mxDateTime

Date/Time Library
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 a 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

The mxDateTime package provides consistent way of transferring date and time data between Python and databases.

Apart from handling dates before the Unix epoch (1.1.1970) they also correctly work with dates beyond the Unix time limit (currently with Unix time values being commonly encoded using 32bit integers, the limit is reached in 2038) and thus is **Year 2000** and **Year 2038** safe.

The package provides three main data structures for working with date and time values.

These are:

- *DateTime* for referring to absolute date/time values,
- *DateTimeDelta* for date/time spans and
- *RelativeDateTime* for representing variable date/time spans (these are the TABs of date/time calculation)

All object, functions and constants are available via the package namespace `mx.DateTime`.

2. Design

The primary absolute date/time type **DateTime** uses the following internal format:

Absolute date

> This is a C `long` defined as being the number of days in the Gregorian calendar since the day before January 1 in the year 1 (0001-01-01), the *Gregorian Epoch*, also known as the Epoch of the Common Era (CE), thus the Gregorian date 0001-01-01 corresponds to absolute date 1. Note that the *Julian Epoch* starts two days before the Gregorian one.

Absolute time

> This is a C `double` defined as the number of seconds since midnight (0:00:00.00) of the day expressed by the above value.

The *Epoch* used by the module is January 1st of the year 1 at midnight (0:00:00.00) in the Gregorian calendar. This date corresponds to absolute day 1 and absolute time 0. Dates before the Epoch are handled by extrapolating the calendars using negative years as basis (the year 1 BCE corresponds to the year 0, 2 BCE is represented as year -1 and so on).

For the purpose of storing absolute time differences, the package provides a second type called *DateTimeDelta*. The internal representation for this type is seconds and stored in a signed C `double`.

To handle relative time deltas a third object type is available: *RelativeDateTime*. This object is currently implemented in Python and may be used to store relative time deltas (see below for an exact description). It's main purpose is providing an intuitive way to calculate e.g. the "first of next month".

Designing the types wasn't as easy as expected, since many criteria had to be taken into account. Here are some of them and their implementation:

2.1 Time Zones, Daylight Savings Time (DST) and Leap Seconds

Time zones are among the most difficult to handle issues when it comes to implementing and using types for date and time. We chose to move the time zone handling functionality out of the C implementation and into Python. This means that the types know nothing about the time zones of the values they store and calculations are done using the raw data.

If you need to store and use these informations in calculations, you can "subclass" the types to implement your ideas rather than having to stick to what the C implementation defines. The included ODMG submodule is an example of how this can be done.

Leap seconds are not supported either. You can implement classes respecting these by "subclassing" DateTime and DateTimeDelta and then overriding the calculation methods with methods that work on Unix ticks values (provided the underlying C lib knows about leap seconds -- most don't and the POSIX standard even enforces *not* to use leap seconds).

2.2 Calendars

The module supports two calendars, the Gregorian (default and needed for most conversions) and the Julian, which is handy for dates prior to the year 1582 when the calendar was revised by Pope Gregory XIII.

Construction of Julian dates can be done using either the `JulianDateTime()` constructor or indirect through the `.Julian()` method of DateTime instances. To check which calendar a DateTime instance uses, query the `calendar` instance attribute.

Note that Julian dates output the Julian date through the instances date attributes and broken down values. Not all conversions are available on instances using the Julian calendar. Even though in the Julian calendar days start at noon (12:00:00.0), mxDateTime will use the Gregorian convention of using the date for the period from 00:00:00.0 to 23:59:59.99 of that day (this may change in future versions of mxDateTime).

Both calendars use mathematical models as basis -- they do not account for the many inaccuracies that occurred during their usage history. For this reason, the `.absdate` values should be interpreted with care, esp. for dates using the Julian calendar. As a result of the mathematical models, the

Epochs in the calendars differ by a few days. This was needed in order to synchronize the calendars in the switching year 1582: `JulianDate(1,1,1)` points to a date two days before `Date(1,1,1)`.

2.3 Conversion from and to other formats

For the purpose of converting the stored values to Unix ticks (number of seconds since the Unix epoch; the C lib also uses this representation) we assume that the values are given in **local time**. This assumption had to be made because the C lib provides no standard way to convert a broken down date/time value in any other way into a ticks value.

Conversions to COM dates and tuples are done without any assumption on the time zone. The raw values are used.

Conversion from other formats to DateTime instances is always done by first calculating the corresponding absolute time and date values (which are also used as basis for calculations).

2.4 Rounding Errors and Roundtrip-Safety

The internal representation of date/times behaves much like floats do in Python, i.e. *rounding errors* can occur when doing calculations. There is a special compare function included (`cmp()`) in the package that allows you to compare two date/time values using a given accuracy, e.g. `cmp(date1, date2, 0.5)` will allow 12:00:00.5 and 12:00:01.0 to compare equal.

Special care has been taken to prevent these rounding errors from occurring for COM dates. If you create a DateTime instance using a COM date, then the value returned by the `.COMDate()` method is guaranteed to be exactly the same as the one used for creation. The same is true for creation using absolute time and absolute date and broken down values.

2.5 Immutability

One other thing to keep in mind when working with DateTime and DateTimeDelta instances is that they are immutable (like tuples). Once an instance is created you can not change its value. Instead, you will have to create a new instance with modified values.

The advantage of having immutable objects is that they can be used as dictionary keys.

2.6 UTC and GMT

UTC (a mix of: Temps Universel Coordonné and Coordinated Universal Time) and GMT (Greenich Mean Time) are two names for more or less the same thing: they both refer to the international universal time which is used throughout the world to coordinate events in time regardless of time zone, day light savings time or other local time alterations. See the *Calendar FAQ* for more infos and the exact definitions of UTC and GMT.

The mx.DateTime package uses these two names interchangeably. Sometimes API only refer to one name for simplicity. The name preference (GMT or UTC) is often chosen according to common usage.

2.7 Interaction with other Types

DateTime and DateTimeDelta instances can be compared and hashed, making them compatible to the dictionary implementation Python uses (they can be used as keys). The copy protocol, simple arithmetic and pickleing are also supported (see below for details).

2.8 String formats

DateTime and DateTimeDelta instances know how to output themselves as ISO8601-strings. The format is very simple: `YYYY-MM-DD HH:MM:SS.ss` for DateTime instances and `[-][DD:]HH:MM:SS.ss` for DateTimeDelta instances (the DD-part (days) is only given if the absolute delta value is greater than 24 hours). Customized conversion to strings can be done using the `strftime`-methods or the included submodules.

String parsing is supported through the `strptime()` constructor which implements a very strict parsing scheme and the included submodules (e.g. *ISO* and *ARPA*), which allow a little more freedom.

2.9 Speed and Memory

Comparing the types to time-module based routines is not really possible, since the used strategies differ. You can compare them to tuple-based date/time classes though: DateTime[Delta] are much faster on creation, use less storage and are faster to convert to the supported other formats than any equivalent tuple-based implementation written in Python.

Creation of time-module values using `time.mktime()` is much slower than doing the same thing with DateTime(). The same holds for the reverse conversion (using `time.localtime()`).

The storage size of ticks (floats, which the time module uses) is about 1/3 of the size a DateTime instance uses. This is mainly due to the fact that DateTime instances cache the broken down values for fast access.

To summarize: DateTime[Delta] are faster, but also use more memory than traditional time-module based techniques.

2.10 Background and Resource Information on the Web

Here is a small list of links I used as starting points to find some of the date/time related information included in this package:

- The *Calendar FAQ* by Claus Tondering.
- The *Calendar Links* by Rudy Limeback.
- The *Ecclesiastical Calendar* by Marcos J. Montes.
- The *Systems of Time* page provided by the Time Service Dept., U.S. Naval Observatory, Washington, DC.
- The *Calendar Conversion* page by Scott E. Lee.
- For the interested reader, I also suggest *A walk through time* presented by the NIST Time and Frequency Division.

3. mx.DateTime.DateTime Object

DateTime objects encapsulate an absolute point in the date/time continuum.

3.1 DateTime Object Constructors

Several constructors are available in the module `DateTime`. All of these return DateTime instances using the Gregorian calendar except for `JulianDateTime()` which returns instances using the Julian calendar.

`DateTime(year,month=1,day=1,hour=0,minute=0,second=0.0)`

Constructs a DateTime instance from the given values.

