eGenix.com

mxODBC

ODBC Database Interface for Python

Version 3.1

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 years 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.

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 either 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 directly by linking to the database ODBC drivers.

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. Included are several *preconfigured subpackages* for a wide range of common ODBC drivers and managers to support this.

mxODBC uses the *mxDateTime* package for handling date/time value, eliminating the problems you normally face when handling dates before 1.1.1970 and after 2038. This also makes the package Year 2000/2038 safe.

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 Installation on Windows

The binaries 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.

2.1.1 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 database 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 section 13 *Hints & Links to other Resources*.

2.1.2 Procedure

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 NT, 2000, XP, Vista and 7 to successfully complete the installation.

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.

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.

2.1.3 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.

2.2 Installation on Unix using the RPM packages

On Linux you can use the binary RPM packages provided by eGenix.com to simplify the install process.

These binaries only include the `mx.ODBC.iODBC` and `mx.ODBC.unixODBC` subpackages of mxODBC. These two subpackage interface directly to the iODBC or unixODBC ODBC managers, one of which is usually preinstalled on Linux systems.

You can use the available GUI-configuration helpers for these ODBC managers 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 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.

- *root* access to the target machine

- You will need ODBC drivers for all database you wish to connect to. If you can't find the driver you are looking for have a look at section 13 *Hints & Links to other Resources*.

Download the right .rpm package for your platform and Python version to a temporary directory and execute the standard rpm commands for installation as *root* user:

```
rpm -i egenix-mxodbc-3.0.0-py2.5_1.i386.rpm
```

The RPM files provided by eGenix.com include the subpackages for the iODBC and unixODBC ODBC managers.

If the install command complains about unresolved dependencies, it is likely that you only have one of these two ODBC managers installed. In this case, install the package by overriding the dependency checks done by the `rpm` command:

```
rpm -i --nodeps egenix-mxodbc-3.0.0-py2.5_1.i386.rpm
```

This will install the package even though only one of the subpackages `mx.ODBC.iODBC` or `mx.ODBC.unixODBC` will actually work. You get an `ImportError` for the subpackage which has the unresolved dependencies.

2.2.2 Uninstall

To uninstall the distribution, run the RPM uninstall command:

```
rpm -e egenix-mxodbc
```

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.

2.3 Installing from Source

This section describes installation of mxODBC from source. This is usually only necessary on platforms for which eGenix.com does not provide binary packages or if you have special needs.

The procedure is the same for Windows and Unix.

2.3.1 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 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.

- Also make sure that you have the Python development files installed on your system: these are usually located in /usr/local/lib/pythonX.X/config/ or /usr/lib/pythonX.X/config/. If you don't, look for a `python-devel` or similar package for your OS distribution and install this first.

- You will need ODBC drivers for all database you wish to connect to. If you can't find the driver you are looking for have a look at section 13 *Hints & Links to other Resources*.

- A C compiler is required.

  GNU CC (*gcc*) will do in most cases on Unix, but it is advised to use the same C compiler and linker that you have used to compile Python itself on the target platform. The installation process will automatically choose the compiler depending on the Python installation for you.

  On Windows, *Visual C++* is preferred, but *cygwin* or *mingw32* should also work.

2.3.2 Procedure

To install mxODBC on a non-Linux Unix platform such as Sun Solaris, AIX or BSD, download the source distribution of the `egenix-mxodbc` distribution, e.g. egenix-mxodbc-3.0.0.zip.

Unzip this source archive to a temporary directory and change into the distribution directory (e.g. egenix-mxodbc-3.0.0/).

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

To install the `egenix-mxodbc` distribution, run the setup.py script with the Python version you want to install the packages to, e.g.

```
python setup.py install
```

This automatically scans the system for available ODBC drivers and managers and try to build all subpackages for which it finds the right libraries and header files.

If the automatic setup does not find the ODBC you have installed on the system, please use your favorite Python editor and edit the file mxODBC.py according to the instructions given in that file. You should normally only have to adapt the paths given in the different subpackages setup sections to your actual ODBC driver install paths.

After successful installation, try to import all the subpackages that were found. If you get an error like 'unresolved symbol: SQLxxx', try to add a

```
define_macros=[..., ('DONT_HAVE_SQLDescribeParam', 1),],
```

to the setup lines in mxODBC.py and reinstall.

2.3.3 Reinstall

You can rerun the above install command as often as you like until the compile and install process works as configured.

However, if you change a configuration setting, please make sure that the build/ subdirectory of the distribution directory is being removed prior to executing the install. Otherwise, the installation process could pick up files from a previous run and not use the modified settings.

There also is a command to automate this:

```
python setup.py clean
```

2.3.4 Uninstall

To uninstall the distribution, run the uninstall setup command:

