eGenix.com

mxODBC

ODBC Database Interface
for Python

Version 3.3

Printed in Germany.

Contents

1. Introduction

mxODBC has proven to be the most stable and versatile ODBC interface available for Python. It has been in active use for more than a decade and is actively maintained by eGenix.com to meet the requirements of modern database applications which our customers have built on top of mxODBC.

This manual will give you an in-depth overview of mxODBC's capabilities and features. It is written as technical manual, so background in Python and database programming is needed.

mxODBC tries to hide many of the complicated details of the ODBC specification from the user, but does provide access to many of the introspection APIs defined in that standard. If you don't need introspection for your applications, you can easily make use of mxODBC without any further knowledge of the underlying ODBC interface.

1.1 Technical Overview

The mxODBC package provides a *Python Database API 2.0* compliant interface to databases that are accessible via the ODBC application programming interface (API). Since ODBC is the de-facto standard for connecting to databases, this allows connecting Python to most available databases on the market today.

Accessing the databases can be done through an ODBC manager, e.g. the ODBC manager that comes with Windows, *iODBC* or *unixODBC* which are free ODBC managers available for Unix, or the DataDirect ODBC manager, which is a proprietary ODBC manager for Unix.

The package supports parallel database interfacing, meaning that you can access multiple different databases from within one process, e.g. one database through the iODBC manager and another through unixODBC.

mxODBC is customizable to many different needs via configuration attributes, e.g. you can use the Python datetime module or the *eGenix.xom mxDateTime* package for handling date/time value, eliminating the problems you often face when handling dates before 1.1.1970 and after 2038.

It also supports the Python decimal module, long integer interfacing, Unicode, large binary and text data, as well as stored procedures, prepared statements, database introspection and in-flight customization of connections and cursors.

1.2 Features

- **Python Database API 2.0 Compliance**: the mxODBC API is fully *Python DB-API 2.0* compatible and implements a large number of powerful extensions.

- **Support for all recent ODBC Version** : mxODBC works with ODBC drivers implementing the ODBC version specifications 2.0 - 3.8.

- **Uses ODBC 3 APIs**: provided the ODBC driver/manager is capable of using ODBC 3 APIs, mxODBC will use them for added efficiency.

- **32-bit and 64-bit ODBC**: mxODBC supports both 32-bit and 64-bit versions of the ODBC standard - including special 64-bit builds on Unix.

- **Supports all major ODBC driver managers:** mxODBC can work with the MS ODBC Driver Manager on Windows, unixODBC, iODBC and the DataDirect ODBC Driver Manager on Unix and the Mac OS X ODBC Driver Manager on Mac OS X. If needed, multiple ODBC managers can be used at the same time, giving you full flexibility.

- **Stable, robust and reliable**: the mxODBC API has been in active production use since 1997.

- **Stored Procedure / Function Calls:** mxODBC support calling stored procedures and functions, using output and input/output parameters, as well as result sets for passing back data to Python.

- **ODBC Catalog & Introspection Functions**: mxODBC Client API provides methods e.g. to list tables, find column specifications, query index relationships, etc.

- **Support for Multiple Result Sets**: call stored procedures and access all returned result sets using an easy to API. Easily free up resources in case result sets are no longer needed.

- **Support for custom Row objects:** in addition to using standard Python tuples, mxODBC can automatically return result set rows as custom objects. mxODBC comes with a set of optimized row factories for: TupleRows, ListRows and NamespaceRows. All of these provide both index and attribute access to row column fields.

- **Dynamic ODBC Configuration**: adjust ODBC connection parameters dynamically, e.g. set timeouts, read-only access, auto-commit, etc.

- **Many useful DB-API Extensions:**

    - **Adjustable .paramstyle:** mxODBC supports both the ODBC question mark positional parameter binding style as well as the named parameter styles used by e.g. Oracle.

  - **cursor.scroll()** to scroll the cursor in result sets without actually fetching data.
  - **cursor.prepare()** to prepare SQL statements for execution, without actually running them. This allows creating pools of cursors for dedicated purposes.
  - **connection.autocommit** to easily turn on/off the ODBC autocommit feature
  - **cursor** and **connection objects usable as context managers**
  - **cursor.executemany() accepts iterators/generators** as parameter "sequence".
  - **cursor.cursortype** to easily adjust the used ODBC cursor type to your application's needs.
  - **ODBC cursor/connection option methods** to adjust ODBC cursors/connections to your application's needs and optimize performance by e.g. declaring a connection read-only.

- **Configurable Data Type Mappings**:
  - Supports Python type binding and Database type binding for efficient data exchange.
  - Supports **mxDateTime** and **Python's time and datetime modules** for date/time value exchange.
  - Supports standard Python floats, integers, longs and **Python's decimal module** for loss-less numeric value exchange.
  - Supports **Python 2.7 memoryview objects**.
  - Automatically handles and **supports unknown data types and user data types** via string conversion.

- **Full Unicode Support**: use Unicode for managing text data in your client applications - even if the database does not natively support Unicode, mxODBC will automatically provide the necessary conversions on-the-fly. mxODBC supports both the Unicode or the ANSI ODBC APIs. You can chose the optimal approach for your driver.

- **Multi-Version Python Support**: mxODBC works with **Python 2.4, 2.5, 2.6 and 2.7**.[1]

- **Full 64-bit Support:** mxODBC runs on the following 64-bit platforms natively: Windows, Linux, FreeBSD and Mac OS X.

[1] Please note that mxODBC 3.3 will be the last release to support Python 2.4, 2.5 and 2.6. The next release will only support Python 2.7 and Python 3.

- **Highly Portable Codebase**: in addition to the already supported platforms for mxODBC, eGenix.com provides *custom porting services* for more exotic platforms.

- **Easy installation**: using Windows installers, .egg file package or our Python distutils compatible prebuilt Python packages.

- **Easy configuration**: use ODBC manager GUI tools for easy configuration of ODBC data sources, then access these data sources by name from Python, or use a connection-less way to connect to databases by specifying the driver name and database details in the application.

1.3 Requirements

mxODBC needs these environment on Windows, Unix or Mac OS X for successful installation:

Windows

- All Windows platforms starting with Windows 2000 are supported.

- Python 2.4 or later needs to be installed and working.

- The Windows version of the mxODBC uses the Windows ODBC manager as ODBC manager, so you have to configure your ODBC data sources using its GUI interface which is available through the system settings folder. Alternatively, you can choose to use a DSN-less setup which defines all connection details in the connection string.

- You should setup at least one configured and running ODBC data source for testing purposes.

Unix

- SuSE and RedHat Linux distributions for x86 and x86_64 (AMD64/EM64T) processors, FreeBSD and Sun Solaris are supported Unix platforms. eGenix.com can also provide custom builds for other Unix platforms on request. Please write to *sales@egenix.com* for details.

- Python 2.4 or later needs to be installed and working.

- On Linux, FreeBSD and Solaris, the binary package includes support for the iODBC, the unixODBC and the DataDirect managers. You must have at least one of these installed in order to be able to connect to ODBC data sources. Please use the ODBC manager GUI interfaces to configure the data sources. Alternatively, you can choose to use a DSN-less setup which defines all connection details in the connection string.

`mx.ODBC.Manager` prefers iODBC over unixODBC over DataDirect if more than one ODBC driver manager is installed.

- You should setup at least one configured and running ODBC data source for testing purposes.

Mac OS X

- Mac OS X 10.4/10.5 Intel and PPC 32-bit and Mac OS X 10.6 Intel 64-bit are supported.

- Python 2.4 or later needs to be installed and working.

- Mac OS X uses a variant of iODBC as system ODBC manager. On Mac OS X 10.4 and 10.5 this comes pre-installed with the system. On Mac OS X 10.6, the ODBC manager is available from Apple as separate download. Please use the ODBC manager GUI interfaces to configure the data sources. Alternatively, you can choose to use a DSN-less setup which defines all connection details in the connection string.

- If you want to use the unixODBC manager from *MacPorts* instead of the system iODBC manager, you first have to install unixODBC from the *MacPorts* and then tell the Mac OS X linker where to find the ports libraries by adjusting the environment variable DYLD_LIBRARY_PATH prior to starting Python:

  ```
  export DYLD_LIBRARY_PATH=/opt/local/lib
  ```

- You should setup at least one configured and running ODBC data source for testing purposes.

2. Installation

The mxODBC database package is distributed as add-on for the eGenix.com mx Base Distribution (`egenix-mx-base`).

Please visit the *eGenix.com web-site* to download the latest versions of both the eGenix.com mx Base Distribution and the eGenix.com mxODBC distribution for your platform and Python version.

> **IMPORTANT NOTE:**
>
> Before installing the `egenix-mxodbc` package, you will have to install the `egenix-mx-base` distribution which contains packages needed by mxODBC.
>
> Even though both distributions use the same installation procedure, please refer to the `egenix-mx-base` installation instructions on how to install that package.

2.1 Download the Software

2.1.1 Automatic download

If you want to use .egg package archives for the mxODBC installation, package tools such as *easy_install* or *zc.buildout* will download the archives automatically from a special package index on the eGenix.com website.

A separate **manual download is normally not needed**. However, you can still download the files manually and point the package tools directly at the downloaded .egg package files, if needed. This may be needed in case the package tools cannot determine which .egg package files to download.

For installation using Windows installers or our distutils compatible prebuilt package format, you will also have to manually download the files.

2.1.2 Manual Download

You can download the binary archives (Windows installers, .egg files or prebuilt archives) for your combination of platform, Python version and Unicode variant from the eGenix.com web-site at *http://www.egenix.com/*.

> Please make sure that you download the right version for your Python installation. If you get import errors or notices of failed initialization, you likely have the wrong product version installed.

These parameters make a difference:

Operating System Platform

All recent versions of these operating systems are supported:

- Windows
- Linux
- Mac OS X
- FreeBSD

Please check the *eGenix.com web-site* for the detailed list of available downloads for these platforms.

If your platform is not among those listed above or on the web-site, eGenix also provides custom porting services to have mxODBC ported to your platform. Please write to *sales@egenix.com* for details.

Python Build Version

To check which Python version you are using, startup the Python interpreter using the –V option:

```
python -V
```

This will print out the Python version number.

mxODBC supports Python versions 2.4 - 2.7 on most platforms.

Python Build Architecture (32 bit or 64 bit)

On most platforms, eGenix.com supports x86 32-bit and x86_64 (AMD64/EM64T) 64-bit versions of Python.

To find out which Python version you are using, run the following command:

```
python -c 'import struct; print struct.calcsize("P")*8,"bit"'
```

This will print out “32 bit” or “64 bit”.

Unicode Variant (UCS2 or UCS4)

On Unix and Mac OS X, Python can be built using two different Unicode variants: UCS2 and UCS4. Windows builds are always UCS2 builds.

To find out which variant your Python version was compiled with, run the following command:

```
python -c 'print "UCS%s"%len(u"x".encode("unicode-internal"))'
```

This will either print out “UCS2” or “UCS4”.

2.2 Installation using Windows installers

The installers provided by eGenix.com for use on Windows only include the `mx.ODBC.Windows` subpackage of mxODBC. This subpackage interfaces directly to the Microsoft ODBC Manager, so you can use all available Windows system tools to configure your ODBC data sources.

2.2.1 Prerequisites

- Please make sure that you have a working installation of the `egenix-mx-base` distribution prior to continuing with the installation of the `egenix-mxodbc` add-on. You can easily check this by checking the Windows Software Setup dialog for an entry of the form "Python x.x eGenix.com mx Base Distribution" or by running the following at the command prompt:

  ```
  python -c "import mx.DateTime"
  ```

  If you get an import error, please visit the *eGenix.com web-site* and install the `egenix-mx-base` package first.

- You will need ODBC drivers for all databases you wish to connect to. Windows comes with a very complete set of such drivers, but if you can't find the driver you are looking for, have a look at 15. Hints & Links to other Resources.

2.2.2 Before You Start

Upgrading

When upgrading from a previous version of mxODBC, you can normally install the new version in place of the previous one. If you want to be extra careful, you can also uninstall the previous version using the standard Windows software setup tool. See 2.2.4 Uninstall for details.

If you used a different packaging format for installing the previous version, please see the relevant installation section of this guide for instructions on how to uninstall that variant.

License Files

In order to use mxODBC, you will need license files from eGenix.com.

If you want to test the product before buying it, you can request evaluation licenses via the eGenix.com web-site at *http://www.egenix.com/*.

When buying licenses from the eGenix.com online shop (*http://shop.egenix.com/*), you will receive the license files immediately after purchase.

In both cases, the license files are sent to the email address you specified during the purchase process or from which you wrote the evaluation license request in form of a ZIP license archive attached to the license email – usually named licenses.zip.

The license archive licenses.zip contains one subdirectory per license you bought. The directories are named after the license key for each license. A typical license archive will have these contents:

```
2100-8789-0322-0926-2568-6429/mxodbc_license.py
2100-8789-0322-0926-2568-6429/mxodbc_license.txt
2100-8089-0312-0926-2668-6529/mxodbc_license.py
2100-8089-0312-0926-2668-6529/mxodbc_license.txt
```

(in the above example, the license archive contains the files for two product licenses).

In order to install the license files, please unzip the license archive to a temporary directory.

In order for mxODBC to pick up the correct license files, please copy them to a location on your `sys.path` or `PYTHONPATH`. If you installed Python to e.g. C:\Python27, the typical location for installation of the license files would be C:\Python27\Lib\site-packages\.

Use the following command to see the `sys.path` that your Python version uses:

```
python -c "import sys; print ':'.join(sys.path)"
```

2.2.3 Step-by-step Installation Guide

Step 1

After you have downloaded the Windows installer of the `egenix-mxodbc` distribution, double-click on the .exe file to start the installer.

> **Note:**
> Depending on your Python installation, you may need admin privileges on Windows to successfully complete the installation.

Step 2

The installer will then ask you to accept the license, choose the Python version and then to start the install process.

If the listbox showing the installed Python versions is empty, it is likely that you have chosen the wrong Windows installer for your Python version. Please go back to the eGenix.com web-site and download the correct version for the installed Python version.

Step 3

In case you are upgrading to a new mxODBC version, the installer will ask you whether you want to overwrite existing files. Answer "yes" to this question. It is safe to allow the installer overwrite files.

The installer will then install all the needed files. Note that it does not setup any links on the desktop or in the start menu.

Step 4

Test the installation by trying to import mxODBC:

```
$ python

>>> import mx.ODBC.Manager
>>>
```

If you don't get any ImportError, you have successfully installed mxODBC.

2.2.4 Uninstall

The Windows installer will automatically register the installed software with the standard Windows software setup tool.

To uninstall the distribution, run the Windows Software Setup tool and select the "Python x.x eGenix mxODBC x.x" entry for deinstallation.

This will uninstall all files that can safely be removed from the system. It will not remove files which were added to the subpackages after installation, nor will it remove the license files you manually installed.

2.3 Installation using egg package archives

We assume that you have already have *setuptools* and *easy_install* installed in your Python installation. The examples in this section refer to a Unix or Mac OS X installation, but it is also possible to install .egg packages on Windows.

You can check this by searching for *easy_install* in the directory where you've installed the Python interpreter binary:

```
python -c "import setuptools; print 'setuptools installed'"
```

If this reports an ImportError, you don't have *setuptools* installed. In that case, please see the next section.

Note that you can also use the information from this section to create a *zc.buildout* setup.

Setuptools

In order to be able to install eggs, you need to install a Python package called *setuptools*.

To get this package installed, download the file *ez_setup.py* from the URL *http://peak.telecommunity.com/dist/ez_setup.py* and run it using the Python interpreter that you will be using with mxODBC:

```
python ez_setup.py
```

This will install setuptools into your Python site-packages/ directory as well as a script called *easy_install* in your bin/ directory. The `easy_install` script is later need to install the mxODBC.

2.3.2 Before You Start

Upgrading

When upgrading from a previous version of mxODBC, you should uninstall the *egenix-mxodbc* package first. Please see section 2.3.4 Uninstall for instructions.

License Files

In order to use mxODBC, you will need license files from eGenix.com.

If you want to test the product before buying it, you can request evaluation licenses via the eGenix.com web-site at *http://www.egenix.com/*.

When buying licenses from the eGenix.com online shop (*http://shop.egenix.com/*), you will receive the license files immediately after purchase.

In both cases, the license files are sent to the email address you specified during the purchase process or from which you wrote the evaluation license request in form of a ZIP license archive attached to the license email – usually named licenses.zip.

The license archive licenses.zip contains one subdirectory per license you bought. The directories are named after the license key for each license. A typical license archive will have these contents:

```
2100-8789-0322-0926-2568-6429/mxodbc_license.py
2100-8789-0322-0926-2568-6429/mxodbc_license.txt
2100-8089-0312-0926-2668-6529/mxodbc_license.py
2100-8089-0312-0926-2668-6529/mxodbc_license.txt
```

(in the above example, the license archive contains the files for two product licenses).

In order to install the license files, please unzip the license archive to a temporary directory.

In order for mxODBC to pick up the correct license files, please copy them to a location on your `sys.path` or `PYTHONPATH`. If you installed Python to e.g. C:\Python27 on Windows or /usr/local/bin/python on Unix or Mac OS X, the typical location for installation of the license files would be C:\Python27\Lib\site-packages\ or /usr/local/lib/python2.7/site-packages/.

Use the following command to see the `sys.path` that your Python version uses:

```
python -c "import sys; print ';'.join(sys.path)"
```

2.3.3 Step-by-step Installation Guide

Step 1

Determine whether you are using a UCS2 or UCS4 build of Python (Windows users always need the UCS2 version, Mac OS X should also try the UCS2 version first, Unix users will most likely need the UCS4 version).

To find out which variant your Python version was compiled with, run the following command:

```
python -c 'print("UCS%s"%len(u"x".encode("unicode-internal")))'
```

This will either print out “UCS2” or “UCS4”.

Step 2

Next, install the *egenix-mxodbc* egg package in your Python installation.

> Note that you may need to have *admin* or *root* privileges in order to successfully complete the following step, unless you are using a *virtualenv-based setup*.

If you got **UCS2** in step 1, run the following command using the `easy_install` script from the Python installation you intend to use:

```
easy_install -i http://downloads.egenix.com/python/index/ucs2/ \
             egenix-mxodbc
```

If you got **UCS4** in step 1, use this command:

```
easy_install -i http://downloads.egenix.com/python/index/ucs4/ \
             egenix-mxodbc
```

If you **manually downloaded** the egg archive from the eGenix.com website to a temporary directory, pass the file name directly to `easy_install` to start the installation:

```
easy_install \
    /path-to-egg-file/egenix_mxodbc-3.3.0-py2.7-win32.egg
```

(replace the file name with the one of the file you downloaded)

After installation, the egg file can be removed from the temporary directory without causing harm.

For more information on how to use `easy_install`, please see the *easy_install documentation*.

Step 3

Now that you have installed the product code, you need to install the proper licenses in order for the mxODBC to startup correctly.

Go to the temporary directory where you unzipped the license archive and change to the license subdirectory which contains the license for the installation you are currently working on.

Copy the two files mxodbc_license.py and mxodbc_license.txt from the license subdirectory to the Python site-packages/ directory.

Step 4

Test the installation by trying to import mxODBC:

```
$ python

>>> import mx.ODBC.Manager
>>>
```

If you don't get any ImportError, you have successfully installed mxODBC.

2.3.4 Uninstall

Since *setuptools* doesn't provide an uninstall command you have to manually remove the installation:

1. remove the egenix-mxodbc.* egg directory from your Python site-packages/ directory and

2. edit the file easy-install.pth in that directory to remove the corresponding egg entry.

2.4 Installation using prebuilt package archives

Prebuilt package archives are standard distutils source distribution packages, which have been built on the respective platforms without performing the installation step. The source parts are also removed from those packages.

You can think of the prebuilt packages as source package installations that were frozen just before running the install command.

When installing these , you just need to run the last step after unpacking the package: the distutils `install` command.

No additional software is needed to install these packages.

2.4.1 Before You Start

Upgrading

When upgrading from a previous version of mxODBC, you can either install the new version over the previous version or first uninstall the previous version. See section 2.4.3 Uninstall for instructions.

License Files

In order to use mxODBC, you will need license files from eGenix.com.

If you want to test the product before buying it, you can request evaluation licenses via the eGenix.com web-site at *http://www.egenix.com/*.

When buying licenses from the eGenix.com online shop (*http://shop.egenix.com/*), you will receive the license files immediately after purchase.

In both cases, the license files are sent to the email address you specified during the purchase process or from which you wrote the evaluation license request in form of a ZIP license archive attached to the license email – usually named licenses.zip.

The license archive licenses.zip contains one subdirectory per license you bought. The directories are named after the license key for each license. A typical license archive will have these contents:

```
2100-8789-0322-0926-2568-6429/mxodbc_license.py
2100-8789-0322-0926-2568-6429/mxodbc_license.txt
2100-8089-0312-0926-2668-6529/mxodbc_license.py
2100-8089-0312-0926-2668-6529/mxodbc_license.txt
```

(in the above example, the license archive contains the files for two product licenses).

In order to install the license files, please unzip the license archive to a temporary directory.

In order for mxODBC to pick up the correct license files, please copy them to a location on your `sys.path` or `PYTHONPATH`. If you installed Python to e.g. C:\Python27 on Windows or /usr/local/bin/python on Unix or Mac OS X, the typical location for installation of the license files would be C:\Python27\Lib\site-packages\ or /usr/local/lib/python2.7/site-packages/.

Use the following command to see the `sys.path` that your Python version uses:

```
python -c "import sys; print ';'.join(sys.path)"
```

2.4.2 Step-by-step Installation Guide

Step 1

Determine whether you are using a UCS2 or UCS4 build of Python (Windows users always need the UCS2 version, Mac OS X should also try the UCS2 version first, Unix users will most likely need the UCS4 version).

To find out which variant your Python version was compiled with, run the following command:

```
python -c 'print("UCS%s"%len(u"x".encode("unicode-internal")))'
```

This will either print out “UCS2” or “UCS4”.

Step 2

Next, install the *egenix-mxodbc* prebuilt package in your Python installation.

Note that you may need to have *admin* or *root* privileges in order to successfully complete the following step, unless you are using a *virtualenv-based setup*.

1. First, unzip the downloaded prebuilt package archive to a temporary directory.

2. Then run the following command using the Python installation you intend to use in the package directory egenix-mxodbc-3.3.*/:

```
python setup.py install
```

Note that you can use the standard distutils install command options, e.g. to install the package to a different prefix (using `--prefix`) or a home directory (using `--home`). For more information on the available options, please have a look at the distutils *install command documentation*.

After installation, you can remove the temporary directory without causing harm. Please keep the prebuilt package archive around, in case you want to uninstall the package again.

Step 3

Now that you have installed the product code, you need to install the proper licenses in order for the mxODBC to startup correctly.

Go to the temporary directory where you unzipped the license archive and change to the license subdirectory which contains the license for the installation you are currently working on.

Copy the two files mxodbc_license.py and mxodbc_license.txt from the license subdirectory to the Python site-packages/ directory.

Step 4

Test the installation by trying to import mxODBC:

```
$ python

>>> import mx.ODBC.Manager
>>>
```

If you don't get any ImportError, you have successfully installed mxODBC.

2.4.3 Uninstall

Automatic Uninstall

In order to uninstall the mxODBC package, run the setup.py of the installation package using the `uninstall` command and the same options you passed to the install command when you installed the package:

```
python setup.py uninstall
```

Manual Uninstall

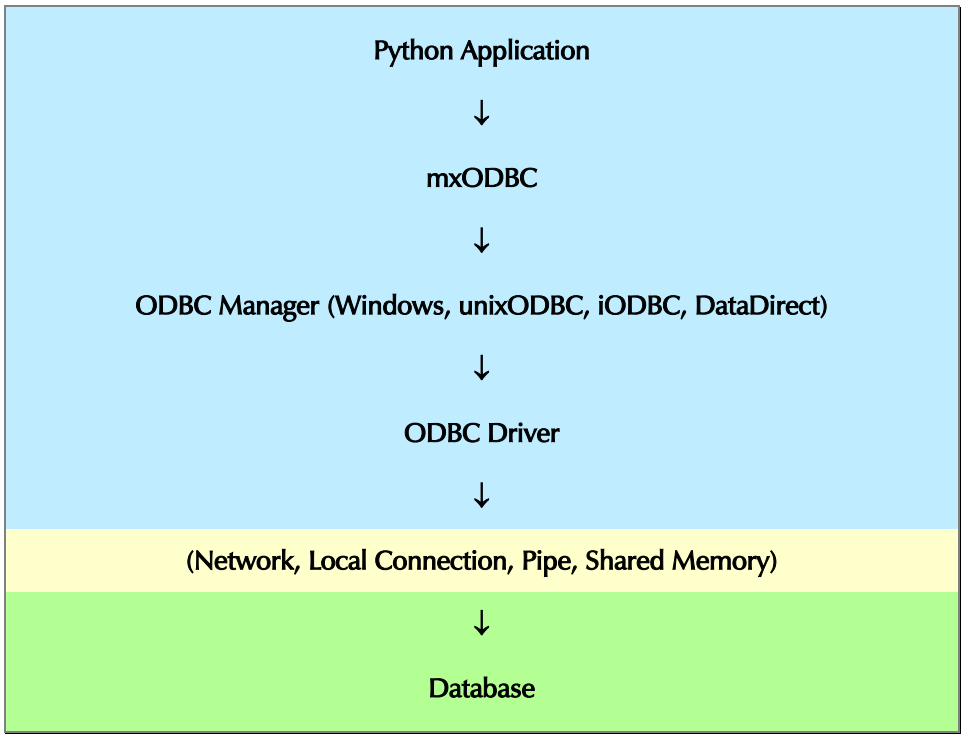
If you no longer have the older installation package, just remove the site-packages/mx/ODBC directory with all its subdirectories.

3. Access Databases using mxODBC

mxODBC provides an easy to use way of accessing the ODBC API of ODBC managers and drivers from Python. Together with a suitable ODBC driver installed on the machine where you are running the Python application, you can connect to your databases with a single Python call.

3.1 ODBC Application Stack

The typical ODBC application setup looks like this:



The upper blue part in the diagram executes within the process of the Python application. The green part usually runs in a separate process and possibly also on a different machine.

3.1.1 Architecture: 32-bit vs. 64-bit

As a result of this process setup outlined in the previous section, it is important that you choose the right ODBC driver type for your application:

- If you are running a **64-bit Python application**, you will also have to have a 64-bit ODBC manager and ODBC driver installed.

- If you are running a **32-bit Python application**, you need an 32-bit ODBC manager and ODBC driver.

Note that the ODBC manager may be capable of translating 32-bit or 64-bit function calls to whatever the ODBC driver supports (this is called *thunking*). Please check the documentation of your ODBC manager for details.

3.2 Accessing Databases from Windows

Most database ship with ODBC drivers for Windows, so setting up database access for Python applications on Windows is fairly straight forward.

Once you've installed the ODBC drivers on the machine you are running your Python application on, you will need to setup an *ODBC Data Source*. This can be done using the *ODBC Manager on Windows*.

To avoid problems with system permissions, eGenix.com recommends setting up System Data Sources, as these are usually accessible by all accounts on a Windows machine.

Using the mxODBC connection constructor `mx.ODBC.Windows.DriverConnect()` you can then setup a connection to the database.

3.2.1 Looking for Windows ODBC Drivers ?

Microsoft supports a whole range of (desktop) ODBC drivers for various databases and file formats. These are available under the name "ODBC Desktop Database Drivers" (search the MS web-site for the exact URL) [wx1350.exe] and also included in the more up-to-date "Microsoft Data Access Components" (MDAC) archive [mdac_typ.exe].

It includes ODBC drivers for: Access, dBase, Excel, Oracle, Paradox, Text (flat file CSV), FoxPro, MS SQL Server.

If you need to connect to databases running on other hosts, please contact the database vendor or check the *SQLSummit list of ODBC drivers*.

3.2.2 Installing Windows ODBC Drivers

Please consult the documentation of your database for ODBC driver installation instructions. These are usually installed in the same way as any other application on Windows, but their respective setup wizards and options are usually different in layout and depend on the target database.

3.2.3 Setting up an ODBC Data Source

Data sources are setup using the *Windows ODBC Manager* on Windows. This can be found in the in the *Control Panel* as *Administrative Tools* and is called *Data Sources (ODBC)*. See the *Windows ODBC documentation* for details.

Starting the ODBC manager will bring up a dialog with tabs for data source creation, ODBC tracing and connection pooling as well as a few information tabs showing the versions of installed drivers.

ODBC on 64-bit Windows Versions

On 64-bit versions of Windows, there are two separate ODBC managers which also keep and manage different lists of ODBC data sources: the 64-bit version can be found in C:\windows\system32\odbcad32.exe, the 32-bit version is located in C:\windows\sysWOW64\odbcad32.exe.

64-bit applications can only use the 64-bit ODBC manager and drivers, whereas 32-bit applications can only use the 32-bit ODBC manager and drivers.

> Please make sure that you use the right variant of the ODBC manager when configuring ODBC data sources on 64-bit Windows. If not, you will get errors from the ODBC manager mentioning problems in finding the given ODBC data source or an architecture mismatch, e.g. `[Microsoft][ODBC Driver Manager] The specified DSN contains an architecture mismatch between the Driver and Application`

3.2.4 ODBC Configuration Files

> On Windows, you should always use the Windows ODBC Manager to configure ODBC data sources.

Even though the Windows ODBC Manager also exposes the standard ODBC configuration files C:\Windows\ODBC.INI and C:\Windows\ODBCINST.INI, these do not contain the full configuration information, since this is stored in the Windows registry.

ODBC.INI - ODBC Data Source Configuration

This INI-file provides data source information using one INI-section per data source.

In addition to the data source sections, there are also a number of higher-level sections:

`[ODBC]`

This section is used to configure driver related ODBC manager settings such as ODBC call tracing. The settings in this section apply to all data sources.

`[ODBC Data Sources]`

This section contains one entry per configured data source, mapping the data source name to a description.

ODBCINST.INI - ODBC Driver Configuration

This INI-file provides one INI-section per installed ODBC driver.

In addition to the data source sections, there are also a number of higher-level sections:

`[ODBC Drivers]`

This section contains one entry per configured and installed ODBC driver, mapping the driver name to the string "Installed".

Windows Registry Keys

The following registry keys provide the ODBC configuration information in much the same way the formerly used INI-files did:

`HKEY_LOCAL_MACHINE\Software\ODBC\ODBC.INI`

One entry per INI-section of the ODBC.INI file, used for system-wide settings and system DSNs. See section 3.2.5 Available Data Source Types (DSNs) for details on the different DSN types.

`HKEY_LOCAL_MACHINE\Software\ODBC\ODBCINST.INI`

One entry per INI-section of the ODBCINST.INI file, used for managing the installed ODBC drivers.

`HKEY_CURRENT_USER\Software\ODBC\ODBC.INI`

One entry per INI-section of the ODBC.INI file, used for user specific settings and user DSNs. See section 3.2.5 Available Data Source Types (DSNs) for details on the different DSN types.

3.2.5 Available Data Source Types (DSNs)

There are three kinds of data sources that you can install on Windows machines:

1. *User Data Sources (User-DSN)*
2. *System Data Sources (System-DSN)*
3. *File Data Sources (File-DSN)*

User Data Sources (User-DSN)

User data sources are only visible to the user creating them. Other users normally do not have access to these data source definitions.

When running an application that is meant to run as service, you have to make sure that you create the user data source under the user name of the service.

If you intend a data source to be available for all users, or to avoid permissions problems, creating a system data source is a better option.

When connecting to a user DSN, you have to specify the DSN name as part of the connection string of `mx.ODBC.Windows.DriverConnect()` using the form `"DSN=mydsn"`. If you use the `mx.ODBC.Windows.Connect`() API to connect, pass the DSN name as first parameter.

System Data Sources (System-DSN)

System data sources are available to all users of the system. This is the recommended setup, if you run services that need to access the data sources from more than just one account.

When connecting to a system DSN, you have to specify the DSN name as part of the connection string of `mx.ODBC.Windows.DriverConnect()` using the form `"DSN=mydsn"`. If you use the `mx.ODBC.Windows.Connect`() API to connect, pass the DSN name as first parameter.

File Data Sources (File-DSN)

File data sources are special in the sense that they store the data source connection information and options in a dedicated file rather than in the registry.

This can be useful if you want to manage data sources across many servers and keep the data source files on a central file server.

You create such DSN files using the ODBC manager.

In order to connect to such a data source, you have to use the `mx.ODBC.Windows.DriverConnect()` API and provide a `"FILEDSN=c:\myfile.dsn"` entry instead of the usual `"DSN=mydsn"` as part for the connection string.

See the *ODBC File Data Source documentation* for more details.

3.2.6 DSN-less Connections

If you don't want to bother setting up a data source in the ODBC manager, you can also use a DSN-less connection setup.

Pros and Cons of using DSN-less Connections

These setups include all required driver and connection information in the connection string itself. All connection information is thus under the control of the application, without any system ODBC manager being aware of the connection setup.

This has both up- and downsides. The most important downside is that changes to the server system can no longer be administered through the ODBC manager, but instead have to be repeated in each application using a DSN-less setup. Even you upgrade an ODBC driver to a newer version, you may have to change all DSN-less connection setups due to changes in the ODBC driver name.

> We recommend to only use DSN-less setups if absolutely necessary, or in cases where access to the ODBC configuration files is otherwise not possible.

DNS-less Connection String

A DNS-less connection provides all configuration information you'd normally configure in the ODBC manager for a data source. Instead of a DSN name, you provide a textual representation of the driver name enclosed in curly brackets, e.g.

```
"Driver={MySQL ODBC 3.51 Driver}; Server=mysql.example.net;
Database=mydb"
```

The name given in curly brackets must match the driver name as listed in the ODBC manager (under *Name* on the *Driver* tab). The ODBC manager will then map the name to the registered driver.

A `"DSN=mydsn"` entry in the connection string is no longer needed.

For a list of common DSN-less connection strings, have a look at the *ConnectionStrings.com* website.

In order to connect to a database using a DSN-less connection string, you simply pass the string to the `mx.ODBC.Windows.DriverConnect()` API.

3.3 Accessing Databases from Unix

mxODBC is often used to access databases across a network. A very typical use case is that of connecting to MS SQL Server, Oracle or DB2 from a Unix machine.

eGenix.com has collected some information in the next section 4. Accessing Popular Databases which may help you in finding the right solution for this kind of setup. We recommend that you always use an ODBC manager on Unix to access these driver setups, e.g. unixODBC, iODBC, or the DataDirect ODBC manager.

Once you've installed the ODBC drivers on the machine you are running your Python application on, you will need to setup an *ODBC Data Source*. This can be done using the ODBC manager GUIs which try to mimic the Windows ODBC Manager, or using a text editor by editing either the system wide ODBC configuration files (usually /etc/odbc.ini and /etc/odbcinst.ini; Mac OS X uses /Library/ODBC/odbc.ini and /Library/ODBC/odbcinst.ini) or the user home directory versions (usually ~/.odbc.ini and ~/.odbcinst.ini).

To avoid problems with system permissions, eGenix.com recommends setting up data sources as System Data Sources using the GUI tools or in the system configuration file /etc/odbc.ini using a text editor, as these are usually accessible by all accounts on a Unix server.

Using the mxODBC connection constructor `mx.ODBC.Manager.DriverConnect()` you can then setup a connection to the database.

3.3.1 Looking for Unix ODBC Drivers ?

Many database vendors also provide ODBC for various Unix platforms. If you are looking for Linux drivers, the situation has cleared up a lot in recent years. On other platforms such as Mac OS X, AIX, Solaris or the BSDs, the situation is a lot less encouraging.

If you have trouble finding a suitable driver, you can contact the database vendor or check the *SQLSummit list of ODBC drivers*.

eGenix.com also provides a generic solution to such problems in form of the *mxODBC Connect* product, which helps you work around the problem of finding a suitable ODBC driver for the client platform. See the next section for detail.

3.3.2 mxODBC Connect - a general purpose client-server solution

Since finding good quality ODBC drivers for Unix platforms is sometimes difficult and managing them on all client systems can introduce quite a bit of

administrative overhead, eGenix has developed a general solution to the problem for Python applications, the client-server product called *eGenix mxODBC Connect*.

eGenix mxODBC Connect provides a highly portable client side Python interface module *mxODBC Connect Client* which connects to a server side service application called *mxODBC Connect Server*.

On the client side, the mxODBC Connect Client provides an interface which is almost fully compatible to the standard mxODBC Python interface, so you can easily port applications using mxODBC or other DB-API compatible adapters to the mxODBC Connect Client.

On the server side, the mxODBC Connect Server takes care of managing the incoming network connections from the mxODBC Connect Clients and interfaces directly to the database using an ODBC driver on the server machine. Since the mxODBC Connect Server is typically installed on the database server itself, the ODBC driver can communicate with the database using low-level and high performance interfaces such as shared memory, pipes, etc.

Using the mxODBC Connect product, you no longer need to search and install ODBC drivers for all your client platforms. Instead, you just need one ODBC driver installation on the server which is then shared by all clients.

The mxODBC Connect also provides better performance, since it doesn't require as many network roundtrips to the server as a low-level ODBC driver on the client side needs in order to provide the database connectivity.

Please see our website *http://www.egenix.com/* for more information on *mxODBC Connect*.

3.3.3 Installing Unix ODBC Drivers

Please consult the documentation of your database for ODBC driver installation instructions.

These often have to be installed by running a shell script or using the system packaging manager.

There are no standards for the driver directory location. Some drivers install in /opt, others in /usr/local, yet others can be unzipped anywhere in the system, provided the linker is setup to find the driver files.

3.3.4 Setting up an ODBC Data Source

Data sources are setup using the ODBC manager GUI tools on Unix or by editing the respective ODBC configuration files.

The GUI tools can typically be found in the System part of the menu.

For more details description please see the ODBC manager manuals:

- *unixODBC User Manual*
- *iODBC User Manual*
- *DataDirect User Manual*
- *Mac OS X ODBC Administrator Manual*[2]

Since the layout and operation of these tools is often similar to the Windows ODBC manager, you can also have a look at the *Windows ODBC documentation* for details.

Starting the ODBC manager will bring up a dialog with tabs for data source creation, ODBC tracing and connection pooling as well as a few information tabs showing the versions of installed drivers.

3.3.5 ODBC Configuration Files

> On Windows, you should always use the Windows ODBC Manager to configure ODBC data sources.

Even though the Windows ODBC Manager also exposes the standard ODBC configuration files C:\Windows\ODBC.INI and C:\Windows\ODBCINST.INI, these do not contain the full configuration information, since this is stored in the Windows registry.

/etc/odbc.ini - System ODBC Data Source Configuration

Depending on your ODBC manager installation or OS, the file may also reside in a different directory. Please consult your ODBC manager documentation for details. On Mac OS X, the file is located in /Library/ODBC/odbc.ini.

This INI-file provides data source information using one INI-section per data source.

In addition to the data source sections, there are also a number of higher-level sections:

`[ODBC]`

This section is used to configure driver related ODBC manager settings such as ODBC call tracing. The settings in this section apply to all data sources.

[2] For Mac OS X 10.6 (Snow Leopard) you may have to install the ODBC Administrator separately. It is available from Apple as disk image: *http://support.apple.com/kb/DL895*

> iODBC needs this section in the odbc.ini file. unixODBC in the odbcinst.ini file. More recent DataDirect ODBC manager versions accept the section in both files, older version need it in the odbc.ini file.

`[ODBC Data Sources]`

This section contains one entry per configured data source, mapping the data source name to a description.

Example:

```
[ODBC]
Trace = 0
TraceFile = /tmp/odbc.log

[ODBC Data Sources]
sybasease12 = Sybase ASE 12 on sybasease12.example.net

[sybasease12]
Driver          = /opt/sybase/DataAccess/ODBC/lib/libsybdrvodb.so
Description     = Adaptive Server Enterprise
Server          = sybasease12.egenix.internal
Port            = 5000
Database        = mydb
TextSize        = 10000000
#UseCursor       = 1
FileUsage       = -1
Trace           = 0
TraceFile       = /tmp/sybase.log
```

~/.odbc.ini - User ODBC Data Source Configuration

This INI-file provides data source information on a per user basis. It uses the same structure as the system wide /etc/odbc.ini file.