This is the standard constructor for DateTime instances.

Assumes that the date is given in the Gregorian calendar (which it the one used in many countries today).

The entry for `day` can be negative to indicate days counted in reverse order, that is the last day becomes -1, the day before that -2, and so on, e.g. `DateTime(1997,12,-2)` gives the 30.12.1997 (this is useful especially for months).

Note that although the above makes it look like this function can handle keywords, it currently cannot.

The following constructors are provided to simplify integration with existing code.

`Date(year,month,day)`

Is just another name binding for `DateTime()`. The time part is set to 00:00:00.0.

`DateFrom(*args,**kws)`

Constructs a DateTime instance from the arguments, but only uses the date parts and drops the time information, if any.

This constructor can parse strings, handle numeric arguments, Python `datetime.date` and `datetime.datetime` objects, and knows about the keywords `year,month,day`.

It uses type inference to find out how to interpret the arguments and makes use of the Parser module.

`DateFromTicks(ticks)`

Constructs a DateTime instance pointing to the local time date at 00:00:00.00 (midnight) indicated by the given ticks value. The time part is ignored.

`DateTimeFrom(*args,**kws)`

Constructs a DateTime instance from the arguments.

This constructor can parse strings, handle numeric arguments, Python `datetime.date` and `datetime.datetime` objects, and knows about the keywords `year,month,day,hour,minute,second`.

It uses type inference to find out how to interpret the arguments and makes use of the Parser module.

`DateTimeFromAbsDateTime(absdate, abstime, calendar=Greogorian)`

Returns a new DateTime instance for the given absolute date and time.

`calendar` may be given to create a DateTime instance for a specific calendar. It defaults to the Gregorian calendar.

This interface can be used by classes written in Python which implement other calendars than the Gregorian, for example.

`DateTimeFromAbsDays(days)`

Constructs a DateTime instance from the days since the (Christian) Epoch value.

`DateTimeFromCOMDate(comdate)`

Constructs a DateTime instance from the *COM date* value.

This is used by the Windows COM interface and represents the date/time difference between 30.12.1899 and the represented date/time, with time being encoded as fraction of a whole day, thus 0.5 corresponds to 12:00:00.00.

Special care is taken that the resulting instance's method `.COMDate()` returns exactly the same value as the one used for constructing it -- even though the internal representation is more accurate.

`DateTimeFromMJD(mjd)`

Constructs a DateTime instance from the given *Modified Julian Day* (MJD) value.

Since MJD values are given in UTC, the instance will represent UTC. See the *Calendar FAQ* for details.

Note: Usage of MJD notation is discouraged by the International Astronomical Union (IAU). Use JDN instead.

`DateTimeFromJDN(jdn)`

Constructs a DateTime instance from the given *Julian Day Number* (JDN).

Since JDN values are given in UTC, the instance will represent UTC. See the *Calendar FAQ* for details.

`DateTimeFromTicks(ticks)`

Constructs a DateTime instance pointing to the local time indicated by the given ticks value. Raises an Error in case the ticks value cannot be converted to a date/time representation.

`DateTimeFromTJD(tjd,tjd_myriad=current_myriad)`

Constructs a DateTime instance from the given *Truncated Julian Day* (TJD) value as used by NASA and the U.S. Naval Observatory, that is TJD = (MJD - 40000) % 10000 or simply TJD = MJD % 10000. Some sources define TJD = MJD - 40000 making it non-periodic; this is not supported by this constructor.

tjd_myriad will default to the tjd_myriad current at package import time, if not given. It refers to the truncated part of the TDJ number. The current myriad (245) started on 1995-10-10 00:00:00.00 UTC and will last until 2023-02-24 23:59:59.99 UTC.

Since TJD values are always given in UTC, the instance will represent UTC.

Please note that usage of TJD is deprecated because of the information loss involved with truncating data: use MJD or JDN instead.

`JulianDate(year,month=1,day=1)`

Is just another name binding for JulianDateTime(). The time part is set to 00:00:00.0.

`JulianDateTime(year,month=1,day=1, hour=0,minute=0,second=0.0)`

Constructs a DateTime instance from the given values assuming they are given in the Julian calendar.

The instance will use the Julian calendar for all date related methods and attributes.

Same comments as for `DateTime()`.

`GregorianDate(year,month,day)`

Is just another name binding for `DateTime()`. The time part is set to 00:00:00.0.

`GregorianDateTime(year,month=1,day=1,hour=0,minute=0,second=0.0)`

Is just another name binding for `DateTime()`.

`Timestamp(year,month,day,hour=0,minute=0,second=0.0)`

Is just another name binding for `DateTime()`.

`TimestampFrom(*args,**kws)`

Alias for `DateTimeFrom()`.

`TimestampFromTicks(ticks)`

Alias for `DateTimeFromTicks()`.

`gmt()`

Returns a new DateTime instance reflecting the current GMT time.

`gmtime(ticks=time.time())`

Constructs a DateTime instance from the ticks value (this is what `time.time()` returns; see the `time` module for details).

The instance will hold the associated UTC time. If ticks is not given, the current time is used. `gmticks()` is the inverse of this function.

`mktime(tuple)`

Same as the `DateTime()` constructor accept that the interface used is compatible to the similar `time.mktime()` API.

`tuple` has to be a 9-tuple `(year,month,day,hour,minute,second,dow,doy,dst)`.

Note that the tuple elements `dow`,`doy` and `dst` are not used in any way.

You should only use this constructor for porting applications from time module based functions to DateTime.

`now()`

Returns a new DateTime instance reflecting the current local time.

mxDateTime tries to use the most accurate clock available on the system. On recent Unix systems this is usually a nanosecond resolution clock, but the actual measured resolution will usually be within a microsecond. On Windows, the resolution is around one millisecond.

`localtime(ticks)`

Constructs a DateTime instance from the ticks value (this is what `time.time()` returns; see the `time` module for details).

The instance will hold the associated local time.

`strptime(string,format_string[,default])`

Parse the given string using the format string and construct a DateTime instance from the found value.

If `default` is given (must be a DateTime instance), it's entries are used as default values. Otherwise, 0001-01-01 00:00:00.00 is used. An `Error` is raised if the underlying C parsing function `strptime()` fails.

Portability note: `default` does not work on Solaris. You will have to reassemble the correct DateTime instance yourself (knowing which parts the `strptime()` function parsed) if you intend to use default values. Solaris sets the defaults to 1900-01-01 00:00:00.00 and then overwrites them with the parsed values.

Note: Since this C API is relatively new, you may not have access to this constructor on your platform. For further information on the format, please refer to the Unix man-page (it is very similar to that of `strftime()` which is documented in the Python library reference for the time module).

`today(hour=0,minute=0,second=0.0)`

Returns a DateTime instance for the current date (in local time) at the given time (defaults to midnight). E.g. `today(14,00)` is today at 1400 hours.

`utc()`

Alias for `gmt()`.

`utctime(ticks=time.time())`

Alias for `gmtime()`.

3.2 DateTime Object Methods

A `DateTime` instance has the following methods. Note that the calendar setting of the instance effects all methods relying on date values.

`.COMDate()`

Returns a float float representing the instances value as COM date (see above).

`.Format(format_string="%c")`

This is just an alias for `.strftime()` to make the type compatible to other date/time types.

`.Julian()`

Returns a DateTime instance pointing to the same point in time but using the Julian calendar.

`.Gregorian()`

Returns a DateTime instance pointing to the same point in time but using the Gregorian calendar.

`.absvalues()`

Returns the instances value as tuple `(absdate, abstime).`

`.gmticks(offset=0.0)`

Returns a float representing the instances value in ticks (see above).

The conversion routine assumes that the stored date/time value is given in *UTC time*. `offset` is subtracted from the resulting value.

The method raises an `RangeError` exception if the objects value does not fit into the system's ticks range.

`.gmtime()`

Assuming that the instance refers to local time, this method returns new DateTime instance holding the corresponding UTC value.

`.gmtoffset()`

Returns a DateTimeDelta instance representing the UTC offset for the instance assuming that the stored values refer to local time. This is also sometimes called timezone.

The UTC offset is defined as: local time - UTC time, e.g. it is negative in the US and positive in eastern Europe and Asia.

`.localtime()`

Assuming that the instance refers to UTC time, this method returns new DateTime instance holding the corresponding local time value.