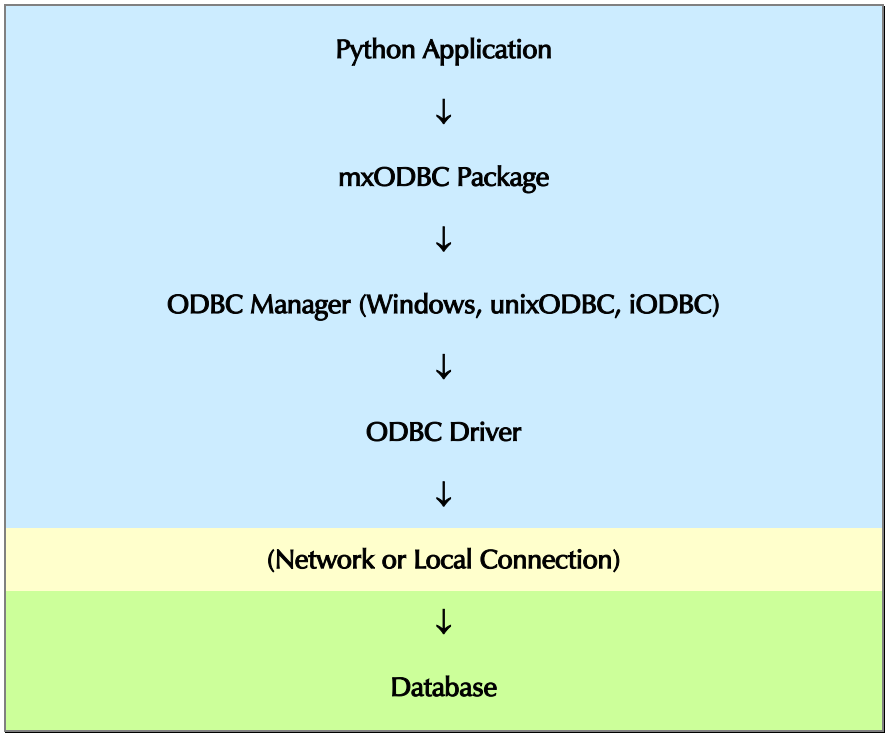
```
python setup.py uninstall
```

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.

3. Access Databases using mxODBC

mxODBC provides a way of accessing the ODBC API of ODBC managers and drivers. In order to connect to a database you still need to have suitable ODBC drivers installed on the machine where you are running the Python application.

The typical ODBC 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.

As a result of this setup, 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.

3.1 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, we recommend setting up System Data Sources, as these are usually accessible by all accounts on a Windows machine.

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

3.1.1 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].

Last time we checked, it included 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 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.

We have collected some information 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. iODBC or unixODBC.

3.2.1 MS SQL Server

Available solutions:

EasySoft's ODBC-ODBC bridge

http://www.easysoft.com/

This was tested with mxODBC and is supported by EasySoft.

It is recommended to use the bridge with unixODBC. Starting with version 1.4.2 of the bridge, there is full functional Unicode support available if the target ODBC driver supports this (the latest MS SQL Server and MS Access drivers do).

MDBTools ODBC driver

http://forums.devshed.com/archive/46/2002/06/4/37357

http://mdbtools.sourceforge.net/

Not tested with mxODBC. Unsupported by eGenix.

ODBC Socket Server

http://odbcsock.sourceforge.net/

Not tested with mxODBC. Unsupported by eGenix.

For a version of the Socket Server with transaction support, you can try:

UniverSQL:

http://www.sidespace.com/products/universql/

Not tested with mxODBC. Unsupported by eGenix.

DataDirect and *Merant* also provide driver sets for connecting to MS SQL Server from Unix.

3.2.2 Oracle

Available solutions:

EasySoft's Oracle Driver for Unix

http://www.easysoft.com/

This was tested with mxODBC and is supported by EasySoft.

DataDirect and *Merant* also provide driver sets for connecting to Oracle from Unix.

3.2.3 IBM DB2

Linux client to Unix/Windows server

IBM DB2 for Linux ships with ODBC drivers for DB2. These can also be used to connect to DB2 database over a network.

Linux client to iSeries / AS/400 server

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-1.ibm.com/servers/eserver/iseries/linux/odbc/

3.2.4 Looking for Unix ODBC drivers ?

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 *DataDirect* (formerly Intersolv/Merant). They have 30-day evaluation packages available.

Another source for commercial ODBC drivers is *OpenLink*. To see if they support your client/server setup check this *matrix*. They are giving away 2-client/10-connect licenses for free.

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

If you would like to connect to a database for which you don't have a Unix ODBC driver, you can also try the ODBC-ODBC bridge from *EasySoft* which redirects the queries to e.g. the NT ODBC driver for the database.

Alternatively, you can have a look at our *mxODBC Connect product* 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.

4. 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 either the `mx.ODBC.iODBC` or `mx.ODBC.unixODBC` package depending on which of these two standard 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 DB-API as possible. Some things cannot easily be mapped onto ODBC, so there are a few deviations from the standard. Section 4.1 *mxODBC and the Python Database API Specification* explains these in more detail.

4.1 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.

Here is a list of 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 available through ODBC interfaces and the values are not commonly used in applications.

- `cursor.callproc()` is only implemented for input parameters for reasons explained in section 4.6 *Stored Procedures*.

- `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.

mxODBC extends the DB-API specification in a number of ways. 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*.

4.2 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 - 3.52 compliant ODBC database driver/manager. mxODBC prefers ODBC 3.x over 2.x in case the driver/manager supports both versions of the standard.

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 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 have not copied the complete ODBC documentation to this page.

You can browse and download the MS ODBC reference from the *Microsoft MDAC web-site*.

4.3 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.

4.3.1 ODBC Managers

All supported ODBC managers (MS ODBC Manager, iODBC and unixODBC) 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 version number, e.g. `'03.51.00'`.

4.3.2 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.

4.4 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.

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

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.

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.

Note:
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 thread"` to find out. If you get an exception, thread support is not available.

4.5 Transaction Support

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.

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.

