pymepps Documentation

Release 0.3.0

Tobias Sebastian Finn

Jul 05, 2017

Content

The python meteorological post-processing system could be used to post-process meteorological data. The main purpose of this project is the post-processing of numerical weather forecast data. This package has some modules to load and process meteorological spatial and time series data. More modules to process and plot the data are planned.

CHAPTER 1

Installation

This package is a python package. Further this package is based on different python tools. So there are some dependencies which should be matched. To install this package you should read the installation section.

1.1 Requirements

This package is programmed with python 3.5. It is not planned to support python 2.7. In the future this package will be back checked with travis ci for version 3.4 – 3.6. But at the moment we couldn't guarantee the compatibility of other versions than 3.5.

- python
- numpy
- scipy
- pandas
- xarray
- netcdf4
- pygrib
- matplotlib
- basemap

The following packages are only recommended to use all features. * cdo * cdo bindings

In the future some requirements will be added for example scikit-learn.

1.2 Installation

At the moment this package is not available on pypi and conda. So you have to clone this package and install it via pip.

It is recommended to install the requirements via a conda virtual environment, but it is also possible to install them via pip.

1.2.1 Installation and activation via conda (recommended)

```
git clone git@github.com:maestrotf/pymepps.git
cd pymepps
conda env create -f environment.yml
source activate pymepps
pip install .
```

1.2.2 Installation via pip

```
git clone git@github.com:maestrotf/pymepps.git
cd pymepps
pip install -r requirements.txt
pip install .
```

CHAPTER 2

History

Let's talk a little about the history and the formation process of this package.

2.1 There are so many python packages why a new one?

Python is a rapidly developing programming language. In the last few years has won more fans, especially in the geoscientific community. There are so many different packages for different purposes, but no package matched my requirements.

2.2 What are the requirements?

In my bachelor thesis I used a simple method for post-processing of numerical weather model data. This method was based on a linear regression and is called model output statistics in the meteorological community. The work for this thesis needed a system for offline statistical processing of data. Later I developed an operational weather forecast system based on the same methods. The requirements of an online and operational weather forecast system are very different to an offline system. So the requirements for this package are offline and online processing of weather model data.

The biggest part of the online processing is outsourced to a companion project called pymepps-streaming. But this package will be the base for offline and online processing of numerical weather model data.

CHAPTER 3

Data structure

Pymepps is a system to read and process meteorological data. So we defined some base types and tools for an easier read and process workflow.

3.1 File handlers

File handlers are used to read in the data. A file handler is working on file basis, such that one file handler could only process one file. There different file handlers for different file types. Some are only to read in spatial data and some could only read in time series data. But all file handlers have three common methods:

- Method to load the data into the memory
- Method to get the variable names within the file
- Method to extract a variable and prepare the variable for the dataset.

The method to extract a variable uses a message based interface so that similar files could be merged within a dataset.

3.1.1 NetCDF handler

The NetCDF handler could be used to read in netcdf files. The NetCDF handler is based on the xarray and the netcdf4 package, so it is also possible to load opendap data streams with this handler. The NetCDF handler could be used to read in spatial and time series files. At the moment the load of time series data with this handler is only tested for measurement data from the "Universtät Hamburg".

3.1.2 Grib handler

The grib handler could be used to read in grib1 and grib2 files. The grib handler is based on the pygrib package. The grib handler could be only used to read in spatial data, due to the requirements of a grib file.

At the moment there are only these two differnt file handlers, but it is planned to implement some other file handlers to read in hdf4/5 and csv based data.

3.2 Dataset

Datasets are used to combine file handlers and to manage the variable selection. A dataset is working at multiple file level. The messages of the file handlers are bundled to spatial or time series data. So the two different dataset types the spatial and the times series dataset have a merge method in common.

3.2.1 Spatial dataset

A spatial dataset is used to combine the file handlers, which are capable to read in spatial data. The spatial dataset interacts on the same level as the climate data operators (cdo). So it is possible to process the data of a spatial dataset with some of the cdos. A method for the general support of the cdos is planned.The spatial dataset also creates the grid for the spatial data. The grid could be either predefined or is read in with the griddes function from the cdo.

3.2.2 Time series dataset

A time series dataset is ised to combine the times series file handlers. A time series dataset is valid for a given coordinates, so it is possible to defined a coordinate tuple. If no coordinate tuple is set the time series dataset tries to get the coordinates from the data origin.

3.3 Data

Two different data types are defined within this package – spatial data and time series data. The data types are used to process and plot the data. The data types are working at variable level. Both data types are like a wrapper around powerful packages – pandas and xarray.

3.3.1 Spatial data

The spatial data is represented within the SpatialData data type. The data type is based on xarray.DataArray and could be seen as NetCDF like cube. So it is easy to save the data as NetCDF file. The spatial data contains a grid, defining the horizontal grid coordinates of the data. With this grid it is further possible to remap the data and to transform the data to time series data. These features are used to process the data in statistical models.

3.3.2 Time series data

The time series data is represented within the TSData data type. The data type is based on pandas.Series and pandas.DataFrame and could be seen as table like data.

CHAPTER 4

API

4.1 File handler package

The file handlers are used to extract the data from files.

4.1.1 Base file handler

class `pymepps.metfile.filehandler.`**`FileHandler`**(*file_path*)

> **`get_messages`**(*var_name*, ***kwargs*)
>
> **`get_timeseries`**(*var_name*, ***kwargs*)
>
> **`load_file`**()
>
> **`var_names`**

4.1.2 Grib file handler

class `pymepps.metfile.gribhandler.`**`GribHandler`**(*file_path*)

> Bases: *`pymepps.metfile.filehandler.FileHandler`*
>
> **`close`**()
>
> **`get_messages`**(*var_name*, ***kwargs*)
>
> Method to get message-wise the data for a given variable as xr.DataArray.
>
> **Parameters** **`var_name`** (*`str`*) – The name of the variable which should be extracted.
>
> **Returns** **data** – The list with the message-wise data as DataArray. The DataArray have six coordinates (analysis, ensemble, time, level, y, x). The shape of DataArray are normally (1,1,1,1,y_size,x_size).
>
> **Return type** list of xr.DataArray

is_type()

open()

4.1.3 NetCDF file handler

class pymepps.metfile.netcdfhandler.**NetCDFHandler**(*file_path*)

Bases: *pymepps.metfile.filehandler.FileHandler*

close()

get_messages(*var_name*, ***kwargs*)

Method to imitate the message-like behaviour of grib files.