`.pydate()`

Returns a Python datetime.date object with just the date values taken from the DateTime object.

New in mxDateTime 3.2.

`.pydatetime()`

Returns a Python datetime.datetime object with the same values as the DateTime object.

New in mxDateTime 3.2.

`.pytime()`

Returns a Python datetime.time object with just the time values taken from the DateTime object.

New in mxDateTime 3.2.

`.strftime(format_string="%c")`

Format the instances value as indicated by the format string.

This is the same function as the one in the `time` module. For further information please refer to the C library documentation for `strftime()` or the Python reference manual.

Note: `strftime()` and `strptime()` try to be the inverse of each other. The output from `strftime()` given to `strptime()` together with the format string passed to `strftime()` will in most cases give you a DateTime instance referring to the same date and time.

Time zone information is *not* available. Use the instance variable `.tz` instead.

`.rebuild(year=None,month=None,day=None,hour=None,minute=None,second=None)`

Returns a new DateTime object copying the attributes of the current object and replacing the attributes given as keyword arguments with new values.

This is useful to e.g. convert a DateTime object into one which maps to midnight on the same day, or one which has the `.seconds` attribute rounded to milliseconds.

`.ticks(offset=0.0,dst=-1)`

Returns a float representing the instances value in ticks (see above).

The conversion routine assumes that the stored date/time value is given in *local time*.

The given value for `dst` is used by the conversion (0 = DST off, 1 = DST on, -1 = unkown) and `offset` is subtracted from the resulting value.

The method raises a `RangeError` exception if the objects value does not fit into the system's ticks range.

Note: On some platforms the C lib's `mktime()` function that this method uses does not allow setting DST to an arbitrary value. The module checks for this and raises a `SystemError` in case setting DST to 0 or 1 does not result in valid results.

`.timetuple()`

Alias for `.tuple()` needed for compatibility with Python's datetime module.

`.tuple()`

Returns the instances value as `time.localtime()` (all integers) tuple.

DST is set assuming local time. It can also be -1, meaning that the information is not available.

`.weekday()`

Returns the day of the week as integer. This is an alternative interface for the `.day_of_week` attribute and needed for compatibility with Python's datetime module.

3.3 DateTime Object Attributes

To make life easier, the instances also provide a more direct interface to their stored values (these are all read-only). Note that the calendar setting of the instance effects all attributes referring to date values.

`.hour, .minute, .second`

Return the indicated values in their standard ranges.

Note that in a future release, leap seconds may also be considered and thus second has a range of 0-60.

`.year, .month, .day`

Return the indicated values in their standard 1-based ranges.

`.date, .time`

Returns the ISO representation of the date part as string. The format is `'[-]YYYY-MM-DD'`.

`.absdate`

Returns the absolute date as used by the instance.

`.absdays`

Returns the absolute date and time of the object converted to a Python float representing absolute days (days since the epoch).

The value is calculated using a 86400.0 seconds/day basis and does not account for leap seconds. This value is handy if you need the date/time value stored in one number. By using a Python float, which is mapped to a C double internally, the accuracy should give a fairly large range of valid dates.

`.abstime`

Returns the absolute time as used by the instance.

`.days_in_month`

Returns the number of days in the object's month.

`.day_of_week`

Returns the day of the week. Monday is returned as 0.

`.day_of_year`

Returns the day of the year; 1.1. is returned as 1.

`.dst`

Integer indicating whether DST is active (1) or not (0) or cannot be determined (-1).

The value is calculated assuming that the stored value is local time.

`.calendar`

Calendar used by the instance. This can either be the constant `Julian` or `Gregorian`.

`.is_leapyear`

Returns 1 iff the instances value points to a leap year in the Gregorian calendar.

`.iso_week`

Returns a tuple `(year,isoweek,isoday)` signifying the *ISO week notation* for the date the object points to.

Note: isoday 1 is Monday !

`.jdn`

Returns a float representing the instance's value as *Julian Day Number* (Julian Day Number 0 starts at 12:00 UTC on 1 January 4713 BC and ends 24 hours later at noon on 2 January 4713 BC).

It is assumed for the calculation that the stored value is given in UTC. Fractions indicate parts of the full day, e.g. JDN 2451170.17393 referrs to Tue, 22 Dec 1998 16:10:27 UTC.

See the *Calendar FAQ* for details.

`.mjd`

Returns a float representing the instance's value in terms of *Modified Julian Days* (1858-11-17 00:00:00.00 UTC being Modified Julian Day 0).

It is assumed for the calculation that the stored value is given in UTC. Fractions indicate parts of the full day, e.g. 0.5 referrs to noon on the 17 November 1858.

See the *Calendar FAQ* or *Systems of Time* for details.

Note: Usage of MJD notation is discouraged by the International Astronomical Union (IAU). Use JDN instead.

`.time`

Returns the ISO representation of the time part as string. The format is `'HH:MM:SS.ss'` with `ss` being the truncated fraction of the seconds value.

`.tjd`

Returns a float representing the instance's value in terms of *Truncated Julian Days* (TJD).

TJDs are calculated using 00:00 UTC on 1 January 4713 BC as epoch, counting the number of days as for the Julian Day Numbers and then omitting the myriad part (div 10000) from it. As a result the TJD will always have at most 4 digits. The divisor is available through the `tjd_myriad` attribute.

It is assumed for the calculation that the stored value is given in UTC. Fractions indicate parts of the full day.

Some people claim that this term is also known under the name *Star Date*. Remember ? ... `"Captain's Log, Star Date 8143.65"`. I wonder which myriad these dates refer to.

`.tjd_myriad`

Returns the truncated part of the TJD representation.

`.tz`

Returns the time zone string, assuming local time, or `'???'` if the information is not available.

`.yearoffset`

Returns the absolute date of the 31.12. in the year before the instance's year.

4. mx.DateTime.DateTimeDelta Object

DateTimeDelta objects provide a way to define an absolute time-span.

Time of day, as we commonly refer it, usually refers to a fixed fraction of a day, measured from the start of the day (typically midnight). DateTimeDelta objects abstract this notion to arbitrary time differences between two absolute points in the date/time continuum.

4.1 DateTimeDelta Object Constructors

Several constructors are available:

`DateTimeDelta(days[,hours=0.0,minutes=0.0,seconds=0.0])`

Returns a new DateTimeDelta instance for the given time delta.

This is the standard constructor for DateTimeDelta instances.

The internal value is calculated using the formula `days*86400.0 + hours*3600.0 + minutes*60.0 + seconds`. Keep this in mind when passing negative values to the constructor.

The following constructors are provided to simplify integration with existing code.

`DateTimeDeltaFrom(*args,**kws)`

Constructs a DateTimeDelta instance from the arguments.

This constructor can parser strings, handle numeric arguments, Python `datetime.time` and `datetime.timedelta` objects, and knows about the keywords `year,month,day,hour,minute,second`.

It uses type inference to find out how to interpret the arguments and makes use of the Parser module.

`DateTimeDeltaFromDays(days)`

Constructs a DateTimeDelta instance from the given `days` value. It can be given as float.

The internal value is calculated using a 86400.0 seconds/day basis.

`DateTimeDeltaFromSeconds(seconds)`

Constructs a DateTimeDelta instance from the given `seconds` value. It can be given as float.

`Time(hour,minute=0.0,second=0.0)`

Is just another name binding for `TimeDelta()`.

`TimeDelta(hour=0.0,minute=0.0,second=0.0)`

Constructs a DateTimeDelta instance from the given values.

The internal value is calculated using the formula `hours * 3600 + minutes * 60 + seconds`. Keep this in mind when passing negative values to the constructor.

The constructor allows usage of keywords, e.g. Time(seconds=1.5) works.

`TimeDeltaFrom(*args,**kws)`

Constructs a DateTimeDelta instance from the arguments.

The interface is the same as for `DateTimeDeltaFrom()` with the exception that numeric arguments are interpreted without day part as for the `TimeDelta()` constructor.

`TimeFrom(*args,**kws)`

Alias for `TimeDeltaFrom()`.

`TimeFromTicks(ticks)`

Constructs a DateTimeDelta instance pointing to the local time indicated by the given ticks value. The date part is ignored.

4.2 DateTimeDelta Object Methods

A `DateTimeDelta` instance has the following methods:

`.absvalues()`

Returns a `(absdays, absseconds)` tuple.

