statements, the value returned will also be a list corresponding to the returned data for the given execution of the statement (as identified by the pos parameter).

Variable.**inconverter**
This read-write attribute specifies the method used to convert data from Python to the Oracle database. The method signature is converter(value) and the expected return value is the value to bind to the database. If this attribute is None, the value is bound directly without any conversion.

Variable.**numElements**
This read-only attribute returns the number of elements allocated in an array, or the number of scalar items that can be fetched into the variable or bound to the variable.

Variable.**outconverter**
This read-write attribute specifies the method used to convert data from the Oracle database to Python. The method signature is converter(value) and the expected return value is the value to return to Python. If this attribute is None, the value is returned directly without any conversion.

Variable.**setvalue**(*pos*, *value*)
Set the value at the given position in the variable.

Variable.**size**
This read-only attribute returns the size of the variable. For strings this value is the size in characters. For all others, this is same value as the attribute bufferSize.

Variable.**type**
This read-only attribute returns the type of the variable. This will be an *Oracle Object Type* if the variable binds Oracle objects; otherwise, it will be one of the *database type constants*.

Changed in version 8.0: Database type constants are now used when the variable is not used for binding Oracle objects.

Variable.**values**
This read-only attribute returns a copy of the value of all actual positions in the variable as a list. This is the equivalent of calling *getvalue()* for each valid position and the length will correspond to the value of the *actualElements* attribute.

2.5 SessionPool Object

Note: This object is an extension to the DB API.

Connection pooling in cx_Oracle is handled by SessionPool objects.

See *Connection Pooling* for information on connection pooling.

SessionPool.**acquire**(*user=None*, *password=None*, *cclass=None*, *purity=cx_Oracle.ATTR_PURITY_DEFAULT*, *tag=None*, *matchanytag=False*, *shardingkey=[]*, *supershardingkey=[]*)
Acquire a connection from the session pool and return a *connection object*.

If the pool is homogeneous, the user and password parameters cannot be specified. If they are, an exception will be raised.

The cclass parameter, if specified, should be a string corresponding to the connection class for database resident connection pooling (DRCP).

The purity parameter is expected to be one of *ATTR_PURITY_NEW*, *ATTR_PURITY_SELF*, or *ATTR_PURITY_DEFAULT*.

The tag parameter, if specified, is expected to be a string with name=value pairs like "k1=v1;k2=v2" and will limit the sessions that can be returned from a session pool unless the matchanytag parameter is set to True. In that case sessions with the specified tag will be preferred over others, but if no such sessions are available a session with a different tag may be returned instead. In any case, untagged sessions will always be returned if no sessions with the specified tag are available. Sessions are tagged when they are *released* back to the pool.

The shardingkey and supershardingkey parameters, if specified, are expected to be a sequence of values which will be used to identify the database shard to connect to. The key values can be strings, numbers, bytes or dates.

SessionPool.**busy**
This read-only attribute returns the number of sessions currently acquired.

SessionPool.**close**(*force=False*)
Close the session pool now, rather than when the last reference to it is released, which makes it unusable for further work.

If any connections have been acquired and not released back to the pool this method will fail unless the force parameter is set to True.

SessionPool.**drop**(*connection*)
Drop the connection from the pool which is useful if the connection is no longer usable (such as when the session is killed).

SessionPool.**dsn**
This read-only attribute returns the TNS entry of the database to which a connection has been established.

SessionPool.**homogeneous**
This read-write boolean attribute indicates whether the pool is considered homogeneous or not. If the pool is not homogeneous different authentication can be used for each connection acquired from the pool.

SessionPool.**increment**
This read-only attribute returns the number of sessions that will be established when additional sessions need to be created.

SessionPool.**max**
This read-only attribute returns the maximum number of sessions that the session pool can control.

SessionPool.**max_lifetime_session**
This read-write attribute returns the maximum length of time (in seconds) that a pooled session may exist. Sessions that are in use will not be closed. They become candidates for termination only when they are released back to the pool and have existed for longer than max_lifetime_session seconds. Note that termination only occurs when the pool is accessed. A value of 0 means that there is no maximum length of time that a pooled session may exist. This attribute is only available in Oracle Database 12.1.

New in version 5.3.

SessionPool.**min**
This read-only attribute returns the number of sessions with which the session pool was created and the minimum number of sessions that will be controlled by the session pool.

SessionPool.**name**
This read-only attribute returns the name assigned to the session pool by Oracle.

SessionPool.**opened**
This read-only attribute returns the number of sessions currently opened by the session pool.

SessionPool.**release**(*connection*, *tag=None*)
Release the connection back to the pool now, rather than whenever __del__ is called. The connection will be unusable from this point forward; an Error exception will be raised if any operation is attempted with the connection. Any cursors or LOBs created by the connection will also be marked unusable and an Error exception will be raised if any operation is attempted with them.

