

mxTools

Fast Everyday Helpers
for Python

Version 3.2

Copyright © 1997-2000 by IKDS Marc-André Lemburg, Langenfeld
Copyright © 2000-2011 by eGenix.com GmbH, Langenfeld

All rights reserved. No part of this work may be reproduced or used in any form or by any means without written permission of the publisher.

All product names and logos are trademarks of their respective owners.

The product names "mxBeeBase", "mxCGIPython", "mxCounter", "mxCrypto", "mxDateTime", "mxHTMLTools", "mxIP", "mxLicenseManager", "mxLog", "mxNumber", "mxODBC", "mxODBC Connect", "mxODBC Zope DA", "mxObjectStore", "mxProxy", "mxQueue", "mxStack", "mxTextTools", "mxTidy", "mxTools", "mxUID", "mxURL", "mxXMLTools", "eGenix Application Server", "PythonHTML", "eGenix" and "eGenix.com" and corresponding logos are trademarks or registered trademarks of eGenix.com GmbH, Langenfeld

Printed in Germany.

Contents

1.	Introduction	1
2.	Naming Scheme used in mxTools	2
3.	Automatic Installation as Python Built-ins and sys-module Functions.....	3
4.	mx.Tools Functions.....	4
4.1	New Built-in Functions	4
4.2	New Built-in Objects	9
4.3	New sys-Module Functions	9
4.4	mx.Tools Functions	11
4.5	mx.Tools Objects	13
5.	Examples of Use	15
6.	Package Structure	16
7.	Support	17
8.	Copyright & License	18

1. Introduction

The mxTools package is a collection of helpers and new built-ins that we collected over time.

Most of the functions are written in C with performance in mind. Many of them have inspired Python built-ins that are now standard part of the language itself, either as built-ins or as part of e.g. the *itertools* module.

The functions defined by the package are installed by the package at import time in different places of the Python interpreter. They work as fast add-ons to the existing set of functions and objects.

2. Naming Scheme used in mxTools

mxTools uses the following naming scheme:

- `i` stands for indexed, meaning that you have access to indices
- `m` stands for multi, meaning that processing involves multiple objects
- `n` stands for n-times, e.g. a function is executed a certain number of times
- `t` stands for tuple
- `x` stands for lazy evaluation

3. Automatic Installation as Python Built-ins and sys-module Functions

Most of the functions defined by this package are installed as Python built-in functions or added to the Python `sys` module at package import time.

They are then available as normal built-in functions in every module without explicit import in each module using them (though it is good practice to still put a `'import mx.Tools.NewBuiltins'` at the top of each module relying on these add-ons).

Note that Python 2.2 introduced a few built-ins having the same name as the ones installed by mxTools. mxTools does *not* overwrite these, so the semantics you get are those of the Python built-in APIs. Fortunately, the differences are only minimal.

4. mx.Tools Functions

The package defines the following functions.

4.1 New Built-in Functions

The following functions are installed as new Python built-ins upon importing mx.Tools.

`acquire(object,name)`, `mx.Tools.acquire(...)`

Looks up the attribute name in `object.baseobj` and returns the result. If `object` does not have an attribute `.baseobj` or that attribute is `None` or the attribute name starts with an underscore, an `AttributeError` is raised.

This function can be used as `.__getattr__()` hook in Python classes to enable implicit acquisition along a predefined lookup chain (`object.baseobj` provides a way to set up this chain). See `Examples/Acquisition.py` for some sample code.

`attrlist(object_list,attrname)`, `mx.Tools.attrlist(...)`

Returns a list of all attributes with name `attrname` found among the objects in the list.

`binary(object)`, `mx.Tools.binary(object)`

Alias for `buffer(object)`.

`boolean(object)`, `mx.Tools.boolean(object)`

Alias for `truth(object)`.

Note that Python 2.2 comes with its own `bool()` constructor which provides the same functionality as this API.

