interface_Model

index

(version v0.1.2, 2011-03-28) /home/gis/Documents/interface_10032011/interface_Model.py

```
Model module for Interface.

This module contains classes to handle operation on data and is controlled by
the class 'ControlModel' of the module 'interface_Control'
```

Modules

dateutil	xml.dom.minidom	os	xml
logging	numpy	time	

Classes

ModelDataRead

- ModelDataGridRead
- ModelDataStationRead

ModelDataWrite
ModelMetadataNcmlRead
ModelMetadataNcmlWrite
ModelNetCdfRead
ModelNetCdfWrite
ModelPrint

class **ModelDataGridRead**(ModelDataRead)

```
Class for reading coordinate metadata file and numpy data array of shape
(variable, time, z, lat, lon). This class inherits from 'ModelDataRead'
```

Methods defined here:

__init__(self, infile_)

```
Constructor.

INPUT_PARAMETERS:
infile        - file name without suffix (string). Both the numpy data array
    and the coordinate metadata file must have the same name (expect of suffix)
```

checkDataModel(self, pDataList_)

```
Check if complete data model is correct and consistent
```

getCoordinateVariables(self, pVarList_)

```
Reading coordinate information in coordinate metadata file and attaching
calculated coordinates to coordinate variables in internal model
```

getDataVariables(self, pVarList_)

```
Reading data in numpy data array and attaching these data as separate numpy arrays
to data variables in internal model. The shape of the separate numpy arrays will
be transformed to (time, z, lat, lon) for the internal model.

IMPORTANT:
The variable order and the number of variables in the numpy array must be coherent
with the variable order and the number of variables in the internal model!
```

class **ModelDataRead**

```
Superclass for reading numpy data array and coordinate metadata file
as well as checking operations for internal datamodel
```

Methods defined here:

__init__(self, infile_)

```
Constructor
```

checkDataModel(self, pDataList_)

```
Check if internal data model is correct.

This function checks dimensions as well as coordinate and data variables of a data model if they
are correct and consistent to each other as they are declared in the NCML and coordinate
metadata and numpy data array. If no error is found, the NetCDF creation should be successfull
out of this data modell.

INPUT_PARAMETERS:
pDataList       - internal data model

RETURN_VALUE:
Boolean: True if no error could be found, False if one or more errors were found
```

```
COMMENT:
This function only checks the correctness of the data model and its consistency so that the
NetCDF creation out of this data model will be successfull. It does not check if any NetCDF convention is
respected.
```

getCoordinateVariables(self, pVarList_, dimVar_, dimTime_, dimZ_, dimLat_, dimLon_, dimId_)

```
Obtain coordinate information.

Get coordinate information out of coordinate metadata file and calculate coordinate values
by the use of this information. Attach coordinate values as numpy array to corresponding
variable in internal model

INPUT_PARAMETERS:
pVarList                                        - variable list of internal model
dimVar, dimTime, dimZ, dimLat, dimLon, dimId    - number of values to generate

RETURN_VALUE:
Actualized variable list with coordinate data arrays attached to corresponding variables

IMPORTANT:
In case that separate values instead of minimum and maximum values for cpordinates are
provided, the number of these values must be the same as the shape of the corresponding
dimension.
```

class **ModelDataStationRead**(ModelDataRead)

```
Class for reading coordinate metadata file and numpy data array of shape
(time, variable). This class inherits from 'ModelDataRead'
```

Methods defined here:

__init__(self, infile_)

```
Constructor.

INPUT_PARAMETERS:
infile        - file name without suffix (string). Both the numpy data array
    and the coordinate metadata file must have the same name (expect of suffix)
```

checkDataModel(self, pDataList_)

```
Check if complete data model is correct and consistent
```

getCoordinateVariables(self, pVarList_)

```
Reading coordinate information in coordinate metadata file and attaching
calculated coordinates to coordinate variables in internal model
```

getDataVariables(self, pVarList_)

```
Reading data in numpy data array and attaching these data as separate numpy arrays
to data variables in internal model. The shape of the separate numpy arrays will
be transformed to (time, z, lat, lon) for the internal model.

IMPORTANT:
The variable order and the number of variables in the numpy array must be coherent
with the variable order and the number of variables in the internal model!
```

class **ModelDataWrite**

```
Class for writing a numpy data array and a coordinate metadata file out of
internal model
```

Methods defined here:

__init__(self, infile_)

```
Constructor.

INPUT_PARAMETERS:
infile        - file name without suffix (string). Both the numpy data array
    and the coordinate metadata file will have the same name (expect of suffix)
```

writeCoordinateVariables(self, pVarList_)

```
Get coordinate information from coordinate variables of internal model and
write coordinates to coordinate metadata file.

IMPORTANT:
- All numpy arrays in the internal model that are attached to a coordinate variable
    need to have a single dimension containing their coordinates
```

writeDataVariables(self, pVarList_)

```
Get data values from data variables of internal model and write data variables
to external numpy array file.

IMPORTANT:
- All numpy data arrays in the internal model (so each numpy array attached to a data variable)
must have the same shape and type so that they can be merged in a new array
- All numpy data arrays in the internal model have to have the shape (time, z, lat, lon)
```

class **ModelMetadataNcmlRead**

Class for reading NCML XML metadata

Methods defined here:

__del__(self)
Destructor

__init__(self, infile_)
Constructor.

INPUT_PARAMETERS:
infile - name of NCML file name without suffix (string)

readDimensions(self)
Read dimensions in NCML file and return new dimension list

readGlobalAttributes(self)
Read global attributes in NCML file and return new attribute list

readVariables(self)
Read variables and appendent local attributes in NCML file and return new variable list

class **ModelMetadataNcmlWrite**

Class for writing a NCML XML metadata file out of data of the internal model

Methods defined here:

__init__(self, infile_)
Constructor.

INPUT_PARAMETERS:
infile - name of NCML file name without suffix (string)

addDimensions(self, pDimList_)
Add dimension entries to NCML file by the use of the internal models dimension list

addGlobalAttributes(self, pAttrList_)
Add global attribute entries to NCML file by the use of the internal models global attribute list

addVariables(self, pVarList_)
Add variable metadata and local attribute entries to NCML file by the use of the internal models variable list

printNcmlOnScreen(self)
Print NCML file on screen

class **ModelNetCdfRead**

Class for reading one or multiple NetCDF files

Methods defined here:

__del__(self)
Destructor

__init__(self, infile_)
Constructor.

INPUT_PARAMETERS:
infile - NetCDF file name without suffix (string) or that part of the
NetCDF file name that is shared by all files (for reading multiple files),
followed by a wildcard (*).

COMMENTS:
For reading and aggregating multiple NetCDF files all files need to be similiar
expect of the time coordinate values (but need to share the same time unit).

readDimensions(self)
Reading dimensions of NetCDF file and saving them to internal model

readGlobalAttributes(self)
"Reading global attributes of NetCDF file and saving them to internal model

readVariables(self)
Reading variables and associated local attributes of NetCDF file and saving
them to internal model

IMPORTANT:
Activate function '.__correctVariableInputData' for manual bug fix of 'issue 34'
(slicing MFDataset variables with dimensions of length 1) if API NetCDF4 is older then version 0.9

class **ModelNetCdfWrite**

Class for writing data from the internal model to a NetCDF file

Methods defined here:

__del__(self)
```
Destructor
```

__init__(self, netCdfFileName_)
```
Constructor.

INPUT_PARAMETERS:
infile        - name of NetCDF file name without suffix (string)
```

writeDimensions(self, pDimList_)
```
Write dimensions from the internal models dimension list to the NetCDF file
```

writeGlobalAttributes(self, pAttrList_)
```
Write global attributes from the internal models global attribute list to the NetCDF file
```

writeVariables(self, pVarList_)
```
Write variables (data and metadata) and attached local attributes from the internal
models variable list to the NetCDF file

IMPORTANT:
- The numpy data array must be consistent with the corresponding variable metadata
    (shape, type, _FillValue, etc.)
- The numpy data array is allowed to have one to four dimensions (in general
    one for coordinate variables and four for data variables)
```

class **ModelPrint**
```
Class for printing data and metadata
```

Methods defined here:

__del__(self)
```
Destructor
```

__init__(self, pDataList_)
```
Constructor.

INPUT_PARAMETERS:
DataList        - List of internal model with dimensions, attributes and variables
    to print on screen
```

printCoordinateVariablesData(self)
```
Print data of coordinate variables of internal model
```

printDataVariablesData(self)
```
Print data of data variables of internal model
```

printDimensions(self)
```
Print dimensions of internal model
```

printGlobalAttributes(self)
```
Print global attributes of internal model
```

printVariables(self)
```
Print variables and attached local attributes of internal model
```

Functions

POINTER(...)