New transactions are started in the following cases:

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

4.6 Stored Procedures

There are two ways to call a stored procedure in mxODBC, directly using the `.callproc()` cursor method or 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]]...)]}
```

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

```
cursor.execute("{call myprocedure(?,?)}", (1,2))
```

will call the stored procedure `myprocedure` with the input parameters `1, 2`.

After calling `.callproc()` or `.execute*()`, you can then access output from the stored procedure as one or more result sets using the standard `.fetch*()` cursor methods. If the stored procedure has generate multiple result sets, skipping to the next result set is possible by calling the `.nextset()` cursor method.

4.6.1 Input/Output and Output Parameters

mxODBC does not support input/output or output parameters in stored procedures. The reason for this is that the interface for passing back data from the stored procedure requires knowledge of the data size before calling the procedure which is often impossible to deduce (e.g. for string data).

Passing back such data in form of one or more result sets gives you a much better alternative which also let's you implement variable length output parameter lists and special output value conversions.

4.6.2 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.

4.7 Debugging

To simplify debugging the mxODBC package can generate debugging output in several important places. The feature is only enabled if the module is compiled with debug support and output is only generated if Python is run in debugging mode (use the Python interpreter flag: `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 have the package compiled using debug support, prepend the distutils command `mx_autoconf --enable-debugging` to the `build` or `install` command. This will then enable the define and compile a debugging version of the code, e.g.

```
cd egenix-mx-commcercial-X.X.X
python setup.py mx_autoconf --enable-debugging install
```

installs a debugging enabled version of mxODBC on both Unix and Windows (provided you have a compiler installed).

Note that the debug version of the module is almost as fast as the regular build, so you might as well leave debugging support enabled.

5. 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.

5.1.1 Same Interface for all Subpackages

Even though mxODBC provides interfaces to many different ODBC backends using different subpackages, each of these subpackages use the same names and signatures, thereby making applications very portable between ODBC backends.

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 the `mx.ODBC.iODBC` subpackage and the constructor then becomes `mx.ODBC.iODBC.DriverConnect()`.

Please note that some ODBC backends do not support all available ODBC features. As a result, not all constructors and methods may be available. The best way to work around this caveat is to always rely on ODBC managers to achieve the best portability and also to simplify the configuration of the application's database needs.

5.1.2 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`.

5.2 Connection Object Constructors

```
Connect(dsn, user='', password='', clear_auto_commit=1,
   errorhandler=None)
```

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.

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 *Default Transaction Settings* for details on `clear_auto_commit`.

```
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.

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 *Default Transaction Settings* for details on `clear_auto_commit`.

> The `DriverConnect()` API is only available if the interface was compiled with the compile time switch `HAVE_SQLDriverConnect` defined. This is the default for ODBC managers such as the one on Windows and iODBC/unixODBC 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.

5.2.1 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.

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.

Note that a compile time switch (DONT_CLEAR_AUTOCOMMIT) allows altering the default value for `clear_auto_commit`.

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.

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.

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.

5.3 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])`

Constructs a new *Cursor Object* with the given name using the connection.

If no name is given, the ODBC driver will determine a unique name on its own. You can query this name with `cursor.getcursorname()` (see the *Cursor Object* section 6).

`.getconnectoption(option)`

Same as above, except that the corresponding ODBC function is `SQLGetConnectOption()`.

`option` must be an integer. Suitable option values are available through the `SQL` object (see the *Constants* section 10 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.

`.getinfo(info_id)`

Interface to the corresponding ODBC function `SQLGetInfo()`.

The `info_id` must be an integer. Suitable values are available through the `SQL` object (see the *Constants* section 10 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 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 database subpackage was compiled with transaction support. For ODBC manager subpackages it 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 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.

5.4 Connection Object Attributes

`.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 for details).

`.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.

`.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 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).

`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.

`.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 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.

Note that changing the values 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'`.

`.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 the *Error Handlers* section 9 for details.

`.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.

`.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 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 connection's encoding setting (see the `.encoding` attribute of connection objects) 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 connection's encoding setting (see the `.encoding` attribute of connection objects).

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 connection's encoding setting (see the `.encoding` attribute of connection objects) 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 connection's encoding setting (see the `.encoding`

attribute of connection objects) 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.py 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 or some form of binary objects to store binary data in some future version, e.g. the new bytes type which will be introduced with Python 3.0.

This variable only has an effect if mxODBC was compiled with Unicode support (default for Python 2.0 and later). If not, mxODBC will always work in `EIGHTBIT_STRINGFORMAT` mode.

5.4.1 Additional 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 9 for details and names of these error attributes.

6. 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.

6.1.1 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.

6.1.2 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.

6.1.3 Same Interface for all Subpackages

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.

6.1.4 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`.

6.2 Cursor Object Constructors

Cursor objects are created using the connection method `connection.cursor()`. Please see 5.3 *Connection Object Methods* for details.

6.3 Cursor Object Methods

The following cursor methods are defined in the DB API:

`.callproc(procname[, parameters])`

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 copy of the input sequence. Input parameters are left untouched, output and input/output parameters replaced with possibly new values.

The procedure may also provide a result set as output. This must then be made available through the standard `fetch*()` methods.

> This method is only implemented for input parameters in mxODBC for reasons explained in 4.6 *Stored Procedures*. Future versions of mxODBC may also support in/out and output parameters.

`.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])`

Prepare and execute a database operation (query or command).

Parameters must be provided as sequence and will be bound to variables found in the `sqlcmd` string on a positional basis.

Variables in the `sqlcmd` string are specified using the ODBC variable 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.

A reference to the `sqlcmd` string will be retained by the cursor. If the same `sqlcmd` string 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 of parameters in one call, e.g. to insert multiple rows in a single call.

`sqlcmd` may be a Unicode object in case the ODBC driver and/or database support this.