The data sources defined in this file are only visible to the user account for which it is defined.

/etc/odbcinst.ini - System ODBC Driver Configuration

Depending on your ODBC manager installation or OS, the file may also reside in a different directory. Please consult your ODBC manager documentation for details. On Mac OS X, the file is located in /Library/ODBC/odbcinst.ini.

This INI-file provides one INI-section per installed ODBC driver.

In addition to the data source sections, there are also a number of higher-level sections:

`[ODBC]`

This section is used to configure driver related ODBC manager settings such as ODBC call tracing. The settings in this section apply to all data sources.

> iODBC needs this section in the odbc.ini file. unixODBC in the odbcinst.ini file. More recent DataDirect ODBC manager versions accept the section in both files, older version need it in the odbc.ini file.

`[ODBC Drivers]`

This section contains one entry per configured and installed ODBC driver, mapping the driver name to the string "Installed".

Example:

```
[ODBC]
Trace = 0
TraceFile = /tmp/odbc.log

[ODBC Drivers]
OracleInstantClient = Installed

[OracleInstantClient]
Description = Oracle 11g ODBC Driver
Driver = /usr/local/oracle/instantclient_11_2/libsqora.so.11.1
Setup =
FileUsage =
CPTimeout =
CPReuse =
```

~/.odbcinst.ini - User ODBC Driver Configuration

This INI-file provides ODBC driver information on a per user basis. It uses the same structure as the system wide /etc/odbcinst.ini file.

The drivers defined in this file are only visible to the user account for which it is defined.

Environment Variables: ODBCINI and ODBCINSTINI

In order to override the default search path used by the ODBC manager for the above configuration files, the ODBC managers honor a few environment variables which can be used to direct them to specific alternate files:

`ODBCINI`

This environment is used by the ODBC manager to find the ODBC data source configuration file, if set.

`ODBCINSTINI`

This environment is used by the ODBC manager to find the ODBC driver configuration file, if set.

> Note that some driver manager do not support this environment variable: unixODBC and iODBC support the variable, the DataDirect ODBC manager doesn't.

3.3.6 Available Data Source Types (DSNs)

There are three kinds of data sources that you can install on Windows machines:

1. *User Data Sources (User-DSN)*
2. *System Data Sources (System-DSN)*
3. *File Data Sources (File-DSN)*

User Data Sources (User-DSN)

User data sources are only visible to the user creating them. Other users normally do not have access to these data source definitions.

The user DSNs can be defined via the ODBC manager GUI administration tools or by editing the user ODBC configuration file ~/.odbc.ini. See section 3.3.5 ODBC Configuration Files for details.

When running an application that is meant to run as service or daemon, you have to make sure that you create the user data source under the user name of the service or daemon.

If you intend a data source to be available for all users, or to avoid permissions problems, creating a system data source is a better option.

When connecting to a user DSN, you have to specify the DSN name as part of the connection string of `mx.ODBC.Manager.DriverConnect()` using the form `"DSN=mydsn"`. If you use the `mx.ODBC.Manager.Connect`() API to connect, pass the DSN name as first parameter.

System Data Sources (System-DSN)

System data sources are available to all users of the system. This is the recommended setup, if you run services that need to access the data sources from more than just one account.

Systems DSNs can be defined via the ODBC manager GUI administration tools or by editing the system ODBC configuration file /etc/odbc.ini. See section 3.3.5 ODBC Configuration Files for details.

When connecting to a system DSN, you have to specify the DSN name as part of the connection string of `mx.ODBC.Windows.DriverConnect()` using the form `"DSN=mydsn"`. If you use the `mx.ODBC.Windows.Connect`() API to connect, pass the DSN name as first parameter.

File Data Sources (File-DSN)

File data sources are special in the sense that they store the data source connection information and options in a dedicated file rather than in the registry.

This can be useful if you want to manage data sources across many servers and keep the data source files on a central file server.

You create such DSN files using the ODBC manager (if supported) or by using a text editor.

A DSN file uses the same syntax as the ODBC connection strings, with the difference that the file must start with the line `[ODBC]` and each keyword-value pair must be on a separate line, e.g. postgresql.dns:

```
[ODBC]
Driver = /usr/local/postgresql/lib/psqlodbcw.so
Database = mydb
ServerName = postgresql.example.net
Port = 5432
#Debug = 0
#Optimizer = 0
#CommLog = 0
#ReadOnly = 0
#SSLmode = require
ByteaAsLongVarBinary = 1
TextAsLongVarchar = 1
```

Please see the *FILEDSN MS Knowledge-Base article 165866* for details regarding the file format.

In order to connect to such a data source, you have to use the `mx.ODBC.Manager.DriverConnect()` API and provide a `"FILEDSN=/etc/postgresql.dns"` entry instead of the usual `"DSN=mydsn"` as part for the connection string.

See the *ODBC File Data Source documentation* for more details.

3.3.7 DSN-less Connections

If you don't want to bother setting up a data source in the ODBC manager, you can also use a DSN-less connection setup.

Pros and Cons of using DSN-less Connections

These setups include all required driver and connection information in the connection string itself. All connection information is thus under the control of the application, without any system ODBC manager being aware of the connection setup.

This has both up- and downsides. The most important downside is that changes to the server system can no longer be administered through the ODBC manager, but instead have to be repeated in each application using a DSN-less setup. Even

you upgrade an ODBC driver to a newer version, you may have to change all DSN-less connection setups due to changes in the ODBC driver name.

> We recommend to only use DSN-less setups if absolutely necessary or in cases where access to the ODBC configuration files is otherwise not possible.

DNS-less Connection String

A DNS-less connection provides all configuration information you'd normally place into the ~/.odbc.ini file, including a textual representation of the driver location (based on the name used in ~/.odbcinst.ini), e.g.

```
"Driver={MySQL ODBC 3.51 Driver}; Server=mysql.example.net;
Database=mydb"
```

Note the curly brackets around the driver name. The name given here must match the one used in the ~/.odbcinst.ini or `/etc/odbcinst.ini` file. The ODBC manager will then map the name to the registered driver file location.

A `"DSN=mydsn"` entry in the connection string is no longer needed.

For a list of common DSN-less connection strings, have a look at the *ConnectionStrings.com* website.

In order to connect to a database using a DSN-less connection string, you simply pass the string to the `mx.ODBC.Manager.DriverConnect()` API.

4. Accessing Popular Databases

This section provides information on available ODBC drivers for various popular database as well as notes regarding setup, functionality or available workarounds for compatibility problems eGenix.com found with the drivers.

We have also included the resp. version information of the drivers we have tested successfully with mxODBC.

4.1 MS SQL Server

4.1.1 Available ODBC Drivers

MS SQL Server Native Client for SQL Server 2005, 2008 and later

Homepage: *http://msdn.microsoft.com/en-us/sqlserver/connectivity.aspx*

Tested with MS SQL Server Native Client for 2005, 2008, 2008R2 and 2012

SQL Server Native Client is the native database client and ODBC driver for SQL Server 2005, 2008 and later on Windows. Versions for 32-bit and 64-bit Windows are available.

Finding the latest version of the SQL Server Native Client for Windows

Microsoft always ships the SQL Server Native Client ODBC driver together with the SQL Server database packages, but also makes it available as separate download in the feature packs for each new SQL Server version.

Since the client can typically also be used with older SQL Server installations, it's worth trying the latest available version in case of problems or to benefit from new features. As of this writing, the latest version is SQL Server Native Client 11.

Here's a list of feature packs that include the SQL Server Native Client. You have to look for an installation file called sqlncli.msi or msodbcsql.msi on the pages:

- *Microsoft ODBC Driver 11 for SQL Server*

  Supports SQL Server 2005, 2008, 2008 R2 and 2012, 2014 and Windows Azure SQL Database.

- *Microsoft SQL Server 2012 SP2 Feature Pack*

Supports SQL Server 2005, 2008, 2008R2, 2012. This version no longer supports SQL Server 2000.

- *Microsoft SQL Server 2008 R2 SP1 Feature Pack*

  Supports SQL Server 2000, 2005, 2008, 2008R2.

- *Microsoft SQL Server 2008 Service Pack 2 Feature Pack*

  Supports SQL Server 2000, 2005, 2008.

- *Microsoft SQL Server 2005 Feature Pack Downloads*

  Supports SQL Server 2000 and 2005.

Use mxODBC direct execution methods for better performance

Python applications using the SQL Server Native Client should try to make use of the available direct execution interfaces in mxODBC, e.g.

- `cursor.executedirect()`
- `cursor.executemany(…, direct=1)`
- `cursor.execute(…, direct=1)`

> Our tests have shown that these interface can give a **2-5x better performance** with SQL Server than the normal APIs (e.g. `cursor.execute()` without `direct=1`)

Optimizing SQL Server Native Client Access Method

When installing the SQL Server Native Client, please make sure that you choose the most efficient database access method. If the ODBC driver resides on the same server as the database, shared memory is the most efficient protocol available. For most other purposes, TCP/IP is the best option. See *http://msdn.microsoft.com/en-us/library/ms187892.aspx* for details.

Configuring the SQL Server Native Client Network Protocol

When configuring an ODBC data source using the SQL Server Native Client you can choose the protocol by providing a server address with protocol prefix:

`tcp:db.example.com,1434` - connect to the server db.example.com using TCP/IP over port 1434 (the default SQL Server post is 1433)

`lpc:LOCALHOST\SQLEXPRESS` - connect to the instance SQLEXPRESS running on the local host via shared memory

`np:\\EXAMPLE\pipe\MSSQL$SQLEXPRESS\sql\query` - connect to the instance SQLEXPRESS running on the computer EXAMPLE via named pipe

See http://msdn.microsoft.com/en-us/library/ms188635.aspx for details on the various formats and how to configure them.

Multiple active result sets (MARS) on a single connection

The SQL Server Native Client per default does not support having more than one active result set per connection. This means that you cannot have two cursors on the same connection, which both have active result sets.

Starting with SQL Server 2005, there is a connection option which can be used to enable the *MARS feature of SQL Server Native Client*, which enables working with multiple active result sets on the same connection:

```
from mx.ODBC.Manager import DriverConnect, SQL

# Enable MARS on this connection:
options = [(SQL.COPT_SS_MARS_ENABLED, SQL.MARS_ENABLED_YES)]

# Pass the options to the connection constructor:
db = DriverConnect('DSN=mssqlserver2008;UID=sa;PWD=sa-passwd',
                   connection_options=options)
```

This option is available on SQL Server Native Client for Windows and Linux. The FreeTDS ODBC driver 0.91 does not support this option.

Parameter Binding with SQL Server 2012 and later

SQL Server 2012 and later have changed the SQL parser or optimizer to use a more stringent method of determining the unkown parameter input types.

This results in SQL statements such as "`WHERE col1 >= ? + ?`" to fail with an error such as `mx.ODBC.Error.ProgrammingError: ('42000', 11503, "[Microsoft][ODBC Driver 11 for SQL Server][SQL Server]The parameter type cannot be deduced because a single expression contains two untyped parameters, '@P1' and '@P2'.", 10191)`.

There are two solutions to this incompatibility between SQL Server 2008R2 and SQL Server 2012:

1. use explicit casts to tell the SQL parser which type to expect from the right hand side operation, e.g. "`WHERE col1 >= ? + CAST(? as int)`". This gives the second argument an explicit type and allows the parser to deduce the type of the result,

2. run the query using `.executedirect()` instead of `.execute()`. This causes the parameter values to be embedded into the SQL command and allows the SQL parser on the server side to determine the types at preparation time by looking at the values, rather than just their placeholders.

Calling stored prodcedures

There are multiple ways to call stored procedures with mxODBC.

You can use the direct approach by using:

```
cursor.execute('StoredProcedure1 @p1=1, @p2=2')
```

or using the ODBC escape sequence for stored procedures:

```
cursor.execute('{StoredProcedure1(1, 2)}')
```

or using the mxODBC method `cursor.callproc()` which takes care of the escaping for you:

```
cursor.execute('StoredProcedure1', (1, 2))
```

The latter method is the preferred method, since the ODBC driver will then internally do all necessary escaping of the procedure name and parameters to call the procedure using the syntax required by the used SQL Server backend version.

> Note that **using "execute "** when calling stored procedures with MS SQL Server, e.g. `cursor.execute('execute StoredProcedure1 @p1=1, @p2=2')` is not necessary (see *https://msdn.microsoft.com/en-us/library/ms188332.aspx*) and can, in fact, result in errors such as `'[Microsoft][SQL Native Client]Invalid Descriptor Index'`. It is better to avoid using "execute " in the call syntax.

Stored procedures with output parameters and result sets

MS SQL Server stored procedures are fully supported by mxODBC, but there is one important detail to know when using stored procedures which use both output or input/output parameters and result sets.

Due to the way the SQL Server Native Client works, the data from the result sets is sent to the client before the data for the output parameters. See e.g. the *IBM Knowledge Center* or *StackOverflow* for details.

Since mxODBC returns the output parameters immediately after executing a statement via the `cursor.callproc()`, `cursors.execute()` or `cursor.executedirect()` APIs, the changes send to the client after the result sets are not seen by mxODBC.

The Python DB-API 2.0 also does not allow for the output parameters to be fetched later, e.g. after the result sets have been fetched using the `cursor.fetch*()` APIs.

> As a result, output and input/output parameters in stored procedures which generate and return result sets **are not updated**.

Work-around:

The work-around for this is to return the output parameter values as additional result sets and then using the `cursor.nextset()` method to switch through the available result sets.

Passing NULL values to VARBINARY columns

mxODBC usually binds None as NULL and using a data type of VARCHAR when talking to databases. This works fine for all databases[3], and also does with MS SQL Server as long as you use regular cursor.execute() methods.

However, it causes problems when using mxODBC in **direct execution mode** (e.g. using `.executedirect()` or `.execute(…, direct=True)`), or in **Python type binding mode**, the binding data type is determined by the input data type passed to the methods, rather than determined by the ODBC driver.

In this mode, passing None to a VARBINARY column causes MS SQL Server to return an error `('42000', 257, '[Microsoft][ODBC Driver 11 for SQL Server][SQL Server]Implicit conversion from data type varchar to varbinary(max) is not allowed. Use the CONVERT function to run this query.', 11091)`.

To work around this, we have added a special singleton **`BinaryNull`** in mxODBC 3.3.4 and later, which is available in all mxODBC sub-packages. If you use this singleton instead of None in those cases where you'd otherwise see a VARCHAR/VARBINARY conversion error, MS SQL Server will accept the value as proper NULL value.

Example:

```
from mx.ODBC.Windows import BinaryNull
…
cursor.executedirect('insert into mytable values (?, ?, ?)',
                     (1, BinaryNull, BinaryNull))
```

An alternative work-around is to use an explicit cast in the SQL statement, e.g. "CAST(? AS VARBINARY)" when inserting data which can be NULL into the database.

Example:

```
…
cursor.executedirect('insert into mytable values '
                     '(?, CAST(? AS VARBINARY), CAST(? AS VARBINARY))',
                     (1, None, None))
```

MS SQL Server ODBC Driver for SQL Server 2000

Tested with MS SQL Server ODBC driver from MDAC 2.8 SP1.

If you still use SQL Server 2000, please get the latest ODBC driver for SQL Server from the Micrsoft MDAC package. MDAC 2.8 SP1 can be downloaded from this page:

[3] In fact, the *ODBC standard mandates* that VARCHAR and CHAR must implicitly convert to any other type. MS SQL Server does not implement this for binary data types - see the *conversion table* for details.

http://www.microsoft.com/downloads/details.aspx?familyid=78CAC895-EFC2-4F8E-A9E0-3A1AFBD5922E&displaylang=en

Configuring the SQL Server ODBC Driver Client Network Protocol

When configuring an ODBC data source using the MDAC SQL Server ODBC Driver you can choose the protocol by clicking on the Client Configuration button on the second wizard page.

Note that the MDAC ODBC drivers do not support shared memory access. Use named pipes as best connectivity option when connecting to a database server running on the same machine.

MS SQL Server Native Client for Linux

Homepage:
http://msdn.microsoft.com/en-us/library/hh477150%28v=sql.10%29.aspx

Tested with MS SQL Server Native Client 11 for Linux

The new Microsoft SQL Server Native Client for Linux is a port of the SQL Server Native Client for Windows to Linux. It provides an ODBC driver for SQL Server 2008 R2 and 2012 on Linux, but also works with SQL Server 2005 and 2008. At the moment, only a 64-bit version is available and then only for RedHat RHEL 5 and 6 systems.

Most of the comments for the Windows driver also apply to the Linux driver.

This driver also supports the MARS feature. See "Multiple active result sets (MARS) on a single connection" further up for details.

Driver Limitations

- The driver still has some issues and can produce segfaults (see below), but it's already good enough for testing and simple setups.
- The driver only works on 64-bit Linux distributions and requires the unixODBC 2.3.2 ODBC manager (2.2 won't work due to ABI differences).
- The mxODBC `DriverConnect()` API cannot be used with Unicode connection strings when using this driver: the **SQL Server Native Driver for Linux causes a segfault** when trying to connect using a Unicode connection string (which triggers the use of Unicode ODBC APIs). We have confirmed this for version 11.0.1790 and 11.0.2270 of the driver. Unfortunately, we cannot add a work-around to mxODBC to guard against this, since mxODBC only receives the driver name after the connect.

Example Configuration for Unix

Here is a sample setup for the SQL Server Native Client for Linux:

- Install the driver following the *instructions given by Microsoft*. Here's quick version:

```
tar xvfz sqlncli-11.0.1790.0.tar.gz
cd sqlncli-11.0.1790.0/
mkdir -p /opt/microsoft/sqlncli/bin
mkdir -p /opt/microsoft/sqlncli/lib
./install.sh install --force --accept-license \
   --bin-dir=/opt/microsoft/sqlncli/bin \
   --lib-dir=/opt/microsoft/sqlncli/lib
```

  You can ignore any warnings. Note that using the driver directly from the untarred archive is not possible, since the installation location appears to be hardwired in the driver.

- Depending on your Linux distribution, you may have to add symlinks to your OpenSSL libraries to match the ones used on RedHat:

```
cd /lib64
ln -sf libssl.so.1.0.0 libssl.so.10
ln -sf libcrypto.so.1.0.0 libcrypto.so.10
ldconfig
```

- Add the driver to the /etc/odbcinst.ini (or ~/.odbcinst.ini):

```
[ODBC Drivers]
MSNativeClient = Installed

[MSNativeClient]
Driver = /opt/microsoft/sqlncli/lib64/libsqlncli-11.0.so.1790.0
Description = MS SQL Server Native Client 11
Threading = 1
```

- Add a data source to the /etc/odbc.ini (or ~/.odbc.ini):

```
[mssqlserver2008]
Driver = /opt/microsoft/sqlncli/lib64/libsqlncli-11.0.so.1790.0
Description = MS SQL Server 2008 running on Picasso (MS Native
Client)
Server = tcp:picasso\SQLSERVER2008,1436
Database = testdb
```

  The syntax for the Server entry is described in the MSDN article *SQL Server Native Client ODBC Connection String Format*.

EasySoft ODBC Driver for SQL Server

Homepage: *http://www.easysoft.com/*

OpenLink ODBC Driver for SQL Server

Homepage: *http://www.openlinksw.com/*

DataDirect ODBC Driver for SQL Server

Homepage: *http://www.datadirect.com/*

Actual Technologies Mac OS X ODBC Driver for SQL Server

Homepage: *http://www.actualtech.com/*

When using the driver on Mac OS X 10.6 (Snow Leopard), be sure to use version 3.0.9 or higher, since earlier versions had a problem with fetching data.

FreeTDS Unix ODBC Driver for SQL Server

Homepage: *http://www.freetds.org/*

Tested with FreeTDS 0.91 ODBC driver compiled against unixODBC 2.3.2.

The FreeTDS ODBC driver implements the client side of the TDS wire protocol used by Sybase ASA and Microsoft SQL Server installations. It allows you to directly connect to a SQL Server database from a Unix machine.

Driver Limitations

- The FreeTDS ODBC driver version 0.91 introduces Unicode support for the first time in its version history. The previous stable version 0.82 did not have Unicode support. We've had most success using the NATIVE_UNICODE_STRINGFORMAT mode.

- Most other operations work as expected, but please note that the driver is still under heavy development in some areas. You should test it thoroughly before using it on a production system.

- The FreeTDS website mentions that the driver has some restrictions. Please see the *FreeTDS user guide* for details.

- Be sure to use the same ODBC manager with FreeTDS as the one you have compiled it with. If you mix e.g. unixODBC and iODBC, you can easily run into **Unicode data corruption issues**. The two ODBC managers use different default data types for Unicode data, so a FreeTDS ODBC driver compiled against unixODBC (2-bytes Unicode data type) will not return correct Unicode data when used with the iODBC ODBC manager (4-bytes Unicode data type).

- FreeTDS ODBC driver only works natively with DATETIME columns. **DATE and TIME column types** introduced in SQL Server 2008 are only supported via strings. FreeTDS 0.91 accepts them as strings and returns them as strings. Using mxDateTime values for DATE and TIME fields does not work. You can use the MS SQL Server Native Client for Linux, if you need these column types supported, since it does not have this limitation.

- There are other ODBC drivers available from commercial vendors which implement the full ODBC3 API, including the free MS SQL Server Native Client for Linux from Microsoft. Alternatively, you can use our *mxODBC Connect* product to use the SQL Server Native Client on Windows from all supported Python platforms and without the need for a client-side ODBC driver.

- The FreeTDS driver does not support the MARS feature[4]. Only one active result set is allowed per connection. If you need the MARS feature, please have a look at the MS SQL Server Native Client for Linux. This driver also support the MARS feature. See "Multiple active result sets (MARS) on a single connection" further up for details.

- Older FreeTDS driver versions do not return correct **`cursor.rowcount` values**. mxODBC overrides these wrong values with -1 to be on the safe side. FreeTDS drivers 0.91 and later return correct results, so this work-around was lifted again in mxODBC 3.3.3 and later.

- FreeTDS does not support **sending binary data** passed in as Python 8-bit string to the MS SQL Server backend. As a result, bindings for e-g- VARBINARY columns result in type conversion errors (e.g. `"[FreeTDS][SQL Server]Implicit conversion from data type varchar to varbinary is not allowed. Use the CONVERT function to run this query."` or `"[FreeTDS][SQL Server]Operand type clash: text is incompatible with varbinary"`). As work-around, pass in the binary data wrapped as Python `buffer()`. In mxODBC 3.3.3 and later, we've added a work-around to enable this automatically.

- FreeTDS has the same issue as the SQL Server Native Client when it comes to **passing NULLs to VARBINARY columns**. Please see the section "Passing NULL values to VARBINARY columns" in the notes for SQL Server Native Client for Windows for details on how to work around this. Essentially, you can either pass in the special singleton `BinaryNull` (a su-package global available in all mxODBC sub-packages since mxODBC 3.3.4) instead of None or rewrite your SQL to explicitly case the parameter to VARBINARY.

Example Configuration for Unix

Here is a sample setup for FreeTDS on Linux talking to MS SQL Server 2008 on Windows:

- Add this section to /usr/local/freetds/etc/freetds.conf (the freetds.conf configuration file may be in a different location on your machine):

```
# MS SQL Server 2000 running on server MONET
[MONET]
host = monet.example.net
port = 1433
tds version = 8.0
```

- Add this section to /etc/odbc.ini (the odbc.ini configuration file may be in a different location on your machine). It is also necessary to point `LD_LIBRARY_PATH` to the directory where the driver itself is located.

```
[mssql]
Driver = /usr/local/freetds/lib/libtdsodbc.so
Description = MS SQL Server 2008 running on Monet
Trace = 0
Servername = MONET
```

[4] There is work underway to add this feature. See the *http://freetds.org/* website.

```
Database = tempdb
```

Note that the libtdsodbc.so file may be located in a different directory on your machine.

- Using these settings, you can then connect to SQL Server using a simple connection string such as:

```
"DSN=mssql;UID=username;PWD=password"
```

4.1.2 General Notes

ODBC API Extensions and the SQL Server Native Client

The ODBC API is the native MS SQL Server call level interface and provides the best performance when interfacing to SQL Server.

Microsoft has also extended ODBC with various custom extensions they make available in their SQL Server Native Client API. If you need support for those extensions, please contact *support@egenix.com*.

Static vs. forward-only Cursors

MS SQL Server supports static ODBC cursors, but mxODBC defaults to forward only cursors. [5]

While static cursors allow scrolling through the result set and also provides ways of accessing the correct `.rowcount` value, it does come with a significant performance penalty. We have seen **slow-downs in fetching rows of 2-3x times** for average queries, **up to 300x** for simple ones, so we recommend not using static cursors on connections that do not need scroll support. To enable static cursors, you can adjust the `connection.cursortype`:

```
connection = mx.ODBC.Windows.DriverConnect(…)
connection.cursortype = mx.ODBC.Windows.SQL.CURSOR_STATIC
# All cursors created on connection will then default to the static
# cursor type.
```

Please refer to section 5.9 ODBC Cursor Types for more details on cursor types.

Timestamp Resolution

Unlike many other databases which support nanosecond precision on timestamps, MS SQL Server has a limitation when it comes to the seconds part of timestamp values. Version of SQL Server earlier and including SQL Server 2005 only accept timestamp precisions of 1 millisecond, while version SQL Server 2008 and later work with precision of 100 nanoseconds.

mxODBC addresses this by defaulting to `connection.timestampresolution = 1000000` for SQL Server 2005 and earlier. For SQL Server 2008 and later, mxODBC uses `connection.timestampresolution = 100`.

[5] Please note that in mxODBC 3.2, mxODBC used to default to static cursors.

This allows using timestamp values with SQL Server which use higher precision values without running into errors from the database such as:

```
HY104 - [Microsoft][SQL Server Native Client 11.0] Invalid precision value.
```

However, please be advised that reducing the precision of input values requires rounding, which can lead to unexpected results. mxODBC does make sure that the seconds whole value is not altered by the rounding to reduce unwanted surprises. Please see the documentation on `connection.timestampresolution` for details.

Multiple Cursors on Connections / MARS

If you have troubles with multiple cursors on connections to MS SQL Server the MS Knowledge Base Article 140896 *INF: Multiple Active Microsoft SQL Server Statements* has some valuable information for you.

You have to explicitly enable support for multiple active cursors (MARS) or force the usage of server side cursors to be able to execute multiple statements on a single connection to MS SQL Server. According to the article this is done by setting the connection option SQL.CURSOR_TYPE to e.g. SQL.CURSOR_DYNAMIC:

```
dbc.setconnectoption(SQL.CURSOR_TYPE, SQL.CURSOR_DYNAMIC)
```

If you are using the MS SQL Server Native Client, you can also enable the MARS feature on the client side by setting a connect option during connect. Please see the section "Multiple active result sets (MARS) on a single connection" further up for details.

International Character Data

If you are experiencing problems with MS SQL Server not storing or fetching international character data (Unicode, Asian encodings, etc.) correctly, please have a look at the following MS Knowledge Base Articles:

- *232580 - INF: Storing UTF-8 Data in SQL Server*
- *257668 - FIX: SQL Server ODBC Driver May Cause Incorrect Code Conversion of Some Characters*
- *234748 - PRB: SQL Server ODBC Driver Converts Language Events to Unicode*

More information about the MS SQL Server ODBC Driver and the various connection parameters and options are available on the MSDN Library site: *MS SQL Server ODBC Driver Programmer's Guide*.

Access Violations

If you are experiencing problems related to access violations, like e.g.

```
ProgrammingError: ('37000', 0, '[Microsoft][ODBC SQL Server
Driver]Syntax error or access violation', 4498)
```

a possible reason could be that you are using a function or stored procedure which is generating output using PRINT or that it uses RAISEERROR to report an error with the parameters or values.

Another possible reason is that the ODBC driver for SQL Server does not support the syntax you are using or that bound parameters are not allowed at that location in the SQL statement. As work-around you can use Python string formatting to insert the data verbatim directly into the SQL statement.

Distributed Transaction Managers

If you are using a transaction manager (e.g. MS DTC), you can sometimes get warnings like the following:

```
mxODBC.Warning: ('01000', 7312, [Microsoft][ODBC SQL Server
Driver][SQL Server][OLE/DB provider returned message: New
transaction cannot enlist in the specified transaction
coordinator.]', 4606)
```

This is a problem related to the used transaction manager rather than mxODBC or the database. Please consult your DBA for help.

Note that even though the above exception is raised by the cursor.execute() method, the fact that it is a warning suggests that the executed operation was indeed executed on the cursor.

Kerberos / Windows Integrated Authentication

In order to use Kerberos authentication the ODBC driver used by mxODBC has to support this. mxODBC itself can help with the setup via connection options (see *Service Principal Names (SPNs) in Client Connections (ODBC)*), but does not need to be setup in a special way to support Kerberos.

We know of these ODBC drivers that support Kerberos Windows integrated authentication with MS SQL Server:

MS SQL Server Native Client for Windows

On Windows, you simply have to replace the "`UID=…;PWD=…`" part of your connection string with "`Trusted_Connection=yes`" in order to use Kerberos authentication.

It may also be necessary to provide a Service Principal Name (SPN) for the server. This can be done via the "`ServerSPN=…`" keyword parameter in the connection string. Please see this article for details: *Service Principal Name (SPN) Support in Client Connections*

MS SQL Server Native Client for Linux

Please see this article for details on how to setup the driver to use Kerberos authentication: *Using Integrated Authentication*

Also see the EasySoft documentation below for some added details around the setup of Kerberos authentication on Linux.

EasySoft SQL Server Driver for Linux

Please see the following page for details on how this driver is setup: *Securing Access to SQL Server from Linux with Kerberos*

FreeTDS ODBC Driver for Linux

The FreeTDS ODBC driver can be built against the Kerberos libraries and provides the same `Trusted_Connection` keyword connection parameter as the MS SQL Server Native Client. Provided you have Kerberos working on the system, you simply have to replace the "`UID=…;PWD=…`" part of your connection string with "`Trusted_Connection=yes`" in order to use Kerberos authentication.

Kerberos on Linux

All of the above have some information about the setup of Kerberos on Linux. This book chapter has all the information you need to complete the setup:

http://commons.oreilly.com/wiki/index.php/Linux_in_a_Windows_World/Centralized_Authentication_Tools/Kerberos_Configuration_and_Use

If you want to use an Active Directory, you will additionally have to use Samba to integrate into the AD, hook winbind into PAM and have pam_winbind use Kerberos authentication:

http://wiki.samba.org/index.php/Samba_&_Active_Directory

A user can then log in and authenticate against the AD and the Kerberos server will provide the necessary credentials to the ODBC driver.

Running multiple statements in a single .execute()

MS SQL Server allows running multiple statements with a single cursor.execute() command, e.g.

```
sqlcmd = """
create table products ( name varchar(50) primary key )
insert into products values ('A')
insert into products values ('B')
insert into products values ('C')
"""
result = cursor.execute(sqlcmd)
```

Error reporting needs extra attention

However, without additional configuration, this will cause mxODBC to not report all errors which may have occurred during processing of the batch of commands.

Example:

```
cursor = db.cursor()
sqlcmd = """
create table products ( name varchar(50) primary key )
insert into products values ('A')
insert into products values ('B')
insert into products values ('A')
"""
result = cursor.execute(sqlcmd)
```

In this example, the last INSERT will fail, since the value 'A' was already inserted with the first INSERT and the database constraint of a primary key does not allow duplicates.

Yet, mxODBC will not report this error, because the ODBC driver only reports back potential errors from the first statement in the command batch (the CREATE in this example).

Work-around for proper error reporting

If you don't need to access the number of affected rows (`cursor.rowcount`), there is a work-around for this. Simply SET NOCOUNT ON before running the statements:

```
cursor = db.cursor()
sqlcmd = """
set nocount on
create table products ( name varchar(50) primary key )
insert into products values ('A')
insert into products values ('B')
insert into products values ('A')
"""
result = cursor.execute(sqlcmd)
```

Running this code will have mxODBC raise an error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
mx.ODBC.Error.IntegrityError: ('23000', 2627, "[Microsoft][ODBC Driver 11
for SQL Server][SQL Server]Violation of PRIMARY KEY constraint
'PK__products__72E12F1A3F039ACD'. Cannot insert duplicate key in object
'dbo.products'.", 11245)
```

as you would expect.

The configuration causes SQL Server not to report the number of affected rows back to the ODBC driver, so `cursor.rowcount` will always remain set to -1. You can still use *@@ROWCOUNT* to access the value, if you need it.

> Note: The NOCOUNT setting remains in effect until reset on the connection.

For more information on SET NOCOUNT, please see the *MS SQL Server documentation*.

Better stored procedure performance with SET NOCOUNT ON

As explained in the previous section, using SET NOCOUNT ON can help with getting errors reported back to the ODBC driver and thus via mxODBC to your Python application.

However, the setting also has a nice side-effect on the performance of running multiple statements, e.g. in a statement batch or in a stored procedure.

> If you frequently run multiple statements, longer stored procedures or stored procedures with loops, we recommend using SET NOCOUNT ON to increase performance of your application.

This can be achieved by running the following code on a connection:

```
cursor = db.cursor()
cursor.execute("set nocount on")
```

Since information about affected rows is no longer reported to the ODBC driver, `cursor.rowcount` will always return -1 with this setting enabled. If you still need the affected row count information, you can explicitly request this using the *@@ROWCOUNT* variable available in Transact-SQL.

The setting will persist until reset to the default using:

```
cursor = db.cursor()
cursor.execute("set nocount off")
```

Other Common Problems and Solutions

A **general description of the problems** you might experience when accessing the MS SQL Server using ODBC is described in the article *Using ODBC with Microsoft SQL Server*. Even though it's dated September 1997 it provides some insights into the design and workings of the MS SQL ODBC driver.

4.2 MS Access Database

4.2.1 Available ODBC Drivers

MS Access ODBC Driver

Tested with MDAC 2.8 SP1 Access ODBC driver.

MS Access ships with an ODBC driver for the database which is then installed on the same machine as MS Access (or Office). The drivers are also available separately as part of the MDAC package.

MDAC 2.8 SP1 can be downloaded from this page:

http://www.microsoft.com/downloads/details.aspx?familyid=78CAC895-EFC2-4F8E-A9E0-3A1AFBD5922E&displaylang=en

MDBTools ODBC Driver

Homepage: http://mdbtools.sourceforge.net/

This package provides a very limited ODBC driver which allows accessing Access database files directly without having to install or run MS Access. It is mostly used on Unix platforms to extract data from existing MS Access database files.

4.3 Oracle

4.3.1 Available ODBC Drivers

Oracle Instant Client ODBC driver

Homepage: http://www.oracle.com/technology/tech/oci/instantclient/index.html

Tested with Oracle Instant Client ODBC driver 11.2.

The Oracle Instant Client ships with an ODBC driver (part of the *ODBC Supplement*) for most supported platforms.

mxODBC works well when using the Oracle Instant Client 11.2 with the unixODBC ODBC manager package `mx.ODBC.unixODBC` on Unix or the `mx.ODBC.Windows` package on Windows.

Driver Notes

- eGenix.com has had reports about memory leaks occurring with the Oracle driver when used in long running applications. mxODBC itself does not have any known memory leaks and there are no problems with other available drivers for Oracle.

- Oracle regards empty strings as NULL values. As a result inserting an empty string into a VARCHAR column can result in the Oracle driver returning NULL for that column when fetching data.

- The Oracle driver returns numeric values as floats, even integers, so unless you use a converter function, you will get floats when querying integer columns.

- The automatic reuse of prepared SQL commands does not work with the Oracle driver, so the optimization for the `cursor.execute()` method does not work with the Oracle driver.

- The Oracle driver does not support scrollable cursors, meaning that `cursor.scroll()` will only work using the built-in mxODBC emulation for forward scrolling.

Example Configuration for Unix

- To be able to use the Oracle Instant Client, you have to create a ~/.tnsnames.ora file providing the network configuration details of the target database:

```
ORACLE11GR2 =
  (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)
      (HOST = oracle11gr2.example.net)
      (PORT = 1521))
    (CONNECT_DATA =
      (SERVER = DEDICATED)
      (SERVICE_NAME = oracle11gr2.example.net)
    )
  )
```

- Add an Oracle driver section to the ~/.odbcinst.ini file (the location of the driver file may be different on your system):

```
[ODBC Drivers]
OracleInstantClient = Installed

[OracleInstantClient]
Description = Oracle 11g ODBC Driver
Driver = /opt/oracle/instantclient_11_2/libsqora.so.11.1
Setup =
FileUsage =
CPTimeout =
CPReuse =
```

- Edit the ~/.odbc.ini file based on the above ~/.tnsnames.ora settings. Note that the location of the driver file depends on your installation. It is also necessary to point `LD_LIBRARY_PATH` to the directory where the driver itself is located.

```
[oracle11gr2]
Driver = /opt/oracle/instantclient_11_2/libsqora.so.11.1
ServerName = oracle11gr2
# DSN driver options
Application Attributes = T
Attributes = W
BatchAutocommitMode = IfAllSuccessful
BindAsFLOAT = F
CloseCursor = F
DisableDPM = F
DisableMTS = T
EXECSchemaOpt =
EXECSyntax = T
Failover = T
FailoverDelay = 10
FailoverRetryCount = 10
FetchBufferSize = 64000
ForceWCHAR = F
Lobs = T
Longs = T
MaxLargeData = 0
```

```
MetadataIdDefault = F
QueryTimeout = T
ResultSets = T
#SQLGetData extensions = F
SQLGetData extensions = T
Translation DLL =
Translation Option = 0
DisableRULEHint = T
#UserID =
StatementCache=F
CacheBufferSize=20
UseOCIDescribeAny=F
```

- Using these settings, you can then connect to Oracle using a simple connection string such as:

```
"DSN=oracle11gr2;UID=username;PWD=password"
```

EasySoft ODBC Driver for Oracle

Homepage: *http://www.easysoft.com/*

OpenLink ODBC Driver for Oracle

Homepage: *http://www.openlinksw.com/*

DataDirect ODBC Driver for Oracle

Homepage: *http://www.datadirect.com/*

Actual Technologies Mac OS X ODBC Driver for Oracle

Homepage: *http://www.actualtech.com/*

When using the driver on Mac OS X 10.6 (Snow Leopard), be sure to use version 3.0.9 or higher, since earlier versions had a problem with fetching data.

4.3.2 General Notes

Oracle tnsnames.ora file

When connecting to Oracle database you typically have to provide a ~/.tnsnames.ora file which has the network connection information of your Oracle database servers.

If you want to use a different file location, be sure to set the environment variable `TNS_ADMIN` to the path of the tnsnames.ora file.

4.4 IBM DB2

4.4.1 Available ODBC Drivers

IBM ODBC Driver for Unix/Windows DB2 servers

Tested with IBM DB2 9.7 ODBC driver.

IBM DB2 ships with ODBC drivers for DB2 on Windows, Linux and other Unix systems. These can also be used to connect to DB2 database over a network.

Please see this page for more information:

http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.apdv.cli.doc/doc/c0023378.htm

Example Configuration for Unix

- Setup the network details of your DB2 database in the ~/.db2cli.ini file:

```
[ibmdb2]
DBAlias=SAMPLE
Hostname=db2.example.net
```

- Using the same section name, add a new section to your ~/.odbc.ini file.