The `absseconds` part is normalized in such way that it is always smaller than 86400.0. Both values are signed.

`.pytime()`

Returns a Python datetime.time object with the same values as the DateTimeDelta object.

Raises a `ValueError` in case the DateTimeDelta object has a non-zero `.day` set.

New in mxDateTime 3.2.

`.pytimedelta()`

Returns a Python datetime.timedelta object with the same values as the DateTimeDelta object.

New in mxDateTime 3.2.

`.rebuild(day=None,hour=None,minute=None,second=None)`

Returns a new DateTimeDelta object copying the attributes of the current object and replacing the attributes given as keyword arguments with new values.

This is useful to e.g. convert a DateTimeDelta object into one which has the `.seconds` attribute rounded to milliseconds.

`.strftime(format_string)`

Format the instance's value as indicated by the format string.

This is the same function as the one in the `time` module. For further information please refer to the Python reference manual.

Since some descriptors don't make any sense for date/time deltas these return undefined values. Only the fields hour, minute, seconds and day are set according to the objects value (the descriptors `%d %H %M %S %I %p %X` work as expected).

Negative values show up positive -- you'll have to provide your own way of showing the sign (the `seconds` instance variable is signed).

`.tuple()`

Returns the instance's value as `(day,hour,minute,second)` (all integers) tuple.

The values are the same those returned by the attributes of the same name.

4.3 DateTimeDelta Object Attributes

To make life easier, the instances also provide a more direct interface to their stored values (these are all read-only):

`.day, .hour, .minute, .second`

Return the indicated values in their standard ranges. The values are negative for negative time deltas.

`.days, .hours, .minutes, .seconds`

Return the internal value of the object expressed as float in the resp. units, e.g. `TimeDelta(12,00,00).days == 0.5`.

5. mx.DateTime.RelativeDateTime Object

RelativeDateTime objects provide a way to abstract date/time differences using date/time characteristics.

They are a mix of both absolute and relative date/time settings which makes it very easy to define more complicated date/time relationships between two absolute points in the date/time continuum.

RelativeDateTime objects are typically used to define recurrences of events, age or schedules bound to a certain day of the week, week or day of a month.

5.1 RelativeDateTime Constructors

These constructors are available:

```
RelativeDateTime(years=0,months=0,days=0, year=0,month=0,day=0,
   hours=0,minutes=0,seconds=0, hour=None,minute=None,second=None,
   weekday=None,weeks=0)
```

Returns a RelativeDateTime instance for the specified relative time.

This is the standard constructor for RelativeDateTime instances.

The constructor handles keywords, so you'll only have to give those parameters which should be changed when you add the relative to an absolute DateTime instance.

Do not pass arguments directly, always use the keyword notation !

Absolute values passed to the constructor will override delta values of the same type. Note that `weeks` is added to `days` so that the instances days values will be `days + 7*weeks`.

weekday must be a 2-tuple if given: (day_of_week, nth). The value is applied after all other calculations have been done resulting in moving the date to the nth weekday in the month that the date points to. Negative values for nth result in the ordering of the month's weekdays to be reversed, e.g. (Monday,-1) will move to the last Monday in that month. Setting nth to 0 results in the date's week to be used as reference, e.g (Tuesday,0) will move to Tuesday that week (which could lie in a different month). weekday is considered an absolute value, so multiplication or negation will not touch it.

The following constructors are provided to simplify integration with existing code.

`Age(date1,date2)`

Is another name binding for `RelativeDateTimeDiff()`.

`RelativeDate(years=0,months=0,days=0, year=0,month=0,day=0, weeks=0)`

Is another name binding for RelativeDateTime. Do not pass arguments directly, always use the keyword notation !

`RelativeDateDiff(date1,date2)`

Is another name binding for `RelativeDateTimeDiff()`.

`RelativeDateFrom(*args,**kws)`

Is another name binding for `RelativeDateTime()`.

Note that in future versions this constructor may explicitly ignore the time parts.

`RelativeDateTimeDiff(date1,date2)`

Returns a RelativeDateTime instance representing the difference between date1 and date2 in relative terms. The following should hold: `date2 + RelativeDateDiff(date1,date2) == date1` for all dates date1 and date2.

Note that due to the algorithm used by this function, not the whole range of DateTime instances is supported; there could also be a loss of precision

This constructor is still *experimental*.

`RelativeDateTimeFrom(*args,**kws)`

Constructs a RelativeDateTime instance from the arguments.

This constructor can parse strings, handle numeric arguments and knows about the same keywords as the `RelativeDateTime()` constructor.

It uses type inference to find out how to interpret the arguments and makes use of the Parser module.

`RelativeTimeFrom(*args,**kws)`

Is another name binding for `RelativeDateTime()`.

Note that in future versions this constructor may explicitly ignore the date parts.

5.2 RelativeDateTime Object Methods

`RelativeDateTime` instances currently don't have any instance methods.

5.3 RelativeDateTime Object Attributes

The following attributes are exposed, but should not be written to directly (the objects are currently implemented in Python, but that could change in future releases).

`.year, .month, .day, .hour, .minute, .second, .weekday`

Absolute values of the instance.

`.years, .months, .days, .hours, .minutes, .seconds`

Relative values of the instance.

The given values are only defined in case they were set at instance creation time.

5.4 RelativeDateTime Object Usage

RelativeDateTime objects store the given settings (plural nouns meaning deltas, singular nouns absolute values) and apply them when used in calculations. Delta values will have the effect of changing the corresponding attribute of the involved absolute DateTime object accordingly, while absolute values overwrite the DateTime objects attribute value with a new one. The effective value of the object is thus determined at calculation time and depends on the context it is used in.

Adding and subtracting RelativeDateTime instances is supported with the following rules: deltas will be added together and right side absolute values override left side ones.

Multiplying RelativeDateTime instances with numbers will yield instances with scaled deltas (absolute values are not effected).

Adding RelativeDateTime instances to and subtracting RelativeDateTime instances from DateTime instances will return DateTime instances with the appropriate calculations applied, e.g. to get a DateTime instance for the first of next month, you'd call `now() + RelativeDateTime(months=+1, day=01)`.

Note that dates like `Date(1999,1,30) + RelativeDateTime(months=+1)` are not supported. The package currently interprets these constructions as `Date(1999,2,1) + 30`, thus giving the 1999-03-02 which may not be what you'd expect (this may be changed in a future version of mxDateTime to raise an exception).

When providing both delta and absolute values for an entity the absolute value is set first and then the delta applied to the outcome.

In tests, RelativeDateTime instances are false in case they do not define any date or time alterations and true otherwise.

RelativeDateTime instances are hashable and can also be compared for equality. Other comparisons are currently not possible.

A few examples will probably make the intended usage clearer:

```
>>> from mx.DateTime import *

>>> print now()
1998-08-11 16:46:02.20

# add one month
>>> print now() + RelativeDateTime(months=+1)
1998-09-11 16:46:24.59

# add ten months
>>> print now() + RelativeDateTime(months=+10)
1999-06-11 16:47:03.07

# ten days from now
>>> print now() + RelativeDateTime(days=+10)
1998-08-21 16:47:10.58

# first of next month
>>> print now() + RelativeDateTime(months=+1,day=1)
1998-09-01 16:47:25.15

# first of this month, same time
>>> print now() + RelativeDateTime(day=1)
1998-08-01 16:47:35.48

# first of this month at midnight
>>> print now() + RelativeDateTime(day=1,hour=0,minute=0,second=0)
1998-08-01 00:00:00.00

# next year, first of previous month, same time
>>> print now() + RelativeDateTime(years=+1,months=-1,day=1)
1999-07-01 16:48:31.87

# Last Sunday in October 1998
>>> print Date(1998) + RelativeDateTime(weekday=(Sunday,-
1),month=10)
1998-10-25 00:00:00.00
```

```
# The result in ARPA notation:
>>> print ARPA.str(Date(1998) + RelativeDateTime(weekday=(Sunday,-
1),month=10))
Sun, 25 Oct 1998 00:00:00 +0200

# Generic way of specifying "next tuesday":
>>> NextTuesday = RelativeDateTime(days=+6,weekday=(Tuesday,0))
```

6. mx.DateTime Functions

The package defines these additional functions:

`cmp(obj1,obj2,accuracy=0.0)`

Compares two DateTime[Delta] objects.