`count(condition,sequence)`, `mx.Tools.count(...)`

Counts the number of objects in `sequence` for which `condition` returns true and returns the result as integer. `condition` must be a callable object.

`datetime(object)`, `mx.Tools.datetime(object)`

Alias for `mx.DateTime.DateTimeFrom(object)`. This requires `mxDateTime` to be installed.

WARNING: Python 2.3 comes with its own `datetime()` constructor which creates a `datetime` object type that behaves differently than `mxDateTime` objects. This built-in is not automatically installed in Python 2.3 and above.

`defined(name), mx.Tools.defined(...)`

Returns true iff a symbol name is defined in the current namespace.

The function has intimate knowledge about how symbol resolution works in Python: it first looks in `locals()`, then in `globals()` and if that fails in `__built-ins__`.

`dict(items), mx.Tools.dict(items)`

Constructs a dictionary from the given items sequence. The sequence items must contain sequence entries with at least two values. The first one is interpreted as key, the second one as associated object. Remaining values are ignored.

Note that Python 2.2 comes with its own `dict()` API, so this built-in is not automatically installed in Python 2.2 and above.

`exists(condition, sequence), mx.Tools.exists(...)`

Return 1 if and only if `condition` is true for at least one of the items in sequence and 0 otherwise. `condition` must be a callable object.

`extract(object, indices[, defaults]), mx.Tools.extract(...)`

Builds a list with entries `object[index]` for each index in the sequence `indices`.

If a lookup fails and the sequence `defaults` is given, then `defaults[nth_index]` is used, where `nth_index` is the index of `index` in `indices` (confused? it works as expected!). `defaults` should have the same length as `indices`.

If you need the indices as well, try the `irange` function. The function raises an `IndexError` in case it can't find an entry in `indices` or `defaults`.

`findattr(object_list, attrname), mx.Tools.findattr(...)`

Returns the first attribute with name `attrname` found among the objects in the list. Raises an `AttributeError` if the attribute is not found.

`forall(condition, sequence), mx.Tools.forall(...)`

Return 1 if and only if `condition` is true for all of the items in sequence and 0 otherwise. `condition` must be a callable object.

mxTools - Fast Everyday Helpers for Python

```
get(object, index[, default]), mx.Tools.get(...)
```

Returns `object[index]`, or, if that fails, `default`. If `default` is not given or the singleton `NotGiven` an error is raised (the error produced by the object).

```
ifilter(condition, object[, indices]), mx.Tools.ifilter(...)
```

Builds a list of tuples `(index, object[index])` such that `condition(object[index])` is true and `index` is found in the sequence `indices` (defaulting to `trange(len(object))`). Order is preserved. `condition` must be a callable object.

```
index(condition, sequence), mx.Tools.index(...)
```

Return the index of the first item for which `condition` is true. A `ValueError` is raised in case no item is found. `condition` must be a callable object.

```
indices(object), mx.Tools.indices(object)
```

Returns the same as `tuple(range(len(object)))` -- a tad faster and a lot easier to type.

```
invdict(dictionary), mx.Tools.invdict(dictionary)
```

Constructs a new dictionary from the given one with inverted mappings. Keys become values and vice versa. Note that no exception is raised if the values are not unique. The result is undefined in this case (there is a `value:key` entry, but it is not defined which key gets used).

```
irange(object[, indices]), mx.Tools.irange(...)
```

Builds a tuple of tuples `(index, object[index])`. If a sequence `indices` is given, the indices are read from it. If not, then the index sequence defaults to `trange(len(object))`.

Note that `object` can be any object that can handle `object[index]`, e.g. lists, tuples, string, dictionaries, even your own objects, if they provide a `__getitem__`-method. This makes very nifty constructions possible and extracting items from another sequence becomes a piece of cake. Give it a try ! You'll soon love this little function.

```
iremove(object, indices), mx.Tools.iremove(...)
```

Removes the items indexed by `indices` from `object`.

This changes the object in place and thus is only possible for mutable types.