`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.

Return values are not defined.

`.executedirect(sqlcmd[,parameters])`

Works just like `.execute()`, except that no prepare step is issued and the `sqlcmd` is not cached. 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. 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.

Return values are not defined.

`.executemany(sqlcmd,batch[,direct])`

Prepare a database operation (query or command) and then execute it against all parameter sequences found in the sequence `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.

Return values are not defined.

`.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.

`.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.

If you plan to write cross database applications, use these methods with care, since at least some of the databases don't support certain APIs or return misleading results.

Warning:
Be sure to check the correct performance of the methods and executes.

> We don't want to see you losing your data due to some error we made, or the fact that the ODBC driver of your database is buggy or not fully ODBC-compliant.

`.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 7 for details on how user-defined converters work.

`.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.
	
	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

Option	*Comment*
	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 execute a prepared statement, pass `cursor.command` to one of the `.execute*()` methods.

`sqlcmd` may be a Unicode object in case the ODBC driver and/or database support this.

Return values are not defined.

> This method is unavailable if mxODBC was compiled with compile time switch `DISABLE_EXECUTE_CACHE`.

`.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 datasource 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.py 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.

`.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. Possible values: Any positive integer or SQL.MAX_ROWS_DEFAULT (no limit)
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

Option	*Comment*

	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).

6.3.1 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.

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`[1]. All catalog methods support keywords and use the indicated default values for parameters which are omitted in the call.

[1] Note that this was changed in mxODBC 3.0. Previously the catalog methods used to return the number of rows in the result set.

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).

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 omissions or additions.

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.

The available catalog methods are:

`.columns(qualifier=None, owner=None, table=None, column=None)`

Catalog method which generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEM	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

Column Name	*Column Datatype*	*Comment*
		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_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.

Column Name	*Column Datatype*	*Comment*
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.
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

Column Name	*Column Datatype*	*Comment*
		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)
```

Catalog method which generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEM	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", "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)
```

Catalog method which generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
PKTABLE_CAT	VARCHAR(128)	Always NULL.
PKTABLE_SCHEM	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_SCHEM	VARCHAR(128)	The name of the schema containing FKTABLE_NAME.
FKTABLE_NAME	VARCHAR(128) not NULL	Name of the table containing the primary key.
FKCOLUMN_NAME	VARCHAR(128) not NULL	Primary 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

Column Name	*Column Datatype*	*Comment*
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(sqltypecode)`

Catalog method which 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".
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

Column Name	*Column Datatype*	*Comment*
		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.
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

Column Name	*Column Datatype*	*Comment*
		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 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

Column Name	*Column Datatype*	*Comment*
		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)
```

Catalog method which generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.

Column Name	*Column Datatype*	*Comment*
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)
```

Catalog method which generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
PROCEDURE_CAT	VARCHAR(128)	Always NULL.
PROCEDURE_SCHEM	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_RESULT_SETS	INTEGER not NULL	Number of result sets returned by the 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

Column Name	*Column Datatype*	*Comment*
		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)
```

Catalog method which generates a result set having the following schema (the term "column" refers to the procedure's call parameters):

Column Name	*Column Datatype*	*Comment*
PROCEDURE_CAT	VARCHAR(128)	Always NULL.
PROCEDURE_ SCHEM	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_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.

Column Name	*Column Datatype*	*Comment*
		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.
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,

Column Name	*Column Datatype*	*Comment*
		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	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 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)
```

Catalog method which 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.

Column Name	*Column Datatype*	*Comment*
		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.
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.ENSURE)
```

Catalog method which generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEM	VARCHAR(128)	The name of the schema containing TABLE_NAME.

Column Name	*Column Datatype*	*Comment*
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 Name	*Column Datatype*	*Comment*
		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.
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)
```