If accuracy is given, then equality will result in case the absolute difference between the two values is less than or equal to accuracy.

`local2gm(datetime)`

Convert a DateTime instance holding local time to a DateTime instance using UTC time.

`local2utc(datetime)`

Alias for `local2gm()`.

`gmticks(datetime)`

Returns a ticks value for datetime assuming the stored value is given in UTC.

DEPRECATED: Use the `.gmticks()` method instead.

`gm2local(datetime)`

Convert a DateTime instance holding UTC time to a DateTime instance using local time.

`tz_offset(datetime)`

Returns a DateTimeDelta instance representing the UTC offset for datetime assuming that the stored values refer to local time. If you subtract this value from datetime, you'll get UTC time.

DEPRECATED: Use the `.gmtoffset()` method instead.

`utcticks(datetime)`

Alias for `gmticks()`.

DEPRECATED: Use the `.gmticks()` method instead.

`utc2local(datetime)`

Alias for `gm2local()`.

7. mx.DateTime Constants

The package defines these constants:

`DateTimeType, DateTimeDeltaType`

The type objects for the two types.

`Epoch`

A DateTime instance pointing to the Christian Epoch, i.e. 0001-01-01 00:00:00.00.

`Error, RangeError`

These are the exception objects. Exceptions will normally only be raised by functions, methods or arithmetic operations. `RangeError` is a subclass of Error. Error is subclass of Python's standard `ValueError`.

`Gregorian, Julian`

The objects returned by `calendar` attribute of DateTime objects. Currently these are the strings 'Gregorian' and 'Julian', but this might change in future versions: always use these objects for checking the calendar type.

`January, February, March, April, May, June, July, August, September, October, November, December`

Months encoded as integers. January maps to 1, February to 2 and so on.

`MaxDateTime, MinDateTime, MaxDateTimeDelta, MinDateTimeDelta`

These constants define the accepted ranges for the basic types. The values depend on the ranges of C `longs` on your platform.

`Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday`

Weekdays encoded as integers. Monday maps to 0, Tuesday to 1 and so on.

`Month`

Mapping that maps months to integers and integers to months. January maps to 1, February to 2 and so on.

`POSIX`

Constant stating the POSIX compatibility of the system with respect to Unix ticks.

If the system's time package uses POSIX `time_t` values (without counting leap seconds), it is set to 1. In case the system's ticks values include leap seconds and thus correctly represent the term "seconds since the epoch", the constant is set to 0.

`Weekday`

Mapping that maps weekdays to integers and integers to weekdays. Monday maps to 0, Tuesday to 1 and so on.

`mxDateTimeAPI`

The C API wrapped by a C object. See mxDateTime.h for details.

`now_resolution`

If available, this attribute is a floating point number stating the resolution of the clock used by `now()` in seconds as reported by the system. The actual resolution may be lower than reported by this attribute. It is not available, if the time module emulation has to be used by mxDateTime in order to determine the current time.

`oneWeek, oneDay, oneHour, oneMinute, oneSecond`

Are set to the indicated values wrapped into DateTimeDelta instances.

8. Date/Time Arithmetic

The three objects DateTime, DateTimeDelta and RelativeDateTime can be used to do simple date/time arithmetic. Addition and subtraction are supported and result in the expected results. In addition to handling arithmetic using only the two types, mixed arithmetic with numbers is also understood to a certain extent.

Note that the datetime module mentioned here was added to Python in version 2.3. It is not available in earlier versions.

Argument 1	*Argument 2*	*Result*

DateTime object v	DateTime object w	`v - w`
		returns a DateTimeDelta object representing the time difference.
		`v + w`
		is not defined and raises a TypeError.
		`v cmp w`
		compares the two values.

DateTime object v	A number w	`v - w`
		returns a new DateTime object with a date/time decremented by `w` days (floats can be used to indicate day fractions).
		`v + w`
		adds `w` days to the DateTime object `v` and returns a new DateTime object.
		`w + v`
		works in the same way.

Argument 1	*Argument 2*	*Result*

		Note: you can use the object `oneDay` to get similar effects in a more intuitive way.
		`w - v`
		is not defined and returns a TypeError.
		`v cmp w`
		Converts `v` to Unix ticks and returns the result of comparing the ticks value to the number `w`. Note: the ticks conversion assumes that the stored value is given in local time.

DateTime object v	DateTimeDelta object w	`v - w`
		returns a new DateTime object with a date/time decremented by `w`'s value.
		`v + w` and `w + v`
		return a new DateTime object with a date/time incremented by `w`'s value.
		`w - v`
		is not defined and raises a TypeError.

DateTime object v	RelativeDateTime object w	`v + w`
		returns a new DateTime object with a date/time adjusted according to `w`'s settings;
		`v - w`
		works accordingly.

DateTime object v	datetime.delta object w	`v - w`

Argument 1	*Argument 2*	*Result*
		returns a new DateTime object with a date/time adjusted according to `w`'s settings; `v + w` works accordingly.

DateTime object v	datetime.date or datetime.datetime object w	`v - w`
		returns a DateTimeDelta object representing the time difference; datetime.date values are interpreted as referring to midnight that day;
		`v + w`
		is not defined.

RelativeDateTime object v	A number w	`v * w`
		returns a new RelativeDateTime object with all deltas multiplied by `float(w)` (`w * v` works in the same way);
		`v / w`
		returns a new RelativeDateTime object with all deltas divided by `float(w);`

DateTimeDelta object v	A number w	`v - w`
		returns a new DateTimeDelta object with a time delta value decremented by `w` seconds (can be given as float to indicate fractions of a second);
		`v + w`
		works accordingly;
		Note: you can use the object

Argument 1	*Argument 2*	*Result*

		oneSecond to get similar effects in a more intuitive way;
		`v * w`
		returns a new DateTimeDelta object with a time delta value multiplied by `float(w)` (`w * v` works in the same way);
		`v / w`
		returns a new DateTimeDelta object with a time delta value divided by `float(w);`
		`v cmp w`
		Converts `v` to a signed float representing the delta in seconds and returns the result of comparing the seconds value to the number `w`.

DateTimeDelta object v	DateTimeDelta object w	`v + w`
		returns a new DateTimeDelta object for the sum of the two time deltas (`(v+w).seconds == v.seconds + w.seconds`);
		`v - w`
		works accordingly;
		`v / w`
		returns a float equal to `v.seconds / w.seconds`.

DateTimeDelta object v	datetime.timedelta or datetime.time w	`v + w` returns a new DateTimeDelta object for the sum of the two time

Argument 1	*Argument 2*	*Result*

		deltas: `(v+w).seconds == v.seconds + w.seconds` (for datetime.time objects, the number of seconds since 00:00:00 is used);
		`v - w`
		works accordingly;
		`v op w`
		work based on the seconds deltas (for datetime.time objects, the number of seconds since 00:00:00 is used).

8.1 Notes:

- Operation and argument order are often important because of the different ways operations are implemented. Use parenthesis to make your intent clear or you will get unwanted results.

- In mixed type operations, the mxDateTime implementation will generally try to return mxDateTime objects.

- Some mixed type combinations don't work as expected due to flaws in the Python datetime module implementation, e.g. `datetime.timedelta < DateTimeDelta` doesn't work due to the timedelta object raising an exception instead of returning `NotImplemented`, allowing the DateTimeDelta to handle the operation.

- Comparing RelativeDateTime instances does not work.

- Operations on DateTime instances cause the result to inherit the calendar of the left operand.

9. mxDateTime Submodules

The package provides additional features in form of the following submodules. All submodules are imported on request only.

The submodules are all available via the package namespace `mx.DateTime`, e.g. as `mx.DateTime.ISO`.

9.1 mx.DateTime.ISO Submodule

The ISO submodule is intended to provide interfacing functions to *ISO 8601 date and time representations* (the ISO document is also available as *PDF file*). The most common format is:

```
YYYY-MM-DD HH:MM:SS[+-HH:MM]
```

Note: *timezone information* (`+-HH:MM`) is only interpreted by the `ParseDateTimeUTC()` constructor. All others ignore the given offset and store the time value as-is.

You can access the functions and symbols defined in the submodule through `DateTime.ISO` -- it is imported on demand.