Internally, references to the connection are held by cursor objects, LOB objects, etc. Once all of these references are released, the connection itself will be released back to the pool automatically. Either control references to these related objects carefully or explicitly release connections back to the pool in order to ensure sufficient resources are available.

If the tag is not None, it is expected to be a string with name=value pairs like "k1=v1;k2=v2" and will override the value in the property *Connection.tag*. If either *Connection.tag* or the tag parameter are not None, the connection will be retagged when it is released back to the pool.

SessionPool.**stmtcachesize**
This read-write attribute specifies the size of the statement cache that will be used as the starting point for any connections that are created by the session pool. Once a connection is created, that connection's statement cache size can only be changed by setting the *Connection.stmtcachesize* attribute on the connection itself.

See *Statement Caching* for more information.

New in version 6.0.

SessionPool.**timeout**
This read-write attribute specifies the time (in seconds) after which idle sessions will be terminated in order to maintain an optimum number of open sessions. Note that termination only occurs when the pool is accessed. A value of 0 means that no idle sessions are terminated.

SessionPool.**tnsentry**
This read-only attribute returns the TNS entry of the database to which a connection has been established.

SessionPool.**username**
This read-only attribute returns the name of the user which established the connection to the database.

SessionPool.**wait_timeout**
This read-write attribute specifies the time (in milliseconds) that the caller should wait for a session to become available in the pool before returning with an error. This value is only used if the getmode parameter to *cx_Oracle.SessionPool()* was the value *cx_Oracle.SPOOL_ATTRVAL_TIMEDWAIT*.

New in version 6.4.

2.6 Subscription Object

Note: This object is an extension the DB API.

Subscription.**callback**
This read-only attribute returns the callback that was registered when the subscription was created.

Subscription.**connection**
This read-only attribute returns the connection that was used to register the subscription when it was created.

Subscription.**id**
This read-only attribute returns the value of REGID found in the database view

USER_CHANGE_NOTIFICATION_REGS or the value of REG_ID found in the database view USER_SUBSCR_REGISTRATIONS. For AQ subscriptions, the value is 0.

Subscription.**ipAddress**
This read-only attribute returns the IP address used for callback notifications from the database server. If not set during construction, this value is None.

New in version 6.4.

Subscription.**name**
This read-only attribute returns the name used to register the subscription when it was created.

New in version 6.4.

Subscription.**namespace**
This read-only attribute returns the namespace used to register the subscription when it was created.

Subscription.**operations**
This read-only attribute returns the operations that will send notifications for each table or query that is registered using this subscription.

Subscription.**port**
This read-only attribute returns the port used for callback notifications from the database server. If not set during construction, this value is zero.

Subscription.**protocol**
This read-only attribute returns the protocol used to register the subscription when it was created.

Subscription.**qos**
This read-only attribute returns the quality of service flags used to register the subscription when it was created.

Subscription.**registerquery**(*statement*[, *args*])
Register the query for subsequent notification when tables referenced by the query are changed. This behaves similarly to cursor.execute() but only queries are permitted and the args parameter must be a sequence or dictionary. If the qos parameter included the flag cx_Oracle.SUBSCR_QOS_QUERY when the subscription was created, then the ID for the registered query is returned; otherwise, None is returned.

Subscription.**timeout**
This read-only attribute returns the timeout (in seconds) that was specified when the subscription was created. A value of 0 indicates that there is no timeout.

2.6.1 Message Objects

Note: This object is created internally when notification is received and passed to the callback procedure specified when a subscription is created.

Message.**consumerName**
This read-only attribute returns the name of the consumer which generated the notification. It will be populated if the subscription was created with the namespace *cx_Oracle.SUBSCR_NAMESPACE_AQ* and the queue is a multiple consumer queue.

New in version 6.4.

Message.**dbname**
This read-only attribute returns the name of the database that generated the notification.

Message.**queries**
This read-only attribute returns a list of message query objects that give information about query result sets

changed for this notification. This attribute will be None if the qos parameter did not include the flag *SUBSCR_QOS_QUERY* when the subscription was created.

Message.**queueName**
This read-only attribute returns the name of the queue which generated the notification. It will only be populated if the subscription was created with the namespace *cx_Oracle.SUBSCR_NAMESPACE_AQ*.

New in version 6.4.

Message.**registered**
This read-only attribute returns whether the subscription which generated this notification is still registered with the database. The subscription is automatically deregistered with the database when the subscription timeout value is reached or when the first notification is sent (when the quality of service flag *cx_Oracle.SUBSCR_QOS_DEREG_NFY* is used).

New in version 6.4.