Catalog method which 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.
TABLE_SCHEM	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)
```

Catalog method which generates a result set having the following schema:

Column Name	*Column Datatype*	*Comment*
TABLE_CAT	VARCHAR(128)	Always NULL.
TABLE_SCHEM	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".

Column Name	*Column Datatype*	*Comment*
IS_GRANTABLE	VARCHAR(3)	Indicates whether the grantee is permitted to grant the privilege to other users. This can be "YES", "NO" or NULL.

6.4 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 observes this value with respect to the `.fetchmany()` method, but currently interacts with the database a single row at a time.

`.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.

`.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.

`.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 date/time format.

`.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 decimal format.

`.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 7 *Type Objects and Constructors* below. mxODBC returns the SQL type integers in this field. These are described in the section 7 *Supported Data Types* and are available through the `SQL` singleton defined at module level.

`.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 the *Error Handlers* section 9 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.

`.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.

You should check whether the database you are interfacing to supports `.rownumber` 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 `.rownumber` will always be `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 `stringformat`.

7. 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.py 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.

7.1 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:

Binding Mode	*Value of* `connection.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

Binding Mode	*Value of* `connection.bindmethod`	*Comments*
		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 settings with which the ODBC subpackage was compiled. If not indicated in the *Subpackages* section 12, it is set to SQL type binding mode (`BIND_USING_SQLTYPE`), since this offers more flexibility than Python type binding.

Note:
For SQL type binding to be usable, mxODBC needs a working ODBC `SQLDescribeParam()` API implementation. This is checked at connect time and the binding style adjusted to Python type binding, if mxODBC cannot rely on `SQLDescribeParam()`. Unfortunately, not all database ODBC drivers tell the truth about the capabilities of their `SQLDescribeParam()` implementation, so workarounds are in place for a few known cases. If you find more such cases, please contact *support* for help.

7.2 SQL Type Input Binding

The following data types are used for SQL type input binding mode (`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.py 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 connection's character `.encoding` setting. In all other modes, Unicode is converted to an 8-bit string before passing it to the ODBC driver using the connection's character `.encoding` setting. Strings are passed as-is.
`SQL.WCHAR`, `SQL.WVARCHAR`, `SQL.WLONGVARCHAR` (TEXT, BLOB or LONG in SQL)	String or Unicode or stringified object	The conversion truncates the string at the SQL field length. Note that currently only very few ODBC drivers can handle native Unicode. The MS Access and SQL Server ODBC drivers are the only ones we have successfully tested. The only way to store Unicode data in a non-Unicode aware database is by encoding it using e.g. UTF-8. The handling of Unicode depends on the setting of the `.stringformat` attribute. In `EIGHTBIT_STRINGFORMAT`

SQL Type	*Python Type*	*Comments*
		and `UNICODE_STRINGFORMAT` mode, Unicode is converted to an 8-bit string before passing it to the ODBC driver using the connection's character `.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 connection's character `.encoding` setting.
`SQL.BINARY`, `SQL.VARBINARY`, `SQL.LONGVARBINARY` (BLOB or LONG BYTE in SQL)	Buffer 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.
`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 via string representation since there is no C type with enough precision to hold the value. Because of this, you might 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

SQL Type	*Python Type*	*Comments*
		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 or a ticks value as Python number	While you should use DateTime instances, the module also excepts 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 or a tocks value as Python number	While you should use DateTimeDelta instances, the module also excepts Python datetime.time instances, tocks (Python numbers indicating the number of seconds since 0:00:00.00) and tuples `(hour,minute,second)` on input.
`SQL.TIMESTAMP`	DateTime instance or datetime.datetime instance or a tuple `(year,month,day, hour,minute,second)` or String or a ticks value as Python number	While you should use DateTime instances, the module also excepts Python datetime.datetime instances, 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.

SQL Type	*Python Type*	*Comments*
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).

7.3 Python Type Input Binding

The following mappings are used for input variables in Python type input binding mode (`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. The long variants are used for strings longer than 256 characters.
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

Python Type	*SQL Type*	*Comments*
		driver/manager. The long variant is used for Unicode data longer than 256 code points.
Buffer	`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) 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

Python Type	*SQL Type*	*Comments*
		SQL column type (DATE or TIMESTAMP).
DateTimeDelta	`SQL.TIME`	Converts the DateTimeDelta instance into a TIME struct defined by the ODBC standard. 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.
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.
`None`	Any nullable column	The Python None singleton is converted to the special SQL `NULL` value.
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.

7.4 Output Conversions

The following data types are used per default for output variable mapping and for SQL type input binding mode (`connection.bindmethod` set to `BIND_USING_SQLTYPE`):

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. Unicode is only available in case mxODBC was compiled with Unicode support. 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	Conversion from the database type to Python is done via a string.
`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 conversion. In DECIMAL_DECIMALFORMAT mode,

SQL Type	*Python Type*	*Comments*
		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`.

7.5 Output Type Converter Functions

The last section defined the standard mapping mxODBC applies when fetching output data from the database.

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 7.2 *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).

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.

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)
```

7.6 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.

7.7 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 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.

7.8 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.py test script.

8. Supported 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 and buffer 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 7 *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.

9. 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.

9.1 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
```

If you are interested in the exact mapping of SQL error codes to exception classes, have a look at the `mxODBC_ErrorCodeTranslations` array defined in the mxODBC source code (mxODBC.c) or inspect the `errorclass` dictionary which is defined at subpackage scope.

If you need to specify your own SQLSTATE to exception mappings, you can assign to the `errorclass` dictionary. It maps SQLSTATE strings to error classes and is used by mxODBC internally.

9.2 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. If you want to mask `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) 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))
```

```
    # Don't raise exceptions for Warnings
    if connection is None or \
       not issubclass(errorclass, connection.Warning):
        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)
```

9.3 Exception Value Format

All ODBC driver generated exceptions use a standard exception value layout.

The 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.c source code which generated the message. This is very useful for support purposes.

9.4 Error Handlers

If you want to provide your own error handler, e.g. to mask database warnings, 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.

An error handler has to 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.

The default 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`.

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.

9.4.1 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. 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.

10.1 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.

10.2 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.

11. mxODBC Globals and Constants

11.1 Subpackage Globals and Constants

Each mxODBC subpackage exports the following constants:

`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`.

`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.

`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.

```
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.

`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.

`apilevel`

String constant stating the supported DB API level. This is set to `'2.0'`, since mxODBC supports nearly all features of the DB API 2.0 standard. Many DB API 1.0 features are still supported too for backward compatibility.

`paramstyle`

String constant stating the type of parameter marker formatting expected by the interface. This is set to `'qmark'`, since ODBC interfaces always expect `'?'` 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.

`BIND_USING_SQLTYPE, BIND_USING_PYTHONTYPE`

Integer values returned by `connection.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 is usually set at compilation time, but can also differ from database to database when accessing them via an ODBC manager.

`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.

`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 available in Python 2.0 and above and enabled per default.

`license`

String with the license information for the installed mxODBC license.

11.2 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 9. mxODBC Exceptions and Error Handling for details.

12. mx.ODBC Subpackages

This section includes specific notes for preconfigured subpackages and setups.

12.1 Subpackage Notes

The following sections provide hints that apply to all mx.ODBC subpackages. Please read carefully.

12.1.1 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.

12.1.2 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 or unixODBC 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.

12.1.3 Compiling from Source

When compiling the package from source, you should always check the paths and filenames used in the corresponding section of the distribution's

mxODBC.py file because these depend on your specific ODBC driver/manager installations.

You may also want to consult Paul Boddie's *mxODBC Configuration* page which has some details about specific database backends he has used with mxODBC.

12.2 mx.ODBC.Windows -- *Windows ODBC Driver Manager*

Tested with Windows 95 - XP

mxODBC compiles on Windows using VC++ and links against the Windows ODBC driver manager. The necessary import libs and header files are included in the VC++ package but are also available for free in the *Microsoft ODBC SDK (now called MDAC SDK)*. Note that the latter is usually more up-to-date.

12.2.1 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.

12.2.2 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 SQLDescribeParam() API. **MS Access** is one candidate for which this API is not useable.

12.2.3 Issues with MS SQL Server

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.

It seems that you'll have to 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)
```