9.1.1 Constructors & Functions

The module defines these constructors and functions:

`DateTime(), Time(), TimeDelta()`

Aliases for the constructors you find in DateTime. Just included for completeness, since these also use ISO style notation for their argument order.

`ParseAny(isostring)`

Returns a DateTime[Delta] instance reflecting the given ISO date and/or time. All ISO formats supported by the module are understood by this constructor.

`ParseDate(isostring)`

Returns a DateTime instance reflecting the given ISO date. Year must be given, month and day default to 1. A time part may not be included.

`ParseDateTime(isostring)`

Returns a DateTime instance reflecting the given ISO date.

A time part is optional and must be delimited from the date by a space or 'T'. Year must be given, month and day default to 1. For the time part, hour and minute must be given, while second defaults to 0.

Time zone information is parsed, but not evaluated.

`ParseDateTimeGMT(isostring)`

Same as `ParseDateTime()` except that timezone information is used to calculate and return the date/time value in UTC.

Note: UTC is practically the same as GMT, the old time standard.

`ParseDateTimeUTC(isostring)`

Alias for `ParseDateTimeGMT()`.

Note: UTC is practically the same as GMT, the old time standard.

`ParseTime(isostring)`

Returns a DateTimeDelta instance reflecting the given ISO time. Hours and minutes must be given, seconds are optional and default to 0. Fractions of a second may also be used, e.g. `'12:23:12.34'`.

`ParseTimeDelta(isostring)`

Returns a DateTimeDelta instance reflecting the given ISO time as delta. Hours and minutes must be given, seconds are optional and default to 0. Fractions of a second may also be used, e.g. `'12:23:12.34'`. In addition to the ISO standard a sign may be prepended to the time, e.g. `'-12:34'`.

`ParseWeek(isostring)`

Returns a DateTime instance reflecting the given ISO date. Year must be given, week number and day are optional and default to 1. A time part may not be included.

`ParseWeekTime(isostring)`

Returns a DateTime instance reflecting the given ISO date. Year must be given, week number and day are optional and default to 1. A time part may not be included.

`Week(year,isoweek,isoday=1)`

Alias for `WeekTime()`.

`WeekTime(year,isoweek=1,isoday=1,hour=0,minute=0,second=0.0)`

Returns a DateTime instance pointing to the given *ISO week and day*. `isoday` defaults to 1, which corresponds to Monday in the ISO numbering. Note that the resulting date can in fact lie in the year before the one given as parameter, e.g. `Week(1998,1,1)` points to the date 1997-12-29. The DateTime instance variable `iso_week` provides an inverse to this function.

`str(datetime)`

Returns the datetime instance as standard ISO date string (omitting the seconds fraction and always adding timezone information). The function assumes that the stored value is given in local time and calculates the correct timezone offset accordingly.

`strGMT(datetime)`

Returns the datetime instance as ISO date string assuming it is given in UTC.

`strUTC(datetime)`

Alias for `strGMT()`.

Notes

The parsing routines strip surrounding whitespace from the strings, but are strict in what they want to see. Additional characters are not allowed and will cause a `ValueError` to be raised.

Timezone information may be included, but will not be interpreted unless explicitly stated.

The parsing routines also understand the ISO 8601 date/time formats without separating dashes and colons, e.g. '19980102T142020', and mixtures of both notations.

9.1.2 ISO 8601 string formats and DateTime[Delta] instances

DateTime and DateTimeDelta instances use a slightly enhanced ISO format for string representation:

DateTime instances are converted to `'YYYY-MM-DD HH:MM:SS.ss'` where the last `ss` indicate hundredths of a second (ISO doesn't define how to display these).

DateTimeDelta instances use `'[-][DD:]HH:MM:SS.ss'` as format, where DD: is only shown for deltas spanning more than one day (24 hours). The `ss` part has the same meaning as for DateTime instances: hundredths of a second. A minus is shown for negative deltas. ISO does not define relative time deltas, but the time representation is allowed to be `'HH:MM:SS'`.

9.2 mx.DateTime.ARPA Submodule

The ARPA submodule is intended to provide interfacing functions to ARPA date representations. These are used throughout the Internet for passing around mails, postings, etc. The format is very simple:

```
[Day, ]DD Mon YYYY HH:MM[:SS] ZONE
```

where `ZONE` can be one of these: MDT, O, EDT, X, Y, CDT, UT, AST, GMT, PST, Z, V, CST, ADT, I, W, T, U, R, S, P, Q, N, EST, L, M, MST, K, H, E, F, G, D, PDT, B, C, UTC, A (the single letter ones being *military time zones*).

Use of explicit time zone names other than UTC and GMT is deprecated, though. The better alternative is providing the offset from UTC being in effect at the given local time: `+-HHMM` (this is the offset you have to subtract from the given time in order to get UTC).

You can access the functions and symbols defined in the submodule through `DateTime.ARPA` -- it is imported on demand.

9.2.1 Constructors & Functions

The module defines these constructors and functions:

`ParseDate(arpastring)`

Returns a DateTime instance reflecting the given ARPA date. Any time part included in the string is silently ignored.

`ParseDateTime(arpastring)`

Returns a DateTime instance reflecting the given ARPA date assuming it is local time (timezones are silently ignored).

`ParseDateTimeGMT(arpastring)`

Returns a DateTime instance reflecting the given ARPA date converting it to UTC (timezones are honored).

`ParseDateTimeUTC(arpastring)`

Alias for `ParseDateTimeGMT()`. Note: UTC is practically the same as GMT, the old time standard.

`str(datetime,tz=DateTime.tz_offset(datetime))`

Returns the datetime instance as ARPA date string. `tz` can be given as DateTimeDelta instance providing the time zone difference from datetime's zone to UTC. It defaults to `mx.DateTime.tz_offset(datetime)` which assumes local time.

`strGMT(datetime)`

Returns the datetime instance as ARPA date string assuming it is given in GMT using the 'GMT' timezone indicator.

Note: Most Internet software expects to find 'GMT' and not 'UTC'.

`strUTC(datetime)`

Returns the datetime instance as ARPA date string assuming it is given in UTC using the 'UTC' timezone indicator.

Notes

The parsing routines strip surrounding whitespace from the strings. Additional characters *are* allowed (because some mail apps add extra information to the date header).

9.3 mx.DateTime.Feasts Submodule

The Feasts submodule is intended to provide easy-to-use constructors for common moveable Christian feasts that can be deduced from the date of Easter Sunday. The algorithm used to calculate Easter Sunday is based on the one presented in the *Calendar FAQ* by Claus Tondering, which in return is based on the algorithm of Oudin (1940) as quoted in "Explanatory Supplement to the Astronomical Almanac", P. Kenneth Seidelmann, editor.

9.3.1 Constructors & Functions

The module defines these constructors and functions:

`EasterSunday(year), Ostersonntag(year), DimanchePaques(year)`

Returns a DateTime instance pointing to Easter Sunday in the given year at midnight.

The other feasts are deduced from this date and all use the same interface. The module defines these sets of constructors the return the corresponding DateTime instance for midnight of the implied day:

`Ascension(year), Himmelfahrt(year)`

`AshWednesday(year), Aschermittwoch(year), MercrediCendres(year)`

`CarnivalMonday(year), Rosenmontag(year)`

`CorpusChristi(year), Fronleichnam(year), FeteDieu(year)`

For further reading, have a look at the *Ecclesiastical Calendar*.

`EasterMonday(year), Ostermontag(year), LundiPaques(year)`

`EasterFriday(year), GoodFriday(year), Karfreitag(year), VendrediSaint(year)`

`MardiGras(year)`

`PalmSunday(year), Palmsonntag(year), DimancheRameaux(year)`

`Pentecost(year), WhitSunday(year), Pfingstsonntag(year), DimanchePentecote(year)`

`TrinitySunday(year)`

`WhitMonday(year), Pfingstmontag(year), LundiPentecote(year)`

9.4 mx.DateTime.Parser Submodule

The Parser submodule provides constructors for DateTime[Delta] values taking a string as input. The module knows about quite a few different date and time formats and will try very hard to come up with a reasonable output given a valid input.

Date/time parsing is a very difficult field of endeavor and that's why the exact definition of what the module can parse and what not is defined by implementation rather than a rigorous set of formats.

Things the module will recognize are the outputs of ISO, ARPA and the `.strftime()` method. Currently only English, German, French, Spanish and Portuguese month and day names are supported. Have a look at the source code (`Parser.py`) for a full list of compatible date/time formats.