addressof(...)
```
addressof(C instance) -> integer
Return the address of the C instance internal buffer
```

alignment(...)
```
alignment(C type) -> integer
alignment(C instance) -> integer
Return the alignment requirements of a C instance
```

byref(...)
```
byref(C instance[, offset=0]) -> byref-object
Return a pointer lookalike to a C instance, only usable
as function argument
```

date2num(...)
```
date2num(dates,units,calendar='standard')

Return numeric time values given datetime objects. The units
of the numeric time values are described by the L{units} argument
and the L{calendar} keyword. The datetime objects must
be in UTC with no time-zone offset.  If there is a
time-zone offset in C{units}, it will be applied to the
```

```
    returned numeric values.

    Like the matplotlib C{date2num} function, except that it allows
    for different units and calendars.  Behaves the same if
    C{units = 'days since 0001-01-01 00:00:00'} and
    C{calendar = 'proleptic_gregorian'}.

    @param dates: A datetime object or a sequence of datetime objects.
     The datetime objects should not include a time-zone offset.

    @param units: a string of the form C{'B{time units} since B{reference time}}'
     describing the time units. B{C{time units}} can be days, hours, minutes
     or seconds.  B{C{reference time}} is the time origin. A valid choice
     would be units=C{'hours since 1800-01-01 00:00:00 -6:00'}.

    @param calendar: describes the calendar used in the time calculations.
     All the values currently defined in the U{CF metadata convention
     <http://cf-pcmdi.llnl.gov/documents/cf-conventions/>} are supported.
     Valid calendars C{'standard', 'gregorian', 'proleptic_gregorian'
     'noleap', '365_day', '360_day', 'julian', 'all_leap', '366_day'}.
     Default is C{'standard'}, which is a mixed Julian/Gregorian calendar.

    @return: a numeric time value, or an array of numeric time values.

    The maximum resolution of the numeric time values is 1 second.
```

get_errno(...)

num2date(...)

```
    num2date(times,units,calendar='standard')

    Return datetime objects given numeric time values. The units
    of the numeric time values are described by the C{units} argument
    and the C{calendar} keyword. The returned datetime objects represent
    UTC with no time-zone offset, even if the specified
    C{units} contain a time-zone offset.

    Like the matplotlib C{num2date} function, except that it allows
    for different units and calendars.  Behaves the same if
    C{units = 'days since 001-01-01 00:00:00'} and
    C{calendar = 'proleptic_gregorian'}.

    @param times: numeric time values. Maximum resolution is 1 second.

    @param units: a string of the form C{'B{time units} since B{reference time}}'
    describing the time units. B{C{time units}} can be days, hours, minutes
    or seconds.  B{C{reference time}} is the time origin. A valid choice
    would be units=C{'hours since 1800-01-01 00:00:00 -6:00'}.

    @param calendar: describes the calendar used in the time calculations.
    All the values currently defined in the U{CF metadata convention
    <http://cf-pcmdi.llnl.gov/documents/cf-conventions/>} are supported.
    Valid calendars C{'standard', 'gregorian', 'proleptic_gregorian'
    'noleap', '365_day', '360_day', 'julian', 'all_leap', '366_day'}.
    Default is C{'standard'}, which is a mixed Julian/Gregorian calendar.

    @return: a datetime instance, or an array of datetime instances.

    The datetime instances returned are 'real' python datetime
    objects if the date falls in the Gregorian calendar (i.e.
    C{calendar='proleptic_gregorian'}, or C{calendar = 'standard'} or C{'gregorian'}
    and the date is after 1582-10-15). Otherwise, they are 'phony' datetime
    objects which support some but not all the methods of 'real' python
    datetime objects.  This is because the python datetime module cannot
    the uses the C{'proleptic_gregorian'} calendar, even before the switch
    occured from the Julian calendar in 1582. The datetime instances
    do not contain a time-zone offset, even if the specified C{units}
    contains one.
```

pointer(...)

resize(...)

```
    Resize the memory buffer of a ctypes instance
```

set_conversion_mode(...)

```
    set_conversion_mode(encoding, errors) -> (previous-encoding, previous-errors)

    Set the encoding and error handling ctypes uses when converting
    between unicode and strings.  Returns the previous values.
```

set_errno(...)

sizeof(...)

```
    sizeof(C type) -> integer
    sizeof(C instance) -> integer
    Return the size in bytes of a C instance
```

Data

ALL_FLOATS = ['float64', 'double', 'Float64', 'f8', 'float', 'float32', 'Float32', 'f4']
ALL_INTS = ['byte', 'int8', 'i1', 'ubyte', 'UByte', 'uint8', 'u1', 'short', 'int16', 'Int16', 'i2', 'ushort',