Parameters

- **var_name** (*str*) – The variable name, which should be extracted.
- **runtime** (*np.datetime64, optional*) – If the dataset has no runtime this runtime is used. If the runtime is not set, the runtime will be inferred from file name.
- **ensemble** (*int or str, optional*) – If the dataset has no ensemble information this ensemble is used. If the ensemble is not set, the ensemble will be inferred from file name.
- **sliced_coords** (*tuple(slice), optional*) – If the cube should be sliced before it is loaded. This is helpful by large opendap requests. These slice will be used from the behind. So (slice(1,2,1), slice(3,5,1)) means [. . . , 1:2, 3:5]. If it is not set all data is used. T

Returns data – The list with the message-wise data as DataArray. The DataArray have six coordinates (analysis, ensemble, time, level, y, x). The shape of DataArray are normally (1,1,1,1,y_size,x_size).

Return type list of xr.DataArray

get_timeseries(*var_name*, ***kwargs*)

Method to get the time series from a NetCDF file. This is designed for measurement site data in netcdf format. At the moment this method is only tested for Wettermast Hamburg data!

Parameters var_name (*str*) – The variable name, which should be extracted.

Returns data – The selected variable is extracted as dict with pandas series as values.

Return type dict with pandas series

is_type()

load_cube(*var_name*)

Method to load a variable from the netcdf file and return it as xr.DataArray.

Parameters var_name (*str*) – The variable name, which should be extracted.

Returns variable – The DataArray of the variable.

Return type xr.DataArray

lon_lat

open()

pymepps.metfile.netcdfhandler.**cube_to_series**(*cube*, *var_name*)

4.1.4 Opendap file handler

class `pymepps.metfile.opendaphandler.`**`OpendapHandler`**(*file_path*)

Bases: *`pymepps.metfile.netcdfhandler.NetCDFHandler`*

`is_type`()

`open`()

4.2 MetData package

The metadata package is used to define data types for different data types.

4.2.1 Dataset module

A dataset could load many different of a specific file type.

Base Dataset

class `pymepps.metdata.metdataset.`**`MetDataset`**(*file_handlers*, *data_origin=None*, *processes=1*)

MetDataset is a base class for handling meteorolgical files.

The normal workroutine would be:

1. load the files (use of file handlers)
2. select the important variables within the files (this object)
3. post-process the variables (MetData/SpatialData/TSData object)

Parameters

- **`file_handlers`** (*`list of childs of FileHandler or None.`*) – The loaded file handlers. This instance load the variables. If the file handlers are None then the dataset is used for conversion between Spatial and TSData.
- **`data_origin`** (*`optional`*) – The class where the data comes from. Normally this would be a model or a measurement site. If this is None, this isn't set. Default is None.
- **`processes`** (*`int`*, *`optional`*) – This number of processes is used to calculate time-consuming functions. For time-consuming functions a progress bar is shown. If the number of processes is one the functions will be processed sequential. For more processes than one the multiprocessing module will be used. Default is 1.

`data_merge`(*data*, *var_name*)

Method to merge the given data by given metadata into one data structure.

`file_handlers`

`processes`

`select`(*var_name*, ***kwargs*)

Method to select a variable from this dataset. If the variable is find in more than one file or message, the method tries to find similarities within the metadata and to combine the data into one array, with several dimensions. This method could have a long running time, due to data loading and combination.

Parameters

- **`var_name`** (*`str`*) – The variable which should be extracted. If the variable is not found within the dataset there would be a value error exception.
- **`kwargs`** (*`dict`*) – Additional parameters that are passed to the file handlers.

Returns extracted_data – A child instance of MetData with the data of the selected variable as data. If None is returned the variable wasn't found within the list with possible variable names.

Return type *SpatialData*, *TSData* or None

`select_by_pattern`(*pattern*, *return_list=True*, ***kwargs*)
Method to select variables from this dataset by keywords. This method uses list comprehension to extract the variable names where the var_name pattern is within the variable name. If the variable names are found the variable is selected with the select method.

Parameters

- **`pattern`** (*`str`*) – The pattern for which should be searched.
- **`return_list`** (*`bool`*) – If the return value should be a list or a dictionary.
- **`kwargs`** (*`dict`*) – Additional parameters that are passed to the file handlers.

Returns data_list – list(SpatialData or TSData) or None The return value is a dict/list with SpatialData instances, one entry for every found variable name. If return_list is False, are the keys the variable names. If None is returned no variable with this pattern was found.

Return type dict(str, *SpatialData* or *TSData*) or

`select_ds`(*include=None*, *exclude=None*, ***kwargs*)
Extract the dataset data into a MetData instance. The include list is handled superior to the exclude list. If both lists are None all available variables are used.

Parameters

- **`include`** (*`iterable or None`*) – Within the include iterable are all variable names, which should be included into the MetData data. The list will be filtered for available variable names. If no variable name is available a ValueError will be raised. If this is None, the include will be skipped and the exclude list will be used. Default is None.
- **`exclude`** (*`iterable or None`*) – If no include iterable is given, this exclude iterable is used. In this case, any available variable name, which is not within this list is used. If this iterable is also None, all available data variables are used to construct the MetData instance. Default is None.
- **`kwargs`** (*`dict`*) – Additional parameters that are passed to the file handlers.

Returns extracted_data – The extracted data instance.

Return type *TSData* or *SpatialData*

Raises ValueError: – A ValueError is raised if no variable was selected from the dataset.

`var_names`
Get the available variable names.

`variables`
Return the variable names and the corresponding file handlers.

Spatial Dataset

class `pymepps.metdata.spatialdataset.`**`SpatialDataset`**(*file_handlers*, *grid=None*, *data_origin=None*, *processes=1*)

Bases: *`pymepps.metdata.metdataset.MetDataset`*

SpatialDataset is a class for a pool of file handlers. Typically a spatial dataset combines the files of one model run, such that it is possible to select a variable and get a SpatialData instance. For memory reasons the data of a variable is only loaded if it is selected.

Parameters

- **`file_handlers`** (*`list of childs of FileHandler or None`*) – The spatial dataset is based on these files. The files should be either instances of GribHandler or NetCDFHandler. If file handlers is None then the dataset is used for conversion from TS-Data to SpatialData.
- **`grid`** (*`str or Grid or None`*) – The grid describes the horizontal grid of the spatial data. The grid will be appended to every created SpatialData instance. If a str is given it will be checked if the str is a path to a cdo-conform grid file or a cdo-conform grid string. If this is a instance of a child of Grid it is assumed that the grid is already initialized and this grid will be used. If this is None the Grid will be automatically read from the first file handler. Default is None.
- **`data_origin`** (*`optional`*) – The data origin. This parameter is important to trace the data flow. If this is None, there is no data origin and this dataset will be the starting point of the data flow. Default is None.
- **`processes`** (*`int, optional`*) – This number of processes is used to calculate time-consuming functions. For time-consuming functions a progress bar is shown. If the number of processes is one the functions will be processed sequential. For more processes than one the multiprocessing module will be used. Default is 1.