9.4.1 Constructors & Functions

The module defines these constructors and functions:

`DateFromString(text[, formats, defaultdate])`

Returns a DateTime instance reflecting the date given in text. A possibly included time part is ignored; the time part is always set to 0:00:00.00.

`formats` and `defaultdate` work just like for `DateTimeFromString()`.

`DateTimeDeltaFromString(text)`

Returns a DateTimeDelta instance reflecting the delta given in text. Defaults to 0:00:00:00.00 for parts that are not included in the textual representation or cannot be parsed.

`DateTimeFromString(text[, formats, defaultdate, time_formats])`

Returns a DateTime instance reflecting the date and time given in text. In case a timezone is given, the returned instance will point to the corresponding UTC time value. Otherwise, the value is set as given in the string.

`formats` may be set to a tuple of strings specifying which of the following parsers to use and in which order to try them. Default is to try all of them in the order given below:

'euro' - the European date parser

'us' - the US date parser

'altus' - the alternative US date parser (with '-' instead of '/')

'iso' - the ISO date parser

'altiso' - the alternative ISO date parser (without '-')

'usiso' - US style ISO date parser (yyyy/mm/dd)

'lit' - the US literal date parser

'altlit' - the alternative US literal date parser

'eurlit' - the Eurpean literal date parser

'unknown' - if no date part is found, use defaultdate

`defaultdate` provides the defaults to use in case no date part is found. Most other parsers default to the current year January 1 if some of these date parts are missing.

If `'unknown'` is not given in formats and the date/time cannot be parsed, a `ValueError` is raised.

`time_formats` may be set to a tuple of strings specifying which of the following parsers to use for parsing the time part and in which order to try them. Default is to try all of them in the order given below:

'standard' - standard time format with ':' delimiter

'iso' - ISO time format (superset of 'standard')

'unknown' - default to 00:00:00 in case the time format cannot be parsed

Defaults to 00:00:00.00 for parts that are not included in the textual representation.

`TimeDeltaFromString(text)`

Alias for `DateTimeDeltaFromString()`.

`TimeFromString(text, [formats])`

Returns a DateTimeDelta instance reflecting the time given in text. A possibly included date part is ignored.

formats may be set to a tuple of strings specifying which of the following parsers to use and in which order to try them. Default is to try all of them in the order given below:

'standard' - standard time format with ':' delimiter

'iso' - ISO time format (superset of 'standard')

'unknown' - default to 00:00:00 in case the time format cannot be parsed

Defaults to 00:00:00.00 for parts that are not included in the textual representation.

`RelativeDateFromString(text)`

Same as `RelativeDateTimeFromString(text)` except that only the date part of `text` is taken into account.

`RelativeDateTimeFromString(text)`

Returns a RelativeDateTime instance reflecting the relative date and time given in text.

Defaults to wildcards (None or 0) for parts or values which are not included in the textual representation or cannot be parsed.

The format used in text must adhere to the following ISO-style syntax:

```
[YYYY-MM-DD] [HH:MM[:SS]]
```

with the usual meanings.

Values which should not be altered may be replaced with '*', '%', '?' or any combination of letters, e.g. 'YYYY'. Relative settings must be enclosed in parenthesis if given and should include a sign, e.g. '(+0001)' for the year part. All other settings are interpreted as absolute values.

Date and time parts are both optional as a whole. Seconds in the time part are optional too. Everything else (including the hyphens and colons) is mandatory.

`RelativeTimeFromString(text)`

Same as `RelativeDateTimeFromString(text)` except that only the time part of `text` is taken into account.

The parsing routines ignore surrounding whitespace. Additional characters and symbols are ignored.

9.5 mx.DateTime.NIST Submodule

The NIST submodule is useful when you are connected to the Internet and want access to the **accurate world standard time**, the NIST atomic clocks.

The module accesses a *special service* provided by NIST and other partner organizations, which allows anyone with Internet access to query the current UTC time. Of the three provided protocols, daytime, time and ntp, we chose the daytime protocol because of its simplicity and robustness.

Since access through the Internet can be slow, the module also provides a way to calibrate itself and then use the computer's clock without the need to go across the Internet for every call to the current time constructors. The defaults are set in such a way that calibration occurs without further interaction on part of the programmer. See the code for details.

9.5.1 Constructors & Functions

The module defines these constructors and functions:

`calibrate(iterations=20)`

Calibrates the `localtime()` and `gmtime()` functions supplied in this module (not the standard ones in DateTime !).

Uses the NIST time service as time base. The computer must have an active internet connection to be able to do calibration using the NIST servers.

iterations sets the number of rounds to be done.

Note: This function takes a few seconds to complete. For long running processes you should recalibrate every now and then because the system clock tends to drift (usually more than the hardware clock in the computer).

`disable_auto_calibration()`

Turns auto calibration off.

`enable_auto_calibration()`

Currently an alias for `reset_auto_calibration()`.

`gmtime()`

Alias for `utctime()`.

`localtime(nist_lookup=0)`

Returns the current local time as DateTime instance.

Same notes as for `utctime()`.

`now()`

Alias for `localtime()`.

`reset_auto_calibration()`

Enables and resets the auto calibration for a new round.

This does not clear possibly available calibration information, so the two time APIs will continue to revert to the calibrated clock in case no connection to the NIST servers is possible.

Auto calibration is on per default when the module is imported.

`set_calibration(calibration_offset)`

Sets the calibration to be use by `localtime()` and `utctime()`.

This also sets the global `calibrated` to 1 and disables auto calibration.

`time_offset(iterations=10)`

Returns the average offset of the computer's clock to the NIST time base in seconds.

If you add the return value to the return value of `time.time()`, you will have a pretty accurate time base to use in your applications.

Note that due to network latencies and the socket overhead, the calculated offset will include a small hopefully constant error.

iterations sets the number of queries done to the NIST time base. The average is taken over all queries.

`utctime(nist_lookup=0)`

Returns the current UTC time as DateTime instance.

Works must like the standard `DateTime.now()`, but tries to use the NIST time servers as time reference -- not only the computer's built-in clock.

Note that the constructor may take several seconds to return in case no calibration was performed (see `calibrate()`). With calibration information, the computer's clock is used as reference and the offset to NIST time is compensated by the constructor.

In case the NIST service is not reachable, the constructor falls back to using either the calibrated (preferred) or uncalibrated computer's clock.

Setting `nist_lookup` to false (default) will cause the constructor to prefer the calibrated CPU time over the expensive Internet queries. If it is true, then Internet lookups are always tried first before using the local clock. A value of 2 will cause an `Error` (see below) to be raised in case the NIST servers are not reachable.

The constructor will use the received NIST information for auto calibration.

9.5.2 Constants

The package defines these constants:

`Error`

This exception is raised by the constructors in case no connection to the NIST service was possible.

`calibrated`

True in the global `calibration` contains valid information.

`calibrating`

If true, the module will Try to auto-calibrate itself whenever the NIST servers are reachable.

`calibration`

Current calibration offset (NIST - CPU time) in seconds.

9.5.3 Examples

There's an example called `AtomicClock.py` in the `Examples/` subdir which demonstrates how easy it is to turn your PC into a fairly accurate time piece.

For even better time accuracy, one would have to use NTP.

10. Examples of Use

For an example of how to use the two types to develop other date/time classes (e.g. ones that support time zones or other calendars), see the included `ODMG` module. It defines types similar to those of the ODMG standard.

Here is a little countdown script:

```
#!/usr/local/bin/python -u

""" Y2000.py - The year 2000 countdown.
"""
from mx.DateTime import *
from time import sleep

while 1:
    d = Date(2000,1,1) - now()
    print 'Y2000... time left: %2i days %2i hours '
     '%2i minutes %2i seconds\r' % \
     (d.day,d.hour,d.minute,d.second),
    sleep(1)
```

This snippet demonstrates some of the possible string representations for DateTime instances:

```
>>> from mx.DateTime import *

>>> ISO.str(now())
'1998-06-14 11:08:27+0200'

>>> ARPA.str(now())
'Sun, 14 Jun 1998 11:08:33 +0200'

>>> now().strftime()
'Sun Jun 14 11:08:51 1998'

>>> str(now())
'1998-06-14 11:09:17.82'
```

More examples are available in the `Examples` subdirectory of the package.