(thanks to Damien Morton for tracking this down and digging up the MS KB article).

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*.

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.

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.

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.

12.2.4 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).

12.3 mx.ODBC.iODBC -- *iODBC Driver Manager*

Tested with iODBC 3.52.5

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.

Notes:

- 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.

- Note: 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.

- Hint: You may experience problems when trying to connect to MySQL via MyODBC hooked to iODBC in case you are using the binary RPMs available from *www.mysql.com*. For some reason, the MyODBC driver does not reference the MySQL shared libs it needs to connect to the MySQL server and there's no way to tell iODBC to load two shared libs. Here's a hack 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
```

Notes regarding 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.

12.4 mx.ODBC.unixODBC -- *unixODBC Driver Manager*

Tested with unixODBC 2.2.12

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.

Notes:

- 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.

- Even though unixODBC does support Unicode data to some extent, there are bugs in unixODBC <= 2.0.7 which will result in random core dumps due to memory being overwritten during the conversion of 8-bit to Unicode data. Later versions of unixODBC may not have this problem, so you should check by running the `mx.ODBC.Misc.test` script in continuous mode against your version.

Notes regarding 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.

- 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.

- 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.

12.5 ODBC Driver Subpackages

The following subpackages are specific to certain ODBC drivers against which mxODBC can be linked directly.

If possible, please use one of the above ODBC managers for connecting to databases, since these often reduce the number of problems you can run into when setting up an ODBC connection.

> Note binary distributions of mxODBC usually don't contain any additional subpackages for specific drivers. Only the ODBC manager packages for the platform are included. However, driver specific subpackages can be made available on request. Please write to *support@egenix.com* for details.

12.5.1 mx.ODBC.Adabas -- *SuSE Adabas D*

Tested with Adabas 6.1.1

The SuSE Linux distribution ships with a free personal edition of Adabas (available in form of *RPMs* from *SuSE*). A commercial version is also available, but we suggest first trying the personal edition.

If you want to trim down the interface module size, try linking against a shared version of the static ODBC driver libraries. You can create a pseudo-shared lib by telling the linker to wrap the static ones into a single shared one:

```
ld -shared --whole-archive odbclib.a libsqlrte.a libsqlptc.a \
   -lncurses -o /usr/local/lib/libadabasodbc.so
```

Note:
The ADABAS ODBC driver returns microseconds in the timestamp fraction field. Because of this the Setup includes a define to do the conversion to seconds using a microseconds scale instead of the ODBC standard nanosecond scale.

Since the ODBC driver for Adabas on Linux also provides the `DriverConnect()` API it is also exposed by the package (even though the driver itself is not an ODBC manager).

12.5.2 mx.ODBC.DB2 -- *IBM DB2 Universal Database*

Tested with IBM DB2 9.1

IBM provides a free express edition of the powerful DB2 database engine which happens to use ODBC as native C level interface.

This package interfaces directly to the ODBC driver included in the Unix edition of the database. If you want to access DB2 from Windows NT, you can use the `Windows` subpackage of mxODBC.

There is one catch you should watch out for: 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))
```

Unfortunately, the provided shell scripts are sometimes buggy, so simply sourcing them won't do any good; you will have to carefully create your own. A typical problem is that the scripts set `LIBPATH` or `LD_LIBRARY_PATH` which then causes the following traceback when trying to load mxODBC:

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

Also note that 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. Please consult the IBM DB2 documentation for details.

If you want to use the `DriverConnect()` API, 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.

12.5.3 mx.ODBC.DBMaker -- *CASEMaker's DBMaker Database*

Tested with DBMaker 3.71

DBMaker has a small lean and mean SQL database that comes with an ODBC driver. This subpackage interfaces directly to that ODBC driver.

Note:
DBMaker's ODBC driver doesn't have all the advertised SQL catalog functions (the privilege functions are missing) and also doesn't support the `.nativesql()` method. It does provide a `DriverConnect()` API, which might be useful for connecting to databases across a network.

mxODBC currently does not support the use of file object for in- and output of large objects. This may change in a future version though (the needed techniques are already in place).

The subpackage links against the DBMaker driver without problems. Testing has only been preliminary though, but since even CASEMaker advertises mxODBC as Python interface for DBMaker, you should be on the safe side.

12.5.4 mx.ODBC.EasySoft -- *EasySoft ODBC-ODBC Bridge*

Tested with ODBC-ODBC bridge 1.4.x

EasySoft has developed an ODBC-ODBC bridge which allows you to connect to any remote ODBC driver, e.g. you can access MS SQL Server running on an NT box from your Linux web-server.