Message.**subscription**
This read-only attribute returns the subscription object for which this notification was generated.

Message.**tables**
This read-only attribute returns a list of message table objects that give information about the tables changed for this notification. This attribute will be None if the qos parameter included the flag *SUBSCR_QOS_QUERY* when the subscription was created.

Message.**txid**
This read-only attribute returns the id of the transaction that generated the notification.

Message.**type**
This read-only attribute returns the type of message that has been sent. See the constants section on event types for additional information.

2.6.2 Message Table Objects

Note: This object is created internally for each table changed when notification is received and is found in the tables attribute of message objects, and the tables attribute of message query objects.

MessageTable.**name**
This read-only attribute returns the name of the table that was changed.

MessageTable.**operation**
This read-only attribute returns the operation that took place on the table that was changed.

MessageTable.**rows**
This read-only attribute returns a list of message row objects that give information about the rows changed on the table. This value is only filled in if the qos parameter to the *Connection.subscribe()* method included the flag *SUBSCR_QOS_ROWIDS*.

2.6.3 Message Row Objects

Note: This object is created internally for each row changed on a table when notification is received and is found in the rows attribute of message table objects.

MessageRow.**operation**
: This read-only attribute returns the operation that took place on the row that was changed.

MessageRow.**rowid**
: This read-only attribute returns the rowid of the row that was changed.

2.6.4 Message Query Objects

Note: This object is created internally for each query result set changed when notification is received and is found in the queries attribute of message objects.

MessageQuery.**id**
: This read-only attribute returns the query id of the query for which the result set changed. The value will match the value returned by Subscription.registerquery when the related query was registered.

MessageQuery.**operation**
: This read-only attribute returns the operation that took place on the query result set that was changed. Valid values for this attribute are *EVENT_DEREG* and *EVENT_QUERYCHANGE*.

MessageQuery.**tables**
: This read-only attribute returns a list of message table objects that give information about the table changes that caused the query result set to change for this notification.

2.7 LOB Objects

See *Using CLOB and BLOB Data* for more information about using LOBs.

Note: This object is an extension the DB API. It is returned whenever Oracle CLOB, BLOB and BFILE columns are fetched.

LOB.**close**()
: Close the LOB. Call this when writing is completed so that the indexes associated with the LOB can be updated – but only if *open()* was called first.

LOB.**fileexists**()
: Return a boolean indicating if the file referenced by the BFILE type LOB exists.

LOB.**getchunksize**()
: Return the chunk size for the internal LOB. Reading and writing to the LOB in chunks of multiples of this size will improve performance.

LOB.**getfilename**()
: Return a two-tuple consisting of the directory alias and file name for a BFILE type LOB.

LOB.**isopen**()
: Return a boolean indicating if the LOB has been opened using the method *open()*.

LOB.**open**()
: Open the LOB for writing. This will improve performance when writing to a LOB in chunks and there are functional or extensible indexes associated with the LOB. If this method is not called, each write will perform an open internally followed by a close after the write has been completed.

LOB.**read**([*offset=1*[, *amount*]])
Return a portion (or all) of the data in the LOB object. Note that the amount and offset are in bytes for BLOB and BFILE type LOBs and in UCS-2 code points for CLOB and NCLOB type LOBs. UCS-2 code points are equivalent to characters for all but supplemental characters. If supplemental characters are in the LOB, the offset and amount will have to be chosen carefully to avoid splitting a character.

LOB.**setfilename**(*dirAlias*, *name*)
Set the directory alias and name of the BFILE type LOB.

LOB.**size**()
Returns the size of the data in the LOB object. For BLOB and BFILE type LOBs this is the number of bytes. For CLOB and NCLOB type LOBs this is the number of UCS-2 code points. UCS-2 code points are equivalent to characters for all but supplemental characters.

LOB.**trim**([*newSize=0*])
Trim the LOB to the new size.

LOB.**type**
This read-only attribute returns the type of the LOB as one of the *database type constants*.

New in version 8.0.

LOB.**write**(*data*[, *offset=1*])
Write the data to the LOB object at the given offset. The offset is in bytes for BLOB type LOBs and in UCS-2 code points for CLOB and NCLOB type LOBs. UCS-2 code points are equivalent to characters for all but supplemental characters. If supplemental characters are in the LOB, the offset will have to be chosen carefully to avoid splitting a character. Note that if you want to make the LOB value smaller, you must use the *trim()* function.

2.8 Object Type Objects

Note: This object is an extension to the DB API. It is returned by the *Connection.gettype()* call and is available as the *Variable.type* for variables containing Oracle objects.

ObjectType([*sequence*])
The object type may be called directly and serves as an alternative way of calling *newobject()*.