`select`()

Method to select a variable.

`selnearest`()

Method to select the nearest grid point for given coordinates.

`sellonlatbox`()

Method to slice a box with the given coordinates.

`data_merge`(*data*, *var_name*)

Method to merge instances of xarray.DataArray into a SpatialData instance. Also the grid is read and inserted into the SpatialData instance.

Parameters

- **`data`** (*`list of xarray.DataArray`*) – The data list.
- **`var_name`** (*`str`*) – The name of the variable which is selected within the data list.

Returns The SpatialData instance with the extracted data and the extracted grid.

Return type *SpatialData*

`get_grid`(*var_name*, *data_array=None*)

Method to get for given variable name a Grid instance. If the grid attribute is already a Grid instance this grid will be returned. If the grid attribute is a str instance, the str will be read from file or from the given grid str. If the grid attribute isn't set the grid instance will be the grid for the variable selected with the first corresponding file handler and cdo.

Parameters

- **var_name** (*str*) – The variable name, which should be used to generate the grid.
- **data_array** (*xarray.DataArray or None, optional*) – If the data array is given the method will try to load the grid from the data array's attributes. If None the DataArray method will be skipped. Default is None.

Returns grid – The returned grid. If the returned grid is None, the grid could not be read.

Return type Instance of child of grid or None

Time series dataset

class pymepps.metdata.tsdataset.**TSDataset**(*file_handlers*, *data_origin=None*, *lonlat=None*, *processes=1*)

Bases: *pymepps.metdata.metdataset.MetDataset*

TSDataset is a class for a pool of file handlers. Typically a time series dataset combines the files of a station, such that it is possible to select a variable and get a TSData instance. For memory reasons the data of a variable is only loaded if it is selected.

Parameters

- **file_handlers** (*list of childs of FileHandler or None*) – The spatial dataset is based on these files. The files should be either instances of NetCDFHandler or TextHandler. If file handlers is None then the dataset is used for conversion from SpatialData to TSData.
- **data_origin** (*optional*) – The data origin. This parameter is important to trace the data flow. If this is None, there is no data origin and this dataset will be the starting point of the data flow. Default is None.
- **lonlat** (*tuple(float, float) or None*) – The coordinates (longitude, latitude) where the data is valid. If this is None the coordinates will be set based on data_origin or based on the first file handler.

select()

Method to select a variable.

data_merge(*data*, *var_name*)

select_by_pattern(*pattern*, *return_list=False*)

4.2.2 Data module

A data module contains all data for one specific variable extracted from a Dataset.

Base data

class pymepps.metdata.metdata.**MetData**(*data*, *data_origin=None*)

MetData is the base class for meteorological data, like station data, nwp forecast data etc.

append(*item*, *inplace=False*)

copy()

data

data_plot(***kwargs*)

Method to refer to the xr_plot layer. :param kwargs:

remove(*item*, *inplace=False*)

update(**items*)

Spatial data

class pymepps.metdata.spatialdata.**SpatialData**(*data*, *grid=None*, *data_origin=None*)

Bases: *pymepps.metdata.metdata.MetData*

SpatialData contains spatial based data structures. This class is the standard data type for file types like netCDF or grib. It's prepared for the output of numerical and statistical weather models. Array based data is always saved to netcdf via xarray.

data

xarray.DataArray or None – The data of this grid based data structure.

grid

Child instance of Grid or None – The corresponding grid of this SpatialData instance. This grid is used to interpolate/remap the data and to select the nearest grid point to a given longitude/latitude pair. The grid is also used to get a basemap instance to determine the grid boundaries for plotting purpose.

data_origin

object of pymepps or None, optional – The origin of this data. This could be a model run, a station, a database or something else. Default is None.

grid

static load(*path*)

Load a SpatialData instance from a given path. The path is loaded as SpatialDataset. A correct saved SpatialData instance will have only one variable within the NetCDF file. So the first variable will be returned as newly constructed SpatialData instance.

Parameters **path** (*str*) – The path to the saved SpatialData instance.

Returns **spdata** – The loaded SpatialData instance.

Return type *SpatialData*

merge(**items*, ***kwargs*)

The merge routine could be used to merge this SpatialData instance with other instances. The merge creates a new merge dimension, name after the variable names. The grid of this instance is used as merged grid.

Parameters

- **items** (*xarray.DataArray or SpatialData*) – The items are merged with this SpatialData instance. The grid of the items have to be same as this SpatialData instance.
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The SpatialData instance with the merged data. If inplace is True, this instance is returned.

Return type *SpatialData*

merge_analysis_timedelta(*analysis_axis='runtime'*, *timedelta_axis='time'*, *inplace=False*)

The analysis time axis will be merged with the valid time axis, which should be given as timedelta. The merged time axis is called validtime and will be the first data axis.

Parameters

- **analysis_axis** (*str, optional*) – The analysis time axis name. This axis will be used as basis for the valid time. Default is runtime.
- **timedelta_axis** (*str, optional*) – The time delta axis name. This axis should contain the difference to the analysis time.
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The SpatialData instance with the replaced axis.

Return type *SpatialData*

plot(*method='contourf'*)

remapbil(*new_grid*, *inplace=False*)
Remap the horizontal grid with a bilinear approach to a given new grid.

Parameters

- **new_grid** (*Child instance of Grid*) – The data is remapped to this grid.
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The SpatialData instance with the replaced grid.

Return type *SpatialData*

remapnn(*new_grid*, *inplace=False*)
Remap the horizontal grid with the nearest neighbour approach to a given new grid.

Parameters

- **new_grid** (*Child instance of Grid*) – The data is remapped to this grid.
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The SpatialData instance with the replaced grid.

Return type *SpatialData*

save(*path*)
To save the SpatialData a copy of this instance is created and the grid dict of the grid is added to the SpatialData attributes. Then the instance is saved as NetCDF file.

Parameters **path** (*str*) – The path where the netcdf file should be saved.

sellonlatbox(*lonlatbox*, *inplace=False*)
The data is sliced with the given lonlatbox. A new grid is created based on the sliced coordinates.