You can download it via the *product page* or via *FTP* as trial version. The personal editions are said to available for free in the near future. Remember to download setups for client (Linux in the example) and server (NT in the example).

It is recommended to use the bridge with unixODBC. Starting with version 1.4.2 of the bridge, there is full functional Unicode support available if the target ODBC driver supports this (the latest MS SQL Server and MS Access drivers do).

12.5.5 mx.ODBC.FreeTDS -- *FreeTDS ODBC Driver for MS SQL Server and Sybase ASA*

Tested with FreeTDS 0.82 compiled against unixODBC

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.

mxODBC compiles against the current FreeTDS release available from the *www.FreeTDS.org* site, but since the driver is not fully developed yet, only very basic operations are supported.

Notes:

- The FreeTDS website mentions that the driver has some restrictions. There are other ODBC drivers available from commercial vendors which implement the full ODBC3 API, e.g. from OpenLink and Data-Direct, which work well with mxODBC.

- Here is a sample setup for FreeTDS on Linux talking to MS SQL Server 2000 on Windows. In the test we used a Linux box talking to a Windows machine over a network:

  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):

  ```
  [mssql]
  Driver = /usr/local/freetds/lib/libtdsodbc.so
  Description = MS SQL Server 2000 running on Monet
  Trace = No
  Servername = MONET
  Database = tempdb
  ```

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

12.5.6 mx.ODBC.Informix -- *Informix SQL Server*

Unsupported contribution

Informix for Unix doesn't come with Unix ODBC drivers, but there a few source for these: Informix sells the driver under the term "Informix CLI"; Data-Direct and OpenLink also support Informix through their driver suites.

Note: There is also a free *Informix SDK* available for a few commercial Unix platforms like HP-UX and Solaris. It includes the needed ODBC libs and header files (named infxcli.h and libifsql.a).

Once you have installed the ODBC drivers following the vendor's instructions, enable the appropriate set of directives in `Setup`, run `make -f Makefile.pre.in boot` in the `Informix/` subdirectory, and then run `make` to finish the compilation.

To use the OpenLink driver setup instead copy `Setup.in` to `Setup` and enable the OpenLink section in `Setup` before compiling.

Gilles Lenfant emailed us these instructions which you might find useful in setting up the Informix subpackage:

> In addition to the change to Setup (or Setup.in) file edition before the "make -f Makefile.pre.in boot", I made it compile and run with the following changes in the Informix section of the "Setup" file (according to the latest "Informix ODBC Driver Programmer's Manual").
>
> Note that this book must be read carefully for the setup of the Informix related environment variables: The user must have $INFORMIXDIR (informix client software root) set. and his LD_LIBRARY_PATH must include "$INFORMIXDIR/lib:$INFORMIXDIR/lib/esql:$INFORMIXDIR/lib/cli"
>
> Compiling mxODBC requires Informix client SDK (compile time free download from intraware.com) and ESQL/C libraries (client run-time libraries provided with the server CD - not free).
>
> In addition, fixes and tuning must be done in $INFORMIXDIR/etc/odbcinst.ini, and the user must configure his data sources in $HOME/.odbc.ini or $ODBCINI file.

12.5.7 mx.ODBC.MySQL -- *MySQL + MyODBC*

Tested with MyODBC 2.50.36 and MySQL Connector ODBC 3.51.20

MySQL is a SQL database for Unix and Windows platforms developed by *TCX*. It is free for most types of usage (see their FAQ for details) and offers good performance and stability. To download MySQL and the ODBC driver MyODBC, check the *www.mysql.com* website.

There is one particularity with the ODBC driver for MySQL: all input parameters are being processed as string -- even integers and floats. The ODBC driver implements the necessary conversions. mxODBC uses the

Python Type binding method to bind the input parameters; see the *Python Type Input Binding* section 7.3 for more details.

Depending on your MySQL setup (whether you use a transactional 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. The subpackage uses auto-commit mode as default. You can turn this workaround off by editing the distribution's setup.py and removing the switch DONT_CLEAR_AUTOCOMMIT e.g. if you are using a MySQL 4.x or 5.x version with transactions enabled.

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

Important:
Some MySQL + MyODBC setups we have tested showed some serious memory leaks on Linux machines; please check your setup using the included mx/ODBC/Misc/test.py script for leaks. There are no known leaks in mxODBC itself.

12.5.8 mx.ODBC.Oracle -- *Oracle*

Unsupported contribution

Oracle for Unix doesn't ship with Unix ODBC drivers. You can get them from Data-Direct or OpenLink.

Once you have installed the ODBC drivers following the vendor's instructions, run `make -f Makefile.pre.in boot` in the `Oracle/` subdirectory, enable the appropriate set of directives in `Setup` and then run `make` to finish the compilation.

Using Data-Direct drivers is reported to work. Shawn Dyer (irin.com) has kindly provided the setup for this combination and some additional notes:

```
...we also set the following environment variables:
LD_LIBRARY_PATH= both the oracle lib path and the Data-Direct
library path
ODBCINI= the odbc.ini file in the Data-Direct install
```

Once you talk to the Data-Direct ODBC driver, it seems to be a simple matter of setting up the ODBC data source name in their .ini file that has that stuff. At that point you can talk to any of their ODBC drivers you have installed.

To use the OpenLink driver setup instead, copy `Setup.in` to `Setup` and enable the OpenLink section in `Setup` before compiling.

12.5.9 mx.ODBC.PostgreSQL -- *PostgreSQL*

Tested with PostgreSQL 8.2.3 and the psqlodbc driver 08.02.0200.