ObjectType.**attributes**
This read-only attribute returns a list of the *attributes* that make up the object type.

ObjectType.**iscollection**
This read-only attribute returns a boolean indicating if the object type refers to a collection or not.

ObjectType.**name**
This read-only attribute returns the name of the type.

ObjectType.**element_type**
This read-only attribute returns the type of elements found in collections of this type, if *iscollection* is `True`; otherwise, it returns `None`. If the collection contains objects, this will be another object type; otherwise, it will be one of the *database type constants*.

New in version 8.0.

ObjectType.**newobject**([*sequence*])
Return a new Oracle object of the given type. This object can then be modified by setting its attributes and then bound to a cursor for interaction with Oracle. If the object type refers to a collection, a sequence may be passed and the collection will be initialized with the items in that sequence.

ObjectType.**schema**
This read-only attribute returns the name of the schema that owns the type.

2.8.1 Object Objects

Note: This object is an extension to the DB API. It is returned by the *ObjectType.newobject()* call and can be bound to variables of type *OBJECT*. Attributes can be retrieved and set directly.

Object.**append**(*element*)
Append an element to the collection object. If no elements exist in the collection, this creates an element at index 0; otherwise, it creates an element immediately following the highest index available in the collection.

Object.**asdict**()
Return a dictionary where the collection's indexes are the keys and the elements are its values.

New in version 7.0.

Object.**aslist**()
Return a list of each of the collection's elements in index order.

Object.**copy**()
Create a copy of the object and return it.

Object.**delete**(*index*)
Delete the element at the specified index of the collection. If the element does not exist or is otherwise invalid, an error is raised. Note that the indices of the remaining elements in the collection are not changed. In other words, the delete operation creates holes in the collection.

Object.**exists**(*index*)
Return True or False indicating if an element exists in the collection at the specified index.

Object.**extend**(*sequence*)
Append all of the elements in the sequence to the collection. This is the equivalent of performing *append()* for each element found in the sequence.

Object.**first**()
Return the index of the first element in the collection. If the collection is empty, None is returned.

Object.**getelement**(*index*)
Return the element at the specified index of the collection. If no element exists at that index, an exception is raised.

Object.**last**()
Return the index of the last element in the collection. If the collection is empty, None is returned.

Object.**next**(*index*)
Return the index of the next element in the collection following the specified index. If there are no elements in the collection following the specified index, None is returned.

Object.**prev**(*index*)
Return the index of the element in the collection preceding the specified index. If there are no elements in the collection preceding the specified index, None is returned.

Object.**setelement**(*index*, *value*)
Set the value in the collection at the specified index to the given value.

Object.**size**()
Return the number of elements in the collection.

`Object.`**`trim`**`(num)`
Remove the specified number of elements from the end of the collection.

2.8.2 Object Attribute Objects

Note: This object is an extension to the DB API. The elements of *`ObjectType.attributes`* are instances of this type.

`ObjectAttribute.`**`name`**
This read-only attribute returns the name of the attribute.

`ObjectAttribute.`**`type`**
This read-only attribute returns the type of the attribute. This will be an *Oracle Object Type* if the variable binds Oracle objects; otherwise, it will be one of the *database type constants*.

New in version 8.0.

2.9 Advanced Queuing (AQ)

See *Oracle Advanced Queuing (AQ)* for more information about using AQ in cx_Oracle.

Note: All of these objects are extensions to the DB API.

2.9.1 Queues

These objects are created using the *`Connection.queue()`* method and are used to enqueue and dequeue messages.

`Queue.`**`connection`**
This read-only attribute returns a reference to the connection object on which the queue was created.

`Queue.`**`deqMany`**`(maxMessages)`
Dequeues up to the specified number of messages from the queue and returns a list of these messages. Each element of the returned list is a *message property* object.

`Queue.`**`deqOne`**`()`
Dequeues at most one message from the queue. If a message is dequeued, it will be a *message property* object; otherwise, it will be the value None.

`Queue.`**`deqOptions`**
This read-only attribute returns a reference to the *options* that will be used when dequeuing messages from the queue.

`Queue.`**`enqOne`**`(message)`
Enqueues a single message into the queue. The message must be a *message property* object which has had its payload attribute set to a value that the queue supports.

`Queue.`**`enqMany`**`(messages)`
Enqueues multiple messages into the queue. The messages parameter must be a sequence containing *message property* objects which have all had their payload attribute set to a value that the queue supports.

Warning: calling this function in parallel on different connections acquired from the same pool may fail due to Oracle bug 29928074. Ensure that this function is not run in parallel, use standalone connections or connections from different pools, or make multiple calls to *Queue.enqOne()* instead. The function *Queue.deqMany()* call is not affected.