Parameters

- **lonlatbox** (*tuple(float)*) – The longitude and latitude box with four entries as degree. The entries are handled in the following way:

  (left/west, top/north, right/east, bottom/south)

- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The sliced SpatialData instance with the replaced grid.

Return type *SpatialData*

set_grid_coordinates(*grid=None*, *data=None*)
Set the coordinates of the data to the coordinates of the given grid.

Parameters

- **grid** (*Child instance of Grid or None, optional*) – The grid of this instance is set to this grid. If this is None instance's grid is used. The last dimensions of instance's data is set according to to the grid. Default is None.
- **data** (*np.ndarray or None, optional*) – The data is set to this data values. The data values should have the same last dimension as the new grid. If this is None, the data values of this instance are used. Default is None.

to_tsdata(*lonlat=None*)
Transform the SpatialData to a TSData based on given coordinates. If coordinates are given this method selects the nearest neighbour grid point to this coordinates. The data is flatten to a 2d-Array with the time as row axis.

Parameters **lonlat** (*tuple(float, float) or None*) – The nearest grid point to this coordinates (longitude, latitude) is used to generate the time series data. If lonlat is None no coordinates will be selected and the data is flatten. If the horizontal grid coordiantes are not a single point it is recommended to set lonlat.

Returns **extracted_data** – The extracted TSData instance. The data is based on either a pandas Series or Dataframe depending on the dimensions of this SpatialData.

Return type *TSData*

update(**items*)
The update routine could be used to update the data of this SpatialData, based on either xarray.DataArrays or other SpatialData. There are some assumptions done:

1. The used data to update this SpatialData instance has the same grid and dimension variables as this instance. 2. Beginning from the left the given items are used to update the data. Such that intersection problems are resolved in favor of the newest data.

Parameters **items** (*xarray.DataArray or SpatialData*) – The items are used to update the data of this SpatialData instance. The grid has to be the same as this SpatialData instance.

Time series data

class pymepps.metdata.tsdata.**TSData**(*data*, *data_origin=None*, *lonlat=None*)
Bases: *pymepps.metdata.metdata.MetData*

TSData is a data structure for time series based data. This class is for meteorological measurement station observations and forecasts. Its instances are based on pandas.dataframe. So it's possible to use every operation on this structure specified in the documentation of pandas [1]. This structure has usually only one dimension. This data type has only one flexible dimension and the other dimensions are fixed in comparison to ArrayBasedData.

[1] (http://pandas.pydata.org/pandas-docs/stable/)

data
pandas.dataframe – The data of this time series based data structure.

data_origin
object of pymepps – The origin of this data.This could be a model run, a station, a database or something else.

`lonlat`
tuple(float, float) or None, optional – The data of this instance is valid for this coordinates (longitude, latitude). If this is None the coordiantes are not set and not all features could be used. Default is None.

Parameters

- **`data`** (*`pandas.dataframe`*) – The data of this time series based data structure.
- **`data_origin`** (*`object of pymepps`*) – The origin of this data.This could be a model run, a station, a database or something else.
- **`lonlat`** (*`tuple(float, float) or None`*) – The data is valid for these coordinates.

`copy`()

static **`load`**(*path*)
Load the given json file and return a TSData instance with the loaded file. The loader uses tries to locate the lonlat and the data keys within the json file. If there are not these keys the loader tries to load the whole json file into pandas.

Parameters **`path`** (*`str`*) – Path to the json file which should be loaded. It is recommended to load only previously saved TSData instances.

Returns **tsdata** – The loaded TSData instance.

Return type *TSData*

`plot`(*variable*, *type*, *color*)

`save`(*path*)
The data is saved as json file. The pandas to_json method is used to generate convert the data to json. If lonlat was given it will be saved under a lonlat key. Json is used instead of HDF5 due to possible corruption problems.

Parameters **`path`** (*`str`*) – Path where the json file should be saved.

`slice_index`(*start=”*, *end=”*, *inplace=False*)

inplace: bool, optional If the new data should be replacing the data of this TSData instance or if the instance should be copied. Default is None.

Returns **tsdata** – The TSData instance with the sliced index.

Return type *TSData*

`update`(**items*)

4.3 Grid package

The grid package is an extension for the SpatialDataset to support horizontal grid transformations and functions.

4.3.1 Grid builder

class `pymepps.grid.builder.`**`GridBuilder`**(*griddes*)

build_grid()

This method build up the grid with the griddes attribute.

Returns **grid** – The built grid. The class of the grid is defined by the gridtype. The values of the grid are calculated with griddes.

Return type child instance of Grid

static **decode_str**(*grid_str*)

Method to clean the given grid str and to get a python dict. Key and value are separated with =. Every new key value pair needs a new line delimiter. Only alphanumeric characters are allowed as key and value. To delimit a value list use spaces and new lines. Lines with # are used as comment lines.

Steps to decode the grid string:

1. String splitting by new line delimiter
2. Clean the lines from unallowed characters
3. Split the non-comment lines to key, value pairs
4. Append elements where no key, value pair is available to the previous value
5. Clean and split the key, value elements from spaces
6. Convert the values to float numbers

Parameters **grid_str** (*str or list(str)*) – The given grid_str which should be decoded. If this is a string the string will be splitten by new line into a list. It is necessary that every list entry has only one key = value entry.

Returns **grid_dict** – The decoded grid dict from the str.

Return type dict(str, str or float)

griddes

static **open_string**(*path_str*)

This method is used to check if the given str is a path or a grid string.

Parameters **path_str** (*str*) – This string is checked and if it is a path it will be read.

Returns **grid_str** – The given str or the read str.

Return type str

Raises `TypeError` – If path_str is not a str type.

4.3.2 Grids

BaseGrid

class pymepps.grid.grid.**Grid**(*grid_dict*)

The base class for every grid type.

static **convert_to_deg**(*field*, *unit*)

Method to convert given field with given unit into degree.

Parameters

- **field** –
- **unit** –

copy()

get_coord_names()

Returns the name of the coordinates.

Returns

- **yname** (*str*) – The name of the y-dimension.
- **xname** (*str*) – The name of the x-dimension

get_coords()

Get the coordinates in a xarray-compatible way.

Returns coords – The coordinates in a xarray compatible coordinates format. The key is the coordinate name. The coordinates have as value a tuple with their own name, indicating that the they are self-describing, and the coordinate values as numpy array.

Return type dict(str, (str, numpy.ndarray))

get_nearest_point(*data*, *coord*)

Get the nearest neighbour grid point for a given coordinate. The distance between the grid points and the given coordinates is calculated with the haversine formula.