```
[ibmdb2]
Driver = /usr/local/odbc-drivers/db2/clidriver/lib/libdb2.so
```

  You will have to change the driver entry to the location where you copied the ODBC driver and add the lib/ directory of the driver to your `LD_LIBRARY_PATH`.

- You can then connect to your database via the connection string `"DSN=ibmdb2;UID=username;PWD=password"`.

IBM ODBC Driver for iSeries / AS/400 DB2 servers

IBM has a Linux ODBC driver which makes this setup possible. See their web-page on the "iSeries ODBC driver for Linux" for details:

http://www-03.ibm.com/systems/i/software/access/linux/guide/index.html

OpenLink ODBC Driver for DB2

Homepage: *http://www.openlinksw.com/*

DataDirect ODBC Driver for DB2

Homepage: *http://www.datadirect.com/*

4.4.2 General Notes

ODBC API Extensions and the IBM CLI

The ODBC API is the native IBM DB2 call level interface and provides the best performance when interfacing to DB2.

IBM has also extended ODBC with various custom extensions they make available as CLI interface. If you need support for those extensions, please contact support@egenix.com.

Configuring Database Access

If you want to use the `DriverConnect()` API with IBM DB2, you'll have to configure the IBM ODBC driver's data source INI file which is named ~/db2cli.ini and usually found in the same directory as the above script files. The file is needed in addition to the ~/.odbc.ini file and the entries must match.

If you place the db2cli.ini file into a different directory, make sure that you setup the environment variable `DB2CLIINIPATH` to point to the full path of the file.

Environment Variables on Unix

If you don't use a db2cli.ini file, you can configure the access details using environment variables:

In order to connect to the IBM DB2 database the `DB2INSTANCE` environment variable must be set to the name of the DB2 instance you would like to connect to.

There may be more environment variables needed, please check the scripts that come with DB2 called `db2profile` (for bash) or `db2cshrc` (for C shell) which set the environment variables. Without having these set, mxODBC will fail to load and give you a traceback:

```
Traceback (most recent call last):
...
    from mxODBC import *
ImportError: initialization of module mxODBC failed
(mxODBC.InterfaceError:failed to retrieve error information (line 6778,
rc=-1))
```

Linker Paths

Unfortunately, the provided db2profile / db2cshrs shell scripts are buggy in some versions of DB2, so simply sourcing them won't necessarily work.

You will have to carefully create your own to work around these issues.

A typical problem is that the scripts set `LIBPATH` or `LD_LIBRARY_PATH` (without paying attention to possibly existing settings) which then causes the following linker-related traceback when trying to load mxODBC:

```
Traceback (most recent call last):
...
ImportError: from module mxODBC.so No such file or directory
```

Database Setup for ODBC Access

Unlike many other databases, DB2 needs to be explicitly told that you want to connect to the database using ODBC.

This is done by binding the IBM CLI driver against the database in order to setup ODBC related views and stored procedures. Please consult the IBM DB2 documentation for details on how this is done.

Static vs. forward-only Cursors

IBM DB2 supports static ODBC cursors, but mxODBC defaults to forward only cursors. [6]

While static cursors allow scrolling through the result set and also provides ways of accessing the correct `.rowcount` value, it does come with a significant performance penalty. We have seen **slow-downs in fetching rows of around 2x times** for average queries, so we recommend not using static cursors on connections that do not need scroll support. To enable static cursors, you can adjust the `connection.cursortype`:

```
connection = mx.ODBC.Windows.DriverConnect(…)
connection.cursortype = mx.ODBC.Windows.SQL.CURSOR_STATIC
# All cursors created on connection will then default to the static
# cursor type.
```

Please refer to section 5.9 ODBC Cursor Types for more details on cursor types.

4.5 Sybase ASE

4.5.1 Available ODBC Drivers

Sybase ASE ODBC driver

Homepage: *http://www.sybase.com/*

Tested with Sybase ASE 15.5 and 15.7 ODBC drivers

Sybase ASE ships with ODBC drivers for both 32-bit and 64-bit platforms. The drivers are part of the ASE server packages.

[6] Please note that in mxODBC 3.2, mxODBC used to default to static cursors.

The ASE 15.5 ODBC driver can also be used to connect to a Sybase ASE 12.x server database. In fact, this setup is recommended, since the 15.5 version of the driver fixes a couple of issues that are present in the 12.x ODBC driver.

NULL issues with Sybase ASE ODBC driver

We have had reports of the Sybase ODBC driver producing errors when trying to set a character column to NULL (by passing None to mxODBC for that column), esp. on 64-bit platforms.

mxODBC 3.3 now has a work-around for this problem in place, so this should no longer be an issue.

Segfaults with Sybase ASE ODBC driver 15.7

On 64-bit **Unix platforms**, we have observed occasional segfaults caused by the Sybase ASE ODBC driver. These occur in the malloc() routine of the glibc after running tests for a while and seem to be caused by the driver corrupting the heap. In some cases, glibc also detects invalid pointers in free() as a result of this.

The specific version of the driver where we found this problem was 15.7.0.104. This may have been resolved in later versions.

We have also had reports of similar issues on 32-bit **Windows platforms**, where the segfaults occur in ntdll.dll.

> The Sybase ASE ODBC driver for 15.5 does not have these issues, so we recommend using this until the problem is resolved in the Sybase ASE ODBC driver 15.7.

BIGINT columns can cause data corruption

Even though Sybase ASE 15 does support a BIGINT column type, the Sybase ESD 3 ODBC driver has problems interfacing to it and data is corrupted. As example, inserting a -2 to a BIGINT columns results in 1 being read back; inserting -2147483648 results in 47493012874424 read from the column.

Since this is a bug in the ODBC driver, future ESDs may fix the issue. In any case, please carefully check for this problem before using BIGINT columns with the driver.

Driver Notes

- In mxODBC 3.3, we switched the default parameter binding method from Python type to SQL type. This resolves issues with the Sybase ODBC driver complaining about the wrong Python object type being used for certain database data types. Especially the Unicode support (UNICHAR, UNIVARCHAR and UNITEXT database column types) now works much better than before.

- The Sybase ASE driver only supports forward scrolling, so `cursor.scroll()` will just work with relative and positive increments.

- There is no support for `cursor.rownumber` in the driver.

- If you are using large text/binary data fields with more than 32k data, be sure to add the connection parameter `TextSize = 10000000` (or larger) to the connection string or the data source definition in your ~/.odbc.ini. Not doing so will otherwise result in data truncations.

Example Configuration for Unix

- Add a Sybase driver section to the ~/.odbcinst.ini file (the location of the driver file may be different on your system):

```
[ODBC Drivers]
SybaseASE = Installed

[SybaseASE]
Description = SybaseASE ODBC Driver
Driver = /opt/sybase/DataAccess/ODBC/lib/libsybdrvodb.so
Setup =
```

- Edit the ~/.odbc.ini file based and add a `sybasease` section(the location of the driver file may be different on your system). It is also necessary to point `LD_LIBRARY_PATH` to the directory where the driver itself is located.

```
[sybasease]
Driver = /opt/sybase/DataAccess/ODBC/lib/libsybdrvodb.so
Description = Adaptive Server Enterprise
Server = sybasease.example.net
Port = 5000
Database = mydb
TextSize = 10000000
#UseCursor = 1
FileUsage = -1
Trace = Off
TraceFile = /tmp/sybase.log
```

- Using these settings, you can then connect to Sybase ASE using a simple connection string such as:

```
"DSN=sybasease;UID=username;PWD=password"
```

EasySoft ODBC Driver for Sybase

Homepage: *http://www.easysoft.com/*

OpenLink ODBC Driver for Sybase

Homepage: *http://www.openlinksw.com/*

DataDirect ODBC Driver for Sybase

Homepage: *http://www.datadirect.com/*

Actual Technologies Mac OS X ODBC Driver for Sybase

Homepage: http://www.actualtech.com/

When using the driver on Mac OS X 10.6 (Snow Leopard), be sure to use version 3.0.9 or higher, since earlier versions had a problem with fetching data.

4.6 PostgreSQL

4.6.1 Available ODBC Drivers

PostgreSQL ODBC Driver

Homepage: http://psqlodbc.projects.postgresql.org/

Tested with psqlodbc 09.03.0210.

The PostgreSQL driver is usually compiled against unixODBC on Unix platforms. Please use the `mx.ODBC.unixODBC` subpackage to connect to PostgreSQL with it. On Mac OS X, you may need to use the `mx.ODBC.iODBC` subpackage instead, since the Mac OS X ODBC manager is derived from the iODBC manager.

Driver Notes

- Because of limitations in the PostgreSQL ODBC drivers, mxODBC always operates in Python type binding mode.
- In mxODBC 3.3, we have added a work-around to make the driver work with binary data and BYTEA columns.
- Unicode data is supported. It works best with the NATIVE_UNICODE_STRINGFORMAT mode. You can also use the auto-transcoding feature of mxODBC with UTF-8 as encoding.
- The driver only supports forward scrolling with relative increments of +1. Other values result in a driver error. As a result, only `cursor.scroll(+1)` can be used.

Example Configuration for Unix

- Add a PostgreSQL driver section to the ~/.odbcinst.ini file (the location of the driver file may be different on your system):

```
[ODBC Drivers]
PostgreSQL = Installed

[PostgreSQL]
```

```
Description = PostgreSQL ODBC Driver
Driver = /usr/local/postgresql/lib/psqlodbcw.so
Setup =
```

- Edit the ~/.odbc.ini file based and add a postgresql section(the location of the driver file may be different on your system; be sure to use the Unicode variant which ends with `'…w.so'`). It is also necessary to point `LD_LIBRARY_PATH` to the directory where the driver itself is located.

```
[postgresql]
Driver = /usr/local/postgresql/lib/psqlodbcw.so
Database = mydb
ServerName = postgresql.example.net
Port = 5432
#Debug = 0
#Optimizer = 0
#CommLog = 0
#ReadOnly = 0
#SSLmode = require
ByteaAsLongVarBinary = 1
TextAsLongVarchar = 1
# This currently doesn't appear to work:
#UseServerSidePrepare = 1
```

- Using these settings, you can then connect to PostgreSQL using a simple connection string such as:

```
"DSN=postgresql;UID=username;PWD=password"
```

EasySoft ODBC Driver for PostgreSQL

Homepage: *http://www.easysoft.com/*

OpenLink ODBC Driver for PostgreSQL

Homepage: *http://www.openlinksw.com/*

DataDirect ODBC Driver for PostgreSQL

Homepage: *http://www.datadirect.com/*

Actual Technologies Mac OS X ODBC Driver for PostgreSQL

Homepage: *http://www.actualtech.com/*

When using the driver on Mac OS X 10.6 (Snow Leopard), be sure to use version 3.0.9 or higher, since earlier versions had a problem with fetching data.

4.7 MySQL

4.7.1 Available ODBC Drivers

MySQL ODBC Driver

Homepage: http://dev.mysql.com/downloads/connector/odbc/

Tested with MySQL ODBC driver 5.2.6 and MySQL 5.5 and 5.6.

Be sure to use the mxODBC ODBC manager subpackage against which the MySQL driver was compiled. This will usually be `mx.ODBC.unixODBC` or `mx.ODBC.iODBC`.

The MySQL ODBC driver documentation recommends using unixODBC with the driver, i.e. the `mx.ODBC.unixODBC` subpackage.

Note that the MySQL 5.2 ODBC driver can connect to all recent versions of the MySQL database server. It is usually best to use the latest available version, even if your database server has a different version number.

Driver Notes

- There is one particularity with the ODBC driver for MySQL: all input parameters are being processed as strings -- even integers and floats. The ODBC driver implements the necessary conversions. mxODBC therefore defaults to Python Type binding mode for binding parameters; see the *Python Type Input Binding* section 8.5 for more details.

- Unicode data is supported, provided you use the Unicode ODBC driver versions - on Unix these are identified with a small "w" at the end of the driver lib name, e.g. libmyodbc5w.so. It works best with the NATIVE_UNICODE_STRINGFORMAT mode. You can also use the auto-transcoding feature of mxODBC with UTF-8 as encoding.

- Using a MySQL 5 Windows ODBC driver character setting of `'utf-8'` (with hyphen) can cause the driver to segfaults, so care must be taken, using the right spelling for the character set setting.

- The MySQL ODBC driver does not always update the `.rownumber` to the correct value, especially when using `.scroll()`. For client side cursors, mxODBC corrects this using an emulation for `.rownumber`.

- When using the ODBC driver RPMs available from *www.mysql.com*, please be sure to also have the MySQL shared libs RPM and the MySQL development RPM installed.

- Some older MySQL + ODBC driver setups eGenix.com has tested showed some serious memory leaks on Linux machines. Please check your setup for such leaks before going into production. There are no known leaks in mxODBC itself.

Example Configuration for Unix

- Add a MySQL driver section to the ~/.odbcinst.ini file (the location of the driver file may be different on your system):

```
[ODBC Drivers]
MySQL = Installed

[MySQL]
Description = MySQL ODBC Driver
Driver = /usr/local/lib/libmyodbc5w.so
Setup =
```

- Edit the ~/.odbc.ini file based and add a `mysql` section(the location of the driver file may be different on your system). It is also necessary to point `LD_LIBRARY_PATH` to the directory where the driver itself is located.

```
[myodbc]
Driver = /usr/local/lib/libmyodbc5w.so
Description = MySQL 5 Server
Server = mysql.example.net
Port =
# Specifying a database is necessary for MySQL, since you'll
# otherwise won't be able to connect
Database = mydb
# Allow big packets for BLOBs, etc.
option = 8
#Socket =
```

- Using these settings, you can then connect to MySQL using a simple connection string such as:

```
"DSN=mysql;UID=username;PWD=password"
```

OpenLink ODBC Driver for MySQL

Homepage: *http://www.openlinksw.com/*

DataDirect ODBC Driver for MySQL

Homepage: *http://www.datadirect.com/*

Actual Technologies Mac OS X ODBC Driver for MySQL

Homepage: *http://www.actualtech.com/*

When using the driver on Mac OS X 10.6 (Snow Leopard), be sure to use version 3.0.9 or higher, since earlier versions had a problem with fetching data.

4.7.2 General Notes

Depending on whether you use a transactional MySQL storage backend or not, clearing the auto-commit flag at connection time, which is normally done per default by the connection constructors, will not work.

Be sure to set `clear_auto_commit=0` if you know that the storage backend cannot handle transactions. mxODBC will then default to auto-commit mode. Rollback will not work in that mode.

4.8 SAP MaxDB / SAPDB

4.8.1 Available ODBC Drivers

MaxDB ODBC driver

Homepage: *http://www.sdn.sap.com/irj/sdn/maxdb*

Tested with MaxDB 7.7 ODBC driver

MaxDB ships with ODBC drivers for all supported platforms. The ODBC driver is included in the distribution tar archive of the database as SDBODBC.TGZ.

You can use the drivers with both unixODBC and iODBC.

Note that the MaxDB 7.7 ODBC driver can also connect to a MaxDB 7.6 database server. It is usually best to use the latest available version, even if your database servers hasn't been upgraded yet.

Example Configuration for Unix

- Edit your ~/.odbcinst.ini file and add the MaxDB driver (the location of the driver and setup file may be different on your system):

```
[ODBC Drivers]
MaxDB = Installed

[MaxDB]
Driver = /usr/local/maxdb/lib/libsdbodbcw.so
Description = MaxDB ODBC Driver
```

- Edit your ~/.odbc.ini file and add a MaxDB section (the location of the driver may be different on your system; be sure to use the Unicode variant which ends with `'…w.so'`). It is also necessary to point `LD_LIBRARY_PATH` to the directory where the driver itself is located.

```
[maxdb]
DRIVER = /usr/local/maxdb/lib/libsdbodbcw.so
```

```
ServerDB = MYDB
ServerNode = maxdb.example.net
SQLMode =
IsolationLevel =
Trace    = 0
TraceFileName=/tmp/maxdb.log
```

- Using these settings, you can then connect to MaxDB using a simple connection string such as:

```
"DSN=maxdb;UID=username;PWD=password"
```

4.8.2 General Database Notes

Warnings when deleting/update more than one row at a time

MaxDB issues a `mx.ODBC.Error.Warning: ('01001', 0, '[SAP AG][LIBSDBOD SO] Cursor operation conflict', 8416)` warning whenever you try to delete or update more than one row with a single database statement.

You can easily work around this by setting the `cursor.warningformat` to `IGNORE_WARNINGFORMAT`, restoring it afterwards to the default `ERROR_WARNINGFORMAT`, if you just want to ignore this particular warning case.

4.9 Teradata

4.9.1 Available ODBC Drivers

Teradata ODBC Driver

Homepage: *http://www.teradata.com/downloadcenter/*

Tested with Teradata 14.1 ODBC driver and DataDirect 7.1 ODBC manager.

The Teradata ODBC driver was developed by DataDirect and requires the DataDirect ODBC manager, so you will need to use the `mx.ODBC.DataDirect` package on Unix to work with the driver.

Driver Notes

- The `mx.ODBC.DataDirect` package is currently only available for Linux 32-bit and 64-bit systems. If you need the package on other platforms, please write to *support@egenix.com* for assistance.

- The DataDirect ODBC driver manager is included in the same directory as the Teradata ODBC driver itself. If you setup `LD_LIBRARY_PATH` to

the directory where the driver is located, mxODBC will automatically use the right DataDirect ODBC driver manager, e.g.

```
export LD_LIBRARY_PATH=\
    /opt/teradata/client/14.10/odbc_64/lib:\
    /opt/teradata/client/14.10/tdicu/lib64
```

- The Teradata ODBC driver uses a separate file for loading error messages. The path for this is set using the `NLSPATH` environment variable and should be set up like this (the directory will have to be adapted to the location of the `tdodbc.cat` file:

  ```
  export NLSPATH=/opt/teradata/client/14.10/odbc_64/msg/%N.cat
  ```

  If not set, you will get exceptions like this from the driver: `[Teradata][ODBC Teradata Driver] Unable to get catalog string`.

- Native Unicode is supported by the driver/manager combination setup with the `CharacterSet = UTF16` setting in the ~/.odbc.ini section for Teradata.

- Trying to use the Teradata ODBC driver with unixODBC or iODBC usually results in an immediate segfault.

- If you use the Teradata ODBC driver in combination with the DataDirect ODBC manager, be sure to keep the ~/.odbc.ini file short. With longer ~/.odbc.ini files, the combination will segfault.

- The 14.10 version of the driver insists on having a section `[ODBC Data Sources]` in the ~/.odbc.ini file. Without it, the driver doesn't connect and fails with the following error message:

  ```
  [Teradata][ODBC Teradata Driver] No DBCName entries were
  found in DSN/connection-string
  ```

  This does not happen with the 13.10 driver.

- We found that it is apparently not possible to use multi-line SQL statements with the Teradata ODBC drivers 13.10 and 14.10. The Teradata SQL parser complains about newline characters in the strings. Removing these results in syntax errors. This makes it difficult to e.g. define stored procedures from a Python application. The problem appears to be a driver limitation which may be fixed in future Teradata driver versions.

- The Teradata driver only supports relative forward scrolling in the result set. Backwards scrolling is not supported.

- Teradata has the tendency to return non-ordered result sets in random order. This is due to the way the database works internally. If you need to rely on a reproducible result set order, please add an `ORDER BY` clause to the `SELECT` statements as necessary.

Example Configuration for Unix

- Setup your OS environment so that the ODBC manager can find and load the driver (this is for the version 14.1 of the driver, your installation directories may be different):

```
export LD_LIBRARY_PATH=\
    /opt/teradata/client/14.10/odbc_64/lib:\
    /opt/teradata/client/14.10/tdicu/lib64

export NLSPATH=/opt/teradata/client/14.10/odbc_64/msg/%N.cat
```

  This last setting is important to make sure the driver can find its error messages. If not set, you will get exceptions like this from the driver: `[Teradata][ODBC Teradata Driver] Unable to get catalog string`.

- Edit your ~/.odbcinst.ini file and add the Teradata driver (the location of the driver and setup file may be different on your system):

```
[ODBC Drivers]
Teradata = Installed

[Teradata]
Driver=/opt/teradata/client/13.10/odbc_64/lib/tdata.so
APILevel=CORE
ConnectFunctions=YYY
DriverODBCVer=3.51
SQLLevel=1
```

- Edit your ~/.odbc.ini file and add a Teradata section (the location of the driver may be different on your system). It is also necessary to point `LD_LIBRARY_PATH` to the directory where the driver itself is located.

```
[ODBC]
Trace = 0
TraceFile = /tmp/odbc.log

[ODBC Data Sources]
teradata = Teradata

[teradata]
Driver = /opt/teradata/client/14.10/odbc_64/lib/tdata.so
Description = Teradata ODBC
DBCName = 192.168.0.250
DefaultDatabase = mydb
RunInQuietMode = Yes
DSNTraceEnable = No
DSNTraceFilePath = /tmp/teradata.txt
DSNTraceOverwrite = Yes
CharacterSet = UTF16
DateTimeFormat = AAA
# Disable preparing statements
#DisablePREPARE = Yes
# Max. response packet size in bytes
MaxRespSize = 10000000
# Disable parsing of SQL statements by the driver; do not set to
Yes
# if using .callproc() in the application.
NoScan = No
# SessionMode can be Teradata or ANSI
SessionMode =
# Cursor open checks
StCheckLevel = 0
# Enable TCP_NODELAY ?
TCPNoDelay = Yes
# Port to use on the database servers
```

```
TDMSTPortNumber =
# Use BLOB and CLOB ?
UseNativeLOBSupport = Yes
```

Also see the above driver notes regarding how to format the .odbc.ini file.

- Using these settings, you can then connect to Teradata using a simple connection string such as:

```
"DSN=teradata;UID=username;PWD=password"
```

DataDirect ODBC Driver for Teradata

Homepage: http://www.datadirect.com/

4.10 Netezza

4.10.1 Available ODBC Drivers

Netezza ODBC Driver

Homepage: http://www.netezza.com/

Tested with Netezza 4.6.6 ODBC driver.

The Netezza ODBC driver is available to Netezza customers or partners. eGenix.com partnered up with Netezza to make sure that mxODBC performs well with Netezza's driver.

Recommended Setup

The recommended setup is to use the `mx.ODBC.unixODBC` subpackage together with a **unixODBC 2.3 or later** ODBC manager.

The Netezza driver will also work with the DataDirect ODBC manager that ships with the Netezza driver. If you use this setup, please interface to the DataDirect ODBC manager using the `mx.ODBC.DataDirect` subpackage.

Also note that when using the DataDirect ODBC Manager we have seen segfaults related to the .odbc.ini file being too big. You can work around this by either keeping the file short, or by setting up a separate netezza-odbc.ini file and pointing the driver manager to it using the `ODBCINI` environment variable.

Netezza and Unicode

Unicode data exchange doesn't work well when using the Netezza driver with the DataDirect manager.

The unixODBC manager interface does not have these issues and works fine with Unicode if the Netezza driver is configured for UTF-16 data using the `UnicodeTranslationOption = utf16` driver configuration option.

Example Configuration for Unix

- Edit your ~/.odbcinst.ini file and add the Netezza driver (the location of the driver and setup file may be different on your system):

```
[ODBC Drivers]
NetezzaSQL = Installed

[NetezzaSQL]
Driver           = /usr/local/nz/lib64/libnzodbc.so
Setup            = /usr/local/nz/lib64/libnzodbc.so
APILevel         = 1
ConnectFunctions = YYN
Description      = Netezza ODBC driver
DriverODBCVer    = 03.00
DebugLogging     = false
LogPath          = /tmp
# For unixODBC, use the following setting:
UnicodeTranslationOption = utf16
# For DataDirect, use this setting:
#UnicodeTranslationOption = utf8
CharacterTranslationOption = all
PreFetch         = 256
Socket           = 8192
```

- Edit your ~/.odbc.ini file and add a Netezza section (the location of the driver may be different on your system). It is also necessary to point `LD_LIBRARY_PATH` to the directory where the driver itself is located.

```
[netezza]
Driver                = /usr/local/nz/lib64/libnzodbc.so
Description           = NetezzaSQL ODBC
Servername            = netezza.example.net
Port                  = 5480
Database              = mydb
Username              =
Password              =
ReadOnly              = false
ShowSystemTables      = false
LegacySQLTables       = false
LoginTimeout          = 0
QueryTimeout          = 0
DateFormat            = 1
NumericAsChar         = false
SQLBitOneZero         = true
StripCRLF             = false
securityLevel         = preferredUnSecured
caCertFile            =

# Needed by the DataDirect ODBC manager, values:
# 1=UTF-16, 2=UTF-8
DriverUnicodeType = 1
```

- Using these settings, you can then connect to Netezza using a simple connection string such as:

```
"DSN=netezza;UID=username;PWD=password"
```

DataDirect ODBC Driver for Netezza

Homepage: *http://www.datadirect.com/*

4.11 Other Databases

If you want to run mxODBC in a Unix environment and your database doesn't provide an Unix ODBC driver, you can try the drivers sold by these ODBC driver specialists:

4.11.1 EasySoft ODBC Driver Packages

Homepage: *http://www.easysoft.com/*

EasySoft also maintains the open source ODBC manager *unixODBC*.

4.11.2 OpenLink

Homepage: *http://www.openlinksw.com/*

OpenLink maintains the open source ODBC manager *iODBC*.

4.11.3 DataDirect

Homepage: *http://www.datadirect.com/*

DataDirect drivers ship their own ODBC manager. Since this ODBC manager is not compatible with unixODBC or iODBC, please use the `mx.ODBC.DataDirect` driver manager package when accessing the DataDirect drivers through the DataDirect ODBC manager.

> The `mx.ODBC.DataDirect` package is currently only available for Linux 32-bit and 64-bit systems. If you need the package on other platforms, please write to *support@egenix.com* for assistance.

4.11.4 Other Vendors

For a fairly large list of sources for ODBC drivers have a look on the *SQLSummit list of ODBC drivers*.

4.11.5 Alternative solution: *mxODBC Connect*

If you would like to connect to a database for which you don't have a Unix ODBC driver, you can also try our *mxODBC Connect Python Database Interface* which

just needs an ODBC driver on the server side and provides a cross-platform networked interface to this for the client side. This makes it very easy to connect to e.g. a Windows-based database from Unix, BSD or Mac OS X.

5. mxODBC Overview

mxODBC is structured as Python package to support interfaces to many different ODBC managers and drivers. Each of these interfaces is accessible as subpackage of the `mx.ODBC` Python package, e.g. on Windows you'd normally use the `mx.ODBC.Windows` subpackage to access the Windows ODBC manager; on Unix this would typically be the `mx.ODBC.iODBC`, `mx.ODBC.unixODBC` or the `mx.ODBC.DataDirect` package depending on which of these Unix ODBC managers you have installed.

Each of these subpackages behaves as if it were a separate Python database interface, so you actually get more than just one interface with mxODBC. The advantage over other Python database interfaces is that all subpackages share the same logic and programming interfaces, so you don't have to change your application logic when moving from one subpackage to another. This enables programs to run (more or less) unchanged on Windows and Unix, for example.

As you may know, there is a standard for Python database interfaces, the Python Database API Specification or Python DB-API for short. Marc-André Lemburg, the author of the mxODBC package, is the editor of this specification, so great care is taken to make mxODBC as compatible to the Python DB-API as possible.

5.1 mxODBC and the Python Database API Specification

The mxODBC package tries to adhere to the *Python DB API Version 2.0* in most details. Many features of the old *Python DB API 1.0* are still supported to maintain backwards compatibility and simplify porting old Python applications to the new interface.

5.1.1 Differences

Here is a list of minor differences between mxODBC and the DB API 2.0 specifications:

- `cursor.description` doesn't return *display_size* and *internal_size*; both values are always `None`, since this information is not readily available through ODBC interfaces used for result set processing, the information can be expensive to determine and the values are not commonly used in applications. If you do need to access this information, you can use the `cursor.getcolattribute()` method with info ids `SQL.DESC_DISPLAY_SIZE` and `SQL.DESC_OCTET_LENGTH`.

- `db.setinputsizes()` and `db.setoutputsizes()` are dummy functions; this is allowed by DB API 2.0.

- The type objects / constructors (formerly found in the `dbi` module defined by DB API 1.0) are only needed if you want to write database independent code.

- The connection constructor is available under three different names: `ODBC()` (DB API 1.0), `connect()` (DB API 2.0) and `Connect()` (mxODBC specific). See the next section for details on the used parameters. mxODBC also defines a `DriverConnect()` constructor which is available for ODBC managers and some ODBC drivers. If you can, please use the `DriverConnect()` API since this provides more flexibility in configuring the connection.

5.1.2 Extensions

mxODBC extends the DB-API specification in a significant number of ways to provide access to as many ODBC features as possible. If you want to stay compatible to other Python DB-API compliant interface, you should only use those interfaces which are mentioned in the Python DB-API *specification documents*.

5.2 mxODBC and the ODBC Specification

Since ODBC is a widely supported standard for accessing databases, it should in general be possible to use the package with any ODBC version 2.0 - 3.8 compliant ODBC database driver/manager. mxODBC prefers ODBC 3.x over 2.x in case the driver/manager supports both versions of the standard.

5.2.1 Full access to most ODBC features

The ODBC API is very rich in terms of accessing information about what is stored in the database. mxODBC makes most of these APIs available as additional connection and cursor methods and these can be put to good use for database and schema introspection.

Since many of the parameters and names of the ODBC function names were mapped directly to Python method names (by dropping the SQL prefix and converting them to lower-case), we kindly refer you to the *Microsoft ODBC Documentation* and your ODBC driver documentation for low-level details on the various APIs.

You also can access the MS ODBC online reference from the *Microsoft MDAC web-site*.

> Please note that not all ODBC drivers and databases support the complete set of available introspection parameters. When using them, please make sure that the databases supported by your application do implement the parameters used in your application.

5.3 Supported ODBC Versions

mxODBC can be configured to use ODBC 2.x or 3.x interfaces by setting the `ODBCVER` symbol in mxODBC.h to the needed value. It uses the value provided by the ODBC driver header files per default which usually is the latest ODBC standard version available.

Most ODBC drivers today support ODBC 3.x and thus mxODBC will try to use APIs from this version if available.

5.3.1 ODBC Managers

All supported ODBC managers (MS ODBC Manager, iODBC, unixODBC and DataDirect) provide the ODBC 3.x interfaces and map these to ODBC 2.x interfaces in case the driver for the database does not comply to ODBC 3.x.

However, some drivers only pretend to be ODBC 3.x compliant and raise `"Driver not capable"` exceptions when using certain ODBC 3.x APIs or features. If you run into such an situation, please contact *support* for help. The only way to solve this problem currently lies in adding workarounds which are specific to a database.

To find out which ODBC version is being supported by the ODBC driver, you can use `connection.getinfo(SQL.DRIVER_ODBC_VER)[1]`. This will return a string giving you the ODBC version number, e.g. `'03.51.00'`.

5.3.2 Changes between ODBC 2.x and 3.x

Please also note that there are some changes in behavior between ODBC 2.x and 3.x compatible drivers/managers which means that certain option settings differ slightly between the two versions and that special cases are treated differently for ODBC 3.x than for ODBC 2.x. See the *ODBC Documentation* for details.

5.4 Thread Safety & Thread Friendliness

mxODBC itself is written in a thread safe way. There are no module globals being used and thus no locking is necessary.

5.4.1 Connections and Cursors

In general when using a separate database connection for each thread, you shouldn't run into threading problems. If you do, it is more likely that the ODBC driver is not 100% thread safe and thus not 100% ODBC compatible. Note that having threads share cursors is *not* a good idea: there are many very strange transaction related problems you can then run into.

5.4.2 Unlocking the Python Global Interpreter Lock (GIL)

Many of the underlying ODBC SQL function calls are wrapped by macros unlocking the global Python interpreter lock before the call and regaining that lock directly afterwards. The most prominent of those are the connection APIs and the execute and fetch APIs.

Unlocking the interpreter lock during long SQL function calls gives your application more responsiveness. This is especially important for GUI based applications, since no other Python thread can run when the global lock is acquired by one thread.

5.4.3 Threading Support

mxODBC will only support threading if you have built Python itself with thread support enabled. Python for Windows and most recent Python versions for Unix have this enabled per default. Try: `python -c "import threading"` to find out. If you get an exception, thread support is not available.

5.5 Transaction Support

Database transactions provide a way to group statements into logical units of recovery.

If a statement within such a unit fails, it is possible to roll back to the state of the database before the unit of recovery (the transaction) was started.

Likewise, you make changes permanent by explicitly committing them to the database and thereby ending the current unit of recovery.

Databases will typically implicitly start a transaction with the first statement run on a connection after opening it, or after having ended the previous transaction by either committing or rolling back the changes made in that transaction.

5.5.1 Auto-Commit

ODBC uses auto-commit on new connections per default. This means that all SQL statement executes will directly have an effect on the underlying database even in those cases where you would really back out of a certain modification, e.g. due to an unexpected error in your program.

mxODBC turns off auto-commit whenever it creates a new connection, ie. it runs the connection in manual commit mode -- unless the connection constructor flag `clear_auto_commit` is set to 0 or the database does not provide transactions.

You can adjust the connection's commit mode after creating it using the `connection.autocommit` attribute. See 5.5.3 Adjusting the Connection Commit Mode for details.

5.5.2 Manual Commit

Using a connection in manual commit mode means that all your commands are grouped in transactions: only the connection will see the changes it has made to the data in the database until an explicit `connection.commit()` is issued.

The commit informs the database to write all changes done during the last transaction into the global data storage making it visible to all other users. A `connection.rollback()` on the other hand, tells the database to discard all modifications processed in the last transaction.

Transaction Start and End

New transactions are started implicitly in the following cases:

- creation of a new connection,
- on return from a `connection.commit()` and
- after having issued a `connection.rollback()`.

Unless you perform an explicit `connection.commit()` prior to deleting or closing the connection, mxODBC will try to issue an *implicit rollback* on that connection before actually closing it.

Errors are only reported in case you use the `connection.close()` method. Implicit closing of the connection through Python's garbage collection will ignore any errors occurring during rollback.

Data Sources without Transaction Support

Data sources that do not support transactions, such as flat file databases (e.g. Excel or CSV files on Windows), cause calls to `.rollback()` to fail with an `NotSupportedError`. mxODBC will *not* turn off auto-commit behavior for these sources. The setting of the connection constructor flag `clear_auto_commit` has no effect in this case.

Some databases for which mxODBC provides special subpackages such as `MySQL` don't have transaction support, since the database does not provide transaction support. For these subpackages, the `.rollback()` connection method is not available at all (i.e. calling it produces an `AttributeError`) and the `clear_auto_commit` flag on connection constructors defaults to `0`.

5.5.3 Adjusting the Connection Commit Mode

Using connection.autocommit

You can adjust the connection's commit mode after creating it using the `connection.autocommit` attribute. Setting the attribute to `True` will cause the connection to operate in auto-commit mode again. Setting it to `False` will have the connection use manual commit. The attribute also allows querying the current commit mode in the same way.

Using connection options

Alternatively, you can use

```
connection.setconnectoption(SQL.AUTOCOMMIT, SQL.AUTOCOMMIT_ON)
```

to turn on auto commit and

```
connection.setconnectoption(SQL.AUTOCOMMIT, SQL.AUTOCOMMIT_OFF)
```

to turn it off again.

Similarly, `connection.getconnectoption(SQL.AUTOCOMMIT)` will return the current option value (as tuple).

5.6 Transaction Isolation

Related to transactions is the concept of transaction isolation. This essentially addresses the question of when changes made within an open transaction are seen by other transactions open on the database and has great influence on how parallel access to the database can be managed by the database.

Since the transaction isolation provides different levels of protection against dynamic changes to result sets, databases typically define various transaction isolation levels.

5.6.1 Common Transaction Isolation Levels

Databases usually provide a number of different transaction isolation levels, but the details are specific to the database and can vary in many important details.

Please check your database documentation for the exact definition of these terms.

Read uncommitted

The transaction will see data changed or added by other transactions using the same database. It is therefore possible to see uncommitted changes and changes which may get rolled back again (*dirty reads*).

This isolation level requires only very little locking and thus gives the best concurrency. The downside is that data consistency is not always available.

mxODBC SQL constant: `SQL.TXN_READ_UNCOMMITTED`

Read committed

Statements in the transaction only see data committed to the database before the statement began running (no *dirty reads*). Changes committed by other transactions are taken into account between statements, so it is possible that rerunning the same statement may return different results (*non-repeatable reads* or *phantom data*).

This is the default for most databases.

mxODBC SQL constant: `SQL.TXN_READ_COMMITTED`

Repeatable read

Transactions only see data which was committed before the transaction began. Statements can be rerun within the transaction and will always return the same data.

This isolation level requires a lot of locking (both read and write locks). Long running transactions should be avoided to not prevent concurrency altogether.

mxODBC SQL constant: `SQL.TXN_REPEATABLE_READ`

Serializable

Transactions are logically serialized, meaning that data access is managed in such a way that concurrent transactions cannot apply changes which would affect the serialization of the transactions, e.g. one transaction cannot add rows which would end up in the result sets of another transaction.

This level is sometimes implemented using locks or with a strategy which allows transactions to fail during commit.

mxODBC SQL constant: `SQL.TXN_SERIALIZABLE`

Please see section 5.6.3 Adjusting the Transaction Isolation Level for details on how to adjust the transaction isolation level with mxODBC.

5.6.2 Database Specific Information

These are pointers to a few database specific explanations of the transaction isolation levels they support:

- *MS SQL Server: SET TRANSACTION ISOLATION LEVEL*
- *Oracle: Data Concurrency*
- *IBM DB2: Isolation levels*
- *Sybase ASE: Choose an Isolation Level*
- PostgreSQL: Transaction Isolation
- *MySQL: SET TRANSACTION ISOLATION LEVEL*

You can also consult the *ODBC documentation on transaction isolation levels* and the *Wikipedia page on Isolation* for more in depth explanations.

5.6.3 Adjusting the Transaction Isolation Level

In order to set a transaction isolation level, you can use the `connection.setconnectoption()` method or the `connect_options` parameter on the connection constructors.

Setting the isolation level after creation of the connection

The isolation level may be set after creation of the connection, but must usually be done before running any statements on it:

```
from mx.ODBC.Windows import SQL
…
# Connect to the database
connection = DriverConnect(dsn)

# Set up the connection
connection.setconnectoption(SQL.ATTR_TXN_ISOLATION,
                            SQL.TXN_SERIALIZABLE)

# Create a cursor to start working
cursor = connection.cursor()
…
```

Setting the isolation level before opening the connection

In some cases, it may be necessary to use the `connection_options` parameter of the mxODBC Connection constructors to set the option before opening it:

```
from mx.ODBC.Windows import DriverConnect, SQL
…
# Connect to the database and set TXN options
connection = DriverConnect(dsn,
                           connect_options = (SQL.ATTR_TXN_ISOLATION,
                                              SQL.TXN_SERIALIZABLE))

# Create a cursor to start working
cursor = connection.cursor()
…
```

The transaction isolation level cannot be set while inside a running transaction.

Available Transaction Isolation Level Values

These are the ODBC standard values for isolation levels. They are available via the `SQL` lookup variable defined in all mxODBC subpackages:

`SQL.TXN_READ_UNCOMMITTED`

Read uncommitted (dirty reads, non-repeatable reads, phantom data all possible)

`SQL.TXN_READ_COMMITTED`

Read committed (no dirty reads, but non-repeatable reads, phantom data are possible)

`SQL.TXN_REPEATABLE_READ`

Repeatable read (no dirty reads or non-repeatable reads, but phantom data is possible)

`SQL.TXN_SERIALIZABLE`

Serializable (no dirty reads, non-repeatable reads or phantom data)

> Please note that not all database support all isolation levels defined for ODBC. Some may issue a warning saying that they chose a different isolation level in such cases.

5.7 Stored Procedures

mxODBC provides several ways to call stored procedures in databases and supports input, input/output and output parameters to make integration with existing database systems easy:

1. directly using the `.callproc()` cursor method, or
2. using the standard ODBC syntax for calling stored procedures and using the standard `.execute*()` cursor methods to initiate the calls.

5.7.1 Calling Stored Procedures with .callproc()

The `.callproc()` cursor method is very straight forward way of calling stored procedures:

```
results = cursor.callproc("myprocedure", parameters)
```

When using this notation, mxODBC will call the procedure named `"myprocedure"` with the variables given in the sequence `parameters` and return a list copy of the parameters as `results`.

Parameters are bound to the procedure parameters by position, just as is done for the `.execute*()` methods when using the `'qmark'` parameter style.

Results can be retrieved through output parameter, input/output parameters, or result sets. Depending on the database backend, it is also possible to combine both.

Retrieving output parameters from stored procedures

When not providing the optional `parametertypes` parameter as in the above example, all parameters are considered to be input parameters, so `results` will be a list copy of `parameters`.

If you want to use input/output or output parameters, you have to specify the `parametertypes` parameter to define how to bind the variables to the procedure parameters:

```
results = cursor.callproc("myprocedure", parameters, parametertypes)
```

Please see section 5.7.3 Input/Output and Output Parameters for more details on how to use `parametertypes`.

Values from input/output and output parameters will then be updated in the results list copy as returned by the stored procedure.

Example:

```
from mx.ODBC.unixODBC import SQL
…
results = cursor.callproc('sp_params',
                          [1, 0],
                          parametertypes=(SQL.PARAM_INPUT,
                                          SQL.PARAM_OUTPUT))
if results == [1, 3]:
    print 'Works.'
```

Retrieving result sets from stored procedures

The `cursor.callproc()` method can also be used to call stored procedures which generate result sets. These can then be fetched using the standard `cursor.fetch*()` methods. If the stored procedure has generate multiple result sets, skipping to the next result set is possible by calling the `.nextset()` cursor method.

Please see section 5.7.5 Using Result Sets for passing back Output Data for details.

Example:

```
cursor.callproc('sp_result_set', [1])
result_set = cursor.fetchall()
```

5.7.2 Calling Stored Procedures with cursor.execute*() Methods

Stored procedures and functions can also be called indirectly using the following standard ODBC syntax for calling stored procedures.

The ODBC syntax for calling a stored procedure is as follows:

```
{call procedure-name [([parameter][,[parameter]]...)]}
```

For stored functions or procedures with return status, the ODBC syntax is as follows:

```
{? = call function-name [([parameter][,[parameter]]...)]}
```

Using the above syntax, you can call stored procedures through one of the `.execute*()` calls, e.g.