Queue.**enqOptions**
This read-only attribute returns a reference to the *options* that will be used when enqueuing messages into the queue.

Queue.**name**
This read-only attribute returns the name of the queue.

Queue.**payloadType**
This read-only attribute returns the object type for payloads that can be enqueued and dequeued. If using a raw queue, this returns the value None.

2.9.2 Dequeue Options

Note: These objects are used to configure how messages are dequeued from queues. An instance of this object is found in the attribute *Queue.deqOptions*.

DeqOptions.**condition**
This attribute specifies a boolean expression similar to the where clause of a SQL query. The boolean expression can include conditions on message properties, user data properties and PL/SQL or SQL functions. The default is to have no condition specified.

DeqOptions.**consumername**
This attribute specifies the name of the consumer. Only messages matching the consumer name will be accessed. If the queue is not set up for multiple consumers this attribute should not be set. The default is to have no consumer name specified.

DeqOptions.**correlation**
This attribute specifies the correlation identifier of the message to be dequeued. Special pattern-matching characters, such as the percent sign (%) and the underscore (_), can be used. If multiple messages satisfy the pattern, the order of dequeuing is indeterminate. The default is to have no correlation specified.

DeqOptions.**deliverymode**
This write-only attribute specifies what types of messages should be dequeued. It should be one of the values *MSG_PERSISTENT* (default), *MSG_BUFFERED* or *MSG_PERSISTENT_OR_BUFFERED*.

DeqOptions.**mode**
This attribute specifies the locking behaviour associated with the dequeue operation. It should be one of the values *DEQ_BROWSE*, *DEQ_LOCKED*, *DEQ_REMOVE* (default), or *DEQ_REMOVE_NODATA*.

DeqOptions.**msgid**
This attribute specifies the identifier of the message to be dequeued. The default is to have no message identifier specified.

DeqOptions.**navigation**
This attribute specifies the position of the message that is retrieved. It should be one of the values *DEQ_FIRST_MSG*, *DEQ_NEXT_MSG* (default), or *DEQ_NEXT_TRANSACTION*.

DeqOptions.**transformation**
This attribute specifies the name of the transformation that must be applied after the message is dequeued from the database but before it is returned to the calling application. The transformation must be created using dbms_transform. The default is to have no transformation specified.

DeqOptions.**visibility**
: This attribute specifies the transactional behavior of the dequeue request. It should be one of the values *DEQ_ON_COMMIT* (default) or *DEQ_IMMEDIATE*. This attribute is ignored when using the *DEQ_BROWSE* mode. Note the value of *autocommit* is always ignored.

DeqOptions.**wait**
: This attribute specifies the time to wait, in seconds, for a message matching the search criteria to become available for dequeuing. One of the values *DEQ_NO_WAIT* or *DEQ_WAIT_FOREVER* can also be used. The default is *DEQ_WAIT_FOREVER*.

2.9.3 Enqueue Options

Note: These objects are used to configure how messages are enqueued into queues. An instance of this object is found in the attribute *Queue.enqOptions*.

EnqOptions.**deliverymode**
: This write-only attribute specifies what type of messages should be enqueued. It should be one of the values *MSG_PERSISTENT* (default) or *MSG_BUFFERED*.

EnqOptions.**transformation**
: This attribute specifies the name of the transformation that must be applied before the message is enqueued into the database. The transformation must be created using dbms_transform. The default is to have no transformation specified.

EnqOptions.**visibility**
: This attribute specifies the transactional behavior of the enqueue request. It should be one of the values *ENQ_ON_COMMIT* (default) or *ENQ_IMMEDIATE*. Note the value of *autocommit* is ignored.

2.9.4 Message Properties

Note: These objects are used to identify the properties of messages that are enqueued and dequeued in queues. They are created by the method *Connection.msgproperties()*. They are used by the methods *Queue.enqOne()* and *Queue.enqMany()* and returned by the methods *Queue.deqOne()* and *Queue.deqMany()*.

MessageProperties.**attempts**
: This read-only attribute specifies the number of attempts that have been made to dequeue the message.

MessageProperties.**correlation**
: This attribute specifies the correlation used when the message was enqueued.

MessageProperties.**delay**
: This attribute specifies the number of seconds to delay an enqueued message. Any integer is acceptable but the constant *MSG_NO_DELAY* can also be used indicating that the message is available for immediate dequeuing.

MessageProperties.**deliverymode**
: This read-only attribute specifies the type of message that was dequeued. It will be one of the values *MSG_PERSISTENT* or *MSG_BUFFERED*.

MessageProperties.**enqtime**
: This read-only attribute specifies the time that the message was enqueued.