Parameters

- **coord** (*tuple(float, float)*) – The data of the nearest grid point to this coordinate (latitude, longitude) will be returned. The coordinate should be in degree.
- **data** (*numpy.array*) – The return value is extracted from this array. The array should have at least two dimensions. If the array has more than two dimensions the last two dimensions will be used as horizontal grid dimensions.

Returns nearest_data – The extracted data for the nearest neighbour grid point. The dimensions of this array are the same as the input data array without the horizontal coordinate dimensions. There is at least one dimension.

Return type numpy.ndarray

lat_lon

Get latitudes and longitudes for every grid point as xarray.Dataset.

Returns lat_lon – The latitude and longitude values for every grid point as xarray.Dataset with latitude and longitude as variables.

Return type xarray.Dataset

len_coords

Get the number of coordinates for this grid.

Returns len_coords – Number of coordinates for this grid.

Return type int

lonlatbox(*data*, *ll_box*)

static **normalize_lat_lon**(*lat*, *lon*, *data=None*)

The given coordinates will be normalized and reorder into basemap conform coordinates. If the longitude values are between 0° and 360°, they will be normalized to values between -180° and 180°. Then the coordinates will be reorder, such that they are in an increasing order.

Parameters

- **lat** (*numpy.ndarray*) – The latitude values. They are representing the first data dimension.

- **lon** (*numpy.ndarray*) – The longitude values. They are representing the second data dimension.
- **data** (*numpy.ndarray or None, optional*) – The data values. They will be also reordered by lat and lon. If this is None, only lat and lon will be reordered and returned. Default is None.

Returns

- **lat** (*numpy.ndarray*) – Ordered latitude values.
- **lon** (*numpy.ndarray*) – Ordered and normalized longitude values.
- **data** (*numpy.ndarray or None*) – The orderd data based on given latitudes and longitudes. This is None if no other data was given as parameter.

raw_dim

Get the raw dimension values, as they are constructed by the grid description.

Returns **constructed_dim** – The constructed dimensions. Depending on the given grid type, it is either a tuple of arrays or a single array.

Return type tuple(numpy.ndarray) or numpy.ndarray

remapbil (*data*, *other_grid*)

The given data will be remapped via bilinear interpolation to the given other grid.

Parameters

- **data** (*numpy.ndarray*) – The data which should be remapped. There have to be at least two dimensions. If the data has more than two dimensions we suppose that the last two dimensions are the horizontal grid dimensions.
- **other_grid** (*child instance of Grid*) – The data will be remapped to this grid.

Returns **remapped_data** – The remapped data. The shape of the last two dimensions is now the shape of the other_grid coordinates.

Return type numpy.ndarray

Notes

Technically basemap's interp with order=1 is used to interpolate the data.

remapnn (*data*, *other_grid*)

The given data will be remapped via nearest neighbour to the given other grid.

Parameters

- **data** (*numpy.ndarray*) – The data which should be remapped. There have to be at least two dimensions. If the data has more than two dimensions we suppose that the last two dimensions are the horizontal grid dimensions.
- **other_grid** (*child instance of Grid*) – The data will be remapped to this grid.

Returns **remapped_data** – The remapped data. The shape of the last two dimensions is now the shape of the other_grid coordinates.

Return type numpy.ndarray

Notes

Technically basemap's interp with order=0 is used to interpolate the data.

`pymepps.grid.grid.`**`distance_haversine`**(*p1*, *p2*)

Calculate the great circle distance between two points on the earth. The formula is based on the haversine formula[1].

Parameters

- **`p1`** (*`tuple (array_like, array_like)`*) – The coordinates (latitude, longitude) of the first point in degrees.
- **`p2`** (*`tuple (array_like, array_like)`*) – The coordinates (latitude, longitude) of the second point in degrees.

Returns **d** – The calculated haversine distance in meters.

Return type float

Notes

Script based on: http://stackoverflow.com/a/29546836

References

LonLat

class `pymepps.grid.lonlat.`**`LonLatGrid`**(*grid_dict*)

Bases: *`pymepps.grid.grid.Grid`*

A LonLatGrid is a grid with evenly distributed longitude and latitude values. This is the right grid if the grid could be described with a evenly distributed range of values for longitude and latitude.

`lonlatbox`(*data*, *ll_box*)

The data is sliced with given lonlat box.

Parameters

- **`data`** (*`numpy.ndarray`*) – The data which should be sliced. The shape of the last two dimensions should be the same as the grid dimensions.
- **`ll_box`** (*`tuple(float)`*) – The longitude and latitude box with four entries as degree. The entries are handled in the following way:

  (left/west, top/north, right/east, bottom/south)

Returns

- **sliced_data** (*numpy.ndarray*) – The sliced data. The last two dimensions are sliced.
- **grid** (*Grid*) – A new child instance of Grid with the sliced coordinates as values.

[1] de Mendoza y Ríos, Memoria sobre algunos métodos nuevos de calcular la longitud por las distancias lunares: y aplication de su teórica á la solucion de otros problemas de navegacion, 1795.

Gaussian

class pymepps.grid.gaussian.**GaussianGrid**(*grid_dict*)

Bases: *pymepps.grid.lonlat.LonLatGrid*

The gaussian grid is similar to the lonlat grid. This is the right grid if longitude and/or latitude could be described with a non-evenly distributed list of values.

Projection

class pymepps.grid.projection.**BaseProj**

BaseProj is a base class for every projection in a proj4-conform way.

transform_from_latlon(*lon*, *lat*)

Transform the given lon, lat arrays to x and y values.

transform_to_latlon(*x*, *y*)

Transform the given x and y arrays to latitude and longitude values.

class pymepps.grid.projection.**ProjectionGrid**(*grid_dict*)

A projection grid could be defined by a evenly distributed grid. The grid could be translated to a longitude and latitude grid by a predefined projection. At the moment only projections defined by a proj4 string or a rotated latitude and longitude are supported.

get_projection()

lonlatbox(*data*, *ll_box*)

The data is sliced with given lonlat box to a unstructured grid.

Parameters

- **data** (*numpy.ndarray*) – The data which should be sliced. The shape of the last two dimensions should be the same as the grid dimensions.
- **ll_box** (*tuple(float)*) – The longitude and latitude box with four entries as degree. The entries are handled in the following way:

  (left/west, top/north, right/east, bottom/south)

Returns

- **sliced_data** (*numpy.ndarray*) – The sliced data. The last two dimensions are flattened and sliced.
- **grid** (*UnstructuredGrid*) – A new instance of UnstructuredGrid with the sliced coordinates as values.

class pymepps.grid.projection.**RotPoleProj**(*npole_lat*, *npole_lon*)

Class for to calculate the transformation from rotated pole coordinates to normal latitude and longitude coordinates. The rotated pole coordinates are calculated in a cf-conform manner, with a rotated north pole. The calculations are based on[1]. If the resulting latitude coordinate equals -90° or 90° the longitude coordinate will be set to 0°.