```
results = cursor.execute("{call myprocedure(?,?)}", parameters)
```

will call the stored procedure `myprocedure` with the given input `parameters`.

It is also possible to use the `'named'` parameter style, e.g.

```
results = cursor.execute("{call myprocedure(:a, :b)}", parameters)
```

Results can be retrieved through output parameter, input/output parameters, or result sets. Depending on the database backend, it is also possible to combine both.

Retrieving output parameters from stored procedures

When not providing the optional `parametertypes` parameter as in the above examples, all parameters are considered to be input parameters, so `results` will be a tuple copy of `parameters`.[7]

If you want to use input/output or output parameters, you have to specify the `parametertypes` parameter to define how to bind the variables to the procedure parameters:

```
results = cursor.execute("{call myprocedure(?,?)}",
                         parameters, parametertypes)
```

Please see section 5.7.3 Input/Output and Output Parameters for more details on how to use `parametertypes`.

Values from input/output and output parameters will then be updated in the `results` tuple copy as returned by the stored procedure.

Example:

```
from mx.ODBC.unixODBC import SQL
…
results = cursor.execute(
    '{call sp_params(?,?)}',
    [1, 0],
    parametertypes=(SQL.PARAM_INPUT, SQL.PARAM_OUTPUT))
if results == [1, 3]:
    print 'Works.'
```

For **stored functions** or **procedures with return status**, the syntax is similar (note the prepended `"? ="`:

```
results = cursor.execute("{? = call myfunction(?,?)}",
                         parameters, parametertypes)
```

The return value from the function will be passed back as first parameter. Accordingly, the first entry in `parametertypes` must be set to `SQL.PARAM_OUTPUT`.

Example:

```
from mx.ODBC.unixODBC import SQL
…
results = cursor.execute(
    '{? = call function_params(?,?)}',
    [0, 1, 0],
    parametertypes=(SQL.PARAM_OUTPUT,
                    SQL.PARAM_INPUT,
                    SQL.PARAM_OUTPUT))
if results == (4, 1, 2):
    print 'Works.'
```

[7] Note that the results sequence is a tuple or list of tuples for the `cursor.execute*()` methods, not a list as for `cursor.callproc()`. This is due to the DB-API requiring a list for `cursor.callproc()`.

Retrieving result sets from stored procedures

In case the stored procedure generates one or more result sets, these can be fetched using the standard `cursor.fetch*()` methods. If the stored procedure has generate multiple result sets, skipping to the next result set is possible by calling the `.nextset()` cursor method.

Please see section 5.7.5 Using Result Sets for passing back Output Data for details.

Example:

```
cursor.execute('{call sp_result_set(?)}', [1])
result_set = cursor.fetchall()
```

5.7.3 Input/Output and Output Parameters

The mxODBC default assumption for bound parameters is to use input parameters. These don't require additional information to be processed by mxODBC.

ODBC and several databases also support output parameters (parameters which don't have an input value, but are used for sending output back to the caller) and input/output parameters (which are read for input and send data as output).

parametertypes Parameter

In order to tell which parameters are input/output or output parameters, mxODBC needs additional information in form of a `parametertypes` sequence. This sequence defines the type of parameter for each bound parameter in the order they appear in the executed command or stored procedure definition.

The following types of parameters are supported. They are defined through the `SQL` lookup variable which is available in all mxODBC subpackages.

`SQL.PARAM_INPUT`

The parameter is an input parameter. Output values are not allowed and may raise a database error or simply be ignored (this depends on the ODBC driver and database).

`SQL.PARAM_OUTPUT`

The parameter is an output parameter. Input values are ignored.

Since mxODBC does not always have an efficient possibility to query the output type of the stored procedure, the type of the corresponding parameter value in the input parameters sequence is used as **hint to the output type**. Even though the value itself is ignored, it should therefore match the expected output type of the output parameter[8].

[8] Some ODBC drivers/database backends can provide type information for output parameters in stored procedures, but others fail to provide usable information. If you run

`SQL.PARAM_INPUT_OUTPUT`

The parameter is a combination of input and output parameter. Input values are passed to the command/stored procedure parameters. After execution of the command/stored procedure, the output values are read back from the database.

Note: The `parametertypes` parameter and - as a result - output and input/output parameters are **not supported for 'named' parameter style mode**. This may change in future mxODBC versions.

Example for cursor.execute():

```
from mx.ODBC.unixODBC import SQL
…
results = cursor.execute('{call get_max_value(?, ?)}',
                         (1, 3),
                         parametertypes=(SQL.PARAM_INPUT,
                                         SQL.PARAM_INPUT_OUTPUT))
```

Example for cursor.callproc():

```
from mx.ODBC.unixODBC import SQL
…
results = c.callproc('get_max_value',
                     (1, 3),
                     parametertypes=(SQL.PARAM_INPUT,
                                     SQL.PARAM_INPUT_OUTPUT))
```

Dynamically determining the Parameter Type

In some situations it may not be possible to determine the parameters beforehand, e.g. when dynamically creating a stored procedure call.

Fortunately, the cursor catalog methods provide a way to determine which parameters are input, output or input/output parameters via the `cursor.procedurecolumns()` method.

This generates a result set with column COLUMN_TYPE which has the needed information.

Please see the documentation in section 7.6.1 Catalog Methods for `cursor.procedurecolumns()` for details.

5.7.4 Special constraints of some ODBC drivers

Mixing output parameters and output result sets

Some ODBC drivers, most notably, the **MS SQL Server Native Client**, send the output parameter data only after any output result sets which the stored procedure may have created.

---

into problems with data type conversion errors, please try setting the `cursor.bindmethod` to `BIND_USING_PYTHONTYPE` before running the procedure call.

Since the mxODBC cursor methods return the output parameter right after executing the stored procedure and before fetching any result sets, the output parameters are not yet updated. Furthermore, there is no method of accessing the output parameters after those methods have returned.

> As a result, you won't see **output parameter updates** in your application, if you are using both output parameters and output result sets in your stored procedures.

The **work-around** for this is to pass back the output parameter data using an additional result set and then fetching this using the `cursor.nextset()` method after the other result sets. Also see section 5.7.5 Using Result Sets for passing back Output Data on this topic.

Example:

```
cursor.callproc('procname', parameters)

# Get the original output result set
output_result_set = cursor.fetchall()

# Switch to the new output parameter result set
cursor.nextset()
output_parameter_result_set = cursor.fetchall()
```

Using None as value for output parameters

mxODBC uses the value passed to the cursor execution method as hint for the output type of the output parameter, in case the ODBC driver does not provide this information.

If you pass in `None` as placeholder for the output parameter value, mxODBC is, in some cases, unable to determine the correct output type. It then defaults to using a `VARCHAR(4000)` output parameter value and sends back the data as character string.

Especially for numeric data, this may both be inefficient and inconvenient, so it's better to pass in a value which matches the output parameter type such as 0 for integer or 0.0 for floating point data.

5.7.5 Using Result Sets for passing back Output Data

It is also possible and to pass back data from the stored procedure via standard result sets which can be fetched from mxODBC using the cursor.execute*() methods.

This method is often preferred when dealing with larger data chunks, or table data.

Using result sets to pass back output data

Passing back such data in form of one or more result sets allow for great flexibility in exchanging data between stored procedures and your Python application, also let's you implement variable length output parameter lists and special output value conversions.

This can easily be done by adding a SELECT to the stored procedure which then returns the data as additional result set:

```
SELECT OutputParam1, OutputParam2
```

or even using multiple result sets:

```
SELECT OutputParam1; SELECT OutputParam2
```

You can then pick up the data using `cursor.nextset()` and `cursor.fetchall()`:

```
rs1 = c.fetchall()
c.nextset()
rs2 = c.fetchall()
c.nextset()
rs3 = c.fetchall()
```

Example:

```
>>> c.execute('select 1; select 2')
>>> c.fetchall()
[(1,)]
>>> c.nextset()
True
>>> c.fetchall()
[(2,)]
```

MS SQL Server and Sybase ASE Cursors in Stored Procedures

MS SQL Server and Sybase ASE both make the result sets from SELECTs in the stored procedures available to the mxODBC cursor via `cursor.fetch*()`.

Example:

```
CREATE PROCEDURE sp_result_set
@a INTEGER
AS
  SELECT @a * 3;
```

Oracle Ref Cursors as Output Parameters

Oracle has the concept of reference cursors, which provide a similar way to pass cursors to the stored procedure caller. Instead of defining an output variable or a set of output variables, you simply define a REF CURSOR as output variable in your stored procedure.

You can then access the open cursor after calling the stored procedure by simply using the standard `.fetch*()` and `.nextset()` APIs to access the results. The key to making this work is by not passing in any variable for the output REF CURSOR when calling the stored procedure. [9]

[9] Thanks to Etienne Desgagné for pointing out this solution.

Oracle Base has a more detailed article on this:

- *Using Ref Cursors To Return Recordsets*

Example:

```
CREATE PROCEDURE sp_result_set
(a INTEGER,
 rs OUT SYS_REFCURSOR)
AS
BEGIN
  OPEN rs FOR
    SELECT a * 3 FROM DUAL;
END;
```

Note that you don't need to call the procedure with the second parameter. The result set will still be available.

IBM DB2 Cursors in Stored Procedures

IBM has a similar feature to the Oracle Ref Cursors. You simply declare a cursor and open it before returning from the stored procedure. The cursor is then available for reading in Python.

Example:

```
CREATE PROCEDURE sp_result_set
(a INTEGER)
RESULT SETS 1
BEGIN
  DECLARE c1 CURSOR WITH RETURN FOR
    SELECT a * 3 FROM SYSIBM.SYSDUMMY1;
  OPEN c1;
END
```

PostgreSQL Cursors in Stored Procedures

PostgreSQL also allow opening cursors in stored procedures which are then available in Python via mxODBC's `cursor.fetch*()` methods.

Example:

```
CREATE FUNCTION sp_result_set
(a INTEGER,
 rs OUT refcursor)
AS $$
BEGIN
  OPEN rs FOR
    SELECT a * 3;
END;
$$ LANGUAGE plpgsql;
```

5.7.6 SQL Output Statements in Stored Procedures

You should not use any output SQL statements such as `"PRINT"` in the stored procedures, since this will cause at least some ODBC drivers (notably the MS SQL Server one) to turn the output into an SQL error which causes the execution to fail.

On the other hand, these error messages can be useful to pass along error conditions to the Python program, since the error message string will be the output of the `"PRINT"` statement.

5.8 Introspection

5.8.1 Database Schema Introspection

mxODBC provides the full set of ODBC supported introspection cursor methods which allows querying most database schema details without having to rely on database specific internal system tables.

Usage is easy: Open a connection to the database, open a cursor on the connection. Then call a catalog method on the cursor and inspect the generated result set using the `cursor.fetch*()` methods.

Please see section 7.6.1 Catalog Methods for full details on the available catalog methods and their generated result sets.

5.8.2 Result Set Introspection

When working with dynamically generated SQL statements, you often need to check the result set layout in order to prepare for processing the result set.

Introspection via cursor.execute()

In mxODBC this can be done right after executing a SQL statement using one of the `cursor.execute*()` methods by looking at the `cursor.description` attribute.

Introspection via cursor.prepare()

Alternatively, you can use the `cursor.prepare()` method to just prepare execution of a SQL statement - without actually executing it. This may be desirable in case the result set is not immediately needed or the query would require a long time to execute.

The cursor.description attribute

This cursor attribute provides access to a sequence of tuples, each describing the result set column at that position: `(name, type_code, display_size, internal_size, precision, scale, null_ok)`.

If no information is available the `cursor.description` is set to None.

The column tuple entries have the following meanings (index given in square brackets):

`name [0]`

Name of the column as returned by the database.

`type_code [1]`

In mxODBC, this is the SQL type integer describing the database data type of the result set column. These are described in the section 8 *Supported Data Types* and are available through the `SQL` singleton defined at subpackage module level for comparisons.

`display_size [2]`

mxODBC will always return `None` for this field. For database tables, this information can be determined by using the `cursor.getcolattribute(position, SQL.DESC_DISPLAY_SIZE)` method, if needed.

`internal_size [3]`

mxODBC will always return `None` for this field. For database tables, this information can be determined by using the `cursor.getcolattribute(position, SQL.DESC_OCTET_LENGTH)` method, if needed.

`precision [4]`

Precision of numeric columns.

`scale [5]`

Scale of numeric columns.

`null_ok [6]`

Returns 1 if the column can contain NULL values (which are returned as `None` in Python).

The cursor.getcolattribute() method

The `cursor.getcolattribute()` method provides more information about the result set columns than the Python DB-API compatible `cursor.description` sequence.

It also allows querying for auto-increment columns, the base column and table name, the database specific type name, etc.

Please see the document for `cursor.getcolattribute()` in section 7.6 Cursor Object Methods for details.

5.9 ODBC Cursor Types

Starting with mxODBC 3.2, mxODBC supports several different ODBC cursor types. These types define how the cursors will be used by the application and whether or not the application will see changes to the result set while fetching the result set rows.

As with other cursor settings, the ODBC cursor type default value can be defined on the mxODBC connection object and the setting is then inherited by the mxODBC cursor objects created on that connection. Subsequent changes to the cursor type on the cursor do not affect the setting on the connection.

5.9.1 Adjusting/Inspecting the ODBC Cursor Type

Both connections and cursors expose a read/write `.cursortype` attribute for this purpose. The attribute uses the ODBC defined values for the cursor types.

The following values are defined in the ODBC standard:[10]

`SQL.CURSOR_FORWARD_ONLY`

The cursor only scrolls forward. This is the default setting used by mxODBC for all databases.[11]

`SQL.CURSOR_STATIC`

The result set is made static by creating a static copy of the result set after opening the cursor. As a result, any changes to the result set after opening the cursor will not be visible to the client. Databases typically require setting the cursor type to static to support backwards scrolling in the result set via the `cursor.scroll()` call. Please note that creating a static copy can result in a significant performance degradation, esp. with MS SQL Server and less so with IBM DB2.

`SQL.CURSOR_KEYSET_DRIVEN`

Keysets are sets of columns in the result set that provide unique keys to the rows in the result set. Keyset driven cursors fix the memberships and order of the rows in the result set using these keysets. Unlike static cursors, they don't create a copy of the result set.

`SQL.CURSOR_DYNAMIC`

Dynamic cursors are the opposite of static cursors. All changes to the result set after opening it are visible on the next fetch operation.

[10] The `SQL` global refers to the mxODBC subpackage global of the same name, e.g. for `mx.ODBC.Windows`, this is accessible as `mx.ODBC.Windows.SQL`.
[11] Please note that in mxODBC 3.2, the default was database dependent.

Please note that using cursor types other than `SQL.CURSOR_FORWARD_ONLY` may have a significant effect on the performance of fetch operations. Not all databases support all listed cursor types.

5.9.2 Default Cursor Type

mxODBC defaults to using forward-only cursors with all databases.[12]

Some databases also support other cursor types, which may be useful in certain application settings, e.g. to make sure that the result set cannot change while fetching it, or to enable backwards scrolling through a result set.

mxODBC therefore allows changing the cursor type on a per connection or cursor basis using the `connection.cursortype` / `cursor.cursortype` attributes.

You can check the currently used cursor type by inspecting the `.cursortype` attribute on a newly created connection/cursor objects:

```
if connection.cursortype == mx.ODBC.Windows.SQL.CURSOR_STATIC:
    print "Connection uses static cursors."
elif connection.cursortype == mx.ODBC.Windows.SQL.CURSOR_FORWARD_ONLY:
    print "Connection uses forward only cursors."
```

As with most connection attributes, the setting is inherited by the cursors created on the connection at creation time. You can adjust the `.cursortype` on a cursor prior to executing a statement by assigning to the `cursor.cursortype` attribute:

```
cursor.cursortype == mx.ODBC.Windows.SQL.CURSOR_STATIC
```

in case you need e.g. static cursor behavior for that cursor.

Backwards Compatibility Notice:
Please note that mxODBC 3.2 used to set the default cursor type depending on whether the database supports static cursors or not. For those that do, it used static cursors, for all others, it used forward-only cursors. In mxODBC 3.3, we have changed this back to defaulting to forward-only cursors for all databases due to performance issues with static cursors.

5.9.3 Effects of the Cursor Type on cursor.rownumber

Some databases do not provide accurate information for `cursor.rownumber` in case forward-only cursors are used. Examples are MS Access, Teradata and Oracle.

Since mxODBC 3.3 switched to forward-only cursors for all databases, we have added a special client-side `.rownumber` emulation to mitigate this problem.

[12] In mxODBC 3.2, mxODBC used to default to static cursors for some databases such as MS SQL Server and IBM DB2, but this was found to cause performance problems.

If the emulation cannot be provided or the value cannot be determined, the `cursor.rownumber` attribute is set to `None`.

5.9.4 Database Specific Cursor Type Notes

MS SQL Server

MS SQL Server supports backwards scrolling and other more advanced scrolling modes when using static cursors. With forward only cursors, these scrolling capabilities are no longer available. The attribute `.rownumber` is emulated by mxODBC, since SQL Server returns wrong results with forward cursors.

We still recommend setting the `.cursortype` to `SQL.CURSOR_FORWARD_ONLY` in case result set scrolling is not needed by the application, since fetch operations on static cursors can result in a significant slow-down compared to forward only cursors.

Adjusting the default cursor type can be done on a per connection basis:

```
connection = mx.ODBC.Windows.DriverConnect(…)
connection.cursortype = mx.ODBC.Windows.SQL.CURSOR_STATIC
# All cursors created on connection will then default to static
# cursor type, allowing backwards scrolling.
```

Performance Hint:
In tests we have seen **slow-downs from 2-3x** for average queries, **up to 300x** for simple ones, when using static cursors compared to forward only ones.

Oracle

mxODBC uses forward only cursors as default, since using other types can cause **segfaults in the ODBC driver**.

PostgreSQL

mxODBC defaults to forward only cursors, since the **driver becomes unusable** with other settings.

IBM DB2

Just like MS SQL Server, IBM DB2 supports static cursors as well, so you can enable these, if you need scrolling support or `.rowcount` information.

There is a performance penalty of about 2x in using static cursors.

```
connection = mx.ODBC.Windows.DriverConnect(…)
connection.cursortype = mx.ODBC.Windows.SQL.CURSOR_STATIC
# All cursors created on connection will then default to static
# cursor type.
```

Note that with forward cursors, the `.rowcount` attribute does not always give correct results with DB2.

5.10 Custom Cursor Row Objects and Row Factory Functions

In some situation you may want to have mxODBC return a different Python object type or class when fetching rows from the database, e.g. ones which offer not only indexes based access to the row fields, but also named attribute or index access.

mxODBC 3.3 introduced two new cursor attributes `cursor.row` and `cursor.rowfactory` which can be used to customize the objects returned by mxODBC in result sets when using the `cursor.fetch*()` methods.

Default is to return standard Python tuples for rows in the result, since this is the most performant way of providing access to the data.

With the new attributes, it is possible to have mxODBC automatically return other sequence types (e.g. lists), hybrids providing sequence/mapping/attribute access to the columns or higher-level abstraction layer objects.

5.10.1 Cursor Row Constructor: cursor.row

In order to have mxODBC use a different row constructor than the tuple constructor, set the `cursor.row` attribute to a function or class which takes a single parameter, the row tuple, and returns an object for the row.

Setting the value has an immediate effect on subsequent `cursor.fetch*()` calls. mxODBC does not reset this attribute after the fetch operation, so the setting persists until set to another constructor or `None`.

Setting `cursor.row` to `None` resets the row constructor to the mxODBC default of using Python tuples.

Examples:

```
# Have mxODBC return lists for rows instead of tuples:
cursor.row = list
result_set = cursor.fetchall()

# Have mxODBC return tuples again:
cursor.row = None
result_set = cursor.fetchall()

# Have mxODBC return a custom object instead of tuples:
cursor.row = MyRow
result_set = cursor.fetchall()
```

Attribute Inheritance: cursor.row and connection.row

As with many other cursor attributes, the `cursor.row` attribute inherits its default value from the connection on which the cursor was created. At creation time, the `cursor.row` attribute is set to `connection.row`.

Adjusting `connection.row` after the cursor was created does not have an effect on `cursor.row` anymore. The connection attribute setting is only used when creating the cursor.

5.10.2 Cursor Row Factories: cursor.rowfactory

The `cursor.row` constructor attribute is useful for object types that don't depend on the result set that is being returned or where you know that the result set rows are going to have a specific layout before running the query.

On-the-fly Creation of Row Classes

In many cases, you will want to use custom row objects even for queries that depend on external settings or where you don't want to create a specific object class in advance.

This is where the `cursor.rowfactory` attribute comes in handy. It allows you to specify a factory function which provides the row constructor depending on the cursor that just executed a query or statement.

When set, the row factory function `cursor.rowfactory` is called with the cursor as first parameter when calling the first `cursor.fetch*()` method on the cursor to fetch a new result set, but before actually fetching the first row.

The return value is then assigned to `cursor.row`, which then results in the `cursor.row` constructor to be used for fetching the rows of the result set.

This makes it easy to define your own factory functions to programmatically define row classes based on the `cursor.description` or other cursor parameters.

Row Factories and multiple Result Sets

If you are using multiple result sets, the `cursor.rowfactory` is called for the first fetch in each of the result sets you are reading from the database.

Predefined Row Factories

mxODBC comes with a set of useful row factories which provide both sequence index access as well as named attribute access. These are defined in the `mx.ODBC.Misc.RowFactory` module, which is imported into all mxODBC subpackages for convenience. See section 13 mx.ODBC.Misc.RowFactory Module for the API documentation.

The module defines these row factories:

`RowFactory.TupleRowFactory`

This is a factory which is a subtype of the Python tuple type and provides a standard tuple index based access to the row column values, as well as an

attribute based one which is derived from the lower-cased column names found in `cursor.description`.

The row objects are immutable, just like standard tuples, but you can also slice them or index them as usual.

Example:

```
from mx.ODBC.Windows import RowFactory
cursor.rowfactory = RowFactory.TupleRowFactory
cursor.execute('select x, Y, z from mytable')
row = cursor.fetchone()
print (row[0], row[1], row[2], row.x, row.y, row.z)
```

Because the row objects implement the sequence protocol, they are also usable as input parameters to the `cursor.execute*()` methods.

Example:

```
cursor.execute('insert into mytable values (x, Y, z)', row)
```

`RowFactory.ListRowFactory`

This factory uses a subtype of the Python list type and also provides a sequence index based access, as well as a named attribute access, just like the `TupleRowFactory`.

Unlike for the `TupleRowFactory`, the row objects created by the `ListRowFactory` are mutable lists, so you can assign to the indexes as well as the attributes.

Example:

```
from mx.ODBC.unixODBC import RowFactory
cursor.rowfactory = RowFactory.ListRowFactory
cursor.execute('select x, Y, z from mytable')
row = cursor.fetchone()
print (row[0], row[1], row[2], row.x, row.y, row.z)
row[0] = 10
row[1] = 'abc'
print (row[:2])
# will print [10, 'abc']
row.x = 20
row.y = 'def'
print (row[:2])
# will print [20, 'def']
```

The row objects are also usable as input for the `cursor.execute*()` methods.

Example:

```
cursor.execute('insert into mytable values (x, Y, z)', row)
```

`RowFactory.NamespaceRowFactory`

The `NamespaceRowFactory` creates `mx.Misc.Namespace.Namespace` objects as row objects. These provide a more complex namespace oriented API.

In addition to the sequence protocol, they also allow mapping access as well as named attribute access based on the lower-cased column names read from `cursor.description`.

Rows created by this factory are mutable.

Example:

```
from mx.ODBC.iODBC import RowFactory
cursor.rowfactory = RowFactory.NamespaceeRowFactory
cursor.execute('select x, Y, z from mytable')
row = cursor.fetchone()
print (row[0], row[1], row[2])
print (row.x, row.y, row.z)
print (row['x'], row['y'], row['z'])
row[0] = 10
row[1] = 'abc'
print (row[:2])
# will print [10, 'abc']
row.x = 20
row.y = 'def'
print (row[:2])
# will print [20, 'def']
row['x'] = 30
row['y'] = 'ghi'
print (row[0], row[1])
# will print [30, 'ghi']
```

The row objects are also usable as input for the `cursor.execute*()` methods.

Example:

```
cursor.execute('insert into mytable values (x, Y, z)', row)
```

All factories create row classes on-the-fly, based on `cursor.description`.

Factory created Row Classes and pickle

Because of the way these row classes are dynamically created, they **are by default not pickleable**.

In order to be pickleable, they would have to be saved to a module namespace, so that pickle can recreate them at load time. Since the creation parameters depend on the cursor state at creation time, this is not easily possible.

If you know that a cursor will use particular result set layout, you can statically create the row class using the factory functions, register the class in a module and then set the `cursor.row` directly to the row class.

Example:

```
# Create a cursor with information about the result set
cursor.execute('select * from mytable')

# Create the row class
MyTableRow = RowFactory.TupleRowFactory(cursor)

# Adjust the MyTableRow class so that pickle can find the right module
# (__name__) and class name ('MyTableRow')
MyTableRow.__name__ = 'MyTableRow'
MyTableRow.__module__ = __name__

# Fetch the data using MyTableRow objects
cursor.row = MyTableRow
rows = cursor.fetchall()
```

The resulting rows list is pickleable, since it can find the module and class name.

Attribute Inheritance: cursor.rowfactory and connection.rowfactory

Just like for cursor.row, the `cursor.rowfactory` attribute inherits its default value from the connection on which the cursor was created. At creation time, the `cursor.rowfactory` attribute is set to `connection.rowfactory`.

Adjusting `connection.rowfactory` after the cursor was created does not have an effect on `cursor.rowfactory`. The connection attribute setting is only used when creating the cursor.

5.11 mxODBC Subpackages

The mxODBC package is organized in subpackages, with one package per support ODBC driver manager and in custom builds, additional subpackages for specific drivers/databases.

See section 14 mx.ODBC Driver/Manager Packages for details on available subpackages.

5.11.1 One API for all Subpackages

To make applications portable between ODBC database backends, each of these subpackages use the same names and API signatures, in fact, the same mxODBC implementation is used for each of the subpackages, customized to meet the respective ODBC driver/manager's specific requirements.

As an example, say if you are using the `mx.ODBC.Windows` subpackage, then the constructor to call would be `mx.ODBC.Windows.DriverConnect()`. When porting the application to Unix you'd use e.g. the `mx.ODBC.iODBC` subpackage and the constructor then becomes `mx.ODBC.iODBC.DriverConnect()`.

In your application you'd just have to change the top-level import from

```
from mx.ODBC import Windows as Database
```

to

```
from mx.ODBC import iODBC as Database
```

The subpackage globals such as exception names, helper functions, connection constructors, etc. remain the same, so no additional changes are necessary.

Of course, you will usually have to rely on different ODBC drivers when switching from Windows to Unix or the other way around. While the mxODBC API names

and signatures don't change, you will likely have to make some application level changes to accommodate for differences in the ODBC drivers you are using.

6. mxODBC Connection Objects

Connection objects provide the communication link between your Python application and the database. They are also the scope of transactions you perform. Each connection can be setup to your specific needs, multiple connections may be opened at the same time.

6.1 Subpackage Support

Connection objects are supported by all subpackages included in mxODBC.

The extent to which the functionality and number of methods is supported may differ from subpackage to subpackage, so you have to verify the functionality of the used methods (esp. the catalog methods) for each subpackage and database that you intend to use.

6.2 Connection Type Object

mxODBC uses a dedicated object type for connections. Each mxODBC subpackage defines its own object type, but all share the same name: `ConnectionType`.

6.3 Connection Object Constructors

```
Connect(dsn, user='', password='', clear_auto_commit=1,
   errorhandler=None, connection_options=())
```

This constructor returns a connection object for the given data source. It accepts keyword arguments. `dsn` indicates the data source to be used, `user` and `password` are optional and used for database login.

`errorhandler` may be given to set the error handler for the Connection object prior to actually connecting to the database. This is useful to mask e.g. certain warnings which can occur at connection time. The `errorhandler` can be changed after the connection has been established by assigning to the `.errorhandler` attribute of the Connection object. The default error handler raises exceptions for all database warnings and errors. See section 10.4 Error Handlers for more details on how to use error handlers.

If you connect to the database through an ODBC manager, you should use the `DriverConnect()` API since this allows passing more configuration information to the manager and thus provides more flexibility over this interface.

See the following section 6.4 Default Transaction Settings for details on `clear_auto_commit`.

`connection_options` may be given as list of `(option, value)` tuples to set pre-connect ODBC connection options. The `option` and `value` arguments must use the same format as the parameters for the `.setconnectoption()` method. This list can be used to e.g. enable the *MARS feature of SQL Server Native Client*, which enables working with multiple active result sets on the same connection:

```
from mx.ODBC.Manager import DriverConnect, SQL
options = [(SQL.COPT_SS_MARS_ENABLED, SQL.MARS_ENABLED_YES)]
db = DriverConnect('DSN=mssqlserver2008;UID=sa;PWD=dbs0R-X9.rxD',
                   connection_options=options)
```

```
connect(dsn, user='', password='', clear_auto_commit=1,
   errorhandler=None)
```

Is just an alias for `Connect()` needed for Python DB API 2.0 compliance.

```
DriverConnect(DSN_string, clear_auto_commit=1, errorhandler=None)
```

This constructor returns a connection object for the given data source which is managed by an ODBC Driver Manager (e.g. the Windows ODBC Manager or iODBC). It allows passing more information to the database than the standard `Connect()` constructor.

`errorhandler` may be given to set the error handler for the Connection object prior to actually connecting to the database. This is useful to mask e.g. certain warnings which can occur at connection time. The `errorhandler` can be changed after the connection has been established by assigning to the `.errorhandler` attribute of the Connection object. The default error handler raises exceptions for all database warnings and errors. See section 10.4 Error Handlers for more details on how to use error handlers.

Please refer to the documentation of your ODBC manager and the database for the exact syntax of the `DSN_string`. It typically has this formatting: `'DSN=datasource_name;UID=userid;PWD=password'` (case can be important and more entries may be needed to successfully connect to the data source).

See the following section 6.4 Default Transaction Settings for details on `clear_auto_commit`.

`connection_options` may be given as list of `(option, value)` tuples to set pre-connect ODBC connection options. The `option` and `value` arguments must use the same format as the parameters for the `.setconnectoption()` method. This list can be used to e.g. enable the *MARS feature of SQL Server Native Client*, which enables working with multiple active result sets on the same connection:

```
from mx.ODBC.Manager import DriverConnect, SQL
options = [(SQL.COPT_SS_MARS_ENABLED, SQL.MARS_ENABLED_YES)]
db = DriverConnect('DSN=mssqlserver2008;UID=sa;PWD=dbs0R-X9.rxD',
                   connection_options=options)
```

> The `DriverConnect()` API is only available if the ODBC driver or ODBC driver manager supports this. It is available on all supported ODBC driver manager subpackages such as the one for Windows and iODBC/unixODBC/DataDirect on Unix platforms. See the *subpackages section* for details.

```
ODBC(dsn, user='', password='', clear_auto_commit=1,
   errorhandler=None)
```

Is just an alias for `Connect()` needed for Python DB API 1.0 compliance.

6.4 Default Transaction Settings

ODBC usually defaults to auto-commit, meaning that all actions on the connection are directly applied to the database. Since this can be dangerous, mxODBC defaults to turning auto-commit off at connection initiation time provided the database supports transactions.

All connection constructors implicitly start a new transaction when connecting to a database in transactional mode.

When connecting to a database with transaction support, you should explicitly do a `.rollback()` or `.commit()` prior to closing the connection. mxODBC does an automatic rollback of the transaction when the connection is closed if the driver supports transactions.

6.4.1 Overriding the Default

The value of the `clear_auto_commit` connection parameter overrides this default behavior. Passing a `0` as value disables the clearing of the auto-commit flag and lets the connection use the database's default commit behavior. Please see the database documentation for details on its default transaction setting.

Use the connection method `connection.setconnectoption(SQL.AUTOCOMMIT, SQL.AUTOCOMMIT_ON|OFF|DEFAULT)` to adjust the connection's behavior to your needs after the connection has been established, but before you have opened a database cursor.

With auto-commit turned on, transactions are effectively disabled. The `rollback()` method will raise a `NotSupportedError` when used on such a connection.

6.4.2 Errors due to missing Transaction Support

If you get an exception during connect telling you that the driver is not capable or does not support transactions, e.g. `mxODBC.NotSupportedError: ('S1C00', 84, '[Microsoft][ODBC Excel Driver]Driver not capable ', 4226)`, try to connect with `clear_auto_commit` set to `0`. mxODBC will then keep auto-commit switched on and the connection will operate in auto-commit mode.

6.5 Connection objects as context managers

6.5.1 Introduction to Context Managers

Python 2.5 introduced the new concept of context manager to Python. Context managers are Python objects implementing the context manager API based on the methods .__enter__() and .__exit__().

The context managers can be used together with the Python with-statement to wrap sections of a program into a block (the context) that is entered and exited in a controlled way:

```
with context_manager as context:
    context.do_something()
```

When entering the block, the `context_manager`'s `.__enter__()` method is called and the returned object assigned to `context`. When exiting the block, the `context.__exit__()` is called, either with the exception that caused the block to be left or without exception in case the block was left normally.

6.5.2 Using connection objects as context object

Connection objects implement this API and use it to automatically commit or roll back the current transaction.