For sequences the index list must be sorted ascending; an `IndexError` will be raised otherwise (and the object left in an undefined state).

4. mx.Tools Functions

```
lists(sequence), mx.Tools.lists(sequence)
```

Same as `tuples(sequence)`, except that a tuple of lists is returned. Can be used as inverse to `tuples()`.

```
mapply(callable_objects[,args=(),kw={}]), mx.Tools.mapply(...)
```

Creates a tuple of values by applying the given arguments to each object in the sequence `callable_objects`.

This function has a functionality dual to that of `map()`. While `map()` applies many different arguments to one callable object, this function applies one set of arguments to many different callable objects.

```
method_mapply(objects,methodname[,args=(),kw={}]),  
mx.Tools.method_mapply(...)
```

Creates a tuple of values by applying the given arguments to each object's `<methodname>` method. The objects are processed as given in the sequence `objects`.

A simple application is e.g. `method_mapply([a,b,c], 'method', (x,y))` resulting in a tuple `(a.method(x,y), b.method(x,y), c.method(x,y))`. Thanks to Aaron Waters for suggesting this function.

```
napply(number_of_calls,function[,args=(),kw={}]),  
mx.Tools.napply(...)
```

Calls the given function `number_of_calls` times with the same arguments and returns a tuple with the return values. This is roughly equivalent to a for-loop that repeatedly calls

```
apply(function,args,kw)
```

 and stores the return values in a tuple.

Example: create a tuple of 10 random integers... `l = napply(10,whrandom.randint,(0,10))`.

```
reverse(sequence), mx.Tools.reverse(sequence)
```

Returns a tuple or list with the elements from `sequence` in reverse order. A tuple is returned, if the sequence itself is a tuple. In all other cases a list is returned.

```
reval(codestring[,locals={}]), mx.Tools.reval(...)
```

Evaluates the given codestring in a restricted environment that only allows access to operators and basic type constructors like `()`, `[]` and `{}`.

No built-ins are available for the evaluation. `locals` can be given as local namespace to use when evaluating the codestring.

After a suggestion by Tim Peters on `comp.lang.python`.

```
setdict(sequence,value=None), mx.Tools.setdict(...)
```

Constructs a dictionary from the given sequence. The sequence must contain hashable objects which are used as keys. The values are all set to `value`. Multiple keys are silently ignored. The function comes in handy

mxTools - Fast Everyday Helpers for Python

whenever you need to work with a sequence in a set based context (e.g. to determine the set of used values).

```
sign(object), mx.Tools.sign(object)
```

Returns the signum of object interpreted as number, i.e. -1 for negative numbers, +1 for positive ones and 0 in case it is equal to 0. The method used is equivalent to `cmp(object,-object)`.

```
sizeof(object), mx.Tools sizeof(object)
```

Returns the number of bytes allocated for the given Python object. Additional space allocated by the object and stored in pointers is not taken into account (though the pointer itself is). If the object defines `tp_itemsize` in its type object then it is assumed to be a variable size object and the size is adjusted accordingly.

```
trange([start=0,]stop[,step=1]), mx.Tools.trange(...)
```

This works like the built-in function `range()` but returns a tuple instead of a list. Since `range()` is most often used in for-loops there really is no need for a mutable data type and construction of tuples is somewhat (20%) faster than that of lists. So changing the usage of `range()` in for-loops to `trange()` pays off in the long run.

```
range_len(object), mx.Tools.range_len(object)
```

Returns the same as `range(len(object))`.

```
truth(object), mx.Tools.truth(object)
```

Returns the truth value of object as truth singleton (True or False). Note that the singletons are ordinary Python integers 1 and 0, so you can also use them in calculations.

This function is different from the one in the `operator` module: the function does not return truth singletons but integers.

```
tuples(sequence), mx.Tools.tuples(sequence)
```

Returns much the same as `apply(map,(None,)+tuple(sequence))` does, except that the resulting list will always have the length of the first sub-sequence in sequence. The function returns a list of tuples `(a[0], b[0], c[0],...), (a[1], b[1], c[1],...), ...` with missing elements being filled in with `None`.