11. mx.DateTime Python C-API

mxDateTime exposes a C-API that can easily be used from other Python extensions. Please have look at the file `mxDateTime.h` for details.

11.1 Example

Interfacing is provided through a Python C object for ticks, `struct tm`, COM doubles, Python tuples and direct input either by giving absolute date/time or a broken down tuple. To access the module, do the following (note the similarities with Python's way of accessing functions from a module):

```
#include "mxDateTime.h"

...
    PyObject *v;

    /* Import the mxDateTime module */
    if (mxDateTime_ImportModuleAndAPI())
        goto onError;

    /* Access functions from the exported C API through
       mxDateTime */
    v = mxDateTime.DateTime_FromAbsDateAndTime(729376, 49272.0);
    if (!v)
        goto onError;

    /* Type checking */
    if (mxDateTime_Check(v))
        printf("Works.\n");

    Py_DECREF(v);
...
```

11.2 C API Import

Other Python modules can easily import the C API without having to link directly to mxDateTime. If another Python extension wants to use the API, it will have to #include "mxDateTime.h":

```
#include "mxDateTime.h"
```

This will define a global struct variable `mxDateTime` providing access to the interface functions which is initialized by calling an import function:

```
/* Import the mxDateTime module */
if (mxDateTime_ImportModuleAndAPI())
    goto onError;
```

After successful import, the functions are then available via the struct:

```
/* Access functions from the exported C API through
   mxDateTime */
v = mxDateTime.DateTime_FromAbsDateAndTime(729376, 49272.0);
if (!v)
    goto onError;
```

11.3 C API Definition

The following function entries are defined in the mxDateTime C API:

`typedef struct {`

`PyTypeObject *`**`DateTime_Type`**`;`

Type object for DateTime()

`PyObject *(*`**`DateTime_FromAbsDateAndTime`**`)(long absdate, double abstime);`

Construct a new object from the given absolute date and time.

Returns NULL in case of an error.

`PyObject *(*`**`DateTime_FromTuple`**`)(PyObject *v);`

Construct new object from Python 6-tuple (year,month,day,hour,minute,second).

Returns NULL in case of an error.

`PyObject *(*`**`DateTime_FromDateAndTime)`**`(long year, int month, int day, int hour, int minute, double second);`

Construct new object from year,month,day,hour,minute,second

Returns NULL in case of an error.

`PyObject *(*`**`DateTime_FromTmStruct`**`)(struct tm *tm);`

Construct new object from a given struct tm. DST, weekday and day of year are ignored.

Returns NULL in case of an error.

`PyObject *(*`**`DateTime_FromTicks`**`)(double ticks);`

Construct new object from the given ticks; these are first converted to a gmtime struct and this is then used as basis for the object value.

Note that you have to pass in the ticks value as double and not as time_t value (see the note below on this).

Returns NULL in case of an error.

```
PyObject *(*DateTime_FromCOMDate)(double comdate);
```

Construct new object from a given COM date double. This is the date/time standard used in the Microsoft COM interface.

Returns NULL in case of an error.

```
struct tm *(*DateTime_AsTmStruct)(mxDateTimeObject *datetime,
   struct tm *tm);
```

Fill the given `struct tm` with the object's value.

Seconds are truncated before assigning them to the `struct tm` seconds integer slot (previous version rounded the seconds part which sometimes resulted in the value being 60).

Returns a pointer to the changed struct or NULL in case of an error.

```
double (*DateTime_AsTicks)(mxDateTimeObject *datetime);
```

Return the objects value as time_t value.

It is assumed that the object contains local time information, so time.localtime(object.as_ticks()) == object.tuple().

Note that this functions returns a double and not a time_t value -- this is because some systems define time_t to be a long which would cause the conversion to lose the fraction part.

Returns -1.0 and sets an error in case of failure.

```
double (*DateTime_AsCOMDate)(mxDateTimeObject *datetime);
```

Return the objects value as COM date double

Returns -1.0 and sets an error in case of failure.

```
PyTypeObject *DateTimeDelta_Type;
```

Type object for DateTimeDelta()

```
PyObject *(*DateTimeDelta_FromDaysAndSeconds)(long days, double
   seconds);
```

Construct a new object from the given days and seconds deltas.

The internal value is calculated using a 86400.0 seconds/day basis.

Returns NULL in case of an error.

```
PyObject *(*DateTimeDelta_FromTime)(int hours, int minutes, double
seconds);
```

Construct a new object from the given values repesenting time. The parameters are used to calculate a number-of-seconds since midnight value.

Returns NULL in case of an error.

`PyObject *(*`**`DateTimeDelta_FromTuple`**`)(PyObject *v);`

Same as DateTimeDelta_FromDaysAndSeconds() except that you pass the two arguments in a Python tuple.

Returns NULL in case of an error.

`PyObject *(*`**`DateTimeDelta_FromTimeTuple`**`)(PyObject *v);`

Same as DateTimeDelta_FromTime() except that you pass the three arguments in a Python tuple.

Returns NULL in case of an error.

`double (*`**`DateTimeDelta_AsDouble`**`)(mxDateTimeDeltaObject *delta);`

Using 86400.0 seconds/day a seconds value is calculated from the days and seconds part of the passed object.

Returns -1.0 and sets an error in case of failure.

`PyObject *(*`**`DateTime_FromAbsDays`**`)(double days);`

Construct a new DateTime object from the given days value which represents absolute days and the absolute time as fraction of a day.

The internal value is calculated using a 86400.0 seconds/day basis.

Returns NULL in case of an error.

`double (*`**`DateTime_AsAbsDays`**`)(mxDateTimeObject *datetime);`

Using 86400.0 seconds/day a days value is calculated from the internal value of the passed object.

Returns -1.0 and sets an error in case of failure.

`PyObject *(*`**`DateTimeDelta_FromDays`**`)(double days);`

Construct a new DateTimeDelta object from the given days value.

The internal value is calculated using a 86400.0 seconds/day basis.

Returns NULL in case of an error.

`double (*`**`DateTimeDelta_AsDays`**`)(mxDateTimeDeltaObject *delta);`

Using 86400.0 seconds/day a days value is calculated from the internal value of the passed object.

Returns NULL in case of an error.

```
int (*DateTime_BrokenDown)(mxDateTimeObject *datetime, long *year,
   int *month, int *day, int *hour, int *minute, double *second);
```

Sets the given variables to values corresponding to the given DateTime object.

You can pass a NULL pointer if you don't want that variable to be set.

Returns -1 and sets an error in case of failure.

```
int (*DateTimeDelta_BrokenDown)(mxDateTimeDeltaObject *delta, long
   *day, int *hour, int *minute, double *second);
```

Sets the given variables to values corresponding to the given DateTimeDelta object.

You can pass a NULL pointer if you don't want that variable to be set.

Returns -1 and sets an error in case of failure.

```
PyObject *(*DateTime_FromAbsDateTime)(long absdate, double abstime,
int calendar);
```

Construct a new object from the given absolute date, time and calendar.

Returns NULL in case of an error.

New in mxDateTime 3.2.

```
} mxDateTimeModule_APIObject;
```

Binary Compatibility

New APIs will generally be added to the end of the `mxDateTime` struct to maintain binary compatibility between the releases.

Type Checking

The `mxDateTime.h` header file also defines two type checking macros which will work once the mxDateTime C API has been loaded:

`mxDateTime_Check(v)`

Returns 1 for DateTime objects, 0 otherwise.

`mxDateTimeDelta_Check(v)`

Returns 1 for DateTimeDelta objects, 0 otherwise.

12. mxDateTime Package Structure

```
[DateTime]
       Doc/
       [Examples]
              AtomicClock.py
              CommandLine.py
              Y2000.py
              alarm.py
              lifespan.py
       [mxDateTime]
              test.py
       ARPA.py
       DateTime.py
       Feasts.py
       ISO.py
       LazyModule.py
       Locale.py
       NIST.py
       ODMG.py
       Parser.py
       Timezone.py
       timegm.py
```

Names with trailing / are plain directories, ones with []-brackets are Python packages, ones with ".py" extension are Python submodules.

The package imports all symbols from the extension module and also registers the types so that they become compatible to the pickle and copy mechanisms in Python.

13. 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*.

14. Copyright & License

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

© 2000-2011, 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.

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