```
from mx.ODBC.Manager import DriverConnect
connection = DriverConnect(…)
with connection:
    cursor = connection.cursor()
    cursor.execute('INSERT INTO table VALUES (?, ?)', (1, 2))
    … other tasks …
    cursor.close()
```

This code will automatically commit the INSERT to the database backend in cae the with-block is left without exception. If the other tasks trigger an unhandled transaction, the connection is rolled back when leaving the block.

For code which doesn't have to do more complex error handling, using the with-statement block can greatly simplify the resulting code. It also gives the transaction section a visible resemblance in the code.

Cursors also support the context manager API, so the above could be simplified even more to:

```
with connection:
    with connection.cursor() as cursor:
        cursor.execute('INSERT INTO table VALUES (?, ?)', (1, 2))
        … other tasks …
```

6.6 Unicode/ANSI Connections

Starting with mxODBC 3.1, it is possible to tell the ODBC driver manager whether to use the Unicode ODBC interface of a supporting ODBC driver or the ANSI (8-bit string) ODBC interface at connection time.

6.6.1 Unicode ODBC Interface

If the ODBC driver supports the ODBC Unicode interface and you select the Unicode interface by using a Unicode string as connection parameter, the ODBC manager will subsequently convert all ANSI-parameters to Unicode and then call the Unicode APIs of the ODBC driver. Unicode parameters are passed through as-is to the ODBC driver.

For ODBC drivers that natively support the ODBC Unicode interface, connecting using a Unicode connection string and subsequently using Unicode parameters for all execution and catalog methods may result in better performance or improved compatibility.

Example:

```
# Use the Unicode ODBC API of the driver by using a Unicode connection
# string
db = mx.ODBC.Windows.DriverConnect(u'DSN=mydb;UID=uid;PWD=pwd')
```

6.6.2 ANSI ODBC Interface

If the ODBC driver does not support the ODBC Unicode interface, or you connect using an ANSI (8-bit string), the ODBC driver manager will subsequently convert all Unicode parameters to the connection's ANSI code page before calling the ANSI API on the ODBC driver. ANSI parameters are passed through as-is to the ODBC driver.

For ODBC drivers that do not support the ODBC Unicode interface, connecting using an ANSI connection string and subsequently using ANSI parameters for all execution and catalog methods may result in better performance or improved compatibility.

Example:

```
# Use the ANSI ODBC API of the driver by using an 8-bit connection
```

```
# string
db = mx.ODBC.Windows.DriverConnect('DSN=mydb;UID=uid;PWD=pwd')
```

6.7 Connection Object Methods

`.close()`

Close the connection now (rather than automatically at garbage collection time). The connection will be unusable from this point on; an `Error` (or subclass) exception will be raised if any operation is attempted with the connection. The same applies to all cursor objects trying to use the connection.

`.commit()`

Commit any pending changes and implicitly start a new transaction.

> For connections which do not provide transaction support or operate in auto-commit mode, this method does nothing.

`.cursor(name=None, cursor_options=())`

Constructs a new *Cursor Object* with the given name using the connection and initializes any provided cursor options.

If no name is given, the ODBC driver or database backend will determine a unique name on its own. You can query this name with `cursor.getcursorname()` (see the *Cursor Object* section 7).

The `cursor_options` may be given as list of `(option, value)` tuples. These are then passed to the cursor's `cursor.setconnectoption()` API (see the *Cursor Object* section 7) and allow configuring the cursor upfront to a specific need.

`.getconnectoption(option)`

Get information about the connection.

`option` must be an integer. Suitable option values are available through the `SQL` object (see the *Constants* section 10.5 for details).

The method returns the data as 32-bit integer. It is up to the user to decode the integer value using the SQL defines available through the `SQL` constant.

This API gives you a very wide range of information about the underlying database and its capabilities. See the *ODBC SQLGetConnectAttr API Documentation* for more information.

`.getinfo(info_id)`

Get general information about the database, the ODBC driver and the ODBC driver manager.

The `info_id` must be an integer. Suitable values are available through the `SQL` object (see the *Constants* section 10.5 for details).

The method returns a tuple `(integer, string)` giving an integer decoding (in native integer byte order) of the first bytes of the API's result as well as the raw buffer data as string. It is up to the caller to decode the data (e.g. using the `struct` module).

This API gives you a very wide range of information about the underlying database and its capabilities. See the *ODBC SQLGetInfo API Documentation* for more information.

`.nativesql(command)`

This method returns the `command` as it would have been modified by the driver to pass to the database engine. It is a direct interface to the ODBC API `SQLNativeSql()`.

In many cases it simply returns the `command` string unchanged. Some drivers unescape ODBC escape sequences in the command string. Syntax checking is usually not applied by this method and errors are only raised in case of command string truncation.

Not all mxODBC subpackages support this API.

`.rollback()`

In case the database connection has transactions enabled, this method causes the database to roll back any changes to the start of the current transaction.

Closing a connection without committing the changes first will cause an implicit rollback to be performed.

This method is only available if the ODBC driver database subpackage was compiled with transaction support. For ODBC manager subpackages it is always available, but may raise a `NotSupportedError` in case the connection does not support transactions.

`.setconnectoption(option, value)`

This method lets you set some ODBC integer options to new values, e.g. to set the transaction isolation level or to turn on auto-commit.

`option` must be an integer. Suitable option values are available through the `SQL` object, e.g. `SQL.ATTR_AUTOCOMMIT` corresponds to the SQL option `SQL_ATTR_AUTOCOMMIT` in C (see the *Constants* section 10.5 for details).

The method is a direct interface to the ODBC `SQLSetConnectOption()` function. Please refer to the *ODBC Documentation* for more information.

Note that while the API function also supports setting character fields, the method currently does not know how to handle these.

Note for ADABAS/SAP DB/MAX DB users:
Adabas, SAP DB and MAX DB can emulate several different SQL dialects. They have introduced an option for this to be set. These are the values you can use: 1 = ADABAS, 2 = DB2, 3 = ANSI, 4 = ORACLE, 5 = SAPR3. The option code is `SQL.CONNECT_OPT_DRVR_START + 2` according to the Adabas documentation. Please consult your driver documentation for details.

`.__enter__()`

Returns the connection itself. This method makes connection objects usable as context manager (together with the .__exit__() method) and is called when entering a *with*-block (new in Python 2.5).

`.__exit__(exc_type, exc_value, exc_tb)`

Returns True in case `exc_type` is set to None (no exception set) and commits the current transaction. Returns False in case `exc_type` is set to an exception and rolls back the current transaction. This method is part of the context manager API and is called when leaving a *with*-block (new in Python 2.5).

6.8 Connection Object Attributes

`.autocommit`

Writeable attribute to query and set the auto-commit status of the connection.

Returns `True` if the connection is operating in auto commit (non-transactional) mode. Returns `False` is the connection is operating in manual commit (transactional) mode.

Setting the attribute to `True` or `False` adjusts the connection's mode accordingly.

This attribute is a shortcut to using `connection.setconnectoption(SQL.AUTO_COMMIT, value)` and can raise the same exceptions, e.g. in case of a closed connection.

`.bindmethod`

Attribute to query and set the input variable binding method used by the connection. This can either be `BIND_USING_PYTHONTYPE` of `BIND_USING_SQLTYPE` (see the *Constants* section 10.5 for details).

The attribute is inherited by cursors created on the connection at creation time. Cursors may override the setting on a per cursor basis.

`.closed`

Read-only attribute that is true in case the connection is closed. Any action on a closed connection will result in a `ProgrammingError` to be raised. This variable can be used to conveniently test for this state.

`.converter`

Read/write attribute that sets the converter callback default for all newly created cursors using the connection. It is `None` per default (meaning to use the standard conversion mechanism). See the *Supported Data Types* section for details.

`.cursortype`

Read/write attribute that sets the default ODBC cursor type for cursors created on this connection. Possible values are:

`SQL.CURSOR_FORWARD_ONLY`

The cursor only scrolls forward. This is the default setting for all databases.[13]

`SQL.CURSOR_STATIC`

The result set is made static by creating a static copy of the result set after opening the cursor. As a result, any changes to the result set after opening the cursor will not be visible to the client.[13]

`SQL.CURSOR_KEYSET_DRIVEN`

Keysets are sets of columns in the result set that provide unique keys to the rows in the result set. Keyset driven cursors fix the memberships and order of the rows in the result set using these keysets. Unlike static cursors, they don't create a copy of the result set.

`SQL.CURSOR_DYNAMIC`

Dynamic cursors are the opposite of static cursors. All changes to the result set after opening it are visible on the next fetch operation.

Please refer to section 5.9 ODBC Cursor Types for more details on cursor types. Not all databases support all listed cursor types.

Performance Warning:
Please note that using cursor types other than `SQL.CURSOR_FORWARD_ONLY` may have a significant effect on the performance of fetch operations.

`.datetimeformat`

Use this instance variable to set the default output format for date/time/timestamp columns of all cursors created using this connection object.

Possible values are (see the *Constants* section 10.5 for details):

`DATETIME_DATETIMEFORMAT (default)`

DateTime and DateTimeDelta instances.

`PYDATETIME_DATETIMEFORMAT`

datetime.date, datetime.time, datetime.datetime instances. Only available using Python 2.4 and later.

`TIMEVALUE_DATETIMEFORMAT`

Ticks (number of seconds since the epoch) and tocks (number of seconds since midnight).

[13] Please note that in mxODBC 3.2, the default was database dependent.

`TUPLE_DATETIMEFORMAT`

Python tuples as defined in the *Supported Data Types* section.

`STRING_DATETIMEFORMAT`

Python strings. The format used depends on the internal settings of the database. See your database's manuals for the exact format and ways to change it.

We strongly suggest always using the DateTime/DateTimeDelta instances. Note that changing the values of this attribute will not change the date/time format for existing cursors using this connection.

This value is inherited by all cursors created from the connection at creation time. Note that changing the value of this attribute will not change the date/time format for existing cursors using this connection.

`.dbms_name`

String identifying the database manager system.

`.dbms_version`

String identifying the database manager system version.

`.decimalformat`

Use this instance variable to set the default output format for decimal and numeric columns of all cursors created using this connection object.

Possible values are (see the *Constants* section 10.5 for details):

`FLOAT_DECIMALFORMAT (default)`

Values are returned as Python floats.

`DECIMAL_DECIMALFORMAT`

Values are returned as Python decimal.Decimal instances. Only available using Python 2.4 and later.

This value is inherited by all cursors created from the connection at creation time. Note that changing the value of this attribute will not change the decimal format for existing cursors using this connection.

`.driver_name`

String identifying the ODBC driver.

`.driver_version`

String identifying the ODBC driver version.

`.encoding`

Read/write attribute which defines the encoding to use for converting Unicode to 8-bit strings and vice-versa. If set to `None` (default), Python's default encoding will be used, otherwise it has to be a string providing a valid encoding name, e.g. `'latin-1'` or `'utf-8'`.

The `connection.encoding` is used on connection related APIs and also passed to cursors created on the connection at creation time. Cursors store the encoding in `cursor.encoding` and cursor related APIs will use the cursor setting instead of the connection setting.

`.errorhandler`

Read/write attribute which defines the error handler function to use. If set to `None`, the default handling is used, i.e. errors and warnings all raise an exception and get appended to the `.messages` list.

An error handler must be a callable object taking the arguments `(connection, cursor, errorclass, errorvalue)` where `connection` is a reference to the connection, `cursor` a reference to the cursor (or `None` in case the error does not apply to a cursor), `errorclass` is an error class which to instantiate using `errorvalue` as construction argument.

See section 10.4 Error Handlers for more details on how to use error handlers.

`.license`

String with the license information of the installed mxODBC license.

`.messages`

This is a Python list object to which mxODBC appends tuples `(exception class, exception value)` for all messages which the interfaces receives from the underlying ODBC driver or manager for this connection.

The list is cleared automatically by all connection methods calls (prior to executing the call) except for the info and connection option methods calls to avoid excessive memory usage and can also be cleared by executing `del connection.messages[:]`.

All error and warning messages generated by the ODBC driver are placed into this list, so checking the list allows you to verify correct operation of the method calls.

`.paramstyle`

Sets the default parameter binding style for cursors created on this connection, i.e. all cursors created on the connection will use `connection.paramstyle` as their default `cursor.paramstyle` value.

The attribute can be set or queried and takes the following string values (following the `paramstyle` module global as defined in the DB-API):

`'qmark' (default)`

This is the default ODBC parameter binding style and also used as native database binding style by MS SQL Server and IBM DB2.

Parameters in SQL statements used on `cursor.execute*()` methods are marked with the question mark letter (`'?'`) and the variables are bound to these parameter locations using a positional mapping. Parameter values for a SQL statement must be specified as sequence, normally a list or a tuple.

Example: `'SELECT * FROM MyTable WHERE A=? AND B=?'` used with a parameter tuple `(1, 2)` would result in the database executing the query `'SELECT * FROM MyTable WHERE A=1 AND B=2'`.

`'named'`

The 'named' parameter binding style is used by the native database interfaces of e.g. Oracle.

Parameters in SQL statements used on `cursor.execute*()` methods are marked with a colon followed by a name, e.g. `':a'` or `':1'`. The variables are bound to these parameter locations using a name based mapping. Parameter values for a SQL statement must be specified as mapping, normally a dictionary, and are bound to the locations based on the names used in the SQL statement.

Example: `'SELECT * FROM MyTable WHERE A=:a AND B=:b'` used with a parameter dictionary `{'a': 1, 'b': 2}` would result in the database executing the query `'SELECT * FROM MyTable WHERE A=1 AND B=2'`.

`.row`

This attribute sets the default `cursor.row` object constructor to be used by all newly created cursor objects on this connection.

The purpose of this attribute is to define a constructor for rows in result sets fetched using the `cursor.fetch*()` methods.

Default is `None`, which means that mxODBC will use regular Python tuples for returning row data in result sets.

Please see the `cursor.row` *cursor attribute documentation* in section 7.7 for more details.

`.rowfactory`

This attribute sets the default `cursor.rowfactory` row object constructor factory to be used by all newly created cursor objects on this connection.

The purpose of the row factory is to dynamically set the `cursor.row` attribute after having executed a statement on the cursor.

Default is `None`, which means that mxODBC will not use a row factory function and leave `cursor.row` untouched.

Please see the `cursor.rowfactory` *cursor attribute documentation* in section 7.7 for more details.

`.stringformat`

Use this attribute to set or query the default input and output handling for string columns of all cursors created using this connection object. Data conversion on input is dependent on the input binding type.

Possible values are (see the *Constants* section 10.5 for details):

`EIGHTBIT_STRINGFORMAT (default)`

This format tells mxODBC to convert all data passed to and read from the ODBC driver to 8-bit strings.

On input, Python 8-bit strings are passed to the ODBC driver as-is. Unicode objects are converted to Python 8-bit strings assuming the cursor's encoding setting (see the `cursor.encoding` attribute) prior to passing them to the ODBC driver.

On output, all string columns are fetched as strings and passed back as Python 8-bit string objects. Unicode data from the database is converted to Python 8-bit string objects assuming the cursor's encoding setting (see the `cursor.encoding` attribute).

This setting emulates the behavior of previous mxODBC versions and is the default.

`MIXED_STRINGFORMAT`

This format lets the ODBC driver decide which string format to use for the communication, providing the most efficient way of communicating with the driver.

Input and output conversion is dependent on the data format the ODBC driver expects or returns for a given column. If the driver returns a string, a Python string is created; if it returns Unicode data, a Python Unicode object is used.

`UNICODE_STRINGFORMAT`

This format can be used to emulate Unicode support with a database backend that doesn't have a native Unicode data type or where the ODBC driver cannot handle Unicode data.

On input, Python strings are passed to the ODBC driver as-is. Unicode objects are converted to 8-bit strings using the cursor's encoding setting (see the `cursor.encoding` attribute) and then passed to the ODBC driver.

On output, string data is converted to Python Unicode objects, based on the same conversion technique.

Use this setting if you plan to use Unicode objects with non-Unicode aware databases (e.g. by setting the encoding to UTF-8 -- be careful though: multibyte character encodings usually take up more space and are not necessarily compatible with the database's string functions).

`NATIVE_UNICODE_STRINGFORMAT`

This format should be used for databases and applications that support native Unicode data communication.

String columns are converted to Python Unicode objects assuming the cursor's encoding setting (see the `cursor.encoding` attribute) and then passed as Unicode to the ODBC driver.

On output, string data is always fetched as Unicode data from the ODBC driver and returned using Python Unicode objects.

Note that even though mxODBC may report that Unicode support is enabled (default in Python 2.0 and later; `HAVE_UNICODE_SUPPORT` is set to `1`), the ODBC driver may still reject Unicode data. In this case, an `InternalError` of

type `'S1003'` is raised whenever trying to read data from the database in this `.stringformat` mode.

You can use the included mx/ODBC/Misc/test.pyc script to find out whether the database backend support Unicode or not.

Binary and other plain data columns will still use 8-bit strings for interfacing, since storing this data in Unicode objects would cause trouble. mxODBC will eventually use buffer/memoryview or some form of binary objects to store binary data in some future version, e.g. the new bytes type which was introduced with Python 3.

This value is inherited by all cursors created from the connection at creation time. Note that changing the value of this attribute will not change the string format for existing cursors using this connection.

`.timestampresolution`

Use this attribute to adjust the rounding applied when passing second values with fractions to the database, i.e. from Python to the database[14]. Some databases complain about their data types not being capable of representing the precision as given in the fraction value. With others, it is possible to get rounding errors due to truncation, e.g. of 0.4999 to 0.49 instead of 0.50.

The attribute value must be given as integer and defines the resolution of the timestamp values in nanoseconds (ns).

Setting the attribute to 1000 would result in Python seconds values to get rounded to the nearest microsecond prior to passing the value to the database. Setting it to 250*1000000 would result in seconds to get rounded to the nearest ¼ second.

Note: Rounding to a full second is prevented to not cause possibly illegal time values.

Default is 1 nanosecond (`.timestampresolution = 1`), with the following exceptions to address limitations in the database engines, which otherwise cause database errors or warnings:

- *MS SQL Server 2005 and earlier:* 1 millisecond (`.timetstampresolution = 1000000`)
- *MS SQL Server 2008 and later:* 100 nanoseoncds (`.timestampresolution = 100`)

This value is inherited by all cursors created from the connection at creation time. Note that changing the value of this attribute will not change the timestamp resolution for existing cursors using this connection.

`.warningformat`

Use this attribute to change the default warning reporting behavior of mxODBC, in case you don't want to define your own `.errorhandler`.

[14] mxODBC applies this rounding when using the ODBC timestamp interface structures and also applies the rounding to mxDateTime input objects in case the database requests the date/time value as string. It currently does not apply the rounding for strings in case Python datetime objects are used on input.

The DB-API 2.0 mandates that database warnings must raised as `mx.ODBC.Warning` exception, but mostly because at the time of writing, the Python warning module did not yet exist.

For some applications it may be more useful to report warnings via Python warnings. The application could then use the standard Python warning filters to report or filter the warnings in an appropriate way.

Another alternative is to simply ignore such warnings. Some ODBC are rather verbose when it comes to warnings.

Note that a possibly registered `.errorhandler` will still be called in all these cases, however, the mxODBC default error handler will use the `.warningformat` to determine how to react to database warnings.

Possible values are (see the *Constants* section 10.5 for details):

`ERROR_WARNINGFORMAT (default)`

Report warnings in the usual DB-API 2.0 way and raise a Warning exception.

`WARN_WARNINGFORMAT`

Instead of raising a `Warning` exception, issue a `mx.ODBC.DatabaseWarning` which is a Python Warning subclass and can be filtered using the standard Python *warnings module* mechanisms.

`IGNORE_WARNINGFORMAT`

Silently ignore the database warning.

The warning will still be added to the `.message` attribute, but no further action is taken.

This value is inherited by all cursors created from the connection at creation time. Note that changing the value of this attribute will not change the warning format for existing cursors using this connection.

6.8.1 Additional Attributes

Error objects exposed as connection attributes

In addition to the above attributes, all exception objects used by the connection's subpackage are also exposed on the connection objects as attributes, e.g. `connection.Error` gives the `Error` exception of the subpackage which was used to create the connection object.

See the *Exceptions and Error Handling* section 10 for details and names of these error attributes.

BinaryNull constant exposed as connection attribute

`connection.BinaryNull` gives access to the subpackage global BinaryNull which must be used in some rare cases to work around driver issues with binding NULL values to binary column types.

7. mxODBC Cursor Objects

These objects represent a database cursor: an object which is used to manage the context of a database query operation.

This includes preparing and parsing the query or command to be executed on the connection, executing the query or command one or multiple times and providing a pointer into the result set or sets generated by queries.

7.1 Relationship between Cursors and Connections

7.1.1 Dependency on the Connection Object

Cursors are created through a database connection. As a result, cursor objects are only usable as long as the connection object exists and the associated database connection is open and working.

All operations of a cursor are done through the connection that was used to create it. The scope and default settings of a cursor are defined by the connection. Once created, you can change various settings of the cursor, e.g. the `cursor.datetimeformat`. Such changes do not affect the connection or any other cursor objects created on the connection.

Using cursors on a closed connection will result in a `ProgrammingError` to be raised.

7.1.2 Using multiple Cursor Objects on a single Connection

Depending on the capabilities of the database and the used ODBC driver, you can have multiple cursors open on a single connection and execute queries and commands on each at will. This makes it possible to e.g. prepare and then cache often used commands.

7.2 Subpackage Support

Cursor objects are supported by all subpackages included in mxODBC.

The extent to which the functionality and number of methods is supported may differ from subpackage to subpackage, so you have to verify the functionality of the used methods (esp. the catalog methods) for each subpackage and database that you intend to use.

7.3 Cursor objects as context managers

Please see section 6.5.1. Introduction to Context Managers for an introduction to context managers and the Python *with*-statement.

7.3.1 Using cursor objects as context objects

Cursor objects implement this API and use it to automatically close the cursor and freeing resources in the ODBC driver when leaving a *with*-block. Instead of writing:

```
cursor = connection.cursor()
try:
    cursor.execute('INSERT INTO table VALUES (?, ?)', (1, 2))
    … other tasks …
finally:
    cursor.close()
```

you can write:

```
with connection.cursor() as cursor:
    cursor.execute('INSERT INTO table VALUES (?, ?)', (1, 2))
    … other tasks …
```

This not only looks a lot better and also takes care of freeing the resources in case of an error in the block.

7.4 Cursor Type Object

mxODBC uses a dedicated object type for cursors.

Each subpackage defines its own object type, but all share the same name: `CursorType`.

7.5 Cursor Object Constructors

Cursor objects are created using the connection method `connection.cursor()`.

`connection.cursor(name=None, cursor_options=())`

Constructs a new *Cursor Object* with the given name using the connection and initializes any provided cursor options. If no name is given, the ODBC driver or database backend will create one dynamically.

Please see section 6.7 Connection Object Methods for details.

7.6 Cursor Object Methods

The following cursor methods are defined in the DB API:

`.callproc(procname, parameters=(), parametertypes=None)`

Call a stored database procedure with the given name.

The sequence of `parameters` must contain one entry for each argument that the procedure expects.

The result of the call is returned as modified list copy of the input sequence. Input parameters are left untouched, output and input/output parameters replaced with possibly new values in the list copy.

If `parametertypes` is given, it defines the parameter types of the parameters used in the `sqlcmd`. The sequence has to provide one integer entry per parameter. Possible values are `SQL.PARAM_INPUT` (input parameter), `SQL.PARAM_OUTPUT` (output parameter) and `SQL.PARAM_INPUT_OUTPUT` (input/output parameter). If `parametertypes` is not given, default is to assume input parameter types for all parameters.

The procedure may also provide one or more result sets as output. This can then be fetched through the standard `cursor.fetch*()` methods.

Please see section 5.6 *Stored Procedures* for more details on how to call stored procedures.

`.close()`

Close the cursor now (rather than automatically at garbage collection time).

The cursor will be unusable from this point forward; an `Error` (or subclass) exception will be raised if any operation is attempted with the cursor.

`.execute(sqlcmd, parameters=(), direct=-1, parametertypes=None)`

Prepare and execute a database operation `sqlcmd` (query or command).

`sqlcmd` may be a Unicode object in case the ODBC driver and/or database support this.

Depending on the current `.paramstyle` setting, `parameters` must be provided as sequence[15] or mapping:

`'qmark'(default)`

With the `'qmark'` parameter style (default) a sequence is expected and parameters be bound to variables found in the `sqlcmd` string on a positional basis. Variables in the `sqlcmd` string are specified using the ODBC variable question mark placeholder `'?'`, e.g. `'SELECT name,id FROM table WHERE amount > ? AND amount < ?'`, and get bound in the order they appear in the SQL statement `sqlcmd` from left to right.

`'named'`

With `'named'` parameter style a mapping is expected and parameters be bound to variables found in the `sqlcmd` string based on the values of the referenced named entries in the mapping. Variables in the `sqlcmd` string are specified using the Oracle style variable marks placeholder `':name'`, e.g. `'SELECT name,id FROM table WHERE amount > :minamount AND amount < :maxamount'`, and get bound to the values defined in the parameters mapping. It is possible to use multiple references to the same named parameter in `sqlcmd`.

A reference to the `sqlcmd` string will be retained by the cursor and made available to Python as `cursor.command`. If the same `sqlcmd` object is passed in again, the cursor will optimize its behavior by reusing the previously prepared statement. This is most effective for algorithms where the same `sqlcmd` is used, but different parameters are bound to it, e.g. in loops iterating over input data items.

Use `.executemany()` if you want to apply the `sqlcmd` to a sequence or iterator/generator of parameters in one call, e.g. to insert multiple rows in a single call. Please note that passing a sequence such as a list of lists as parameter to `.execute()` may result in strange error messages.

`direct` specifies whether to use direct, unprepared execution or not (see `.executedirect()` for details). It defaults to -1, meaning that direct execution is used if no parameters are given, non-direct otherwise.

If `parametertypes` is given, it defines the parameter types of the parameters used in the `sqlcmd`. The sequence has to provide one integer entry per parameter. Possible values are `SQL.PARAM_INPUT` (input parameter), `SQL.PARAM_OUTPUT` (output parameter) and `SQL.PARAM_INPUT_OUTPUT` (input/output parameter). If `parametertypes` is not given, default is to assume input parameter types for all parameters. Please see section 5.6 *Stored Procedures* for more details on how to use this parameter.

If `parametertypes` are given, the method returns a tuple copy of the parameters sequence, with output and input/output parameter values replaced by the updated values from the database.

[15] Note that in mxODBC 3.0 and earlier, the `.execute()` methods used to work like `.executemany()` when passing a list of tuples as parameter (a feature inherited from Python DB-API 1.0). Starting with mxODBC 3.1 this behavior was removed to avoid confusion.

Without `parametertypes`, the method returns `None`.

`.executedirect(sqlcmd, parameters=(), parametertypes=None)`

This method works just like `.execute()`, except that no prepare step is issued and the `sqlcmd` is not cached. This can result in much better performance with some ODBC driver setups and even better support for binding parameters[16], but also implies that Python type binding mode is used to bind the parameters. All SQL command parsing is then pushed from the client side to the server side.

`sqlcmd` may be a Unicode object in case the ODBC driver and/or database support this.

Use `.executemany(…, direct=1)` if you want to apply the `sqlcmd` to a sequence or iterator/generator of parameters in one call, e.g. to insert multiple rows in a single call. Please note that passing a sequence such as a list of lists as parameter to `.executedirect()` may result in strange error messages.

Return values are the same as for `cursor.execute()`.

`.executemany(sqlcmd, batch=(), direct=0, parametertypes=None)`

Prepare a database operation (query or command) and then execute it against all parameter sequences found in the sequence, iterator or generator `batch`.

The same comments as for `.execute()` also apply accordingly to this method.

If the optional integer `direct` is given and true, mxODBC will not cache the `sqlcmd`, but submit it for one-time execution to the database. This can result in better performance with some ODBC driver setups, but also implies that Python type binding mode is used to bind the parameters.

`sqlcmd` may be a Unicode object in case the ODBC driver and/or database support this.

If `parametertypes` are given, the method returns a list of tuple copies of the batch parameter sequence, with output and input/output parameter values replaced by the updated values from the database.[17]

Without `parametertypes`, the method returns `None`.

`.fetchall()`

Fetch all (remaining) rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples).

An `Error` (or subclass) exception is raised if the previous call to `.execute*()` did not produce any result set or no call was issued yet.

[16] Some ODBC drivers such as the MS SQL Server Native Client have limitations on where to place binding parameters which do not apply when using direct execution. See https://msdn.microsoft.com/en-us/library/ms711808%28VS.85%29.aspx and https://msdn.microsoft.com/en-us/library/ms709310%28VS.85%29.aspx for details.

[17] While this can be used to issue multiple stored procedure calls and retrieve data from the database via output parameters, not all databases support multiple batched calls. The database will then raise a function sequence error or similar as a result. PostgreSQL is one such database.

`.fetchmany([size=cursor.arraysize])`

Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples). An empty sequence is returned when no more rows are available.

The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's `.arraysize` determines the number of rows to be fetched. The method will try to fetch as many rows as indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows may be returned.

An `Error` (or subclass) exception is raised if the previous call to `.execute*()` did not produce any result set or no call was issued yet.

`.fetchone()`

Fetch the next row of a query result set, returning a single sequence, or `None` when no more data is available.

An `Error` (or subclass) exception is raised if the previous call to `.execute*()` did not produce any result set or no call was issued yet.

mxODBC will move the associated database cursor forward by one row only.

`.flush()`

Frees any pending result set used by the cursor. If you only fetch some of the rows of large result sets you can optimize memory usage by calling this method.

Note that `.execute*()` and all the catalog methods do an implicit `.flush()` prior to executing a new query.

`.getcolattribute(position, info_id)`

Get information about the result set column `position`. The column index must be given as 0-based integer, i.e. the first result setup column has the index 0.

`info_id` must be an integer and identifies the requested field information. Suitable values are available through the `SQL` object (see the *Constants* section 10.5 for details).

The method returns a tuple `(integer, string)` giving an integer decoding (in native integer byte order) of the first bytes of the API's result as well as the raw buffer data as string. It is up to the caller to decode the data (e.g. using the `struct` module).

This API gives you a wide range of information about the result set column. See the *ODBC SQLColAttribute API Documentation* for more information.

Some of these values are also available through the `cursor.description` attribute.

This is a list of useful info ids:

Option	*Comment*

Option	*Comment*
SQL.DESC_AUTO_UNIQUE_VALUE	Check whether the result set column refers to an auto-increment column of the table. The check only returns valid values for numeric columns that can be defined as auto-increment column in the database. Returns an integer value: SQL.TRUE - column is auto-increment SQL.FALSE - column is not an auto-increment column or not numeric
SQL.DESC_BASE_COLUMN_NAME	Base column name of the result set column. If the base column name cannot be determined, e.g. for expressions, an empty string is returned. Returns a string..
SQL.DESC_BASE_TABLE_NAME	Base table name of the result set column. If the base table name cannot be determined, e.g. for expressions, an empty string is returned. Returns a string..
SQL.DESC_DISPLAY_SIZE	Returns the maximum number of characters needed to display the column data. Returns an integer.
SQL.DESC_LENGTH	Returns the maximum length of the column data in characters. Returns an integer.
SQL.DESC_OCTET_LENGTH	Returns the maximum length of the column data in bytes. Returns an integer.
SQL.DESC_PRECISION	Returns the precision of a numeric column. For date/time columns, this returns the precision of the seconds fraction, if applicable. Returns an integer.
SQL.DESC_SCALE	Returns the scale of a numeric column. Returns an integer.

Option	*Comment*
SQL.DESC_TABLE_NAME	Table name of the table containing the result set column. If the table name cannot be determined, e.g. for expressions, an empty string is returned. Returns a string..
SQL.DESC_TYPE_NAME	Data source dependent type name of the result set column or an empty string if the value cannot be determined. Returns a string..
SQL.DESC_UNSIGNED	Checks whether the result set column is unsigned numeric data or not. Returns an integer value: SQL.TRUE - column data is unsigned or not numeric SQL.FALSE - column data is signed

If the ODBC driver doesn't support an `info_id` or cannot determine the requested value, it either raises an exception, or returns an empty string where applicable.

`.getcursorname()`

Returns the current cursor name associated with the cursor object. This may either be the name given to the cursor at creation time or a name generated by the ODBC driver for it to use.

`.getcursoroption(option)`

Returns the given cursor option. This method interfaces directly to the ODBC function `SQLGetCursorOption()`.

`option` must be an integer. Suitable option values are available through the `SQL` object.

Possible values are:

Option	*Comment*
SQL.ATTR_QUERY_TIMEOUT	Returns the query timeout in seconds used for the cursor. Note that not all ODBC drivers support this option.
SQL.ATTR_ASYNC_ENABLE	Check whether asynchronous execution of commands is enabled.

Option	*Comment*

	Possible values:
	`SQL.ASYNC_ENABLE_OFF` (default)
	`SQL.ASYNC_ENABLE_ON`
	`SQL.ASYNC_ENABLE_DEFAULT`

`SQL.ATTR_MAX_LENGTH`	Returns the length limit for fetching column data. Possible values: Any positive integer or `SQL.MAX_LENGTH_DEFAULT` (no limit)
`SQL.ATTR_MAX_ROWS`	Returns the maximum number of rows a .fetchall() command would return from the result set. Possible values: Any positive integer or `SQL.MAX_ROWS_DEFAULT` (no limit)

`SQL.ATTR_NOSCAN`	Check whether the ODBC driver will scan the SQL commands for ODBC escape sequences or not.
	Possible values:
	`SQL.NOSCAN_OFF` (default)
	`SQL.NOSCAN_ON`
	`SQL.NOSCAN_DEFAULT`

`SQL.ROW_NUMBER`	Returns the row number of the current row in the result set or 0 if it cannot be determined.

The method returns the data as 32-bit integer. It is up to the caller to decode the integer using the `SQL` defines.

`.next()`

Works like `.fetchone()` to make cursors compatible to the iterator interface (new in Python 2.2). Raises a `StopIteration` at the end of a result set.

`.nextset()`

This method will make the cursor skip to the next available set, discarding any remaining rows from the current set.

If there are no more sets, the method returns `None`. Otherwise, it returns a true value and subsequent calls to the fetch methods will return rows from the next result set.

An `Error` (or subclass) exception is raised if the previous call to `.execute*()` did not produce any result set or no call was issued yet.

`.prepare(sqlcmd)`

Prepare a database operation (query or command) statement for later execution and set `cursor.command`. To later execute a prepared statement, pass `cursor.command` to one of the `.execute*()` methods.

`cursor.prepare(sqlcmd)` can also be used to check `sqlcmd` for syntax errors, or to inspect the result set structure of a query without executing the `sqlcmd` operation, by looking at `cursor.description` after calling `cursor.prepare()`.

`sqlcmd` may be a Unicode object in case the ODBC driver and/or database support this.

Return values are not defined.

This method is may not be available in all mxODBC subpackages. Even if it is available, the used ODBC driver or database may not support preparing database operations for later reuse.

`.scroll(value, mode='relative')`

Scroll the cursor in the result set according to `mode`.

If mode is `'relative'` (default), value is taken as offset to the current position in the result set, if set to `'absolute'`, value gives the absolute position.

An `IndexError` is raised in case the scroll operation leaves the result set. In this case, the cursor position is not changed.

This method will use native scrollable cursors, if the data source provides these, or revert to an emulation for forward-only scrollable cursors. Please check whether the data source supports this method using the included mx/ODBC/Misc/test.pyc script.

Warning:
Some ODBC drivers have trouble scrolling in result sets which use BLOBs or other data types for which the data size cannot be determined at prepare time. mxODBC currently raises a `NotSupportedError` in case a request for backward scrolling is made in such a result set. Hopefully, this will change as ODBC drivers become more mature.

`.setconverter(converter)`

This method sets the converter function to use for subsequent fetches. Passing `None` as converter will reset the converter mechanism to its default setting. See the *Supported Data Types* section 8 for details on how user-defined converters work.

The current converter function used on the cursor can be queried through the read-only `cursor.converter` attribute.

`.setcursorname(name)`

Sets the name to be associated with the cursor object.

There is a length limit for names in SQL at 18 characters. An `InternalError` will be raised if the name is too long or otherwise not useable.

`.setcursoroption(option, value)`

Sets a cursor option to a new value.

Only a subset of the possible option values defined by ODBC are available since this method could otherwise easily cause mxODBC to segfault – it makes changes possible which effect the way mxODBC interfaces to the ODBC driver.

Only options with numeric values are currently supported.

Option	*Comment*
`SQL.ATTR_QUERY_TIMEOUT`	Sets the query timeout in seconds used for the cursor. Queries that take longer raise an exception after the timeout is reached. Possible values: Any positive integer or `SQL.QUERY_TIMEOUT_DEFAULT` Note that not all ODBC drivers support this option.

`SQL.ATTR_ASYNC_ENABLE`	Enable asynchronous execution of commands.
	
	Possible values:
	
	`SQL.ASYNC_ENABLE_OFF` (default)
	
	`SQL.ASYNC_ENABLE_ON`
	
	`SQL.ASYNC_ENABLE_DEFAULT`

`SQL.ATTR_MAX_LENGTH`	Maximum length of any fetched column. Default is no limit. Possible values: Any positive integer or `SQL.MAX_LENGTH_DEFAULT` (no limit)
`SQL.ATTR_MAX_ROWS`	Limit the maximum number of rows to fetch in a result set. Default is no limit.

Option	*Comment*
	Possible values: Any positive integer or `SQL.MAX_ROWS_DEFAULT` (no limit)
`SQL.ATTR_METADATA_ID`	Tell the ODBC driver to interpret the catalog method parameters as case-insensitive identifiers. Default is to interpret them as case-sensitive SQL search patterns. Possible values: `SQL.TRUE` - case-insensitive identifers `SQL.FALSE` - case-sensitive search patterns (default)

`SQL.ATTR_NOSCAN`	Tell the ODBC driver not to scan the SQL commands and unescape (expand) any ODBC escape sequences it finds. Default is to scan for them.
	
	Possible values:
	
	`SQL.NOSCAN_OFF` (default)
	
	`SQL.NOSCAN_ON`
	
	`SQL.NOSCAN_DEFAULT`

`.setinputsizes(sizes)`

This methods does nothing in mxODBC, it is just needed for DB API compliance.

`.setoutputsize(size[, column])`

This methods does nothing in mxODBC, it is just needed for DB API compliance.

`.__iter__()`

Returns the cursor itself. This method makes cursor objects usable as iterators (new in Python 2.2).

`.__enter__()`

Returns the cursor itself. This method makes cursor objects usable as context manager (together with the .__exit__() method) and is called when entering a *with*-block (new in Python 2.5).

`.__exit__(exc_type, exc_value, exc_tb)`

Returns True in case `exc_type` is set to None (no exception set) and closes the cursor. Returns False in case `exc_type` is set to an exception and also closes

the cursor. This method is part of the context manager API and is called when leaving a *with*-block (new in Python 2.5).

7.6.1 Catalog Methods

Catalog methods allow you to access meta-level and structural information about a data source in a portable way.

> Some ODBC drivers do not support all of these methods or return unusable data. As a result, you should verify correct operation for your target data sources prior to relying on these methods.

Common Interface

All of the following catalog methods use the same interface: they do an implicit call to `cursor.execute()` and return their output in form of a list of rows which that can be fetched with the `cursor.fetch*()` methods in the usual way. The number of available rows is available via `cursor.rowcount`[18]. All catalog methods support keywords and use the indicated default values for parameters which are omitted in the call.

Please refer to the *ODBC Documentation* for more detailed information about parameters (if you pass `None` as a value where a string would be expected, that entry is converted to NULL before passing it to the underlying ODBC API).

Result Set Layouts

Note that the result set layouts described here may not apply to your data source. Some databases do not provide all the information given here and thus generate slightly different result sets. Expect column additions and even omissions and do not rely on the column names used in the result set descriptions.

Search Pattern Parameters

The standard catalog method parameters `qualifier`, `owner` and `table` accept SQL search patterns as input, e.g. `table='SYS%'` would return all tables whose name starts with *SYS*.

In some cases, the catalog functions provide additional search parameters such as procedure or column. These parameters then also accept SQL search pattern strings.

[18] Note that this was changed in mxODBC 3.0. Previously the catalog methods used to return the number of rows in the result set.

Case-sensitivity of Search Patterns

The search patterns given as parameters to these catalog methods are usually interpreted in a **case-sensitive** *way*. This means that even if the database itself behaves case-insensitive for identifiers, you may still not find what you're looking for if you don't use the case which the database internally uses to store the identifier.

As an example take the SAP DB: it stores all unquoted identifiers using uppercase letters. Trying to fetch e.g. information about a table using a lowercase version of the name will result in an empty result set. You can use `connection.getinfo(SQL.IDENTIFIER_CASE)` to determine how the database stores identifiers. See the *ODBC Documentation* for details.

Switching between Search Patterns and Identifier Matching

Some ODBC drivers support adjusting the catalog method interface to interpret the parameters as case-insensitive identifiers instead.

In mxODBC, this can be enabled using:

```
cursor.setcursoroption(SQL.ATTR_METADATA_ID, SQL.TRUE)
```

The setting persists on the cursor. It can be switched off again using:

```
cursor.setcursoroption(SQL.ATTR_METADATA_ID, SQL.FALSE)
```

which then causes the catalog methods to interpret the parameters as case-sensitive search patterns again.

Whether this really helps with the problem described above depends on the application.

Unicode

All catalog methods accept Unicode parameters, if the ODBC drivers provide the necessary support for this.

Available Catalog Methods

> Please note that the drivers may not implement all catalog methods that mxODBC supports. In such a case, you will get a `NotSupportedError` or `AttributeError` exception when trying to use a method that is not supported by the ODBC driver.

The following catalog methods are supported by mxODBC:

```
.columns(qualifier=None, owner=None, table=None, column=None)
```

Query the database schema for information on table columns.

`column` allows restricting the results to a single column of a table.

Depending on the used query options, the result set will contain information for only one table, the whole database or just a single column.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEMA	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128) not NULL	The name of the table, or view, or alias, or synonym.
COLUMN_NAME	VARCHAR(128) not NULL	Name of the column of the specified table, view, alias, or synonym.
DATA_TYPE	SMALLINT not NULL	SQL data type of column identified by COLUMN_NAME.
TYPE_NAME	VARCHAR(128) not NULL	Character string representing the name of the data type corresponding to DATA_TYPE.
COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in characters for the column. For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.
BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this column if SQL_C_DEFAULT were specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` calls. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign.
DECIMAL_DIGITS	SMALLINT	The scale of the column. NULL is returned for data types where scale is not applicable.
NUM_PREC_RADIX	SMALLINT	Either 10 or 2 or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, then the

Column Name	*Column Datatype*	*Comment*
		COLUMN_SIZE column contains the number of bits allowed in the column. If DATA_TYPE is an exact numeric data type, this column contains the value 10 and the COLUMN_SIZE contains the number of decimal digits allowed for the column. For numeric data types, the database can return a NUM_PREC_RADIX of either 10 or 2.
NULLABLE	SMALLINT not NULL	SQL.NO_NULLS if the column does not accept NULL values.
REMARKS	VARCHAR(254)	May contain descriptive information about the column or NULL. It is possible that no usable information is returned in this column (due to optimizations).
COLUMN_DEF	VARCHAR(254)	The column's default value. If the default value is a numeric literal, then this column contains the character representation of the numeric literal with no enclosing single quotes. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value a pseudo-literal, such as for DATE, TIME, and TIMESTAMP columns, then this column contains the keyword of the pseudo-literal (e.g. CURRENT DATE) with no enclosing quotes. If NULL was specified as the default value, then this column returns "NULL". If the default value cannot be represented without truncation, then this column contains "TRUNCATED" with no enclosing single quotes. If no default value was specified, then this column is NULL. It is possible that no usable information is returned in this column (due to optimizations).
SQL_DATA_TYPE	SMALLINT not NULL	SQL data type. This column is the same as the DATA_TYPE column.
SQL_DATETIME_SUB	SMALLINT	The subtype code for datetime data types: SQL.CODE_DATE, SQL.CODE_TIME, SQL.CODE_TIMESTAMP. For all other data types this column returns NULL.

Column Name	*Column Datatype*	*Comment*
CHAR_OCTET_LENGTH	INTEGER	Contains the maximum length in octets for a character data type column. For Single Byte character sets, this is the same as COLUMN_SIZE. For all other data types it is NULL.
ORDINAL_POSITION	INTEGER not NULL	The ordinal position of the column in the table. The first column in the table is number 1.
IS_NULLABLE	VARCHAR(254)	Contains the string "NO" if the column is known to be not nullable; and "YES" otherwise.

```
.columnprivileges(qualifier=None, owner=None, table=None,
   column=None)
```

Query the database schema for information on column privileges for the given table. This is useful to determine the authorizations granted to a table or column.

`column` allows restricting the results to a single column of a table.

Note that the `table` parameter is mandatory.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEMA	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128) not NULL	The name of the table, or view, or alias, or synonym.
COLUMN_NAME	VARCHAR(128) not NULL	Name of the column of the specified table, view, alias, or synonym.
GRANTOR	VARCHAR(128)	Authorization ID of the user who granted the privilege.
GRANTEE	VARCHAR(128)	Authorization ID of the user to whom the privilege is granted.
PRIVILEGE	VARCHAR(128)	The table privilege. This may be one of the following strings: "INSERT", "REFERENCES",

Column Name	*Column Datatype*	*Comment*
		"SELECT", "UPDATE".
IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users. This can be "YES", "NO" or NULL.