Note that the function is of the single argument type meaning that calling `tuples(a,b,c)` is the same as calling `tuples((a,b,c))`. `tuples()` can be used as inverse to `lists()`.

4.2 New Built-in Objects

These objects are available as built-ins after importing the package:

```
NotGiven, mx.Tools.NotGiven
```

This is a singleton similar to `None`. Its main purpose is providing a way to indicate that a keyword was not given in a call to a keyword capable function, e.g.

```
import mx.Tools.NewBuiltins

def f(a,b=4,c=NotGiven,d=''):
    if c is NotGiven:
        return a / b, d
    else:
        return a*b + c, d
```

It is also considered false in `if`-statements, e.g.

```
import mx.Tools.NewBuiltins

a = NotGiven
# ...init a conditionally...
if not a:
    print 'a was not given as value'
```

```
True, False, mx.Tools.True, mx.Tools.False
```

These two singletons are used by Python internally to express the boolean values true and false. They represent Python integer objects for 1 and 0 resp. All explicit comparisons return these singletons, e.g.

```
(1==1) is True and (1==0) is False.
```

Note that Python 2.2 comes with its own True and False singletons, so these built-ins will not automatically be installed in Python 2.2 and above. Fortunately, the ones used in Python 2.2 are the same as used by mxTools, so no changes to existing code are necessary.

4.3 New sys-Module Functions

The following functions are installed as add-ons to the built-in `sys` module.

```
sys.cur_frame([offset=0]), mx.Tools.cur_frame([offset=0])
```

Return the current execution frame. If level is given, the returned frame is taken from offset levels up the execution stack. None is returned in case the frame is not found, i.e. there are not enough frames on the stack.

mxTools - Fast Everyday Helpers for Python

Note: Storing the execution frame in a local variable introduces a circular reference, since the locals and globals are referenced in the execution frame, so use the return value with caution.

```
sys.debugging([level]), mx.Tools.debugging([level])
```

If level is given, the value of the interpreter's debugging flag is set to level and the previous value of that flag is returned. Otherwise, the current value is returned.

You can use this function to check whether the interpreter was called with '-d' flag or not. Some extensions use this flag to enable/disable debugging log output (e.g. all the [eGenix.com mx Extensions](http://eGenix.com/mx/Extensions)).

```
sys.interactive([level]), mx.Tools.interactive([level])
```

If level is given, the value of the interpreter's interactive flag is set to level and the previous value of that flag is returned. Otherwise, the current value is returned.

You can use this function to e.g. have the interpreter go into interactive mode when an exception occurs, even though the interpreter was not started with -i. Python has to be started in a terminal session for this to be helpful.

```
sys.makeref(id), mx.Tools.makeref(id)
```

Provided that id is a valid address of a Python object (`id(object)` returns this address), this function returns a new reference to it. Only objects that are "alive" can be referenced this way, ones with zero reference count cause an exception to be raised.

You can use this function to reaccess objects lost during garbage collection.

USE WITH CARE: this is an expert-only function since it can cause instant core dumps and many other strange things -- even ruin your system if you don't know what you're doing !

SECURITY WARNING: This function can provide you with access to objects that are otherwise not visible, e.g. in restricted mode, and thus be a potential security hole.

```
sys.optimization([level]), mx.Tools.optimization([level])
```

If level is given, the value of the interpreter's optimization flag is set to level and the previous value of that flag is returned. Otherwise, the current value is returned.

You can use this function to e.g. compile Python scripts in optimized mode even though the interpreter was not started with -O.

```
sys.verbosity([level]), mx.Tools.verbosity([level])
```

If `level` is given, the value of the interpreter's verbosity flag is set to `level` and the previous value of that flag is returned. Otherwise, the current value is returned.