Parameters

- **npole_lat** (*float*) – The latitude of the rotated north pole in degrees.
- **npole_lom** (*float*) – The longitude of the rotated north pole in degrees.

References

[1] http://de.mathworks.com/matlabcentral/fileexchange/43435-rotated-grid-transform

`lonlatbox` (*data*, *ll_box*)
The data is sliced with given lonlat box to a unstructured grid.

Parameters

- **data** (*numpy.ndarray*) – The data which should be sliced. The shape of the last two dimensions should be the same as the grid dimensions.
- **ll_box** (*tuple(float)*) – The longitude and latitude box with four entries as degree. The entries are handled in the following way:

  (left/west, top/north, right/east, bottom/south)

Returns

- **sliced_data** (*numpy.ndarray*) – The sliced data. The last two dimensions are flattened and sliced.
- **grid** (*UnstructuredGrid*) – A new instance of UnstructuredGrid with the sliced coordinates as values.

`north_pole`
Get the north pole for the rotated pole projection.

`transform_from_lonlat` (*lon*, *lat*)
Transform the given lon, lat arrays to x and y values.

`transform_to_lonlat` (*x*, *y*)
Transform the given x and y arrays to latitude and longitude values.

Unstructured

class `pymepps.grid.unstructured.UnstructuredGrid` (*grid_dict*)
Bases: *pymepps.grid.grid.Grid*

In an unstructured grid the grid could have any shape. A famous example is the triangulated ICON grid. At the moment the longitude and latitude values should have been precomputed. The grid could be calculated with the number of vertices and the coordinates of the boundary.

`get_coord_names` ()
Returns the name of the coordinates.

Returns **coord_names** – The coordinate name for this unstructured grid. This is always a list, with only one entry: ncells.

Return type list(str)

`len_coords`
Get the number of coordinates for this grid.

Returns **len_coords** – Number of coordinates for this grid.

Return type int

`lonlatbox` (*data*, *ll_box*)
The data is sliced with given lonlat box to a unstructured grid.

Parameters

- **data** (*numpy.ndarray*) – The data which should be sliced. The shape of the dimension should be the same as the grid dimension.
- **ll_box** (*tuple(float)*) – The longitude and latitude box with four entries as degree. The entries are handled in the following way:

  (left/west, top/north, right/east, bottom/south)

Returns

- **sliced_data** (*numpy.ndarray*) – The sliced data. The last dimension is sliced.
- **grid** (*UnstructuredGrid*) – A new instance of UnstructuredGrid with the sliced coordinates as values.

Curvilinear

class pymepps.grid.curvilinear.**CurvilinearGrid**(*grid_dict*)

Bases: *pymepps.grid.lonlat.LonLatGrid*

A curvilinear grid could be described as special case of a lonlat grid where the number of vertices is 4. The raw grid values are calculated based on the given grid rules. At the moment the lon lat values had to be precomputed.

lonlatbox(*data*, *ll_box*)

The data is sliced with given lonlat box to a unstructured grid.

Parameters

- **data** (*numpy.ndarray*) – The data which should be sliced. The shape of the last two dimensions should be the same as the grid dimensions.
- **ll_box** (*tuple(float)*) – The longitude and latitude box with four entries as degree. The entries are handled in the following way:

  (left/west, top/north, right/east, bottom/south)

Returns

- **sliced_data** (*numpy.ndarray*) – The sliced data. The last two dimensions are flattened and sliced.
- **grid** (*UnstructuredGrid*) – A new instance of UnstructuredGrid with the sliced coordinates as values.

4.4 Data loader package

The data loader package is used to open time series and spatial data.

4.4.1 Base module

class pymepps.loader.base.**BaseLoader**(*data_path*, *file_type=None*, *processes=1*)

load_data()

4.4.2 Open model files

class `pymepps.loader.model.`**`ModelLoader`**(*data_path*, *file_type=None*, *grid=None*, *processes=1*)

Bases: *`pymepps.loader.base.BaseLoader`*

A simplified way to load weather model data into a SpatialDataset. Technically this class is a helper and wrapper around the file handlers and SpatialDataset.

Parameters

- **`data_path`** (*`str`*) – The path to the files. This path could have a glob-conform path pattern. Every file found within this pattern will be used to determine the file type and to generate the SpatialDataset.
- **`file_type`** (*`str or None, optional`*) – The file type determines which file handler will be used to load the data. If the file type is None it will be determined automatically based on given files. All the files with the majority file type will be used to generate the SpatialDataset. The available file_types are:

  nc: NetCDF files grib2: Grib2 files grib1: Grib1 files dap: Opendap urls
- **`grid`** (*`str or Grid or None, optional`*) – The grid describes the horizontal grid of the spatial data. The given grid will be forwarded to the given SpatialDataset instance. Default is None.

`pymepps.loader.model.`**`open_model_dataset`**(*data_path*, *file_type=None*, *grid=None*, *processes=1*)

4.4.3 Open station files

class `pymepps.loader.station.`**`StationLoader`**(*data_path*, *file_type=None*, *lonlat=None*, *processes=1*)

Bases: *`pymepps.loader.base.BaseLoader`*

A simplified way to load station data into a TSDataset. Technically this class is a helper and wrapper around the file handlers and TSData.

Parameters

- **`data_path`** (*`str`*) – The path to the files. This path could have a glob-conform path pattern. Every file found within this pattern will be used to determine the file type and to generate the TSDataset.
- **`file_type`** (*`str or None, optional`*) – The file type determines which file handler will be used to load the data. If the file type is None it will be determined automatically based on given files. All the files with the majority file type will be used to generate the TSDataset. The available file_types are:

  nc: NetCDF files wm: Text files in a specific “Wettermast format”
- **`lonlat`** (*`tuple(float, float), optional`*) – The lonlat coordinate tuple describes the position of the station in degrees. If this is None the position is unknown. Default is None.

`lon_lat`()

`pymepps.loader.station.`**`open_station_dataset`**(*data_path*, *file_type=None*, *lonlat=None*, *processes=1*)

4.5 Utilities package

4.5.1 Path encoder

class `pymepps.utilities.path_encoder.`**`PathEncoder`**(*base_path*, *date=None*, *undet_numbers=None*)

Bases: `object`

`get_encoded`()

Encode the path with given data.