```
.foreignkeys(primary_qualifier=None, primary_owner=None,
   pimary_table=None, foreign_qualifier=None, foreign_owner=None,
   foreign_table=None)
```

Query the database schema for information on foreign keys. The method has two modes of operation, depending on which parameter is set:

`primary_table`

The method returns a list of foreign key columns in other tables that refer to the primary key column of the given table and the primary key column of the given table itself.

`foreign_table`

The method returns a list of foreign key columns in a table that refer to the primary keys of other tables and the primary key columns of those other tables.

This is useful to determine the relationships between the tables in a database schema.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
PKTABLE_CAT	VARCHAR(128)	Always NULL.
PKTABLE_SCHEMA	VARCHAR(128)	The name of the schema containing PKTABLE_NAME.
PKTABLE_NAME	VARCHAR(128) not NULL	Name of the table containing the primary key.
PKCOLUMN_NAME	VARCHAR(128) not NULL	Primary key column name.
FKTABLE_CAT	VARCHAR(128)	Always NULL.
FKTABLE_SCHEMA	VARCHAR(128)	The name of the schema containing FKTABLE_NAME.

Column Name	*Column Datatype*	*Comment*
FKTABLE_NAME	VARCHAR(128) not NULL	Name of the table containing the foreign key.
FKCOLUMN_NAME	VARCHAR(128) not NULL	Foreign key column name.
ORDINAL_POSITION	SMALLINT not NULL	The ordinal position of the column in the key, starting at 1.
UPDATE_RULE	SMALLINT	Action to be applied to the foreign key when the SQL operation is UPDATE: SQL.RESTRICT, SQL.NO_ACTION, SQL.CASCADE, SQL.SET_NULL.
DELETE_RULE	SMALLINT	Action to be applied to the foreign key when the SQL operation is DELETE: SQL.CASCADE, SQL.NO_ACTION, SQL.RESTRICT, SQL.SET_DEFAULT, SQL.SET_NULL
FK_NAME	VARCHAR(128)	Foreign key identifier. NULL if not applicable to the data source.
PK_NAME	VARCHAR(128)	Primary key identifier. NULL if not applicable to the data source.
DEFERRABILITY	SMALLINT	Possible values: SQL.INITIALLY_DEFERRED, SQL.INITIALLY_IMMEDIATE, SQL.NOT_DEFERRABLE.

`.gettypeinfo(sqltype)`

Query the data source for information on a supported data type `sqltype`.

`sqltype` must be one of the SQL type codes as returned in `cursor.description[1]`. See section 8. Data Types supported by mxODBC for a list of SQL type codes and details about their use.

This method is useful to determine characteristics of the given SQL data type and how it is defined in the SQL dialect supported by the data source.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TYPE_NAME	VARCHAR(128) not NULL	Character representation of the SQL data type name, e.g. "VARCHAR", "DATE", "INTEGER".

Column Name	*Column Datatype*	*Comment*
DATA_TYPE	SMALLINT not NULL	SQL data type of column identified by COLUMN_NAME.
COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in characters for the column. For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.
LITERAL_PREFIX	VARCHAR(128)	Prefix for a literal of this data type. This column is NULL for data types where a literal prefix is not applicable.
LITERAL_SUFFIX	VARCHAR(128)	Suffix for a literal of this data type. This column is NULL for data types where a literal prefix is not applicable.
CREATE_PARAMS	VARCHAR(128)	The text of this column contains a list of keywords, separated by commas, corresponding to each parameter the application may specify in parenthesis when using the name in the TYPE_NAME column as a data type in SQL. The keywords in the list can be any of the following: "LENGTH", "PRECISION", "SCALE". They appear in the order that the SQL syntax requires that they be used. NULL is returned if there are no parameters for the data type definition, (such as INTEGER). Note: The intent of CREATE_PARAMS is to enable an application to customize the interface for a DDL builder.
NULLABLE	SMALLINT not NULL	Indicates whether the data type accepts a NULL value SQL.NO_NULLS - NULL values are disallowed. SQL.NULLABLE - NULL values are allowed.

Column Name	*Column Datatype*	*Comment*
CASE_SENSITIVE	SMALLINT not NULL	Indicates whether the data type can be treated as case sensitive for collation purposes; valid values are SQL.TRUE and SQL.FALSE.
SEARCHABLE	SMALLINT not NULL	Indicates how the data type is used in a WHERE clause. Valid values are: SQL.UNSEARCHABLE: if the data type cannot be used in a WHERE clause. SQL.LIKE_ONLY: if the data type can be used in a WHERE clause only with the LIKE predicate. SQL.ALL_EXCEPT_LIKE: if the data type can be used in a WHERE clause with all comparison operators except LIKE. SQL.SEARCHABLE: if the data type can be used in a WHERE clause with any comparison operator.
UNSIGNED_ATTRIBUTE	SMALLINT	Indicates where the data type is unsigned. The valid values are: SQL.TRUE, SQL.FALSE or NULL. A NULL indicator is returned if this attribute is not applicable to the data type.
FIXED_PREC_SCALE	SMALLINT not NULL	Contains the value SQL.TRUE if the data type is exact numeric and always has the same precision and scale; otherwise, it contains SQL.FALSE.
AUTO_INCREMENT	SMALLINT	Contains SQL.TRUE if a column of this data type is automatically set to a unique value when a row is inserted; otherwise, contains SQL.FALSE.
LOCAL_TYPE_NAME	VARCHAR(128)	This column contains any localized (native language) name for the data type that is different from the regular name of the data type. If there is no localized name, this column is NULL. This column is intended for display only. The character set of the string is locale-dependent and is typically the default character set of the database.
MINIMUM_SCALE	INTEGER	The minimum scale of the SQL data type. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain the same value. NULL is returned where scale is not

Column Name	*Column Datatype*	*Comment*
		applicable.
MAXIMUM_SCALE	INTEGER	The maximum scale of the SQL data type. NULL is returned where scale is not applicable. If the maximum scale is not defined separately in the database, but is defined instead to be the same as the maximum length of the column, then this column contains the same value as the COLUMN_SIZE column.
SQL_DATA_TYPE	SMALLINT not NULL	SQL data type. This column is the same as the DATA_TYPE column.
SQL_DATETIME_SUB	SMALLINT	The subtype code for datetime data types: SQL.CODE_DATE, SQL.CODE_TIME, SQL.CODE_TIMESTAMP. For all other data types this column returns NULL.
NUM_PREC_RADIX	SMALLINT	Either 10 or 2 or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, then the COLUMN_SIZE column contains the number of bits allowed in the column. If DATA_TYPE is an exact numeric data type, this column contains the value 10 and the COLUMN_SIZE contains the number of decimal digits allowed for the column. For numeric data types, the database can return a NUM_PREC_RADIX of either 10 or 2.
INTERVAL_PRECISION	SMALLINT	Datetime interval precision or NULL is interval types are not supported by the database.

`.primarykeys(qualifier=None, owner=None, table=None)`

Query the data source for information on the primary keys of a given table. The `table` parameter is mandatory.

The method is useful when inspecting unknown database schemas. It only supports returning the primary key column(s) for a single table.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEMA	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128) not NULL	The name of the table, or view, or alias, or synonym.
COLUMN_NAME	VARCHAR(128) not NULL	Primary Key column name.
ORDINAL_POSITION	SMALLINT not NULL	Column sequence number in the primary key, starting with 1.
PK_NAME	VARCHAR(128)	Primary key identifier. NULL if not applicable to the data source.

`.procedures(qualifier=None, owner=None, procedure=None)`

Query the data source for information on procedures stored in a data source.

`procedure` can be used to limit the results to a set of procedures or a single procedure.

The method is useful for determining the availability of stored procedures and also for database schema introspection purposes. It can be used to check whether a stored requires output parameters or fetching result sets.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
PROCEDURE_CAT	VARCHAR(128)	Always NULL.
PROCEDURE_SCHEMA	VARCHAR(128)	The name of the schema containing PROCEDURE_NAME.
PROCEDURE_NAME	VARCHAR(128) not NULL	The name of the procedure.
NUM_INPUT_PARAMS	INTEGER not NULL	Number of input parameters.
NUM_OUTPUT_PARAMS	INTEGER not NULL	Number of output parameters.
NUM_RESULT_SETSNUM	INTEGER not NULL	Number of result sets returned by the

Column Name	*Column Datatype*	*Comment*
_RESULT_SETS		procedure.
REMARKS	VARCHAR(254)	Contains the descriptive information about the procedure.
PROCEDURE_TYPE	SMALLINT	Defines the procedure type: SQL.PT_UNKNOWN: It cannot be determined whether the procedure returns a value. SQL.PT_PROCEDURE: The returned object is a procedure; that is, it does not have a return value. SQL.PT_FUNCTION: The returned object is a function; that is, it has a return value.

`.procedurecolumns(qualifier=None, owner=None, procedure=None, column=None)`

Query the data source for information on parameter details of procedures stored in a data source.

`procedure` can be used to limit the results to a set of procedures or a single procedure. `column` allows restricting the results to a single procedure parameter.

The method can be used to e.g. determine whether a parameter is an input, output or input/output parameter. The `COLUMN_TYPE` column information can directly be passed to the `parametertypes` parameter in `cursor.callproc()` and the `cursor.execute*()` methods.

The catalog method generates a result set having the following schema (the term "*column*" used here refers to the procedure's call parameters):

Column Name	*Column Datatype*	*Comment*
PROCEDURE_CAT	VARCHAR(128)	Always NULL.
PROCEDURE_ SCHEMA	VARCHAR(128)	The name of the schema containing PROCEDURE_NAME.
PROCEDURE_ NAME	VARCHAR(128)	The name of the table, or view, or alias, or synonym.
COLUMN_NAME	VARCHAR(128)	Name of the column of the specified table, view, alias, or synonym.

Column Name	*Column Datatype*	*Comment*
COLUMN_TYPE	SMALLINT not NULL	Identifies the type information associated with this column. Possible values: SQL.PARAM_TYPE_UNKNOWN: the parameter type is unknown. SQL.PARAM_INPUT: this parameter is an input parameter. SQL.PARAM_INPUT_OUTPUT: this parameter is an input / output parameter. SQL.PARAM_OUTPUT: this parameter is an output parameter. SQL.RETURN_VALUE: the procedure column is the return value of the procedure. SQL.RESULT_COL: this parameter is actually a column in the result set.
DATA_TYPE	SMALLINT not NULL	SQL data type of column.
TYPE_NAME	VARCHAR(128) not NULL	Character string representing the name of the data type corresponding to DATA_TYPE.
COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in characters for the column. For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.
BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this column if SQL.C_DEFAULT were specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` ODBC calls used internally by mxODBC. This length does not include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign. Note: This column is of little value to Python applications.
DECIMAL_DIGITS	SMALLINT	The scale of the column. NULL is returned for data types where scale is not applicable.

Column Name	*Column Datatype*	*Comment*
NUM_PREC_ RADIX	SMALLINT	Either 10 or 2 or NULL. If DATA_TYPE is an approximate numeric data type, this column contains the value 2, then the COLUMN_SIZE column contains the number of bits allowed in the column. If DATA_TYPE is an exact numeric data type, this column contains the value 10 and the COLUMN_SIZE contains the number of decimal digits allowed for the column. For numeric data types, the database can return a NUM_PREC_RADIX of either 10 or 2.
NULLABLE	SMALLINT not NULL	SQL.NO_NULLS if the column does not accept NULL values.
REMARKS	VARCHAR(254)	May contain descriptive information about the column or NULL. It is possible that no usable information is returned in this column (due to optimizations).
COLUMN_DEF	VARCHAR(3)	The column's default value. If the default value is a numeric literal, then this column contains the character representation of the numeric literal with no enclosing single quotes. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value a pseudo-literal, such as for DATE, TIME, and TIMESTAMP columns, then this column contains the keyword of the pseudo-literal (e.g. CURRENT DATE) with no enclosing quotes. If NULL was specified as the default value, then this column returns "NULL". If the default value cannot be represented without truncation, then this column contains "TRUNCATED" with no enclosing single quotes. If no default value was specified, then this column is NULL. It is possible that no usable information is returned in this column (due to optimizations).
SQL_DATA_TYPE	SMALLINT not NULL	ODBC3 SQL data type. This column is the same as the DATA_TYPE column, except for date/time types.
SQL_DATETIME_ SUB	SMALLINT	The subtype code for datetime data types: SQL.CODE_DATE, SQL.CODE_TIME, SQL.CODE_TIMESTAMP. For all other data types this column returns NULL.

Column Name	*Column Datatype*	*Comment*
CHAR_OCTET_ LENGTH	INTEGER	Contains the maximum length in octets for a character data type column. For Single Byte character sets, this is the same as COLUMN_SIZE. For all other data types it is NULL.
ORDINAL_ POSITION	INTEGER not NULL	The ordinal position of the parameter column in the procedure call. The first column has an ordinal position of 1.
IS_NULLABLE	VARCHAR(254)	Contains the string "NO" if the column is known to be not nullable, "" if this cannot be determined, or "YES" if it is known to be nullable.

```
.specialcolumns(qualifier=None, owner=None, table=None,
   coltype=SQL.BEST_ROWID, scope=SQL.SCOPE_SESSION,
   nullable=SQL.NO_NULLS)
```

Query the data source for information on "special" columns of a given table. The `table` parameter is mandatory.

Special columns in this sense are columns which can be used to uniquely identify a row in the table (e.g. primary keys) or which are automatically updated by the database (e.g. auto-increment columns).

Possible input values for `coltype`:

`SQL_BEST_ROWID`

Return the optimal column or set of columns for uniquely identifying a row in the table (the *rowid*).

`SQL_ROWVER`

Return columns that are automatically updated by the database when the row is updated.

Possible input values for `scope`:

`SQL.SCOPE_CURROW`

The rowid column(s) are only guaranteed to be valid as long as the rows remain unchanged.

`SQL.SCOPE_TRANSACTION`

The rowid is guaranteed to be valid for the duration of the current transaction.

`SQL.SCOPE_SESSION`

The rowid is guaranteed to be valid for the duration of the connection.

Possible input values for `nullable`:

`SQL.NO_NULLS`

Exclude special columns that nullable. Using this option can result in an empty result set, if the table, driver or database don't support such requirements.

`SQL.NULLABLE`

Return special columns, even if they can have NULL values.

The method is useful to determine columns that can be used as to determine query columns that allow retrieving rows which have been inserted in a table without primary key or in a table with a primary key which is defined as auto-increment column.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
SCOPE	SMALLINT	The duration for which the name in COLUMN_NAME is guaranteed to point to the same row. Contains one of the following values: SQL.SCOPE_CURROW, SQL.SCOPE_TRANSACTION, SQL.SCOPE_SESSION.
COLUMN_NAME	VARCHAR(128) not NULL	Name of the column that is (or part of) the table's primary key.
DATA_TYPE	SMALLINT not NULL	SQL data type of column identified by COLUMN_NAME.
TYPE_NAME	VARCHAR(128) not NULL	Character string representing the name of the data type corresponding to DATA_TYPE.
COLUMN_SIZE	INTEGER	If the DATA_TYPE column value denotes a character or binary string, then this column contains the maximum length in characters for the column. For date, time, timestamp data types, this is the total number of characters required to display the value when converted to character. For numeric data types, this is either the total number of digits, or the total number of bits allowed in the column, depending on the value in the NUM_PREC_RADIX column in the result set.
BUFFER_LENGTH	INTEGER	The maximum number of bytes for the associated C buffer to store data from this column if SQL.C_DEFAULT were specified on the `SQLBindCol()`, `SQLGetData()` and `SQLBindParameter()` ODBC calls used internally by mxODBC. This length does not

Column Name	*Column Datatype*	*Comment*
		include any null-terminator. For exact numeric data types, the length accounts for the decimal and the sign. Note: This column is of little value to Python applications.
DECIMAL_DIGITS	SMALLINT	The scale of the column. NULL is returned for data types where scale is not applicable.
PSEUDO_COLUMN	SMALLINT	Indicates whether or not the column is a pseudo-column. Possible values: SQL.PC_NOT_PSEUDO, SQL.PC_UNKNOWN, SQL.PC_PSEUDO.

```
.statistics(qualifier=None, owner=None, table=None,
   unique=SQL.INDEX_ALL, accuracy=SQL.QUICK)
```

Query the data source for information on statistics and available indexes for a given table. The `table` parameter is mandatory.

Possible input values for `unique`:

`SQL.INDEX_UNIQUE`

Return only unique indexes.

`SQL.INDEX_ALL`

Return all indexes.

Possible input values for `accuracy`:

`SQL.ENSURE`

The data returned for CARDINALITY and PAGES must be current and accurate. This mode is not widely supported and its use is discouraged.

`SQL.QUICK`

The data for CARDINALITY and PAGES is returned if available, but must not be current.

This method is mainly useful for identifying the indexes of a table in a database schema.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.

Column Name	*Column Datatype*	*Comment*
TABLE_SCHEMA	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128) not NULL	The name of the table, or view, or alias, or synonym.
NON_UNIQUE	SMALLINT	Indicates whether the index prohibits duplicate values. Returns: SQL.TRUE if the index allows duplicate values. SQL.FALSE if the index values must be unique. NULL is returned if the TYPE column indicates that this row is SQL.TABLE_STAT (statistics information on the table itself).
INDEX_QUALIFIER	VARCHAR(128)	The string that would be used to qualify the index name in the DROP INDEX statement. Appending a period (.) plus the INDEX_NAME results in a full specification of the index.
INDEX_NAME	VARCHAR(128)	The name of the index. If the TYPE column has the value SQL.TABLE_STAT, this column has the value NULL.
TYPE	SMALLINT not NULL	Indicates the type of information contained in this row of the result set: SQL.TABLE_STAT - Indicates this row contains statistics information on the table itself. SQL.INDEX_CLUSTERED - Indicates this row contains information on an index, and the index type is a clustered index. SQL.INDEX_HASHED - Indicates this row contains information on an index, and the index type is a hashed index. SQL.INDEX_OTHER - Indicates this row contains information on an index, and the index type is other than clustered or hashed.
ORDINAL_POSITION	SMALLINT	Ordinal position of the column within the index whose name is given in the INDEX_NAME column. A NULL value is returned for this column if the TYPE column has the value of SQL.TABLE_STAT.
COLUMN_NAME	VARCHAR(128)	Name of the column in the index. A NULL value is returned for this column if the TYPE column has the value of SQL.TABLE_STAT.

Column Name	*Column Datatype*	*Comment*
ASC_OR_DESC	CHAR(1)	Sort sequence for the column; "A" for ascending, "D" for descending. NULL value is returned if the value in the TYPE column is SQL.TABLE_STAT.
CARDINALITY	INTEGER	If the TYPE column contains the value SQL.TABLE_STAT, this column contains the number of rows in the table. If the TYPE column value is not SQL.TABLE_STAT, this column contains the number of unique values in the index. A NULL value is returned if the information cannot be determined.
PAGES	INTEGER	If the TYPE column contains the value SQL.TABLE_STAT, this column contains the number of pages used to store the table. If the TYPE column value is not SQL.TABLE_STAT, this column contains the number of pages used to store the indexes. A NULL value is returned if the information cannot be determined.
FILTER_CONDITION	VARCHAR(128)	If the index is a filtered index, this is the filter condition. NULL is returned if TYPE is SQL.TABLE_STAT or the database does not support filtered indexes.

`.tables(qualifier=None, owner=None, table=None, type=None)`

Query the data source for information on tables stored in the database.

`type` may be set to a comma-separated string of database table types using all uppercase characters. The exact list of available types is data source dependent. Common types include: `TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS, SYNONYM`.

This method is useful for checking whether a table exists and is accessible by the current user or not. It also aids in database schema introspection.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	The name of the catalog containing TABLE_SCHEM. This column contains a NULL value.

Column Name	*Column Datatype*	*Comment*
TABLE_SCHEMA	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128)	The name of the table, or view, or alias, or synonym.
TABLE_TYPE	VARCHAR(128)	Identifies the type given by the name in the TABLE_NAME column. It can have the string values "TABLE", "VIEW", "INOPERATIVE VIEW", "SYSTEM TABLE", "ALIAS", or "SYNONYM".
REMARKS	VARCHAR(254)	Contains the descriptive information about the table.

```
.tableprivileges(qualifier=None, owner=None, table=None)
```

Query the data source for information on table privileges associated with database tables.

The method is useful for determining and extracting table access permissions from the database.

The catalog method generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEMA	VARCHAR(128)	The name of the schema containing TABLE_NAME.
TABLE_NAME	VARCHAR(128) not NULL	The name of the table, or view, or alias, or synonym.
GRANTOR	VARCHAR(128)	Authorization ID of the user who granted the privilege.
GRANTEE	VARCHAR(128)	Authorization ID of the user to whom the privilege is granted.
PRIVILEGE	VARCHAR(128)	The table privilege. This may be one of the following strings: "ALTER", "CONTROL", "INDEX", "DELETE", "INSERT", "REFERENCES", "SELECT", "UPDATE".
IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users. This can be "YES", "NO" or NULL.

7.7 Cursor Object Attributes

`.arraysize`

This read/write attribute specifies the number of rows to fetch at a time with `.fetchmany()`. It defaults to 1 meaning to fetch a single row at a time.

mxODBC uses this value as default for the number of rows to fetch with `.fetchmany()` method.

`.bindmethod`

Attribute to query and set the input variable binding method used by the cursor. This can either be `BIND_USING_PYTHONTYPE` of `BIND_USING_SQLTYPE` (see the *Constants* section 10.5 for details).

The attribute is inherited by cursors from their connections at creation time. Cursors may override the setting on a per cursor basis without affecting the connection that was used to create them.

`.closed`

This read-only attribute is true if the cursor or the underlying connection was closed by calling the `.close()` method.

Any action on a closed connection or cursor will result in a `ProgrammingError` to be raised. This variable can be used to conveniently test for this state.

`.colcount`

This read-only attribute specifies the number of columns in the current result set.

The attribute is `-1` in case no `.execute*()` has been performed on the cursor.

Please note that accessing this attribute may result in database errors in case it is used on cursors with prepared but not yet executed statements.

One of the reasons for this is that the ODBC drivers have not yet seen the parameter values which will be bound to the parameter markers in the statement, e.g. `"[Microsoft][ODBC SQL Server Driver][SQL Server]Procedure or function 'myfunc' expects parameter '@param1', which was not supplied."`

`.command`

Provides access to the last SQL command string or Unicode object that was passed to `.prepare()` or `.execute*()`. If no such command is available, `None` is returned.

It is set by `.prepare()` and `.execute*()` and reset by calling one of the catalog methods or `.close()` on the cursor.

Note that `.command` may be a Unicode object in case a Unicode object was passed to one of the above methods.

`.connection`

Connection object on which the cursor operates.

`.converter`

Read-only access to the converter function set using the `cursor.setconverter()` method or inherited from the `connection.converter` attribute at cursor creation time.

The attribute is `None` in case no converter function was set or inherited. mxODBC will then use the default type conversions when fetching data from the database. See section 8.6 Output Conversions for details.

`.cursortype`

Read/write attribute that sets the ODBC cursor type this cursor. It takes the same values as the `connection.cursortype` instance variable and defaults to the creating connection object's settings for `connection.cursortype`. For possible values, please see the connection `.cursortype` attribute in section 6.8 Connection Object Attributes.

Changes to the cursor type must be made before opening the cursor, ie. before executing a statement on it.

Please refer to section 5.9 ODBC Cursor Types for more details on cursor types.

`.datetimeformat`

Attribute to set the output format for date/time/timestamp columns on a per cursor basis. It takes the same values as the `connection.datetimeformat` instance variable and defaults to the creating connection object's settings for `connection.datetimeformat`.

`.decimalformat`

Attribute to set the output format for decimal/numeric columns on a per cursor basis. It takes the same values as the `connection.decimalformat` instance variable and defaults to the creating connection object's settings for `connection.decimalformat`.

`.description`

This read-only attribute is a sequence of 7-item sequences for operations that produce a result set (which may be empty).

Each of these sequences contains information describing one result column: `(name, type_code, 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.

mxODBC always returns None for `display_size` and `internal_size`. This information can be obtained via `connection.gettypeinfo()`, if needed.

The `type_code` can be interpreted by comparing it to the type objects specified in the section 8 *Type Objects and Constructors* below. mxODBC returns the SQL type integers in this field. These are described in the section 8

Supported Data Types and are available through the `SQL` singleton defined at module level.

Please see section 5.8.2 Result Set Introspection for more information on this attribute and how to use it.

`.encoding`

Read/write attribute which defines the encoding to use for converting Unicode to 8-bit strings and vice-versa. If set to `None` (default), Python's default encoding will be used, otherwise it has to be a string providing a valid encoding name, e.g. `'latin-1'` or `'utf-8'`.

The setting is inherited from the `connection.encoding` at cursor creation time, but can be adjust independently from the connection after its creation. All cursor related APIs such `cursor.execute*()` and `cursor.fetch*()` methods use the `cursor.encoding` for Unicode conversions.

`.messages`

This is a Python list object to which the standard mxODBC error handler appends tuples `(exception class, exception value)` for all messages which the interfaces receives from the underlying ODBC driver or manager for this cursor. See section 10. *Error Handlers* for details.

The list is cleared by all cursor methods calls (prior to executing the call) except for the `.fetch*()` calls to avoid excessive memory usage and can also be cleared explicitly by executing `del cursor.messages[:]`.

An application can use the information in this list to verify correct operation of the method calls. This is particularly useful if the ODBC driver or database splits the error information across multiple error messages. In such a case, only one of the messages will be used to raise the exception by mxODBC (usually the top-most), but this message may not provide enough information to track down the problem.

`.paramcount`

This read-only attribute specifies the number of parameters in the current prepared command.

The attribute is `-1` in case this information is not available.

`.paramstyle`

Sets the default parameter binding style of the cursor. The value is initially set to the value of `connection.paramstyle` of the creating connection. The value takes affect on the next call to a `cursor.execute*()` method.

The attribute can be set or queried and takes the following string values (following the `paramstyle` module global as defined in the DB-API):

`'qmark' (default)`

This is the default ODBC parameter binding style and also used as native database binding style by MS SQL Server and IBM DB2.

Parameters in SQL statements used on `cursor.execute*()` methods are marked with the question mark letter (`'?'`) and the variables are bound to

these parameter locations using a positional mapping. Parameter values for a SQL statement must be specified as sequence, normally a list or a tuple.

Example: `'SELECT * FROM MyTable WHERE A=? AND B=?'` used with a parameter tuple `(1, 2)` would result in the database executing the query `'SELECT * FROM MyTable WHERE A=1 AND B=2'`.

`'named'`

The 'named' parameter binding style is used by the native database interfaces of e.g. Oracle.

Parameters in SQL statements used on `cursor.execute*()` methods are marked with a colon followed by a name, e.g. `':a'` or `':1'`. The variables are bound to these parameter locations using a name based mapping. Parameter values for a SQL statement must be specified as mapping, normally a dictionary, and are bound to the locations based on the names used in the SQL statement.

Example: `'SELECT * FROM MyTable WHERE A=:a AND B=:b'` used with a parameter dictionary `{'a': 1, 'b': 2}` would result in the database executing the query `'SELECT * FROM MyTable WHERE A=1 AND B=2'`.

`.row`

This attribute sets the default `cursor.row` object constructor to be used by all newly created cursor objects on this connection.

If set to an object constructor taking a row tuple as argument (i.e. `cursor.row(row_tuple)`), mxODBC will use the constructor to wrap all rows fetched through one of the `.fetch*()` methods. Instead of row tuples, the `.fetch*()` methods will then return the objects created by this constructor.

Default is `None`, which means that mxODBC will use regular Python tuples for returning row data in result sets.

Please see section 5.10 Custom Cursor Row Objects and Row Factory Functions for details on how to use `cursor.row` and `cursor.rowfactory`.

`.rowfactory`

This attribute sets the default `cursor.rowfactory` row object constructor factory to be used by all newly created cursor objects on this connection.

If set to a factory function, mxODBC will call this factory function with the cursor as argument when starting to fetch a result set and set the `cursor.row` attribute to the object returned by the factory function (i.e. `cursor.row = cursor.rowfactory(cursor)`).

The purpose of the row factory is to dynamically set the `cursor.row` attribute after having executed a statement on the cursor. This allows the factory function to use the `cursor.description` for building a row object constructor customized to the available result set.

Default is `None`, which means that mxODBC will not use a row factory function and leave `cursor.row` untouched.

Please see section 5.10 Custom Cursor Row Objects and Row Factory Functions for details on how to use `cursor.row` and `cursor.rowfactory`.

`.rowcount`

This read-only attribute specifies the number of rows that the last `.execute*()` produced (for DQL statements like `select`) or affected (for SQL DML statements like `update` or `insert`).

The attribute is `-1` in case no `.execute*()` has been performed on the cursor or the rowcount of the last operation is not determinable by the interface or the database.

> You should check whether the database you are interfacing to supports `.rowcount` before writing code which relies on it. Many databases such as MS Access and Oracle do not provide this information to the ODBC driver, so `.rowcount` will always be `-1`.

`.rownumber`

This read-only attribute provides the current 0-based row position of the cursor in the result set. The next `.fetch*()` will return rows starting at the given position.

The row position is automatically updated whenever the cursor moves through the result set, either due to fetches or scrolls.

The attribute is `None` in case no `.execute*()` has been performed on the cursor or the cursor position cannot be determined.

mxODBC provides a `.rownumber` emulation on the client-side for databases that do not implement the ODBC feature, such as e.g. MS Access, Teradata or Oracle. If the emulation cannot be provided, the attribute will return `None`.

`.stringformat`

Attribute to set the conversion format for string columns on a per cursor basis. It takes the same values as the `connection.stringformat` instance variable and defaults to the creating connection object's settings for `connection.stringformat`.

`.timestampresolution`

Attribute to set the timestamp resolution for timestamp input columns on a per cursor basis. It works in the same ways as the `connection.timestampresolution` instance variable and defaults to the creating connection object's settings for `connection.timestampresolution`.

`.warningformat`

Attribute to set the database warning reporting method used by the mxODBC default error handler. It takes the same values as the `connection.warningformat` instance variable and defaults to the creating connection object's settings for `connection.warningformat`.

8. Data Types supported by mxODBC

mxODBC tries to maintain as much of the available information across the Python-ODBC bridge as possible. In order to implement this, mxODBC converts between the ODBC and the Python world by using native data types in both worlds.

You should note however, that some ODBC drivers return data using different types than the ones accepted for input, e.g. a database might accept a time value, convert it internally to a timestamp and then return it in a subsequent SELECT as timestamp value.

mxODBC cannot know that the value only contains valid time information and no date information and thus converts the output data into an mxDateTime DateTime instance instead of an `mx.DateTime.DateTimeDelta` instance (which would normally be returned for time values).

The included mx/ODBC/Misc/test.pyc can help to check for this behavior. It tests many common column types and other database features which are useful to know when writing applications for a particular database backend.

8.1 mxODBC Parameter Binding

When defining SQL statements that use parameters, mxODBC provides a way to bind Python values to those parameters called parameter binding.

Instead of using the literal parameter values in the SQL statement passed to the `cursor.execute*()` methods, you can use a parameter binding character or character sequence to define the parameter locations in the SQL statement and then pass the Python parameter values to the `cursor.execute*()` methods as additional parameter. The ODBC driver or the database backend will then take the values and use them to run the SQL statement.

Example:

> Use `"SELECT * FROM MyTable WHERE A=? AND B=?"` and `(1, 2)` (parameter binding) instead of `"SELECT * FROM MyTable WHERE A=1 AND B=2"` (embedding parameters literally).

This has both a performance and a security advantage.

Performance is much better if the database backend can easily identify whether it has already created an access plan for a SQL statement by simply looking at the parameterized version of the statement, than first having to convert a SQL

statement with embedded literal parameters to a normalized form and then find that it already has an access plan.

If you plan to run the same statement over and over again or use `cursor.executemany()`, then the ODBC driver only has to pass the SQL statement and the list of parameters to the database, rather than build and send hundreds of statements across the wire to the database.

Security is better since the ODBC driver or database backend based building of the final SQL statement prevents the popular *SQL injection attack* on applications.

With this attack method, an attacker tries to trick an application into inserting a specially prepared SQL statement string sequence into an application defined SQL statement template. Say the application uses `"SELECT * FROM MyTable WHERE A=%s"`. An attacker could then try to send the parameter value `"1; DROP TABLE MyTable"` to the application, which would then result in the SQL statement `"SELECT * FROM MyTable WHERE A=1; DROP TABLE MyTable"` to be executed - in case the application doesn't very carefully check, parse and quote the parameter value for A.

8.1.1 Parameter Binding Styles

mxODBC uses the *ODBC parameter style* as binding parameter marker style per default. This style is called `'qmark'` because it uses positional question mark markers (`'?'`) to locate the parameters, e.g. `'SELECT * FROM MyTable WHERE A=?'`.

Starting with mxODBC 3.2, mxODBC also provides a way to adjust the parameter style on a per connection and per cursor basis. In addition of the `'qmark'` parameter style, mxODBC also supports the Oracle style `'named'` parameter style.

The default style is still the `'qmark'` style, but you can set the `connection.paramstyle` to `'named'` to have all new cursors created on the connection default to the `'named'` style. The default `cursor.paramstyle` is set to the value `connection.paramstyle` of the connection on which the cursor was created.

It is also possible to adjust existing cursors to use the `'named'` parameter style for all subsequent `cursor.execute*()` method calls by simply setting `cursor.paramstyle` to `'named'`. This has no affect on other cursors created on the same connection.

Example:

```
cursor.paramstyle = 'qmark'
cursor.execute("SELECT * FROM MyTable WHERE A=? AND B=?",
               (1, 2))
print cursor.fetchall()

cursor.paramstyle = 'named'
cursor.execute("SELECT * FROM MyTable WHERE A=:a AND B=:b",
               {'a': 1, 'b': 2})
```

```
print cursor.fetchall()
```

More information about the connection and cursor attribute `.paramstyle` is available in section 6.8 Connection Object Attributesand 7.7 Cursor Object Attributes.

8.1.2 Limitations of parameter markers

The ODBC standard allows some limitations on where parameter markers can be used in an SQL statement (see *https://msdn.microsoft.com/en-us/library/ms709310%28VS.85%29.aspx*), e.g.

- in a SELECT list
- as both expressions in a comparison-predicate
- as both operands of a binary operator
- as both the first and second operands of a BETWEEN operation
- as both the first and third operands of a BETWEEN operation
- as both the expression and the first value of an IN operation
- as the operand of a unary + or – operation
- as the argument of a set-function-reference

Client cannot determine parameter types

Most of these restrictions are needed since it would be impossible to determine the required parameter types when binding parameters are used in these cases.

However, since mxODBC supports more than just one binding mode (see the next section 8.3 mxODBC Input Binding Modes), these limitations can be avoided in some cases, provided the ODBC drivers play along.

To put it simple: In **SQL type binding mode**, mxODBC asks the ODBC driver for the parameter types. In **Python type binding mode**, mxODBC looks at the parameter you pass to the `.execute*()` methods and passes this to the ODBC driver, so it doesn't rely on having the ODBC driver determine the parameter type beforehand.

By using Python type binding mode, mxODBC does not have to rely on the ODBC driver determining the parameter types and thus the driver does not need to apply the above restrictions.

Use direct execution mode

If this still doesn't work, please see the section 8.2 mxODBC Direct Execution Mode on direct execution mode provided by mxODBC.

8.2 mxODBC Direct Execution Mode

As we've seen in section 8.1.2 Limitations of parameter markers, using Python type mode can be a useful way to work around restrictions in the ODBC drivers.

However, even when using Python type mode, the driver may still complain about restricted use of parameter types. For this reason, mxODBC supports the so called **direct mode of execution**. In this mode, the SQL string you pass to the `.execute*()` method is not parsed on the client side, but instead sent as-is to the server side together with the parameters provided by the client application. Evaluation and parsing then takes place on the server side and the client does not have to be able to determine the parameter types beforehand.

The **MS SQL Server Native Client** is one such ODBC driver which imposes the above restrictions when using `cursor.execute()`. It lifts many of these restrictions when using `cursor.executedirect()` or one of the other direct execution modes.

These are the available APIs for direct execution:

- `cursor.execute(…, direct=1)`
- `cursor.executedirect(…)`
- `cursor.executemany(…, direct=1)`

Please see the section 7.6 Cursor Object Methods for details on these methods.

Direct execution implies Python type mode

When using direct execution mode, mxODBC defaults to Python type binding mode, so you have to be careful to pass in the right parameters to the `.execute*()` methods to avoid errors.

In particular, it is sometimes difficult for mxODBC to determine whether to send data as binary or text data (when using 8-bit strings). You may have to wrap binary data which looks like text data in a Python buffer() object to force sending it as binary data.

Please section 8.3 mxODBC Input Binding Modes for details about the parameter binding modes supported by mxODBC.

Better performance with direct execution mode

Another positive side-effect of using direct execution is that of increased performance you can find with some databases and drivers. Again, the MS SQL Server Native Client / MS SQL Server combination is one such example where direct execution can result in better performance.

8.3 mxODBC Input Binding Modes

When passing parameters to the `.execute*()` methods of a cursor, mxODBC has to apply type conversions to the parameters in order to send them to the database in an appropriate form. This process is called binding a variable.

mxODBC implements two different input variable binding modes depending on what the ODBC driver can deliver. The currently used binding mode can be determined by looking at the `cursor.bindmethod` (which is inherited from the `connection.bindmethod` at cursor creation time).

Binding Mode	*Value of* `.bindmethod`	*Comments*
SQL type binding	`BIND_USING_SQLTYPE`	The database is asked for the appropriate data type and mxODBC tries to convert the input variable into that type. This is the preferred binding mode since it allows to choose the right conversion before passing the data to the ODBC driver.
Python type binding	`BIND_USING_PYTHONTYPE`	mxODBC looks at the type of the input variable and passes its value to the database directly; conversion is done by the ODBC driver/manager as necessary.

The default depends on the capabilities of the ODBC driver being used on the connection. mxODBC will always try to use the SQL type binding mode (`BIND_USING_SQLTYPE`), since this offers more flexibility than Python type binding.

> Please note that mxODBC will try to use SQL type binding if possible, but always falls back to Python type binding mode in case it cannot access the needed type information from the ODBC driver or database. This even applies if the binding mode is set to SQL type binding.

8.3.1 Adjusting the Type Binding Mode

As for many other attributes, mxODBC provide ways of defining the binding method on a per connection or a per cursor basis.

Per Connection Type Binding Setting

If you run into problems when using mxODBC in SQL type binding mode, please try to use Python type binding mode by configuring the connections to use Python type binding mode:

```
connection.bindmethod = BIND_USING_PYTHONTYPE
```

After setting the `connection.bindmethod` all cursors created on the connection will use the new bind method as default.

Per Cursor Type Binding Setting

If you want to adjust the bind method on a per-cursor basis, this is possible as well, by setting the `cursor.bindmethod` attribute.