MessageProperties.**exceptionq**
: This attribute specifies the name of the queue to which the message is moved if it cannot be processed successfully. Messages are moved if the number of unsuccessful dequeue attempts has exceeded the maximum number of retries or if the message has expired. All messages in the exception queue are in the *MSG_EXPIRED* state. The default value is the name of the exception queue associated with the queue table.

MessageProperties.**expiration**
: This attribute specifies, in seconds, how long the message is available for dequeuing. This attribute is an offset from the delay attribute. Expiration processing requires the queue monitor to be running. Any integer is accepted but the constant *MSG_NO_EXPIRATION* can also be used indicating that the message never expires.

MessageProperties.**msgid**
: This attribute specifies the id of the message in the last queue that generated this message.

MessageProperties.**payload**
: This attribute identifies the payload that will be enqueued or the payload that was dequeued when using a *queue*. When enqueuing, the value is checked to ensure that it conforms to the type expected by that queue. For RAW queues, the value can be a bytes object or a string. If the value is a string it will first be converted to bytes by encoding in the encoding identified by the attribute *Connection.encoding*.

MessageProperties.**priority**
: This attribute specifies the priority of the message. A smaller number indicates a higher priority. The priority can be any integer, including negative numbers. The default value is zero.

MessageProperties.**state**
: This read-only attribute specifies the state of the message at the time of the dequeue. It will be one of the values *MSG_WAITING*, *MSG_READY*, *MSG_PROCESSED* or *MSG_EXPIRED*.

2.10 SODA

Oracle Database Simple Oracle Document Access (SODA) allows documents to be inserted, queried, and retrieved from Oracle Database using a set of NoSQL-style cx_Oracle methods.

See *Simple Oracle Document Access (SODA)* for a cx_Oracle example.

SODA requires Oracle Client 18.3 or higher and Oracle Database 18.1 and higher. The role SODA_APP must be granted to the user.

See this tracking issue for known issues with SODA.

2.10.1 SODA Database Object

Note: This object is an extension the DB API. It is returned by the method *Connection.getSodaDatabase()*.

SodaDatabase.**createCollection**(*name*, *metadata=None*, *mapMode=False*)
: Creates a SODA collection with the given name and returns a new *SODA collection object*. If you try to create a collection, and a collection with the same name and metadata already exists, then that existing collection is opened without error.

  If metadata is specified, it is expected to be a string containing valid JSON or a dictionary that will be transformed into a JSON string. This JSON permits you to specify the configuration of the collection including storage options; specifying the presence or absence of columns for creation timestamp, last modified timestamp and version; whether the collection can store only JSON documents; and methods of key and version generation.

The default metadata creates a collection that only supports JSON documents and uses system generated keys. See this collection metadata reference for more information.

If the mapMode parameter is set to True, the new collection is mapped to an existing table instead of creating a table. If a collection is created in this way, dropping the collection will not drop the existing table either.

New in version 7.0.

`SodaDatabase.`**`createDocument`**(*content*, *key=None*, *mediaType="application/json"*)

Creates a *SODA document* usable for SODA write operations. You only need to use this method if your collection requires client-assigned keys or has non-JSON content; otherwise, you can pass your content directly to SODA write operations. SodaDocument attributes 'createdOn', 'lastModified' and 'version' will be None.

The content parameter can be a dictionary or list which will be transformed into a JSON string and then UTF-8 encoded. It can also be a string which will be UTF-8 encoded or it can be a bytes object which will be stored unchanged. If a bytes object is provided and the content is expected to be JSON, note that SODA only supports UTF-8, UTF-16LE and UTF-16BE encodings.

The key parameter should only be supplied if the collection in which the document is to be placed requires client-assigned keys.

The mediaType parameter should only be supplied if the collection in which the document is to be placed supports non-JSON documents and the content for this document is non-JSON. Using a standard MIME type for this value is recommended but any string will be accepted.

New in version 7.0.

`SodaDatabase.`**`getCollectionNames`**(*startName=None*, *limit=0*)

Returns a list of the names of collections in the database that match the criteria, in alphabetical order.

If the startName parameter is specified, the list of names returned will start with this value and also contain any names that fall after this value in alphabetical order.

If the limit parameter is specified and is non-zero, the number of collection names returned will be limited to this value.

New in version 7.0.

`SodaDatabase.`**`openCollection`**(*name*)

Opens an existing collection with the given name and returns a new *SODA collection object*. If a collection with that name does not exist, None is returned.

New in version 7.0.

2.10.2 SODA Collection Object

Note: This object is an extension the DB API. It is used to represent SODA collections and is created by methods `SodaDatabase.createCollection()` and `SodaDatabase.openCollection()`.

`SodaCollection.`**`createIndex`**(*spec*)

Creates an index on a SODA collection. The spec is expected to be a dictionary or a JSON-encoded string. See this overview for information on indexes in SODA.