Returns List with encoded paths.

Return type list of str

`get_file_number`()

4.5.2 TestCase

class `pymepps.utilities.testcase.`**`TestCase`**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

`assertAttribute`(*obj*, *attr*)

`assertCallable`(*obj*, *method*)

`assertMethod`(*obj*, *method*)

4.5.3 File

4.6 Submodule contents

class `pymepps.metdata.`**`SpatialDataset`**(*file_handlers*, *grid=None*, *data_origin=None*, *processes=1*)

Bases: `pymepps.metdata.metdataset.MetDataset`

SpatialDataset is a class for a pool of file handlers. Typically a spatial dataset combines the files of one model run, such that it is possible to select a variable and get a SpatialData instance. For memory reasons the data of a variable is only loaded if it is selected.

Parameters

- **`file_handlers`** (`list of childs of FileHandler or None`) – The spatial dataset is based on these files. The files should be either instances of GribHandler or NetCDFHandler. If file handlers is None then the dataset is used for conversion from TSData to SpatialData.
- **`grid`** (`str or Grid or None`) – The grid describes the horizontal grid of the spatial data. The grid will be appended to every created SpatialData instance. If a str is given it will be checked if the str is a path to a cdo-conform grid file or a cdo-conform grid string. If this is a instance of a child of Grid it is assumed that the grid is already initialized and this grid will be used. If this is None the Grid will be automatically read from the first file handler. Default is None.
- **`data_origin`** (`optional`) – The data origin. This parameter is important to trace the data flow. If this is None, there is no data origin and this dataset will be the starting point of the data flow. Default is None.

- **processes** (*int, optional*) – This number of processes is used to calculate time-consuming functions. For time-consuming functions a progress bar is shown. If the number of processes is one the functions will be processed sequential. For more processes than one the multiprocessing module will be used. Default is 1.

select()
Method to select a variable.

selnearest()
Method to select the nearest grid point for given coordinates.

sellonlatbox()
Method to slice a box with the given coordinates.

data_merge(*data*, *var_name*)
Method to merge instances of xarray.DataArray into a SpatialData instance. Also the grid is read and inserted into the SpatialData instance.

Parameters

- **data** (*list of xarray.DataArray*) – The data list.
- **var_name** (*str*) – The name of the variable which is selected within the data list.

Returns The SpatialData instance with the extracted data and the extracted grid.

Return type *SpatialData*

get_grid(*var_name*, *data_array=None*)
Method to get for given variable name a Grid instance. If the grid attribute is already a Grid instance this grid will be returned. If the grid attribute is a str instance, the str will be read from file or from the given grid str. If the grid attribute isn't set the grid instance will be the grid for the variable selected with the first corresponding file handler and cdo.

Parameters

- **var_name** (*str*) – The variable name, which should be used to generate the grid.
- **data_array** (*xarray.DataArray or None, optional*) – If the data array is given the method will try to load the grid from the data array's attributes. If None the DataArray method will be skipped. Default is None.

Returns **grid** – The returned grid. If the returned grid is None, the grid could not be read.

Return type Instance of child of grid or None

class pymepps.metdata.**SpatialData**(*data*, *grid=None*, *data_origin=None*)
Bases: *pymepps.metdata.metdata.MetData*

SpatialData contains spatial based data structures. This class is the standard data type for file types like netCDF or grib. It's prepared for the output of numerical and statistical weather models. Array based data is always saved to netcdf via xarray.

data
xarray.DataArray or None – The data of this grid based data structure.

grid
Child instance of Grid or None – The corresponding grid of this SpatialData instance. This grid is used to interpolate/remap the data and to select the nearest grid point to a given longitude/latitude pair. The grid is also used to get a basemap instance to determine the grid boundaries for plotting purpose.

data_origin
object of pymepps or None, optional – The origin of this data. This could be a model run, a station, a database or something else. Default is None.

grid

static load(*path*)

Load a SpatialData instance from a given path. The path is loaded as SpatialDataset. A correct saved SpatialData instance will have only one variable within the NetCDF file. So the first variable will be returned as newly constructed SpatialData instance.

Parameters **path** (*str*) – The path to the saved SpatialData instance.

Returns **spdata** – The loaded SpatialData instance.

Return type *SpatialData*

merge(**items*, ***kwargs*)

The merge routine could be used to merge this SpatialData instance with other instances. The merge creates a new merge dimension, name after the variable names. The grid of this instance is used as merged grid.

Parameters

- **items** (*xarray.DataArray or SpatialData*) – The items are merged with this SpatialData instance. The grid of the items have to be same as this SpatialData instance.
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The SpatialData instance with the merged data. If inplace is True, this instance is returned.

Return type *SpatialData*

merge_analysis_timedelta(*analysis_axis='runtime'*, *timedelta_axis='time'*, *inplace=False*)

The analysis time axis will be merged with the valid time axis, which should be given as timedelta. The merged time axis is called validtime and will be the first data axis.

Parameters

- **analysis_axis** (*str, optional*) – The analysis time axis name. This axis will be used as basis for the valid time. Default is runtime.
- **timedelta_axis** (*str, optional*) – The time delta axis name. This axis should contain the difference to the analysis time.
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The SpatialData instance with the replaced axis.

Return type *SpatialData*

plot(*method='contourf'*)

remapbil(*new_grid*, *inplace=False*)

Remap the horizontal grid with a bilinear approach to a given new grid.

Parameters

- **new_grid** (*Child instance of Grid*) – The data is remapped to this grid.
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The SpatialData instance with the replaced grid.

Return type *SpatialData*

remapnn (*new_grid*, *inplace=False*)
Remap the horizontal grid with the nearest neighbour approach to a given new grid.

Parameters

- **new_grid** (*Child instance of Grid*) – The data is remapped to this grid.
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The SpatialData instance with the replaced grid.

Return type *SpatialData*

save (*path*)
To save the SpatialData a copy of this instance is created and the grid dict of the grid is added to the SpatialData attributes. Then the instance is saved as NetCDF file.

Parameters **path** (*str*) – The path where the netcdf file should be saved.

sellonlatbox (*lonlatbox*, *inplace=False*)
The data is sliced with the given lonlatbox. A new grid is created based on the sliced coordinates.