```
cursor.bindmethod = BIND_USING_PYTHONTYPE
```

Doing so will not affect the connection the cursor was created on or any other cursors created on the connection.

Per-Statement Binding Mode

With some database drivers, it is also possible to trigger the Python type binding mode in a more fine-grained way on a per statement basis.

This is done by using `cursor.executedirect()` method or the `direct=1` parameters on other execution methods for running SQL statements against the database.

mxODBC will then send the statements as-is to the database server and apply Python type binding for the parameters.

8.4 SQL Type Input Binding

The following data types are used for SQL type input binding mode - `cursor.bindmethod` (inherited from `connection.bindmethod`) set to `BIND_USING_SQLTYPE`.

The SQL type is what the database ODBC driver expects from mxODBC. The interface then tries to convert the Python input objects to the Python type given in the table before passing it on to the ODBC driver.

SQL Type	*Python Type*	*Comments*
`SQL.CHAR`, `SQL.VARCHAR`, `SQL.LONGVARCHAR` (TEXT, BLOB or LONG in SQL)	String or Unicode or stringified object	The conversion truncates the string at the SQL field length. The handling of special characters depends on the codepage the database uses. Some database drivers/managers can't handle binary data in these column types, so you better check the database's capabilities with the included mx/ODBC/Misc/test.pyc first before using them. The handling of Unicode depends on the setting of the `.stringformat` attribute. In `NATIVE_UNICODE_STRINGFORMAT` mode, Unicode is passed to the ODBC driver as native Unicode. Strings are converted to Unicode using the `cursor.encoding` setting. In all other modes, Unicode is converted to an 8-bit string before passing it to the ODBC driver using the `cursor.encoding` setting. Strings are passed as-is. Note that for DateTime input objects, seconds rounding is applied just like for SQL.TIMESTAMP SQL types. For DateTimeDelta input objects, seconds are truncated to whole seconds.
`SQL.WCHAR`, `SQL.WVARCHAR`,	String or Unicode or	The conversion truncates the string at the SQL field length.

SQL Type	*Python Type*	*Comments*
`SQL.WLONGVARCHAR` (TEXT, BLOB or LONG in SQL)	stringified object	Non-string objects are passed through `unicode(obj, cursor.encoding)` to convert them to Unicode objects, except numbers, which are passed through `unicode(obj)`, i.e. without using the connection encoding. The handling of Unicode depends on the setting of the `.stringformat` attribute. In `EIGHTBIT_STRINGFORMAT` and `UNICODE_STRINGFORMAT` mode, Unicode is converted to an 8-bit string before passing it to the ODBC driver using the `cursor.encoding` setting. Strings are passed as-is. In all other modes, Unicode is passed to the ODBC driver as native Unicode. Strings are converted to Unicode before passing them to the ODBC driver using the `cursor.encoding` setting. Note that for mxDateTime input objects, seconds rounding is applied just like for SQL.TIMESTAMP SQL types. For DateTimeDelta input objects, seconds are truncated to whole seconds.
`SQL.BINARY`, `SQL.VARBINARY`, `SQL.LONGVARBINARY` (BLOB or LONG BYTE in SQL)	buffer, memoryview or String	Truncation at the SQL field length. These columns can contain embedded 0-bytes and other special characters. Handling of these column types is database dependent. Please refer to the database's documentation for details. Many databases store the passed in data as-is and thus make these columns types useable as storage facility for arbitrary binary data. Note that for mxDateTime input objects, seconds rounding is applied just like for SQL.TIMESTAMP SQL types. For DateTimeDelta input objects, seconds are truncated to whole seconds. Some database backends and some ODBC drivers don't support sending binary data to the database as string.

SQL Type	*Python Type*	*Comments*
		Please wrap the strings as Python buffer() if you run into such cases.
`SQL.TINYINT`, `SQL.SMALLINT`, `SQL.INTEGER`, `SQL.BIT`	Integer or any other object which can be converted to a Python integer	Conversion from the Python integer (a C long) to the SQL type is left to the ODBC driver/manager, so expect the usual truncations.
`SQL.BIGINT`	Long integer or any other object which can be converted to a Python long integer	Conversion to and from the Python long integer is done directly, if possible, or via the string representation if the C data types are not sufficient to hold the numeric data. If mxODBC has to use the string representation for interfacing, you may receive errors indicating truncation or errors because the database sent string data that cannot be converted to a Python long integer. Not all SQL databases implement this type or impose size limits.
`SQL.DECIMAL`, `SQL.NUMERIC`	Python decimal.Decimal or Float or any other object which can be converted to a Python float	Conversion from the Python float (a C double) to the SQL type is left to the ODBC driver/manager, so expect the usual truncations. Python decimals are passed to that database as strings, so no truncation or loss of precision occurs.
`SQL.REAL`, `SQL.FLOAT`, `SQL.DOUBLE`	Float or any other object which can be converted to a Python float	Conversion from the Python float (a C double) to the SQL type is left to the ODBC driver/manager, so expect the usual truncations.
`SQL.DATE`	DateTime instance or datetime.date instance or a tuple `(year,month,day)` or String/Unicode or a ticks value as Python number	While you should use DateTime instances, the module also accepts Python datetime.date instances, ticks (Python numbers indicating the number of seconds since the Unix Epoch; these are converted to local time and then stored in the database) and tuples `(year,month,day)` on input.
`SQL.TIME`	DateTimeDelta instance or datetime.time instance or a tuple `(hour,minute,second)` or String/Unicode or a tocks value as Python	While you should use DateTimeDelta instances, the module also accepts Python datetime.time instances, tocks (Python numbers indicating the number of seconds since 0:00:00.00) and tuples

SQL Type	*Python Type*	*Comments*
	number	`(hour,minute,second)` on input. Seconds are rounded to the nearest nanosecond in order to avoid issues with float second values which sometimes cannot be represented with full accuracy.
`SQL.TIMESTAMP`	DateTime instance or datetime.datetime instance or datetime.date instance or a tuple `(year,month,day, hour,minute,second)` or String/Unicode or a ticks value as Python number	While you should use DateTime instances, the module also accepts Python datetime.datetime instances, datetime.date instances (the time part is then set to 00:00:00), ticks (Python numbers indicating the number of seconds since the epoch; these are converted to local time and then stored in the database) and tuples `(year,month,day, hour,minute,second)` on input. Seconds are rounded according to the setting of the .timestampresolution setting, which defines the resolution of the timestamps in nanoseconds. mxODBC will round the timestamp's second value to the nearest nanosecond fraction defined by this setting in order to avoid issues with float second values which sometimes cannot be represented with full accuracy. If not set, the .timestampresolution attribute defaults to 1 nanosecond, so rounding usually is done to the nearest nanosecond, which is also the smallest fraction supported by the ODBC standard.
Any nullable column	`None`	The Python `None` singleton is converted to the special SQL NULL value.
Unsupported Type	String or stringified object	Input binding to these columns is done via strings (or stringified versions of the input data). Note that for mxDateTime input objects, seconds rounding is applied just like for SQL.TIMESTAMP SQL types.

8.5 Python Type Input Binding

The following mappings are used for input variables in Python type input binding mode - `cursor.bindmethod` (inherited from `connection.bindmethod`) set to `BIND_USING_PYTHONTYPE`. The table shows how the different Python types are converted to SQL types.

Python Type	*SQL Type*	*Comments*
String	`SQL.VARCHAR`, `SQL.LONGVARCHAR`, `SQL.VARBINARY`, `SQL.LONGVARBINARY` (char *)	The conversion truncates the string at the SQL field length. If the string contains binary data, SQL.VARBINARY is used for passing the data to the ODBC driver/manager for backends which don't support sending binary data as SQL_CHAR. The long variants are used for strings longer than 256 characters. If you get type conversion errors for binary data, it is possible that mxODBC has white-listed the database backend as supporting sending binary data using SQL_CHAR, but your ODBC driver does not support this (e.g. FreeTDS for MS SQL Server). In such cases, please wrap the strings as Python buffer() to make the binding as binary data explicit.
Unicode	`SQL.WVARCHAR`, `SQL.WLONGVARCHAR` (wchar_t *)	The conversion truncates the string at the SQL field length. Note that not all ODBC drivers/managers support Unicode data at C level. This binding is used for all cursors which do not have the `.stringformat` attribute set to `EIGHTBIT_STRINGFORMAT` or `UNICODE_STRINGFORMAT`. In `EIGHTBIT_STRINGFORMAT` mode (default) and `UNICODE_STRINGFORMAT` mode, Unicode objects are converted to a 8-bit strings first and then passed to the ODBC driver/manager. The long variant is used for Unicode data longer than 256 code points.
buffer or memoryview	`SQL.VARBINARY`, `SQL.LONGVARBINARY` (char *)	The conversion truncates the string at the SQL field length. The string may contain binary data. If the ODBC driver/manager doesn't support processing binary data using strings, wrap the data object using Python buffers (via the `buffer()` constructor) or Python memory

Python Type	*SQL Type*	*Comments*
		views (via the `memoryview()` constructor) to have mxODBC use a binary SQL type for interfacing to the driver/manager. The Oracle ODBC drivers usually need this. The long variant is used for binary data longer than 256 bytes.
Integer	`SQL.SLONG` (signed long)	Conversion from the signed long to the SQL column type is left to the ODBC driver/manager, so expect the usual truncations.
Long Integer	`SQL.CHAR` (char *)	Conversion from the Python long integer is done via the string representation since there usually is no C type with enough precision to hold the value.
Float	`SQL.DOUBLE` (double)	Conversion from the Python float (a C double) to the SQL column type is left to the ODBC driver/manager, so expect the usual truncations.

decimal.Decimal	`SQL.VARCHAR`,
	`SQL.LONGVARCHAR`
	`(char *)`
		Conversion from a Python decimal.Decimal instance is done via the string representation to avoid losing precision.
		
		The long variant is used for decimal representations longer than 256 characters.

DateTime	`SQL.TIMESTAMP`
	or
	`SQL.DATE`
		Converts the DateTime instance into a TIMESTAMP or DATE struct defined by the ODBC standard.
		
		The ODBC driver may use the time part of the instance or not depending on the SQL column type (DATE or TIMESTAMP).
		
		Seconds are rounded according to the setting of the .timestampresolution setting, which defines the resolution of the timestamps in nanoseconds.
		
		mxODBC will round the timestamp's second value to the nearest nanosecond fraction defined by this setting in order to avoid issues with float second values which sometimes cannot be represented with full accuracy. If not set, the .timestampresolution attribute defaults to 1 nanosecond, so rounding usually is done to the nearest nanosecond, which is also the smallest fraction supported by the ODBC standard.

DateTimeDelta	`SQL.TIME`	Converts the DateTimeDelta instance into a TIME struct defined by the ODBC standard.

Python Type	*SQL Type*	*Comments*
		Fractions of a second will be lost in this conversion.
datetime.date	`SQL.DATE`	Converts the datetime.date instance into a DATE struct defined by the ODBC standard. Note that some database backends don't support date column types and give an error when using datetime.date objects. MS SQL Server 2000 and 2005 are examples. MS SQL Server 2008 introduced a date column type.
datetime.time	`SQL.TIME`	Converts the datetime.time instance into a TIME struct defined by the ODBC standard.
datetime.datetime	`SQL.TIMESTAMP`	Converts the datetime.datetime instance into a TIMESTAMP struct defined by the ODBC standard. Seconds are rounded to the nearest nanosecond in order to avoid issues with float second values which sometimes cannot be represented with full accuracy.
`None`	Any nullable column	The Python None singleton is converted to the **special SQL NULL value**. and bound as VARCHAR data type (normally compatible with all other data types).
`BinaryNull`	Binary nullable column	Converted to the special SQL NULL value and bound as binary column type, avoiding conversion errors which some ODBC drivers issue when using the standard None value to represent NULL. This value is only used on input. Database NULL output is always the Python None singleton.[19]
Any other type	`SQL.VARCHAR`, `SQL.LONGVARCHAR`, `SQL.VARBINARY`, `SQL.LONGVARBINARY` (char *)	Conversion is done by calling str(variable) and then passing the resulting string value to the ODBC driver/manager. Same notes as for strings apply.

See the *ODBC Documentation* and your ODBC driver's documentation for more information on how these C data types are mapped to SQL column types.

[19] Added as work-around for MS SQL Server in mxOBDC 3.3.4.

8.6 Output Conversions

The following data types are used per default for output variable:

SQL Type	*Python Type*	*Comments*
`SQL.CHAR`, `SQL.VARCHAR`, `SQL.LONGVARCHAR` (TEXT, BLOB or LONG in SQL)	String	The handling of special characters depends on the codepage the database uses. In `NATIVE_UNICODE_STRINGFORMAT` and `UNICODE_STRINGFORMAT` mode, the string data is converted to a Python Unicode object based on the connection's encoding setting.
`SQL.WCHAR`, `SQL.WVARCHAR`, `SQL.WLONGVARCHAR` (TEXT, BLOB or LONG in SQL)	String or Unicode	Whether a Python string or Unicode object is returned depends on the setting of the `.stringformat` attribute of the cursor fetching the data. In `EIGHTBIT_STRINGFORMAT` mode, the Unicode data is converted to a Python string object based on the connection's encoding setting.
`SQL.BINARY`, `SQL.VARBINARY`, `SQL.LONGVARBINARY` (BLOB or LONG BYTE in SQL)	String	These can contain embedded 0-bytes and other special characters. Handling of these column types is database dependent. Please refer to the database's documentation for details.
`SQL.TINYINT`, `SQL.SMALLINT`, `SQL.INTEGER`, `SQL.BIT`	Integer or Long Integer	Bits are converted to Python integers 0 and 1 resp. Unsigned short integers are fetched as Python integers, unsigned integers as Python long integers.
`SQL.BIGINT`	Long Integer	mxODBC tries to fetch the long integer data directly and falls back to using string interfacing, if the platform does not provide the necessary C types for this.
`SQL.DECIMAL`, `SQL.NUMERIC`	Float or decimal.Decimal	In FLOAT_DECIMALFORMAT mode (default), mxODBC will fetch the numeric data as Python float. Since Python stores floats as double precision C float, rounding errors may occur during the

SQL Type	*Python Type*	*Comments*
		conversion. In DECIMAL_DECIMALFORMAT mode, mxODBC will fetch the numeric data as string and create a Python decimal.Decimal instance from it which is then returned. This avoids any rounding errors.
`SQL.REAL`, `SQL.FLOAT`, `SQL.DOUBLE`	Float	Python stores floats as double precision C float, so rounding errors may occur during the conversion.
`SQL.DATE`	DateTime instance or datetime.date instance or ticks or `(year,month,day)` or String	The type of the return values depends on the setting of `cursor.datetimeformat` and whether the ODBC driver/manager does return the value with proper type information. Default is to return DateTime instances.
`SQL.TIME`	DateTimeDelta instance or datetime.time instance or tocks or `(hour,minute,second)` or String	The type of the return values depends on the setting of `cursor.datetimeformat` and whether the ODBC driver/manager does return the value with proper type information. Default is to return DateTimeDelta instances.
`SQL.TIMESTAMP`	DateTime instance or datetime.datetime instance or ticks or `(year,month,day, hour,minute,second)` or String	The type of the return values depends on the setting of `cursor.datetimeformat` and whether the ODBC driver/manager does return the value with proper type information. Default is to return DateTime instances.
`SQL NULL value`	`None`	The Python `None` singleton is used to represent the special SQL `NULL` value in Python.
Unsupported Type	String	mxODBC will try to fetch data from columns using unsupported SQL data types as strings. This is likely to always work but may cause unwanted conversions and or truncations or loss of precision.

Output bindings can only be applied using the above mapping by mxODBC if the database correctly identifies the type of the output variables.

The SQL type given in the above table is also made available though the cursor's `.description` tuple as `type_code` entry (position 1) for result set generating SQL commands. You can compare this value directly to the appropriate `SQL` object values, e.g. test for `SQL.CHAR` or `SQL.VARCHAR`.

8.7 Output Type Converter Functions

The last section defined the standard mapping mxODBC applies when fetching output data from the database.

8.7.1 Converter Function Signatures

You can modify this mapping on-the-fly by defining a *cursor converter function* which takes three arguments and has to return a 2-tuple:

```
def converter(position,sqltype,sqllen):
    # modify sqltype and sqllen as appropriate
    return sqltype,sqllen

# Now tell the cursor to use this converter:
cursor.setconverter(converter)
```

or 3-tuple:

```
def converter(position,sqltype,sqllen):
    # modify sqltype and sqllen as appropriate, provide binddata as
    # input (e.g. for file names which should be used for file
    # binding)
    return sqltype,sqllen,binddata

# Now tell the cursor to use this converter:
cursor.setconverter(converter)
```

The converter function is called for each output column prior to the first `.fetch*()` operation executed on the cursor. The returned values are then interpreted as defined in the table in section 8.4 *Output Conversions and SQL Type Input Binding*.

The parameters have the following meanings:

`position`

identifies the 0-based position of the column in the result set.

`sqltype`

is usually one of the SQL data type constants, e.g. `SQL.CHAR` for string data, but could also have database specific values. mxODBC only understands the ones defined in the above table, so this gives you a chance to map user defined types to ones that Python can process.

`sqllen`

is only used for string data and defines the maximum length of strings that can be read in that column (mxODBC allocates a memory buffer of this size for the data transfer).

Returning `0` as `sqllen` will result in mxODBC dynamically growing the data transfer buffer when fetching the column data. This is sometimes handy in case you want to fetch data that can vary in size.

`binddata`

is optional and only needed for some special `sqltypes`. It will be used in future versions to e.g. allow binding output columns to files which some ODBC drivers support (the column data is transferred directly to a file instead of copied into memory).

8.7.2 Adjusting/Querying the Converter Function

Cursor objects will use the connection's `.converter` attribute as default converter. It defaults to `None`, meaning that no converter function is in effect. `None` can also be used to disable the converter function on a cursor:

```
# Don't use a converter function on the cursor
cursor.setconverter(None)
```

You can switch converter functions even in between fetches. mxODBC will then reallocate and rebind the column buffers for you.

The currently used converter function can be queried through the read-only `cursor.converter` attribute, e.g. to check whether the default mxODBC conversions are being used or not.

8.7.3 Example Converter Function

Example (always return INTEGER values as FLOATS):

```
def converter(position,sqltype,sqllen):
    if sqltype == SQL.INTEGER:
        sqltype = SQL.FLOAT
    return sqltype,sqllen

# Now tell the cursor to use this converter:
cursor.setconverter(converter)
```

8.8 Auto-Conversions

While you should always try to use the above Python types for passing input values to the respective columns, the package will try to automatically convert the types you give into the ones the database expects when using the SQL Type bind method, e.g. an integer literal '123' will be converted into an integer 123 by mxODBC if the database ODBC driver requests an integer.

The situation is different in Python type binding mode (`BIND_USING_PYTHONTYPE`): the Python type used in the parameter is passed directly to the database, thus passing '123' or 123 does make a difference and could result in an error from the database.

8.9 Unicode and String Data Encodings

mxODBC also supports Unicode objects to interface with databases. As more databases and ODBC drivers support Unicode natively, using Unicode for text data stored in database becomes more attractive than ever and allows you to avoid the problems you typically face when having to deal with different text encodings and code pages in databases.

Even if you don't have access to an ODBC capable of dealing with Unicode natively, you can still take advantage of the auto-conversion mechanisms in mxODBC to simulate Unicode capabilities.

mxODBC provides several different run-time configurations to deal with passing Unicode to and fetching it from an ODBC driver. The `.stringformat` attribute of connection and cursor objects allows defining how to convert string data into Python objects and vice-versa.

Unicode conversions to and from 8-bit strings in Python usually assume the Python default encoding (which is ASCII unless you modify the Python installation). Since the database may be using a different encoding, mxODBC allows defining the encoding to be used on a per-connection basis.

The `.encoding` attribute of connection and cursor objects is writeable for this purpose. Its default value is `None`, meaning that Python's default encoding (usually *ASCII*) is to be used. You can change the encoding by simply assigning a valid encoding name to the attribute. Make sure that Python supports the encoding (you can test this using the `unicode()` built-in).

The **default conversion mechanism** used in mxODBC is `EIGHTBIT_STRINGFORMAT` (Unicode gets converted to 8-bit strings before passing the data to the driver, output is always an 8-bit string), the default encoding Python's default encoding.

To store Unicode in a database, one possibility is to use the `UNICODE_STRINGFORMAT` and set the encoding attribute to e.g. `'utf-8'`.

mxODBC will then convert the Unicode input data to UTF-8, store this in the database and convert it back to Unicode during fetch operations. Note however that UTF-8 encoded data usually takes up more room in the database than the Unicode equivalent, so may experience data truncations which then cause the decoding process to fail.

Another possibility is to use the `MIXED_STRINGFORMAT` which allows mxODBC to interface to the database using the best suitable data type. For e.g. MS SQL Server this usually means passing all string data as Unicode data to and from the database. In `MIXED_STRINGFORMAT` mode mxODBC will return string data in the default format of the database driver, leaving the conversion to the Python program.

Note:
mxODBC only supports Unicode objects at the data storage interface level meaning that it can insert and fetch Unicode data from a database provided that the database can handle Unicode and that the used mxODBC subpackage was configured with Unicode support. It also supports SQL commands given as Unicode data. However, it does *not* handle Unicode at the schema interface level, that is e.g. `cursor.description` will not return Unicode objects for the column names. This may be added to a future version of mxODBC, but is currently not supported by the package.

8.10 Additional Comments

The above SQL types are provided by each subpackage in form of SQL type code integers through attributes of the singleton object `SQL`, e.g. `SQL.CHAR` is the type integer for a CHAR column.

You can decode the `type_code` value in the `cursor.description` tuple by comparing it to one of those constants. A reverse mapping of integer codes to code names is provided by the dictionary `sqltype` which is provided by all subpackages.

Note:
You may run into problems when using the tuple versions for date/time/timestamp arguments. This is because some databases (notably MySQL) want these arguments to be passed as strings. mxODBC does the conversion internally but tuples turn out as: '(1998,4,6)' which it will refuse to accept. The solution: use DateTime[Delta] instances instead. These convert themselves to ISO dates/times which most databases (including MySQL) do understand.

To check the ODBC driver/manager capabilities and support for the above column types, run the included mx/ODBC/Misc/test.pyc test script.

9. DB-API Type Objects and Constructors

Since many database have problems recognizing some column's or parameter's type beforehand (e.g. for LONGs and date/time values), the Python DB-API provides a set of standard constructors to create objects that can hold special values. When passed to the cursor methods, the module can then detect the proper type of the input parameter and bind it accordingly.

In mxODBC these constructors are not needed: it uses the objects defined in *mxDateTime* for date/time values and is able to pass strings, buffer and memoryview objects to LONG and normal CHAR columns without problems. You only need them to write code that is portable across database interfaces.

A Cursor Object's `description` attribute returns information about each of the result columns of a query. The `type_code` compares *equal* to one of Type Objects defined below. Type Objects may be equal to more than one type code (e.g. DATETIME could be equal to the type codes for date, time and timestamp columns).

mxODBC returns more detailed description about type codes in the `description` attribute. See the section 8 *Supported Data Types* for details. The type objects are only defined for compatibility with the DB API standard and other database interfaces.

Each subpackage exports the following constructors and singletons:

`Date(year,month,day)`

This function constructs an mxDateTime DateTime object holding the given date value. The time is set to 0:00:00.

`Time(hour,minute,second)`

This function constructs an mxDateTime DateTimeDelta object holding the given time value.

`Timestamp(year,month,day,hour,minute,second)`

This function constructs an mxDateTime DateTime object holding a time stamp value.

`DateFromTicks(ticks)`

This function constructs an mxDateTime DateTime object holding the date value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python `time` module for details).

Usage of Unix ticks (number of seconds since the Epoch) for date/time database interfacing can cause troubles because of the limited date range they cover.

`TimeFromTicks(ticks)`

This function constructs an mxDateTime DateTimeDelta 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).

`TimestampFromTicks(ticks)`

This function constructs an mxDateTime DateTime 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).

Usage of Unix ticks (number of seconds since the Epoch) for date/time database interfacing can cause troubles because of the limited date range they cover.

`Binary(string)`

This function constructs a buffer object pointing to the (long) string value. On Python versions without buffer objects (prior to 1.5.2), the string is taken as is.

`STRING`

This type object is used to describe columns in a database that are string-based: `SQL.CHAR`, `SQL.BINARY`.

`BINARY`

This type object is used to describe (long) binary columns in a database: `SQL.LONGVARCHAR`, `SQL.LONGVARBINARY` (e.g. LONG, RAW, BLOB, TEXT).

`NUMBER`

This type object is used to describe numeric columns in a database: `SQL.DECIMAL`, `SQL.NUMERIC`, `SQL.DOUBLE`, `SQL.FLOAT`, `SQL.REAL`, `SQL.DOUBLE`, `SQL.INTEGER`, `SQL.TINYINT`, `SQL.SMALLINT`, `SQL.BIT`, `SQL.BIGINT`.

`DATETIME`

This type object is used to describe date/time columns in a database: `SQL.DATE`, `SQL.TIME`, `SQL.TIMESTAMP`.

`ROWID`

This type object is used to describe the "Row ID" column in a database. mxODBC does not support this special column type and thus no type code is equal to this type object.

SQL NULL values are represented by the Python `None` singleton on input and output.

10. mxODBC Exceptions and Error Handling

The mxODBC package and all its subpackages use the DB API 2.0 exceptions layout. All exceptions are defined in the submodule `mx.ODBC.Error` but also imported into the top-level package module `mx.ODBC` as well as all sub-packages.

Note that all sub-packages use the same exception classes, so writing cross-database applications is simplified this way.

The exception values are either

- a single string, or
- a tuple having the format `(sqlstate, sqltype, errortext, lineno)`

SQL state (`sqlstate`) and type (`sqltype`) are defined by the ODBC standard and may be extended by the specific ODBC driver handling the connection. Please see the ODBC driver manual for details. `lineno` refers to the line number in the mxODBC.c file to ease debugging the package.

Note on the mx.ODBC.Error Module

If you want to import the exception classes from the `mx.ODBC.Error` submodule, you have to use the `from…import` form:

```
from mx.ODBC.Error import ProgrammingError
```

The reason is that the `Error` base class is imported into the top-level `mx.ODBC` package when loading it, shadowing the module of the same name. With the above form, Python will lookup `mx.ODBC.Error` in the module dictionary instead of the `mx.ODBC` package and find the module instead of the `mx.ODBC.Error` exception class.

10.1 Exception Classes

These exceptions are defined in the modules scope and also available as attributes of the connection objects to easy writing applications using different mxODBC sub-packages.

`Error`

Baseclass for all other exceptions related to database or interface errors.

You can use this class to catch all errors related to database or interface failures. `error` is just an alias to `Error` needed for DB-API 1.0 compatibility.

Error is a subclass of `exceptions.StandardError`.

`Warning`

Exception raised for important warnings like data truncations while inserting, etc.

Warning is a subclass of `exceptions.StandardError`. This may change in a future release to some other baseclass indicating warnings.

`InterfaceError`

Exception raised for errors that are related to the interface rather than the database itself.

`DatabaseError`

Exception raised for errors that are related to the database.

`DataError`

Exception raised for errors that are due to problems with the processed data like division by zero, numeric out of range, etc.

`OperationalError`

Exception raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc.

`IntegrityError`

Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails.

`InternalError`

Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc.

`ProgrammingError`

Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, performing operations on closed connections etc.

`NotSupportedError`

Exception raised in case a method or database API was used which is not supported by the database, e.g. requesting a .rollback() on a connection that does not support transaction or has transactions turned off.

This is the exception inheritance layout:

```
StandardError
|__Warning
|__Error
   |__InterfaceError
   |__DatabaseError
      |__DataError
```

```
|__OperationalError
|__IntegrityError
|__InternalError
|__ProgrammingError
|__NotSupportedError
```

10.2 SQL Error Mappings

If you are interested in the exact mapping of SQL error codes to exception classes, have a look at the `errorclass` dictionary which is defined at subpackage scope, e.g. `mx.ODBC.Windows.errorclass`.

`errorclass`

The `errorclass` dictionary maps SQLSTATE strings to error classes and is used by mxODBC to determine which Python exception class to use for reporting the database error within the Python application.

If you need to specify your own SQLSTATE to exception mappings, you can assign to the `errorclass` dictionary.

10.3 Exception Value Format

All ODBC driver generated exceptions use a standard exception value layout.

The error value will always be a tuple `(sqlstate, sqlcode, messagetext, lineno)` with the following meanings:

`sqlstate`

SQL state as string; these values are defined in the *ODBC Documentation* and by the ODBC driver/manager.

`sqlcode`

Numeric SQL error code as integer; these values are defined in the *ODBC Documentation* and by the ODBC driver/manager.

`messagetext`

Message text as string explaining the error. These strings usually have the format "[Vendor][Driver][Database] Message Text".

`lineno`

Line number in the mxODBC source code which generated the message. This is very useful for support purposes.

10.4 Error Handlers

If you want to provide your own error handler, e.g. to mask database warnings or debug connection setups, you can do so by assigning to the `.errorhandler` attribute of connections and cursors or passing a callback function to the connection constructors at connection creation time using the `errorhandler` keyword argument.

Error handlers are inherited from connections to cursors, so it normally suffices to set an error handler on the connection object to have it take affect for all subsequently created cursors.

> Cursors created prior to setting the error handler on the connection will not see or use the new error handler.

Error handler signature

An error handler has to be a callable object taking the arguments `(connection, cursor, errorclass, errorvalue)` where `connection` is a reference to the connection (or `None` in case the connection has not yet been setup), `cursor` a reference to the cursor (or `None` in case the error does not apply to a cursor or the cursor has not yet been setup), `errorclass` is an error class which to instantiate using `errorvalue` as construction argument.

Return values are currently not defined.

This a typical error handler function:

```
from mx.ODBC.Windows import Warning

# Error handler function
def myerrorhandler(connection, cursor, errorclass, errorvalue):
    if issubclass(errorclass, Warning):
        print ('Ignoring warning %s' % errorvalue)
    else:
        raise errorclass(*errorvalue)
```

Default error handler

The default mxODBC error handler will append the tuple `(errorclass, errorvalue)` to the `.messages` list of the cursor or connection (if cursor is `None`) and then raise the exception by instantiating `errorclass` with `errorvalue`.

Error processing

Note that only database and ODBC driver/manager related errors are processed through the error handlers. Other errors such as mxODBC internal or `AttributeErrors` are not processed by these handlers.

10.4.2 Examples

Here's an example of an error handler that allows to flexibly ignore warnings or only record messages.

```
# Error handler configuration
record_messages_only = 0
ignore_warnings = 0

# Error handler function
def myerrorhandler(connection, cursor, errorclass, errorvalue):

    """ Default mxODBC error handler.
        The default error handler reports all errors and warnings
        using exceptions and also records these in
        connection.messages as list of tuples (errorclass,
        errorvalue).

    """
    # Append to messages list
    if cursor is not None:
       cursor.messages.append((errorclass, errorvalue))
    elif connection is not None:
       connection.messages.append((errorclass, errorvalue))

    # Ignore warnings
    if (record_messages_only or
        (ignore_warnings and
         issubclass(errorclass, mx.ODBC.Error.Warning))):
        return

    # Raise the exception
    raise errorclass, errorvalue

# Installation of the error handler on the connection
connection.errorhandler = myerrorhandler
```

In case the connection or one of the cursors created from it cause an error, mxODBC will call the `myerrorhandler()` function to let it decide what to do about the error situation.

Possible error resolutions are to raise an exception, log the error in some way, ignore it or to apply a work-around.

Typical use-cases for error handlers are situations where warnings need to be masked or an application requires an on-demand reconnect.

If you need to catch errors or warnings at connection time, you can use the optional keyword argument `errorhandler` to have the error handler installed early enough to be able to deal with such errors or warnings:

```
connection = mx.ODBC.Windows.DriverConnect('DSN=test',
                errorhandler=myerrorhandler)
```

10.5 Warning Classes

The Python DB-API 2.0 does not define a warning class hierarchy. At the time the DB-API 2.0 was defined, the Python warning was not yet in existence. It is expected that a future revision will add such a hierarchy.

Until then mxODBC uses it's own warning hierarchy which currently just has one warning class:

`DatabaseWarning`

Warning issued for important warnings like data truncations while inserting, etc., if the mxODBC default error handler is active and the `connection.warningformat` or `cursor.warningformat` as set to `WARN_WARNINGFORMAT`.

`DatabaseWarning` is a subclass of the standard Python Warning base class. This may change in a future release if the DB-API is changed to provide a warning class hierarchy as well.

10.6 Database Warnings

The default behavior of mxODBC is to raise all errors, including `Warnings`, which many ODBC drivers issue for truncations, loss of precision in data conversions, etc.

This may not always be desirable. For this reason, mxODBC provides a way to handle database warnings in different ways.

10.6.1 Default Error Handler

The mxODBC default error handler can be adjusted to handle database warnings in three different ways:

1. raise a `Warning` exception for all database warnings (this is the default),
2. issue a Python `Warning` for all database warnings (compatible with the warning framework in Python),
3. ignore all database warnings.

Adjusting the mxODBC behavior is possible using the `connection.warningformat` or `cursor.warningformat` attributes. As always for these format settings, the cursors inherit the setting from the connection they were created from using the value set on the connection at creation time.

These `mx.ODBC` constants are available for the `.warningformat` attribute:

`ERROR_WARNINGFORMAT (default)`

Report warnings in the usual DB-API 2.0 way and raise a `Warning` exception.

`WARN_WARNINGFORMAT`

Instead of raising a `Warning` exception, issue a `mx.ODBC.DatabaseWarning` which is a Python `Warning` subclass and can be filtered using the standard Python *warnings module* mechanisms.

`IGNORE_WARNINGFORMAT`

Silently ignore the database warning.

The warning will still be added to the `.message` attribute, but no further action is taken.

10.6.2 Custom Warning Error Handler

If you need a more fine-grained approach to dealing with warnings, you can also setup a special error handler which then overrides the behavior of the default handler.

If you want to mask only certain `Warnings`, simply set a `connection.errorhandler` like the one below to disable raising exceptions for database warnings:

```
# Error handler function
def myerrorhandler(connection, cursor, errorclass, errorvalue):

    """ This error handler ignores (but logs) 01000 warnings issued
        by the database.

    """
    # Append to messages list
    if cursor is not None:
       cursor.messages.append((errorclass, errorvalue))
    elif connection is not None:
       connection.messages.append((errorclass, errorvalue))

    # Ignore 01000 database warning
    if (issubclass(errorclass, connection.Warning) and
        errorvalue[0] == '01000'):
        return

    # Raise all other database errors and warnings
    raise errorclass, errorvalue

# Installation of the error handler
connection.errorhandler = myerrorhandler
```

If you need to catch errors or warnings at connection time, you can use the optional keyword argument `errorhandler` to have the error handler installed early enough to be able to deal with such errors or warnings:

```
connection = mx.ODBC.Windows.DriverConnect('DSN=test',
```

```
                        errorhandler=myerrorhandler)
```

11. mxODBC Functions

mxODBC includes a few helper functions and generic APIs which aid in everyday ODBC database programming or allow introspection at the ODBC manager level. The next sections describe these functions in detail.

11.1 Subpackage Functions

For some subpackages, mxODBC also defines a few helpers which you can use to query additional information from the ODBC driver or manager. These are available through the subpackage, e.g. as `mx.ODBC.Windows.DataSources()`.

`DataSources()`

This helper function is only available for ODBC managers and some ODBC drivers which have internal ODBC manager support, e.g. IBM's DB2 ODBC driver, and allows you to query the available data sources.

It returns a dictionary mapping data source names to descriptions

Notes:

Older versions of unixODBC had a bug in some versions which makes the manager only return information about data sources on the first call to this function. Older versions of iODBC truncated the descriptions to two characters.

`getenvattr(option)`

Returns the given ODBC environment option. This method interfaces directly to the ODBC function `SQLGetEnvAttr()`.

`option` must be an integer. Suitable option values are available through the `SQL` singleton object.

The method returns the data as 32-bit integer. It is up to the caller to decode the integer using the SQL defines.

This function is only available for ODBC 3.x compatible managers and ODBC drivers.

`setenvattr(option, value)`

This function lets you set ODBC environment attributes which are encoded as 32-bit integers.

This method interfaces directly to the ODBC function `SQLSetEnvAttr()`.

`option` must be an integer. Suitable option values are available through the `SQL` singleton object.

This function is only available for ODBC 3.x compatible managers and ODBC drivers.

Note:
The function allows setting environment attributes which mxODBC itself uses to define the way it interfaces to the database. Changing these attributes can result in unwanted behavior or even segmentation faults. USE AT YOUR OWN RISK !

`statistics()`

Returns a tuple (connections, cursors) stating the number currently open connections and cursors for this subpackage.

Note that broken connections or cursors are not correctly counted.

11.2 mx.ODBC Functions

In addition to subpackage specific helpers, mxODBC also provides a few additional functions available through the top-level `mx.ODBC` package. These are:

```
format_resultset(cursor, headers=None, colsep=' | ', headersep='-
   ', stringify=repr)
```

Fetch the result set from cursor and format it into a list of strings (one for each row):

```
-header-
-headersep-
-row1-
-row2-
...
```

headers may be given as list of strings. It defaults to the header names from `cursor.description`. The function will add numbered columns as appropriate if it finds more columns than given in headers.

Columns are separated by `colsep`; the header is separated from the result set by a line of `headersep` characters.

The function calls stringify to format the value data returned by the driver into a string. It defaults to `repr()`.