Note that a commit should be performed before attempting to create an index.

New in version 7.0.

`SodaCollection.`**`drop`**()

Drops the collection from the database, if it exists. Note that if the collection was created with mapMode set to True the underlying table will not be dropped.

A boolean value is returned indicating if the collection was actually dropped.

New in version 7.0.

SodaCollection.**dropIndex**(*name*, *force=False*)
Drops the index with the specified name, if it exists.

The force parameter, if set to True, can be used to force the dropping of an index that the underlying Oracle Database domain index doesn't normally permit. This is only applicable to spatial and JSON search indexes. See here for more information.

A boolean value is returned indicating if the index was actually dropped.

New in version 7.0.

SodaCollection.**find**()
This method is used to begin an operation that will act upon documents in the collection. It creates and returns a *SodaOperation object* which is used to specify the criteria and the operation that will be performed on the documents that match that criteria.

New in version 7.0.

SodaCollection.**getDataGuide**()
Returns a *SODA document object* containing property names, data types and lengths inferred from the JSON documents in the collection. It can be useful for exploring the schema of a collection. Note that this method is only supported for JSON-only collections where a JSON search index has been created with the 'dataguide' option enabled. If there are no documents in the collection, None is returned.

New in version 7.0.

SodaCollection.**insertMany**(*docs*)
Inserts a list of documents into the collection at one time. Each of the input documents can be a dictionary or list or an existing *SODA document object*.

Note: This method requires Oracle Client 18.5 and higher and is available only as a preview.

New in version 7.2.

SodaCollection.**insertManyAndGet**(*docs*)
Similarly to `insertMany()` this method inserts a list of documents into the collection at one time. The only difference is that it returns a list of *SODA Document objects*. Note that for performance reasons the returned documents do not contain the content.

Note: This method requires Oracle Client 18.5 and higher.

New in version 7.2.

SodaCollection.**insertOne**(*doc*)
Inserts a given document into the collection. The input document can be a dictionary or list or an existing *SODA document object*.

New in version 7.0.

SodaCollection.**insertOneAndGet**(*doc*)
Similarly to `insertOne()` this method inserts a given document into the collection. The only difference is that it returns a *SODA Document object*. Note that for performance reasons the returned document does not contain the content.

New in version 7.0.

SodaCollection.**metadata**
: This read-only attribute returns a dictionary containing the metadata that was used to create the collection. See this collection metadata reference for more information.

  New in version 7.0.

SodaCollection.**name**
: This read-only attribute returns the name of the collection.

  New in version 7.0.

SodaCollection.**save**(*doc*)
: Saves a document into the collection. This method is equivalent to *insertOne()* except that if client-assigned keys are used, and the document with the specified key already exists in the collection, it will be replaced with the input document.

  New in version 8.0.

SodaCollection.**saveAndGet**(*doc*)
: Saves a document into the collection. This method is equivalent to *insertOneAndGet()* except that if client-assigned keys are used, and the document with the specified key already exists in the collection, it will be replaced with the input document.

  New in version 8.0.

SodaCollection.**truncate**()
: Removes all of the documents in the collection, similarly to what is done for rows in a table by the TRUNCATE TABLE statement.

  New in version 8.0.

2.10.3 SODA Document Object

Note: This object is an extension the DB API. It is returned by the methods *SodaDatabase.createDocument()*, *SodaOperation.getDocuments()* and *SodaOperation.getOne()* as well as by iterating over *SODA document cursors*.

SodaDoc.**createdOn**
: This read-only attribute returns the creation time of the document in ISO 8601 format. Documents created by *SodaDatabase.createDocument()* or fetched from collections where this attribute is not stored will return None.

  New in version 7.0.

SodaDoc.**getContent**()
: Returns the content of the document as a dictionary or list. This method assumes that the content is application/json and will raise an exception if this is not the case. If there is no content, however, None will be returned.

  New in version 7.0.

SodaDoc.**getContentAsBytes**()
: Returns the content of the document as a bytes object. If there is no content, however, None will be returned.

  New in version 7.0.

SodaDoc.**getContentAsString**()
: Returns the content of the document as a string. If the document encoding is not known, UTF-8 will be used. If there is no content, however, None will be returned.

New in version 7.0.

SodaDoc.**key**
: This read-only attribute returns the unique key assigned to this document. Documents created by *SodaDatabase.createDocument()* may not have a value assigned to them and return None.

  New in version 7.0.

SodaDoc.**lastModified**
: This read-only attribute returns the last modified time of the document in ISO 8601 format. Documents created by *SodaDatabase.createDocument()* or fetched from collections where this attribute is not stored will return None.

  New in version 7.0.