'uint16', 'UInt16', 'u2', 'int', 'int32', 'Int32', 'integer', 'i4', ...]
BOOL = ['bool', 'Bool']
BYTE = ['byte', 'int8', 'i1']
BasicContext = Context(prec=9, rounding=ROUND_HALF_UP, Emin=-99...sionByZero, InvalidOperation, Clamped, Overflow])
COORD_KEYWORDS = ['time', 'height', 'elev', 'depth', 'lat', 'latitude', 'lon', 'longitude', '_id']
DECLARATION_NETCDF_STATION = '_time_series'
DEFAULT_MODE = 0
DOUBLE = ['float64', 'double', 'Float64', 'f8']
DefaultContext = Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=...aps=[DivisionByZero, InvalidOperation, Overflow])
ExtendedContext = Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-..., Emax=999999999, capitals=1, flags=[], traps=[])
FILENAME_DEFAULT_SETTINGS_XML = 'interface_Settings.xml'
FILENAME_SUFFIX_NCML = '__ncml.xml'
FILENAME_SUFFIX_NETCDF = '.nc'
FILENAME_SUFFIX_NUMPYDATA = '__data.npy'
FILENAME_SUFFIX_NUMPYXML = '__coords.xml'
FLOAT = ['float', 'float32', 'Float32', 'f4']
GDAL_DTYPES = ['byte', 'int8', 'i1', 'short', 'int16', 'Int16', 'i2', 'ushort', 'uint16', 'UInt16', 'u2', 'int', 'int32', 'Int32', 'integer', 'i4', 'uint', 'uint32', 'UInt32', 'unsigned_integer', ...]
HEIGHT = ['height', 'elev', 'depth']
HEIGHT_UNITS = ['m', '1']
ID = ['_id']
INTEGER = ['int', 'int32', 'Int32', 'integer', 'i4']
INTERFACE_LOGGER_ROOT = 'interface'
LATITUDE = ['lat', 'latitude']
LATITUDE_UNITS = ['degrees_north']
LONG = ['long', 'int64', 'Int64', 'i8']
LONGITUDE = ['lon', 'longitude']
LONGITUDE_UNITS = ['degrees_east']
MODEL_REFERENCE_TIME_UNITS = ['hours since 1970-01-01 00:00:0.0', 'msec since 1970-01-01 00:00:0.0']
NETCDF3_DTYPES = ['byte', 'int8', 'i1', 'short', 'int16', 'Int16', 'i2', 'int', 'int32', 'Int32', 'integer', 'i4', 'float', 'float32', 'Float32', 'f4', 'float64', 'double', 'Float64', 'f8', ...]
NETCDF_FORMAT = 'NETCDF3_CLASSIC'
NUMPY_DTYPES = ['bool', 'Bool', 'byte', 'int8', 'i1', 'ubyte', 'UByte', 'uint8', 'u1', 'short', 'int16', 'Int16', 'i2', 'ushort', 'uint16', 'UInt16', 'u2', 'int', 'int32', 'Int32', ...]
ROUND_05UP = 'ROUND_05UP'
ROUND_CEILING = 'ROUND_CEILING'
ROUND_DOWN = 'ROUND_DOWN'
ROUND_FLOOR = 'ROUND_FLOOR'
ROUND_HALF_DOWN = 'ROUND_HALF_DOWN'
ROUND_HALF_EVEN = 'ROUND_HALF_EVEN'
ROUND_HALF_UP = 'ROUND_HALF_UP'
ROUND_UP = 'ROUND_UP'
RTLD_GLOBAL = 256
RTLD_LOCAL = 0
SHORT = ['short', 'int16', 'Int16', 'i2']
STRING = ['char', 'string', 'S1']
TIME = ['time']
U_BYTE = ['ubyte', 'UByte', 'uint8', 'u1']
U_INTEGER = ['uint', 'uint32', 'UInt32', 'unsigned_integer', 'u4']
U_LONG = ['ulong', 'uint64', 'UInt64', 'u8']
U_SHORT = ['ushort', 'uint16', 'UInt16', 'u2']
__author__ = 'Nicolai Holzer'
__author_email__ = 'first-name dot last-name @ mailbox.tu-dresden.de'
__date__ = '2011-03-28'
__version__ = 'v0.1.2'
cdll = <ctypes.LibraryLoader object>
environ = {'LANG': 'en_US.UTF-8', 'USERNAME': 'root', 'TER...36:*.spx=00;36:*.xspf=00;36:', 'DISPLAY': ':0.0'}
memmove = <CFunctionType object>
memset = <CFunctionType object>
pydll = <ctypes.LibraryLoader object>
pythonapi = <PyDLL 'None', handle a17918 at b73899ec>

Author

Nicolai Holzer