```
print_resultset(cursor, headers=None)
```

Pretty-prints the current result set available through cursor.

See `format_resultset()` for details on formatting.

12. mxODBC Globals and Constants

12.1 Subpackage Globals and Constants

Each mxODBC subpackage exports the following globals and constants:

`BIND_USING_SQLTYPE, BIND_USING_PYTHONTYPE`

Integer values returned by or used for setting `connection.bindmethod` and `cursor.bindmethod`.

SQL type binding means that the interface queries the database to find out which conversion to apply and which input type to expect, while Python type binding looks at the parameters you pass to the methods to find out the type information and then lets the database apply any conversions.

The bind method default is database dependent, but can also be adjusted on a per connection or cursor basis.

`CHAR, VARCHAR, LONGVARCHAR, BINARY, VARBINARY, LONGVARBINARY, TINYINT, SMALLINT, INTEGER, BIGINT, DECIMAL, NUMERIC, BIT, REAL, FLOAT, DOUBLE, DATE, TIME, TIMESTAMP [, CLOB, BLOB, TYPE_DATE, TYPE_TIME, TYPE_TIMESTAMP, UNICODE, UNICODE_LONGVARCHAR, UNICODE_VARCHAR, WCHAR, WVARCHAR, WLONGVARCHAR]`

ODBC 2.0 type code integers for the various natively supported SQL types. These map to integers as returned in the type field of `cursor.description`.

They are also available through the SQL singleton, e.g. `SQL.CHAR`. The dictionary `sqltype` provides the inverse mapping.

The codes mentioned in square brackets are optional and only available if the ODBC driver/manager supports a later ODBC version than 2.5.

Note that mxODBC has support for unknown SQL types: it returns these types converted to strings. The conversion is done by the ODBC driver and may be driver dependent.

`DATETIME_DATETIMEFORMAT, PYDATETIME_DATETIMEFORMAT, TIMEVALUE_DATETIMEFORMAT, TUPLE_DATETIMEFORMAT, STRING_DATETIMEFORMAT`

Integer values which are used by `connection.datetimeformat` and `cursor.datetimeformat`.

mxODBC can handle different output formats for date/time values on a per connection and per cursor basis. See the documentation of the two attributes for more information.

`EIGHTBIT_STRINGFORMAT, MIXED_STRINGFORMAT, UNICODE_STRINGFORMAT, NATIVE_UNICODE_STRINGFORMAT`

Integer values which are used by `connection.stringformat` and `cursor.stringformat`.

mxODBC can handle different string conversion methods on a per connection and per cursor basis. See the documentation of the two attributes for more information.

`ERROR_WARNINGFORMAT, WARN_WARNINGFORMAT, IGNORE_WARNINGFORMAT`

Integer values which are used by `connection.warningformat` and `cursor.warningformat`.

mxODBC can use different ways of reporting database warnings on a per connection and per cursor basis. See the documentation of the two attributes for more information.

`FLOAT_DECIMALFORMAT, DECIMAL_DECIMALFORMAT`

Integer values which are used by `connection.decimalformat` and `cursor.decimalformat`.

mxODBC can handle different output formats for numeric and decimal database column types on a per connection and per cursor basis. See the documentation of the two attributes for more information.

`HAVE_UNICODE_SUPPORT`

Integer flag which is either `0` or `1` depending on whether mxODBC was compiled with Unicode support or not. Unicode support is always available in mxODBC 3.1 and later so this flag is always set to `1`.

`BinaryNull`

`BinaryNull` is a special singleton which can be used to bind a NULL value to binary column types for ODBC drivers which return a conversion error when trying to do the same with the standard Python None singleton.

The special value is never returned by mxODBC and only used as parameter input value.

At the moment, this work-around is only needed for MS SQL Server, when using the FreeTDS ODBC driver, or when using direct execution or Python type mode with the SQL Server Native Client ODBC driver on both Windows and Linux. Please see the respective driver notes for details.

`RowFactory`

A reference to the `mx.ODBC.Misc.RowFactory` module which provides access to a set of standard row factory functions which can be used for `cursor.rowfactory`. See section 13 mx.ODBC.Misc.RowFactory Module for details on the available API and section 5.10 Custom Cursor Row Objects and Row Factory Functions for usage examples.

`SQL`

Singleton object which defines nearly all values available in the ODBC 3.5 header files. The "`SQL_`" part of the ODBC symbols is omitted, e.g. `SQL_AUTOCOMMIT` is available as `SQL.AUTOCOMMIT`.

`apilevel`

String constant stating the supported DB API level. This is set to `'2.0'`, since mxODBC supports the features of the DB API 2.0 standard. Many DB API 1.0 features are still supported too for backward compatibility.

`errorclass`

Writeable dictionary mapping SQL error code strings (ODBC's SQLSTATE) to exception objects used by the module.

If you need to specify your own SQLSTATE to exception class mappings, you can assign to this dictionary. Changes will become visible immediately.

`license`

String with the license information for the installed mxODBC license.

`paramstyle`

String constant stating the type of parameter marker formatting expected by the interface. This is set to `'qmark'`, since the ODBC default is to use `'?'` to be used as positional placeholder for variables in an SQL statement.

Parameters are bound to these placeholders in the order they appear in the SQL statement, e.g. the first parameter is bound to the first question mark, the second to the second and so on.

Note that the parameter style can be changes on a per-connection or per-cursor basis using the `connection.paramstyle` and `cursor.paramstyle` read/write attributes.

`sqltype`

Dictionary mapping SQL type codes (these are returned in the type field of `cursor.description`) to type strings. All natively supported SQL type codes are included in this dictionary. The contents may vary depending on whether the ODBC driver/manager defines these types or not.

`threadsafety`

Integer constant stating the level of thread safety the interface supports. It is always set to `1`, meaning that each thread must use its own connection.

Some ODBC drivers also support sharing connections and even cursors between threads. Please have a look at your ODBC driver documentation for details.

12.2 mx.ODBC Globals and Constants

At the top-level, the mx.ODBC package defines these globals and constants:

```
Error, Warning, InterfaceError, DatabaseError, DataError,
   OperationalError, IntegrityError, InternalError,
   ProgrammingError, NotSupportedError
```

Exception objects used by the mxODBC subpackages. See section 10. mxODBC Exceptions and Error Handling for details.

13. mx.ODBC.Misc.RowFactory Module

This module defines a set of factory functions which can be used together with `cursor.rowfactory` to customize the row objects returned by the cursor.fetch*() methods in mxODBC.

This section describes the available module APIs. Please see section 5.10 Custom Cursor Row Objects and Row Factory Functions for details on how to use these row factory functions.

Classes

`Row`

Common base class of all Row classes built by the factory functions in this module.

This is useful to have in order to easily type check Row classes via `isinstance()`.

Base class(es): `object`

Functions

`TupleRowFactory(cursor)`

This is a factory which is a subtype of the Python tuple type and provides a standard tuple index based access to the row column values, as well as an attribute based one which is derived from the lower-cased column names found in `cursor.description`.

The row objects are immutable, just like standard tuples, but you can also slice them or index them as usual.

`cursor` has to be an mxODBC Cursor object.

`ListRowFactory(cursor)`

This factory uses a subtype of the Python list type and also provides a sequence index based access, as well as a named attribute access, just like the `TupleRowFactory`.

Unlike for the `TupleRowFactory`, the row objects created by the `ListRowFactory` are mutable lists, so you can assign to the indexes as well as the attributes.

`cursor` has to be an mxODBC Cursor object.

`NamespaceRowFactory(cursor)`

This row factory function creates `mx.Misc.Namespace.Namespace` objects as row objects. These provide a more complex namespace oriented API.

In addition to the sequence protocol, they also allow mapping access as well as named attribute access based on the lower-cased column names read from `cursor.description`.

Rows created by this factory are mutable.

`cursor` has to be an mxODBC Cursor object.

14. mx.ODBC Driver/Manager Packages

This section includes specific notes for preconfigured subpackages and setups.

14.1 Driver/Manager Subpackage Notes

The following sections provide hints that apply to all mx.ODBC subpackages. Please read carefully.

14.1.1 Windows Platform Notes

You should always use the `mx.ODBC.Windows` subpackage and access the databases through the MS ODBC Driver Manager. The other packages provide Unix based interfaces to the databases.

14.1.2 Unix Platform Notes

Even though there are many subpackages for specific databases which then sometimes provide more functionality for that particular database, we would like to encourage the use of ODBC managers such as the iODBC, unixODBC or DataDirect ODBC managers, since these provide the best flexibility in terms of database setup and configuration.

Using ODBC managers also enables you to easily switch from local databases to cross-network databases by adding additional tiers in-between.

The binary distributions of mxODBC for Unix platforms usually only contain the `mx.ODBC.unixODBC` and `mx.ODBC.iODBC` subpackages. For some platforms, the `mx.ODBC.DataDirect` subpackage is also included, e.g. Linux x86 and x86_64.

14.2 mx.ODBC.Manager -- Generic ODBC Driver Manager

In order to make writing cross-platform application easier with mxODBC, the package provides a meta-subpackage to access the default platform ODBC driver manager.

Windows Platforms

mxODBC selects the subpackage by trying to import the available ODBC driver subpackages in the following order:

1. *mx.ODBC.Windows*

No other subpackage is currently tried, since the Windows ODBC manager is always present in all recent Windows versions.

Unix Platforms

mxODBC selects the subpackage by trying to import the available ODBC driver subpackages in the following order:

1. *mx.ODBC.unixODBC*
2. *mx.ODBC.iODBC*
3. *mx.ODBC.DataDirect*

The `mx.ODBC.Manager` package then behaves just like the driver manager chosen by this process.

> Please note: The order was changed in mxODBC 3.2. Previous mxODBC versions preferred iODBC over unixODBC. Since unixODBC is widely supported nowadays and provides better Unicode support, selecting unixODBC over iODBC when both are present provides a better user experience.

14.3 mx.ODBC.Windows -- *Windows ODBC Driver Manager*

Tested with Windows XP, Vista, 7.

mxODBC links against the Windows ODBC driver manager on Windows. This is the only mxODBC interface subpackage available on Windows.

14.3.1 Connecting to a Database

Always use the `DriverConnect()` API to connect to the data source if you need to pass in extra configuration information such as names of log files, etc.

14.3.2 Supported Datatypes

The subpackage defaults to SQL type binding mode (see the *Datatypes* section for details), but reverts to Python type binding in case the connection does not support the ODBC SQLDescribeParam() API. **MS Access** is one candidate for which this API is not useable.

14.3.3 File Data Sources

If you want to connect to a file data source (without having to configure it using the ODBC manager), you can do so by using the FILEDSN= parameter instead of the DSN= parameter:

```
DriverConnect('FILEDSN=test.dsn;UID=test;PWD=test')
```

This is sometimes useful when you want to dynamically setup a data source, e.g. a MS Access database.

For more information about the FILEDSN-keyword and the other Windows ODBC manager features, see the *Microsoft SQLDriverConnect() documentation*.

Also note that ODBC drivers working on single files, e.g. the **MS Excel** file driver, usually do not support transactions. mxODBC will not clear auto-commit for these drivers (it may sometimes still be necessary to set the `clear_auto_commit` flag in the connect constructors to 0).

14.4 mx.ODBC.iODBC -- *iODBC Driver Manager*

Tested with iODBC 3.52.7.

iODBC is an Open Source ODBC manager for Unix maintained by *OpenLink*. It compiles against mxODBC without problems and is the preferred way of talking to an ODBC data source from Unix using mxODBC.

14.4.1 Notes

General Recommendations

- Please always use the `DriverConnect()` API to connect to the data source if you need to pass in extra configuration information such as names of log files, etc.

- When interfacing to MySQL using the MySQL ODBC driver, we have observed problems with using Unicode statements passed to `cursor.execute()` when using iODBC 3.52.5. These problems appear to be related to iODBC. As work-around, you can use unixODBC, which works fine with Unicode statements.

- You may experience problems when trying to connect to MySQL via MyODBC hooked to iODBC in case you are using the binary RPMs available. For some reason, the MyODBC driver does not reference the MySQL shared libraries it needs to connect to the MySQL server and there's no way to tell iODBC to load two shared libraries. Here's a trick which will allow you to create an import lib which solves the problem on Linux:

```
rm -f /usr/local/lib/libmyodbc.so
ld -shared --whole-archive \
       /usr/local/lib/libmyodbc-2.50.34.so \
       /usr/lib/libmysqlclient.so.10 \
       -o /usr/local/lib/libmyodbc.so
ldconfig
```

64-bit Platforms

- You may run into problems with iODBC since it uses 64-bit SQL Unicode types. Most ODBC drivers follow the Windows standard of using 32-bit Unicode types. Support for Unicode with iODBC is therefore limited.

- You may also run into problems with ODBC drivers compiled against unixODBC. While iODBC follows the ODBC standard of using 64-bit SQL length types, unixODBC has only recently (starting with version 2.2.13) switched to these longer types. As a result ODBC drivers compiled against older versions of unixODBC will not work reliably with iODBC.

- Commercial ODBC drivers for Unix are often compiled using 64-bit SQL length types and 32-bit Unicode types. iODBC uses 64-bit types for both.

14.5 mx.ODBC.unixODBC -- *unixODBC Driver Manager*

Tested with unixODBC 2.3.2.

unixODBC is an alternative Open Source ODBC manager for originally designed for Linux and later extended to other Unixes maintained by *EasySoft*. It compiles against mxODBC without problems.

Many open-source ODBC drivers are compiled against this driver manager per default, so it may provide better support for those drivers than iODBC.

14.5.1 Notes

General Recommendations

- Please always use the `DriverConnect()` API to connect to the data source if you need to pass in extra configuration information such as names of log files, etc.

- Between **unixODBC 2.3.0** and **unixODBC 2.3.1**, the unixODBC project switched the library name of the ODBC manager library from libodbc.so.1 to libodbc.so.2 to signal the change in their ABI on 64-bit platforms (see below). This renaming affects both 32- and 64-bit versions of unixODBC.

  Since eGenix.com compiles against unixODBC 2.3.1 (or later), mxODBC will look for a **libodbc.so.2** library file and this may not be available if your system comes with unixODBC 2.3.0. If you have unixODBC 2.3.0 installed you can safely create a symlink from the libodbc.so.1 library to the new name libodbc.so.2 to overcome this problem. Please see the *unixODBC website* for instructions.

Debugging ODBC Configurations

A common error you can see when trying to configure a database connection or ODBC driver installation is `mx.ODBC.Error.OperationalError: ('08003', 0, '[unixODBC][Driver Manager]Connnection does not exist', 11593)`.

Unfortunately, this doesn't tell you much about the true cause of the problem - only that the connection could not be established.

The reason is that unixODBC does not report the true cause of the problem as error, but instead only issues a warning and these warnings are ignored by mxODBC during connect, since they would prevent successful connects with some popular drivers that regularly issues context switch warnings during connects.

Finding the cause using an ODBC trace

If you run an ODBC trace and look at the log file, you'd see that unixODBC reports a warning such as this during the connection attempt: `[01000][unixODBC][Driver Manager]Can't open lib '/usr/…/libodbcdriver.so' : file not found]`

For details on how to setup an ODBC trace, please have a look at section 19.1 ODBC Call Level Tracing.

Finding the cause using a custom error handler

Alternatively, you can use a custom error handler to debug such situations during connects.

Example:

```
def debug_errorhandler(connection, cursor, errorclass, errorvalue):
    sys.stderr.write('debug_errorhandler: %s: %r\n' %
                     (errorclass, errorvalue))

db = DriverConnect('DSN=drivernotfound;UID=sa;PWD=test',
                   errorhandler=debug_errorhandler)
```

The error handler will be called for all messages coming from the ODBC driver and manager, including warnings which are then ignored by mxODBC, and print these to `stderr` for review.

Depending on how the ODBC driver is written, the above may result in lengthy output or even cause more severe problems, because the error handler does not raise an exception which would cause the connect processing to stop. mxODBC will continue setting up the connection as if it were existing and this can result in segfaults with some drivers.

A safer variant looks like this:

```
errors = 0
def debug_errorhandler(connection, cursor, errorclass, errorvalue):
    global errors
    errors += 1
    sys.stderr.write('debug_errorhandler: %s: %r\n' %
                     (errorclass, errorvalue))
    if errors > 5:
        raise errorclass(*errorvalue)
```

64-bit Platforms

- On 64-bit platforms you may run into problems with unixODBC since it uses 32-bit SQL length types for versions prior to 2.2.13. Some ODBC drivers on Unix instead use 64-bit SQL length values and will therefore not return correct results when used with unixODBC.

  The binary version eGenix.com ships was compiled against **unixODBC 2.3.1** (or later) and expects 64-bit SQL length types. If you need a version for unixODBC 2.2.12 or earlier, please either use our older mxODBC 3.0 release or write to *support@egenix.com* for help.

- Commercial ODBC drivers for Unix are often compiled using 64-bit SQL length types and 32-bit Unicode types. unixODBC uses the same types starting with version 2.3.

- You may run into problems with ODBC drivers compiled against iODBC. While unixODBC follows the ODBC standard of using 32-bit Unicode types, iODBC defaults to using the Unix 64-bit standard. As a result, ODBC drivers compiled against iODBC will not work reliably with Unicode data when used with unixODBC.

Threading

- In unixODBC versions 2.3.0 and below, the ODBC manager used a little known odbc.ini setting called "Threading" which determined the default thread level protection of the ODBC data source.

- The default used to be lock level 3 (the ODBC driver does not allow multiple threads to use it and everything is serialized). This could result in the application using mxODBC and unixODBC to hang during long running queries. Fixing this was easy, but not well documented in unixODBC. Setting the thread lock level to 0 (driver is fully thread safe) allowed the application to run other queries in parallel, e.g.

```
[PostgreSQL]
Description     = PostgreSQL driver for Linux & Win32
Driver          = /usr/local/lib/libodbcpsql.so
Setup           = /usr/local/lib/libodbcpsqlS.so
Threading       = 0
```

- **Starting with unixODBC 2.3.1, the default thread lock level now is 0**, so the above is no longer necessary.

- These are the available thread lock levels (from unixODBC's __handle.c):

```
Level 0 - Only the DM internal structures are protected
the driver is assumed to take care of it's self

Level 1 - The driver is protected down to the statement level
each statement will be protected, and the same for the connect
level for connect functions, note that descriptors are considered
equal to statements when it comes to thread protection.

Level 2 - The driver is protected at the connection level. only
one thread can be in a particular driver at one time

Level 3 - The driver is protected at the env level, only one thing
at a time.

By default the driver open connections with a lock level of 3,
this can be changed by adding the line

Threading = N

to the driver entry in odbcinst.ini, where N is the locking level
(0-3)
```

14.6 mx.ODBC.DataDirect -- *DataDirect ODBC Manager*

Tested with DataDirect ODBC Manager 7.1

DataDirect is a proprietary ODBC manager for Unix developed by DataDirect. It is used by a number of ODBC drivers available for Unix platforms.

eGenix.com provides binary subpackages for this ODBC driver manager **only on Linux x86 and x64** platforms. If you need the subpackage on other platforms as well, please contact *support@egenix.com* for help.

14.6.1 Notes

General Recommendations

- Please always use the `DriverConnect()` API to connect to the data source if you need to pass in extra configuration information such as names of log files, etc.

- The DataDirect ODBC manager may not work correctly with Unicode data, even though the driver may support Unicode.

  This is due to the fact that the DataDirect manager provides three different options of encoding Unicode data: UTF-8, UTF-16 (ODBC standard on Windows) and UTF-32. Most drivers assume UTF-16, but don't necessarily implement the special call needed to configure the DataDirect manager to assume this as well. The manager then defaults to UTF-8 and this causes the Unicode transport to fail.

  You can try to work around this by adding an entry

  ```
  # Needed by the DataDirect ODBC manager,
  # possible values: 1=UTF-16, 2=UTF-8
  DriverUnicodeType = 1
  ```

  to the driver section in your .odbc.ini file. See this page for details:

  http://web.datadirect.com/resources/odbc/unicode/unix.html

- In some cases we have observed segfaults when using ODBC drivers with the DataDirect ODBC manager. These were related to the ~/.odbc.ini being to large. Reducing the size of the ODBC configuration file resolved the problem. It is not clear whether the segfaults was caused by the driver or the driver manager or just a specific combination of both.

64-bit Platforms

- There are no known issues regarding 64-bit platforms.

14.7 ODBC Driver Subpackages

In previous mxODBC releases, eGenix.com included a limited set of additional subpackages with support for directly linking against specific ODBC drivers on Unix platforms.

Since these setups caused a lot of support requests due to configuration problems and version mismatches between the driver versions we used to build the binary mxODBC distribution and the ones deployed at customer sites, **we have decided to drop general support for these additional subpackages**.

It is usually better to use one of the available ODBC driver manager packages to configure and manage the data sources. These driver managers also provide a further abstraction layer between ODBC applications and the drivers, removing ODBC level compatibility issues, which makes the ODBC setup a lot less error prone.

> eGenix.com can still provide specific subpackages or build custom ones on request, if there is a need. Please contact *support@egenix.com* for details.

15. Hints & Links to other Resources

15.1 Running mxODBC from a CGI script

ODBC drivers and managers are usually compiled as a shared library. When running CGI scripts most HTTP daemons (or web servers) don't pass through the path for the dynamic loader (e.g. `LD_LIBRARY_PATH`) to the script, thus importing the mxODBC C extension will fail with unresolved symbols because the loader doesn't find the ODBC driver/manager's libs.

To have the loader find the path to those shared libs you can either wrap the Python script with a shell script that sets the path according to your system configuration or tell the HTTP daemon to set or pass these through (see the daemon's documentation for information on how to do this; for Apache the directives are named `SetEnv` and `PassEnv`).

On Windows, you also have to take into account that the ODBC data sources defined in the ODBC manager are usually restricted to specific user accounts. You can work around this by either setting up the ODBC data sources for the web server service account or by configuring the data as *system data sources*.

15.2 Running mxODBC with mod_wsgi

Using mxODBC with *mod_wsgi* is generally possible. However, since the script will run under a restricted user account, some care has to be taken to make the setup work. Please see 15.1 Running mxODBC from a CGI script for more details on getting ODBC drivers to work in such an environment.

mod_wsgi and Python 2.7

On Windows, there is also another issue to consider when running the combination Apache, mod_wsgi and Python 2.7. Due to changes in Python 2.7, manifests for the Visual C++ runtime environment, needed by Windows to find the right DLL to load, are no longer added to Python extensions, since this caused problems with loading them into Python processes (see *Python Issue 4120*).

Unfortunately, neither mod_wsgi nor Apache appear to include the required manifests either. This causes an import error when trying to load mxODBC into a mod_wsgi run process, since Windows cannot resolve the DLL references in mxODBC without the manifest.

Since this affects not only mxODBC, but other Python C extensions as well, you may want to use a work-around until either Apache or the mod_wsgi team solves the problem:

Manifest work-around

Adding the VC++ manifests to the Apache process is explained in *this posting*.

You will also have to install the *MS VC++ 2008 CRT SP1 redistributable package* on the server running Apache.

With those changes in place, mxODBC should load without problems.

15.3 Freezing mxODBC using py2exe

Thomas Heller has written a great tool which is based on distutils. The tool allows you to freeze your application into a single standalone Windows application and is called py2exe.

> **Note:**
> Freezing mxODBC together with an application and redistributing the resulting executables requires that you have obtained developer licenses from eGenix.com permitting you to redistribute mxODBC along with a product. Please see the *License section* for more information.

When freezing mxODBC you may experience problems with py2exe related to py2exe not finding the DLLs needed by mxODBC. In this case you have to help py2exe to find the correct subpackage for Windows, ie. *mx.ODBC.Windows* and *mx.DateTime*. This can be done by adding `-i mx.ODBC.Windows,mx.DateTime` to the py2exe command line:

```
python py2exe -i mx.ODBC.Windows,mx.DateTime yourapp.py
```

After doing so, py2exe should have no problem finding the files mxODBC.pyd and mxDateTime.pyd needed by `mx.ODBC.Windows` and `mx.DateTime`.

mxODBC also uses the *md5* or *hashlib* module (depending on the Python version) and the license module *mxodbc_license* internally. You will have to add them to the above list, if you run into license verification problems when running the py2exe compiled application.

15.4 More Sources of Information

There are several resources available online that should help you getting started with ODBC. Here is a small list of links useful for further reading:

Microsoft MDAC Site

Microsoft is constantly developing new forms of database access. For a close up on what they have come up recently take a look at their ODBC site. Note that they now call their ODBC SDK "Microsoft Data Access Components SDK" (MDAC). It does not only focus on ODBC but also on OLE DB and ADO.

Note: If you are not happy about the size of the SDK download (over 31MB), you can also grab the older 3.0 SDK which might still be available from a FTP server. Look for "odbc3sdk.exe" using e.g. FTP Search.

Microsoft also supports a whole range of (desktop) ODBC drivers for various databases and file formats. These are available under the name "ODBC Desktop Database Drivers" (search the MS web-site for the exact URL) [wx1350.exe] and also included in the more up-to-date "Microsoft Data Access Components" (MDAC) archive [mdac_typ.exe].

Microsoft ODBC Portal

This portal page has a few interesting links into the Microsoft ODBC site. If you're looking for the latest SQL Server or Oracle ODBC drivers this is the place to look first.

ODBC Documentation

The ODBC documentation is included in the free MS MDAC SDK which you can download from their *ODBC site*.

SQLSummit List of ODBC drivers

A collection of available ODBC driver packages. This should be the first place to look in case you are searching for OBDC connectivity to your database.

16. Examples

Here is a very simple example of how to use mxODBC. More elaborate examples of using Python Database API compatible database interfaces can be found in the *Database Topic Guide on http://www.python.org/*. *Andrew Kuchling's introduction to the Python Database API* is an especially good reading. There are also a few books on using Python DB API compatible interfaces, some of them cover mxODBC explicitly.

On Unix:

```
>>> import mx.ODBC.iODBC
>>> db = mx.ODBC.iODBC.DriverConnect('DSN=database;UID=user;PWD=passwd')
>>> c = db.cursor()
>>> c.execute('select count(*) from test')
>>> c.fetchone()
(305,)
>>> c.tables(None,None,None,None)
8
>>> mx.ODBC.print_resultset(c)
Column 1 | Column 2 | Column 3       | Column 4 | Column 5
-------------------------------------------------------------
''       | ''       | 'test'         | 'TABLE'  | 'MySQL table'
''       | ''       | 'test1'        | 'TABLE'  | 'MySQL table'
''       | ''       | 'test4'        | 'TABLE'  | 'MySQL table'
''       | ''       | 'testblobs'    | 'TABLE'  | 'MySQL table'
''       | ''       | 'testblobs2'   | 'TABLE'  | 'MySQL table'
''       | ''       | 'testdate'     | 'TABLE'  | 'MySQL table'
''       | ''       | 'testdates'    | 'TABLE'  | 'MySQL table'
''       | ''       | 'testdatetime' | 'TABLE'  | 'MySQL table'
>>> c.close()
>>> db.close()
>>>
```

On Windows:

```
>>> import mx.ODBC.Windows
>>> db =
mx.ODBC.Windows.DriverConnect('DSN=database;UID=user;PWD=passwd')
>>> c = db.cursor()
>>> c.execute('select count(*) from test')
>>> c.fetchone()
(305,)
>>> c.tables(None,None,None,None)
8
>>> mx.ODBC.print_resultset(c)
Column 1 | Column 2 | Column 3       | Column 4 | Column 5
-------------------------------------------------------------
''       | ''       | 'test'         | 'TABLE'  | 'MySQL table'
''       | ''       | 'test1'        | 'TABLE'  | 'MySQL table'
''       | ''       | 'test4'        | 'TABLE'  | 'MySQL table'
''       | ''       | 'testblobs'    | 'TABLE'  | 'MySQL table'
''       | ''       | 'testblobs2'   | 'TABLE'  | 'MySQL table'
''       | ''       | 'testdate'     | 'TABLE'  | 'MySQL table'
''       | ''       | 'testdates'    | 'TABLE'  | 'MySQL table'
''       | ''       | 'testdatetime' | 'TABLE'  | 'MySQL table'
>>> c.close()
>>> db.close()
>>>
```

As you can see, mxODBC has the same interface on Unix and Windows which makes it an ideal basis for writing cross-platform database applications.

Note:
When connecting to a database with transaction support, you should explicitly do a `.rollback()` or `.commit()` prior to closing the connection. In the example this was omitted since the used MySQL database backend does not support transactions and we were only reading from the database.

17. Testing the Database Connection

The package includes a test script that checks some of the database's features. As side effect this also provides a good regression test for the mxODBC interface.

To start the test, simply run the script in mx/ODBC/Misc/test.pyc.

```
python mx/ODBC/Misc/test.pyc
```

The script will generate a few temporary tables (named `mxODBC0001`, `mxODBC0002`, etc; no existing tables will be overwritten) and then test the interface - database communication including many database related features such as data types and support of various SQL dialects. The tables are automatically removed after the tests have run through.

18. mxODBC Package Structure

This is the Python package structure setup when installing mxODBC:

```
[ODBC]
       Doc/
       [Misc]
              proc.py
              test.pyc
       [DataDirect]
              dbi.py
              dbtypes.py
       [Manager]
       [Windows]
              dbi.py
              dbtypes.py
       [iODBC]
              dbi.py
              dbtypes.py
       [unixODBC]
              dbi.py
              dbtypes.py
       LazyModule.py
       ODBC.py
```

Entries enclosed in brackets are packages (i.e. they are directories that include a `__init__.py` file). Ones with slashes are just simple subdirectories that are not accessible via `import`.

19. Support

eGenix.com is provides commercial support for this package, including adapting it to special needs for use in customer projects. If you are interested in receiving information about this service please contact *support@egenix.com* for details.

This section describes methods which are useful to track down interoperability problems with ODBC drivers. eGenix support may ask you to apply some of the methods when working with you to resolve driver-related problems.

19.1 ODBC Call Level Tracing

In support some cases, eGenix may ask you to create an ODBC trace of a session demonstrating a problem you may have with a particular ODBC driver. This section explains how to enable ODBC call level tracing.

The ODBC trace log is a text file that ODBC managers (and some drivers) can generate in order to help with debugging the interaction between the driver, the driver manager and the application.

The method of how to enable tracing depends on the used ODBC driver manager.

19.1.1 Windows ODBC Manager

Open the *Windows ODBC Data Source Administrator* on Windows. This can be found in the in the *Control Panel* as *Administrative Tools* and is called *Data Sources (ODBC)*. See the *Windows ODBC documentation* for details.

Go to the *Tracing* tab and select a trace output file under *Log File Path* and then click on Start Tracing to enable ODBC tracing output.

After that is done, start you application or script using mxODBC and run the code that is causing problems with the driver in question.

After you've run the application or script, open the ODBC administrator again and click on the same button, now called Stop Tracing.

Finally, pick up the ODBC trace file from the location you've chosen and email it to *support@egenix.com*.

19.1.2 iODBC Driver Manager

To enable ODBC level tracing, open the /etc/odbc.ini or ~/.odbc.ini file and add this section to it:

```
[ODBC]
Trace = 1
TraceFile = /tmp/odbc.log
```

If you already have such entries in the `[ODBC]` section, make sure that the settings are correct and that `Trace` is set to 1.

After that is done, start you application or script using mxODBC and run the code that is causing problems with the driver in question.

After you've run the application or script, open the odbc.ini file again and set Trace to 0. This will disable ODBC call level tracing.

Finally, pick up the ODBC trace file from the location you've chosen (/tmp/odbc.log in the example) and email it to *support@egenix.com*.

19.1.3 unixODBC Driver Manager

To enable ODBC level tracing, open the /etc/odbcinst.ini or ~/.odbcinst.ini file and add this section to it:

```
[ODBC]
Trace = 1
TraceFile = /tmp/odbc.log
```

If you already have such entries in the `[ODBC]` section, make sure that the settings are correct and that `Trace` is set to 1.

After that is done, start you application or script using mxODBC and run the code that is causing problems with the driver in question.

After you've run the application or script, open the odbcinst.ini file again and set Trace to 0. This will disable ODBC call level tracing.

Finally, pick up the ODBC trace file from the location you've chosen (/tmp/odbc.log in the example) and email it to *support@egenix.com*.

19.1.4 DataDirect ODBC Driver Manager

Please use the same approach as for the iODBC Driver Manager.

19.1.5 Mac OS X ODBC Driver Manager

Open the *Mac OS X ODBC Administrator*. This can be found under *Applications/Utilities*.

Go to the *Tracing* tab and select a trace output file under *Log File* and then click the checkbox *Enable Tracing* to enable ODBC tracing output. Click on Apply have the change take effect.

After that is done, start you application or script using mxODBC and run the code that is causing problems with the driver in question.

After you've run the application or script, open the ODBC administrator again and disable the *Enable Tracing* checkbox.

Finally, pick up the ODBC trace file from the location you've chosen and email it to *support@egenix.com*.

19.2 mxODBC Call Level Tracing

To simplify debugging the mxODBC package can generate debugging output in several important places. The feature is only enabled if the mxODBC package was compiled with debug support and output is only generated if Python is run in debugging mode (use the Python interpreter flag `-d`):

```
python -d script.py
```

The resulting log file is named mxODBC.log. It will be created in the current working directory; messages are always appended to the file so no trace is lost until you explicitly erase the log file. If the log file can not be opened, the module will use `stderr` for reporting.

To obtain a debugging version of mxODBC, please contact *support@egenix.com* for help.

20. History & Changes

Please visit the change log on the *mxODBC product page* for a list of changes in the various product versions.

21. Copyright & License

EGENIX.COM COMMERCIAL LICENSE AGREEMENT

Version 1.3.0

1. Introduction

This "License Agreement" is between eGenix.com Software, Skills and Services GmbH ("eGenix.com"), having an office at Pastor-Loeh-Str. 48, D-40764 Langenfeld, Germany, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Terms and Definitions

The "Software" covered under this License Agreement includes without limitation, all object code, source code, help files, publications, documentation and other programs, products or tools that are included in the official "Software Distribution" available from eGenix.com.

The "Proof of Authorization" for the Software is a written and signed notice from eGenix.com providing evidence of the extent of authorizations the Licensee has acquired to use the Software and of Licensee's eligibility for future upgrade program prices (if announced) and potential special or promotional opportunities. As such, the Proof of Authorization becomes part of this License Agreement.

Installation of the Software ("Installation") refers to the process of unpacking or copying the files included in the Software Distribution to an Installation Target.

"Installation Target" refers to the target of an installation operation. Targets are defined, among other parameters, in terms of the following definitions:

1) "CPU" refers to a central processing unit which is able to store and/or execute the Software (a server, personal computer, virtual machine, or other computer-like device) using at most two (2) processors,
2) "Site" refers to a single site of a company,
3) "Corporate" refers to an unlimited number of sites of the company,
4) "Developer CPU" refers to a single CPU used by at most one (1) developer.

Additional terms may be defined as part of the Proof of Authorization.

When installing the Software on a server CPU for use by other CPUs in a network, Licensee must obtain a License for the server CPU and for all client CPUs attached to the network which will make use of the Software by copying the Software in binary or source form from the server into their CPU memory. If a CPU makes use of more than two (2) processors, Licensee must obtain additional CPU licenses to cover the total number of installed processors. The number of cores per processor does not count towards this license limitation. Virtual machines always count as one (1) CPU. If a Developer CPU is used by more than one developer, Licensee must obtain additional Developer CPU licenses to cover the total number of developers using the CPU.

"Commercial Environment" refers to any application environment which is aimed at directly or indirectly generating profit. This includes, without limitation, for-profit organizations, private educational institutions, work as independent contractor, consultant and other profit generating relationships with organizations or individuals. Governments and related agencies or organizations are also regarded as being Commercial Environments.

"Non-Commercial Environments" are all those application environments which do not directly or indirectly generate profit. Public educational institutions and officially acknowledged private non-profit organizations are regarded as being Non-Commercial Environments in the aforementioned sense.

"Educational Environments" are all those application environments which directly aim at educating children, pupils or students. This includes, without limitation, class room installations and student server installations which are intended to be used by students for educational purposes. Installations aimed at administrational or organizational purposes are not regarded as Educational Environment.

3. License Grant

Subject to the terms and conditions of this License Agreement, eGenix.com hereby grants Licensee a non-exclusive, world-wide license to

1) use the Software to the extent of authorizations Licensee has acquired and
2) distribute, make and install copies to support the level of use authorized, providing Licensee reproduces this License Agreement and any other legends of ownership on each copy, or partial copy, of the Software.

If Licensee acquires this Software as a program upgrade, Licensee's authorization to use the Software from which Licensee upgraded is terminated.

Licensee will ensure that anyone who uses the Software does so only in compliance with the terms of this License Agreement.

Licensee may not

1) use, copy, install, compile, modify, or distribute the Software except as provided in this License Agreement;
2) reverse assemble, reverse engineer, reverse compile, or otherwise translate the Software except as specifically permitted by law without the possibility of contractual waiver; or
3) rent, sublicense or lease the Software.

4. Authorizations

The extent of authorization depends on the ownership of a Proof of Authorization for the Software.

Usage of the Software for any other purpose not explicitly covered by this License Agreement or granted by the Proof of Authorization is not permitted and requires the written prior permission from eGenix.com.

5. Modifications

Software modifications may only be distributed in form of patches to the original files contained in the Software Distribution.

The patches must be accompanied by a legend of origin and ownership and a visible message stating that the patches are not original Software delivered by eGenix.com, nor that eGenix.com can be held liable for possible damages related directly or indirectly to the patches if they are applied to the Software.

6. Experimental Code or Features

The Software may include components containing experimental code or features which may be modified substantially before becoming generally available.

These experimental components or features may not be at the level of performance or compatibility of generally available eGenix.com products. eGenix.com does not guarantee that any of the experimental components or features contained in the eGenix.com will ever be made generally available.

7. Expiration and License Control Devices

Components of the Software may contain disabling or license control devices that will prevent them from being used after the expiration of a period of time or on Installation Targets for which no license was obtained.

Licensee will not tamper with these disabling devices or the components. Licensee will take precautions to avoid any loss of data that might result when the components can no longer be used.

8. NO WARRANTY

eGenix.com is making the Software available to Licensee on an “AS IS” basis. SUBJECT TO ANY STATUTORY WARRANTIES WHICH CAN NOT BE EXCLUDED, EGENIX.COM MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, EGENIX.COM MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

9. LIMITATION OF LIABILITY

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL EGENIX.COM BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR (I) ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE

THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF; OR (II) ANY AMOUNTS IN EXCESS OF THE AGGREGATE AMOUNTS PAID TO EGENIX.COM UNDER THIS LICENSE AGREEMENT DURING THE TWELVE (12) MONTH PERIOD PRECEEDING THE DATE THE CAUSE OF ACTION AROSE.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE EXCLUSION OR LIMITATION MAY NOT APPLY TO LICENSEE.

10. Termination

This License Agreement will automatically terminate upon a material breach of its terms and conditions if not cured within thirty (30) days of written notice by eGenix.com. Upon termination, Licensee shall discontinue use and remove all installed copies of the Software.

11. Indemnification

Licensee hereby agrees to indemnify eGenix.com against and hold harmless eGenix.com from any claims, lawsuits or other losses that arise out of Licensee's breach of any provision of this License Agreement.

12. Third Party Rights

Any software or documentation in source or binary form provided along with the Software that is associated with a separate license agreement is licensed to Licensee under the terms of that license agreement. This License Agreement does not apply to those portions of the Software. Copies of the third party licenses are included in the Software Distribution.

13. High Risk Activities

The Software is not fault-tolerant and is not designed, manufactured or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems, in which the failure of the Software, or any software, tool, process, or service that was developed using the Software, could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities").

Accordingly, eGenix.com specifically disclaims any express or implied warranty of fitness for High Risk Activities.

Licensee agree that eGenix.com will not be liable for any claims or damages arising from the use of the Software, or any software, tool, process, or service that was developed using the Software, in such applications.

14. General

Nothing in this License Agreement affects any statutory rights of consumers that cannot be waived or limited by contract.

Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between eGenix.com and Licensee.

If any provision of this License Agreement shall be unlawful, void, or for any reason unenforceable, such provision shall be modified to the extent necessary to render it enforceable without losing its intent, or, if no such modification is possible, be severed from this License Agreement and shall not affect the validity and enforceability of the remaining provisions of this License Agreement.

This License Agreement shall be governed by and interpreted in all respects by the law of Germany, excluding conflict of law provisions. It shall not be governed by the United Nations Convention on Contracts for International Sale of Goods.

This License Agreement does not grant permission to use eGenix.com trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party.

The controlling language of this License Agreement is English. If Licensee has received a translation into another language, it has been provided for Licensee's convenience only.

15. Agreement

By downloading, copying, installing or otherwise using the Software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

For question regarding this License Agreement, please write to:

eGenix.com Software, Skills and Services GmbH

Pastor-Loeh-Str. 48

D-40764 Langenfeld

Germany

EGENIX.COM PROOF OF AUTHORIZATION

1 CPU License (Example)

This is an example of a "Proof of Authorization" for a 1 CPU License. These proofs are either wet-signed by the eGenix.com staff or digitally PGP-signed using an official eGenix.com PGP-key.

1. License Grant

eGenix.com Software, Skills and Services GmbH ("eGenix.com"), having an office at Pastor-Loeh-Str. 48, D-40764 Langenfeld, Germany, hereby grants the Individual or Organization ("Licensee")

Licensee: <name of the licensee>

a non-exclusive, world-wide license to use the software listed below in source or binary form and its associated documentation ("the Software") under the terms and conditions of this License Agreement and to the extent authorized by this Proof of Authorization.

2. Covered Software

Software Name: <product name>

Software Version: <product version>

(including all patch level releases)

Software Distribution: As officially made available by

eGenix.com on http://www.egenix.com/

Operating System: any compatible operating system

3. Authorizations

eGenix.com hereby authorizes Licensee to copy, install, compile, modify and use the Software on the following Installation Targets under the terms of this License Agreement.

Installation Targets: one (1) CPU

Use of the Software for any other purpose or redistribution IS NOT PERMITTED BY THIS PROOF OF AUTHORIZATION.

4. Proof

This Proof of Authorization was issued by

<name>, <title>

Langenfeld, <date>

Proof of Authorization Key:

<license key>

EGENIX.COM PROOF OF AUTHORIZATION

1 Developer CPU License (Example)

This is an example of a "Proof of Authorization" for a 1 Developer CPU License. These proofs are either wet-signed by the eGenix.com staff or digitally PGP-signed using an official eGenix.com PGP-key.

5. License Grant

eGenix.com Software, Skills and Services GmbH (“eGenix.com”), having an office at Pastor-Loeh-Str. 48, D-40764 Langenfeld, Germany, hereby grants the Individual or Organization (“Licensee”)

Licensee: <name of the licensee>

a non-exclusive, world-wide license to use the software listed below in source or binary form and its associated documentation (“the Software”) under the terms and conditions of this License Agreement and to the extent authorized by this Proof of Authorization.

6. Covered Software

Software Name: <product name>

Software Version: <product version>

(including all patch level releases)

Software Distribution: As officially made available by

eGenix.com on http://www.egenix.com/

Operating System: any compatible operating system

7. Authorizations

7.1 Application Development

eGenix.com hereby authorizes Licensee to copy, install, compile, modify and use the Software on the following Developer Installation Targets for the purpose of developing products using the Software as integral part.

Developer Installation Targets: one (1) Developer CPU

7.2 Redistribution

eGenix.com hereby authorizes Licensee to redistribute the Software bundled with a product developed by Licensee on the Developer Installation Targets ("the Product") subject to the terms and conditions of this License Agreement for installation and use in combination with the Product on the following Redistribution Installation Targets, provided that:

1. Licensee shall not and shall not permit or assist any third party to sell or distribute the Software as a separate product;

2. Licensee shall not and shall not permit any third party to

   i. market, sell or distribute the Software to any end user except subject to the terms and conditions of this License Agreement,

   ii. rent, sell, lease or otherwise transfer the Software or any part thereof or use it for the benefit of any third party,

   iii. use the Software outside the Product or for any other purpose not expressly licensed hereunder;

3. the Product does not provide functions or capabilities similar to those of the Software itself, i.e. the Product does not introduce commercial competition for the Software as sold by eGenix.com;

4. Licensee has obtained Developer CPU Licenses for all developers and CPUs used in developing the Product.

Redistribution Installation Targets:

any number of CPUs capable of running the Product and the Software

8. Proof

This Proof of Authorization was issued by

<name>, <title>

Langenfeld, <date>

Proof of Authorization Key:

<license key>