You can use this function to e.g. enable verbose lookup output to `stderr` for import statements even when the interpreter was not invoked with `-v` or `-vv` switch or to force verbosity to be switched off.

4.4 mx.Tools Functions

The following functions are *not* installed in any built-in module. Instead, you have to reference them via the `mx.Tools` module.

```
mx.Tools.dictscan(dictobj[,prevposition=0])
```

Dictionary scanner.

Returns a tuple (`key,value,position`) containing the `key,value` pair and slot position of the next item found in the dictionaries hash table after slot `prevposition`.

Raises an `IndexError` when the end of the table is reached or the `prevposition` index is out of range.

Note that the dictionary scanner does *not* produce an items list. It provides a very memory efficient way of iterating over large dictionaries.

```
mx.Tools.dlopen(libname[, mode])
```

Load the shared library `libname` using the given mode.

This function is a direct interface to the Unix `dlopen()` function which allows loading arbitrary shared libraries into the process. `mode` defaults to the Python default `dlopen` flags (these can be set using `sys.setdlopenflags()`).

`libname` may include a relative or absolute pathname of the shared library. If no path is included in the `libname`, the standard system linker strategy for finding shared libraries is used, which usually means looking on the `LD_LIBRARY_PATH` and then in the `ld.so` cache.

The advantage of using this function lies in the possibility to provide the full path to the shared library, ie. you don't have to rely on a properly configured `LD_LIBRARY_PATH` environment variable (which cannot be set after process start).

Raises an `OSError` in case of an error while loading or search for the shared library.

mxTools - Fast Everyday Helpers for Python

```
mx.Tools.fqhostname(hostname=None, ip=None)
```

Tries to return the fully qualified (`hostname`, `ip`) for the given hostname.

If `hostname` is None, the default name of the local host is chosen. `ip` then defaults to '127.0.0.1' if not given.

The function modifies the input data according to what it finds using the socket module. If that doesn't work the input data is returned unchanged.

```
mx.Tools.scanfiles(files, dir=None, levels=0, filefilter=None)
```

Build a list of filenames starting with the filenames and directories given in files.

The filenames in are made absolute relative to `dir`. `dir` defaults to the current working directory if not given.

If `levels` is greater than 0, directories in the files list are recursed into up the given number of levels.

If `filefilter` is given, as re match object, then all filenames (the absolute names) are matched against it. Filenames which do not match the criteria are removed from the list.

Note that directories are not included in the resulting list. All filenames are non-directories.

If no user name can be determined, default is returned.

```
mx.Tools.setproctitle(title)
```

Set the process title to title.

Note that the title length is usually limited to what the original process title was at start-up time. The function will truncate the given title as necessary.

Note:

This function is disabled per default since it relies on a hidden API in the Python interpreter which is not always exposed. If you would like to use it, please edit the `egenix_mx_base.py` configuration and enable the line (`'HAVE_PY_GETARGCARGV', None`). If you get import errors from `mx.Tools`, chances are high that your Python version does not support the hidden API. It is known to work with Python 2.1 - 2.6.

```
mx.Tools.srange(string)
```

Converts a textual representation of integer numbers and ranges to a Python list.

Supported formats: "2,3,4,2-10,-1 - -3, 5 - -2"

Values are appended to the created list in the order specified in the string.

```
mx.Tools.username(default='')
```

Return the user name of the user running the current process.

If no user name can be determined, default is returned.

```
mx.Tools.verscmp(a,b)
```

Compares two version strings and returns a `cmp()` function compatible value (< returns -1, = returns 0, > returns +1). The function is useful for sorting lists containing version strings.

The logic used is as follows: the strings are compared at each level, empty levels defaulting to '0', numbers with attached strings (e.g. '1a1') compare less than numbers without attachment (e.g. '1a1' < '1').

4.5 mx.Tools Objects

The following objects are not installed in any built-in module. Instead, you have to reference them via the `mx.Tools` module.

```
mx.Tools.DictScan(dictionary)
```

Creates a forward iterator for the given dictionary. It is based on `mx.Tools.dictscan()`.