PostgreSQL is a free SQL database for Unix. This subpackage is for linking directly against the PostgreSQL ODBC driver on Unix. An alternative setup would be connecting to the database via one of the Unix ODBC managers iODBC or unixODBC also supported by mxODBC.

Because of deficiencies in the PostgreSQL ODBC drivers, the package operates in Python type binding mode. The ODBC driver still has serious problems with data conversion, e.g. it doesn't properly quote binary data and has problems dealing with large objects (BLOBs). It does support smaller strings, numbers and date/time values.

Note: Connecting to a PostgreSQL database from Windows using the PostgreSQL Windows ODBC Driver through the Windows ODBC Manager was reported to work fine.

12.5.10 mx.ODBC.SAPDB -- *SAP DB*

Tested with SAP DB 7.4

SAP DB is a free Open Source, high-performance and fully SQL-92 compliant database made available by SAP which originated from a branch of Adabas D. There are binary versions available for all major platforms and all come with ODBC drivers.

This subpackage is preconfigured to link against the Linux version of the ODBC driver.

Since the ODBC driver for SAP DB also provides the `DriverConnect()` API it is also exposed by the package (even though the driver itself is not an ODBC manager).

> *MaxDB*, the successor to SAPDB is also support by mxODBC. You can use the mx.ODBC.SAPDB package to connect to the MaxDB ODBC driver or one of the ODBC manager packages available in mxODBC.

12.5.11 mx.ODBC.Solid -- *Solid Server*

Tested with Solid Embedded Engine SDK 3.52

Solid Tech. offers an embedded database engine for many different platforms. More information about prices, licenses and downloads is available on their *website*.

The Solid Server's low-level database API uses ODBC as interface standard (most other vendors have proprietary interfaces), so mxODBC should deliver the best performance possible.

Note that Unicode with Solid only works if you link directly against the driver. Accessing Solid through an ODBC manager such as unixODBC or iODBC currently mangles the data because Solid uses 4 bytes to store a character while the ODBC managers use 2.

The Solid ODBC driver only supports scrolling in increments of 1. mxODBC tries to emulate most other modes, but not all work, e.g. you can't scroll out of the result and then back in again.

12.5.12 mx.ODBC.SybaseASA -- *Sybase Adaptive Server Anywhere*

Unsupported contribution

Sybase Adaptive Server Anywhere comes with its own ODBC driver against which mxODBC can link directly. The included Setup is for version 7 of the server.

In case you are running Linux, Sybase has *some information* on its web-site about the *ASA ODBC driver* and its setup on Linux. This *whitepaper* should also be of interest.

You can also use the OpenLink drivers for Sybase ASA: copy `Setup.in` to `Setup` and enable the OpenLink section in `Setup` before compiling.

In any case, you should also consult Paul Boddie's *mxODBC Configuration* page for the Sybase Adaptive Server Anywhere. It includes valuable information about the setup.

Thanks to Paul Boddie and Sam Rushing for helping in getting the package together.

12.5.13 mx.ODBC.SybaseASE -- *Sybase Adaptive Server Enterprise*

Unsupported contribution

Note that you will first have to get the Sybase ASE ODBC drivers from Data-Direct (formerly Merant) in order to use this subpackage -- Sybase ASE does not include ODBC drivers (it's a completely different product than Sybase ASA). This *whitepaper* has some details about ODBC connectivity of ASE.

Gary Pennington from Sun Microsystems reported that the Data-Direct (formerly Merant) evaluation drivers work with Sybase Adaptive Server 11.5 on Solaris 2.6.

You can also use the OpenLink drivers for Sybase ASE: copy `Setup.in` to `Setup` and enable the OpenLink section in `Setup` before compiling.

In any case, you should also consult Paul Boddie's *mxODBC Configuration* page for the Sybase Adaptive Server Enterprise version. It includes valuable information about the setup.

13. Hints & Links to other Resources

13.1 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`).

13.2 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 *mx.ODBC.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.

13.3 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.

14. 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 database backend MySQL does not support transactions.

15. 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.py.

```
python mx/ODBC/Misc/test.py
```

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.

16. mxODBC Package Structure

This is the Python package structure setup when installing mxODBC:

```
[ODBC]
        [Adabas]
                dbi.py
                dbtypes.py
                showdb.py
        [DB2]
                dbi.py
                dbtypes.py
        [DBMaker]
                dbi.py
                dbtypes.py
        Doc/
        [EasySoft]
                dbi.py
                dbtypes.py
        [Informix]
                dbi.py
                dbtypes.py
        [Misc]
                proc.py
                test.py
        [MySQL]
                dbi.py
                dbtypes.py
        [Oracle]
                dbi.py
                dbtypes.py
        [PostgreSQL]
                dbi.py
                dbtypes.py
        [SAPDB]
                dbi.py
                dbtypes.py
        [Solid]
                dbi.py
                dbtypes.py
        [SybaseASA]
                dbi.py
                dbtypes.py
        [SybaseASE]
                dbi.py
                dbtypes.py
        [Windows]
                dbi.py
                dbtypes.py
        [iODBC]
                dbi.py
                dbtypes.py
        [mxODBC]
                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`.

17. Support

eGenix.com is providing 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 see the *eGenix.com Support Conditions*.

18. Copyright & License

EGENIX.COM COMMERCIAL LICENSE AGREEMENT

Version 1.2.0

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 as follows:

1) "CPU" refers to a central processing unit which is able to store and/or execute the Software (a server, personal computer, 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.

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. Likewise, 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>