SodaDoc.**mediaType**
: This read-only attribute returns the media type assigned to the document. By convention this is expected to be a MIME type but no checks are performed on this value. If a value is not specified when calling *SodaDatabase.createDocument()* or the document is fetched from a collection where this component is not stored, the string "application/json" is returned.

  New in version 7.0.

SodaDoc.**version**
: This read-only attribute returns the version assigned to this document. Documents created by *SodaDatabase.createDocument()* or fetched from collections where this attribute is not stored will return None.

  New in version 7.0.

2.10.4 SODA Document Cursor Object

Note: This object is an extension the DB API. It is returned by the method *SodaOperation.getCursor()* and implements the iterator protocol. Each iteration will return a *SODA document object*.

SodaDocCursor.**close**()
: Close the cursor now, rather than whenever __del__ is called. The cursor will be unusable from this point forward; an Error exception will be raised if any operation is attempted with the cursor.

  New in version 7.0.

2.10.5 SODA Operation Object

Note: This object is an extension to the DB API. It represents an operation that will be performed on all or some of the documents in a SODA collection. It is created by the method *SodaCollection.find()*.

SodaOperation.**count**()
: Returns a count of the number of documents in the collection that match the criteria. If *skip()* or *limit()* were called on this object, an exception is raised.

  New in version 7.0.

SodaOperation.**fetchArraySize**(*value*)
: This is a tuning method to specify the number of documents that are internally fetched in batches by calls to

getCursor() and *getDocuments()*. It does not affect how many documents are returned to the application. A value of 0 will use the default value (100). This method is only available in Oracle Client 19.5 and higher.

As a convenience, the SodaOperation object is returned so that further criteria can be specified by chaining methods together.

New in version 8.0.

SodaOperation.**filter**(*value*)
Sets a filter specification for complex document queries and ordering of JSON documents. Filter specifications must be provided as a dictionary or JSON-encoded string and can include comparisons, regular expressions, logical and spatial operators, among others. See the overview of SODA filter specifications for more information.

As a convenience, the SodaOperation object is returned so that further criteria can be specified by chaining methods together.

New in version 7.0.

SodaOperation.**getCursor**()
Returns a *SODA Document Cursor object* that can be used to iterate over the documents that match the criteria.

New in version 7.0.

SodaOperation.**getDocuments**()
Returns a list of *SODA Document objects* that match the criteria.

New in version 7.0.

SodaOperation.**getOne**()
Returns a single *SODA Document object* that matches the criteria. Note that if multiple documents match the criteria only the first one is returned.

New in version 7.0.

SodaOperation.**key**(*value*)
Specifies that the document with the specified key should be returned. This causes any previous calls made to this method and *keys()* to be ignored.

As a convenience, the SodaOperation object is returned so that further criteria can be specified by chaining methods together.

New in version 7.0.

SodaOperation.**keys**(*seq*)
Specifies that documents that match the keys found in the supplied sequence should be returned. This causes any previous calls made to this method and *key()* to be ignored.

As a convenience, the SodaOperation object is returned so that further criteria can be specified by chaining methods together.

New in version 7.0.

SodaOperation.**limit**(*value*)
Specifies that only the specified number of documents should be returned. This method is only usable for read operations such as *getCursor()* and *getDocuments()*. For write operations, any value set using this method is ignored.

As a convenience, the SodaOperation object is returned so that further criteria can be specified by chaining methods together.

New in version 7.0.

SodaOperation.**remove**()

Removes all of the documents in the collection that match the criteria. The number of documents that have been removed is returned.

New in version 7.0.

SodaOperation.**replaceOne**(*doc*)

Replaces a single document in the collection with the specified document. The input document can be a dictionary or list or an existing *SODA document object*. A boolean indicating if a document was replaced or not is returned.

Currently the method `key()` must be called before this method can be called.

New in version 7.0.

SodaOperation.**replaceOneAndGet**(*doc*)

Similarly to `replaceOne()`, this method replaces a single document in the collection with the specified document. The only difference is that it returns a *SODA document object*. Note that for performance reasons the returned document does not contain the content.

New in version 7.0.

SodaOperation.**skip**(*value*)

Specifies the number of documents that match the other criteria that will be skipped. This method is only usable for read operations such as `getCursor()` and `getDocuments()`. For write operations, any value set using this method is ignored.

As a convenience, the SodaOperation object is returned so that further criteria can be specified by chaining methods together.

New in version 7.0.

SodaOperation.**version**(*value*)

Specifies that documents with the specified version should be returned. Typically this is used with `key()` to implement optimistic locking, so that the write operation called later does not affect a document that someone else has modified.

As a convenience, the SodaOperation object is returned so that further criteria can be specified by chaining methods together.

New in version 7.0.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

O

P

Q

R

S

T

U

V

W