The dictionary scanner does *not* produce an items list. It provides a very memory efficient way of iterating over large dictionaries.

Note that no precaution is taken to insure that the dictionary is not modified in-between calls to the `.__getitem__()` method. It is the user's responsibility to ensure that the dictionary is neither modified, nor changed in size, since this would result in skipping entries or double occurrence of items in the scan.

The iterator inherits all methods from the underlying dictionary for convenience.

The returned object inherits all methods from the underlying dictionary and additionally provides the following methods:

```
.reset()
```

Resets the iterator to its initial position.

```
mx.Tools.DictItems(dictionary)
```

mxTools - Fast Everyday Helpers for Python

Is an alias for `mx.Tools.DictScan()`.

5. Examples of Use

A few simple examples:

```
import mx.Tools.NewBuiltins

sequence = range(100)

# In place calculations:
for i,item in irange(sequence):
    sequence[i] = 2*item

# Get all odd-indexed items from a sequence:
odds = extract(sequence, trange(0, len(sequence), 2))

# Turn a tuple of lists into a list of tuples:
chars = 'abcdefghji'
ords = map(ord, chars)
table = tuples(chars, ords)

# The same as dictionary:
chr2ord = dict(table)

# Inverse mapping:
ord2chr = invdict(chr2ord)

# Range checking:
if exists( lambda x: x > 10, sequence ):
    print 'Warning: Big sequence elements!'

# Handle special cases:
if forall( lambda x: x > 0, sequence ):
    print 'Positive sequence'
else:
    print 'Index %i loses' % (index( lambda x: x <= 0, sequence ),)

# dict.get functionality for e.g. lists:
print get(sequence, 101, "Don't have an element with index 101")

# Filtering away false entries of a list:
print filter(truth, [1, 2, 3, 0, '', None, NotGiven, 4, 5, 6])
```

More elaborate examples can be found in the Examples/ subdirectory of the package.

6. Package Structure

```
[Tools]
  Doc/
  [Examples]
    Acquisition.py
  [mxTools]
    vc5/
    bench1.py
    bench2.py
    hack.py
    test.py
  NewBuilt-ins.py
  Tools.py
```

Entries enclosed in brackets are packages (i.e. they are directories that include a `__init__.py` file) or submodules. Ones with slashes are just ordinary subdirectories that are not accessible via `import`.

Importing `mx.Tools` will automatically install the functions and objects defined in this package as built-ins. They are then available in all other modules without having to import them again every time. If you don't want this feature, you can turn it off in `mx/Tools/__init__.py`.

7. Support

eGenix.com is providing commercial support for this package. If you are interested in receiving information about this service please see the [eGenix.com Support Conditions](#).

8. Copyright & License

© 1997-2000, Copyright by IKDS Marc-André Lemburg; All Rights Reserved. mailto: mal@lemburg.com

© 2001-2011, Copyright by eGenix.com Software GmbH, Langenfeld, Germany; All Rights Reserved. mailto: info@egenix.com

This software is covered by the ***eGenix.com Public License Agreement***, which is included in the following section. The text of the license is also included as file "LICENSE" in the package's main directory.

By downloading, copying, installing or otherwise using the software, you agree to be bound by the terms and conditions of the following *eGenix.com Public License Agreement*.

EGENIX.COM PUBLIC LICENSE AGREEMENT

Version 1.1.0

This license agreement is based on the [Python CNRI License Agreement](#), a widely accepted open-source license.

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. License

Subject to the terms and conditions of this eGenix.com Public License Agreement, eGenix.com hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the eGenix.com Public License Agreement is retained in the Software, or in any derivative version of the Software prepared by Licensee.

3. 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.

4. LIMITATION OF LIABILITY

EGENIX.COM SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR 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.

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.

5. Termination

This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. 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.

7. 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.

8. Copyright & License

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.

8. 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