Parameters

- **lonlatbox** (*tuple(float)*) – The longitude and latitude box with four entries as degree. The entries are handled in the following way:

  (left/west, top/north, right/east, bottom/south)
- **inplace** (*bool, optional*) – If the new data should be replacing the data of this SpatialData instance or if the instance should be copied. Default is False.

Returns **spdata** – The sliced SpatialData instance with the replaced grid.

Return type *SpatialData*

set_grid_coordinates (*grid=None*, *data=None*)
Set the coordinates of the data to the coordinates of the given grid.

Parameters

- **grid** (*Child instance of Grid or None, optional*) – The grid of this instance is set to this grid. If this is None instance's grid is used. The last dimensions of instance's data is set according to to the grid. Default is None.
- **data** (*np.ndarray or None, optional*) – The data is set to this data values. The data values should have the same last dimension as the new grid. If this is None, the data values of this instance are used. Default is None.

to_tsdata (*lonlat=None*)
Transform the SpatialData to a TSData based on given coordinates. If coordinates are given this method selects the nearest neighbour grid point to this coordinates. The data is flatten to a 2d-Array with the time as row axis.

Parameters **lonlat** (*tuple(float, float) or None*) – The nearest grid point to this coordinates (longitude, latitude) is used to generate the time series data. If lonlat is None no coordinates will be selected and the data is flatten. If the horizontal grid coordiantes are not a single point it is recommended to set lonlat.

Returns **extracted_data** – The extracted TSData instance. The data is based on either a pandas Series or Dataframe depending on the dimensions of this SpatialData.

Return type *TSData*

update(**items*)

The update routine could be used to update the data of this SpatialData, based on either xarray.DataArrays or other SpatialData. There are some assumptions done:

1. The used data to update this SpatialData instance has the same grid and dimension variables as this instance. 2. Beginning from the left the given items are used to update the data. Such that intersection problems are resolved in favor of the newest data.

Parameters **items** (*xarray.DataArray or SpatialData*) – The items are used to update the data of this SpatialData instance. The grid has to be the same as this SpatialData instance.

class pymepps.metdata.**TSDataset**(*file_handlers*, *data_origin=None*, *lonlat=None*, *processes=1*)

Bases: *pymepps.metdata.metdataset.MetDataset*

TSDataset is a class for a pool of file handlers. Typically a time series dataset combines the files of a station, such that it is possible to select a variable and get a TSData instance. For memory reasons the data of a variable is only loaded if it is selected.

Parameters

- **file_handlers** (*list of childs of FileHandler or None*) – The spatial dataset is based on these files. The files should be either instances of NetCDFHandler or TextHandler. If file handlers is None then the dataset is used for conversion from SpatialData to TSData.
- **data_origin** (*optional*) – The data origin. This parameter is important to trace the data flow. If this is None, there is no data origin and this dataset will be the starting point of the data flow. Default is None.
- **lonlat** (*tuple(float, float) or None*) – The coordinates (longitude, latitude) where the data is valid. If this is None the coordinates will be set based on data_origin or based on the first file handler.

select()

Method to select a variable.

data_merge(*data*, *var_name*)

select_by_pattern(*pattern*, *return_list=False*)

class pymepps.metdata.**TSData**(*data*, *data_origin=None*, *lonlat=None*)

Bases: *pymepps.metdata.metdata.MetData*

TSData is a data structure for time series based data. This class is for meteorological measurement station observations and forecasts. Its instances are based on pandas.dataframe. So it's possible to use every operation on this structure specified in the documentation of pandas [1]. This structure has usually only one dimension. This data type has only one flexible dimension and the other dimensions are fixed in comparison to ArrayBasedData.

[1] (http://pandas.pydata.org/pandas-docs/stable/)

data

pandas.dataframe – The data of this time series based data structure.

data_origin

object of pymepps – The origin of this data.This could be a model run, a station, a database or something else.

lonlat

tuple(float, float) or None, optional – The data of this instance is valid for this coordinates (longitude, latitude). If this is None the coordiantes are not set and not all features could be used. Default is None.

Parameters

- **`data`** (*`pandas.dataframe`*) – The data of this time series based data structure.
- **`data_origin`** (*`object of pymepps`*) – The origin of this data.This could be a model run, a station, a database or something else.
- **`lonlat`** (*`tuple(float, float) or None`*) – The data is valid for these coordinates.

`copy`()

static **`load`**(*path*)

Load the given json file and return a TSData instance with the loaded file. The loader uses tries to locate the lonlat and the data keys within the json file. If there are not these keys the loader tries to load the whole json file into pandas.

Parameters **`path`** (*`str`*) – Path to the json file which should be loaded. It is recommended to load only previously saved TSData instances.

Returns **tsdata** – The loaded TSData instance.

Return type *TSData*

`plot`(*variable*, *type*, *color*)

`save`(*path*)

The data is saved as json file. The pandas to_json method is used to generate convert the data to json. If lonlat was given it will be saved under a lonlat key. Json is used instead of HDF5 due to possible corruption problems.

Parameters **`path`** (*`str`*) – Path where the json file should be saved.

`slice_index`(*start=”*, *end=”*, *inplace=False*)

inplace: bool, optional If the new data should be replacing the data of this TSData instance or if the instance should be copied. Default is None.

Returns **tsdata** – The TSData instance with the sliced index.

Return type *TSData*

`update`(**items*)

class `pymepps.grid.`**`GridBuilder`**(*griddes*)

Bases: `object`

`build_grid`()

This method build up the grid with the griddes attribute.

Returns **grid** – The built grid. The class of the grid is defined by the gridtype. The values of the grid are calculated with griddes.

Return type child instance of Grid

static **`decode_str`**(*grid_str*)

Method to clean the given grid str and to get a python dict. Key and value are separated with =. Every new key value pair needs a new line delimiter. Only alphanumeric characters are allowed as key and value. To delimit a value list use spaces and new lines. Lines with # are used as comment lines.

Steps to decode the grid string:

1. String splitting by new line delimiter

2. Clean the lines from unallowed characters
3. Split the non-comment lines to key, value pairs
4. Append elements where no key, value pair is available to the previous value
5. Clean and split the key, value elements from spaces
6. Convert the values to float numbers

Parameters **`grid_str`** (*`str or list(str)`*) – The given grid_str which should be decoded. If this is a string the string will be splitten by new line into a list. It is necessary that every list entry has only one key = value entry.

Returns **grid_dict** – The decoded grid dict from the str.

Return type dict(str, str or float)

`griddes`

static `open_string`(*path_str*)

This method is used to check if the given str is a path or a grid string.

Parameters **`path_str`** (*`str`*) – This string is checked and if it is a path it will be read.

Returns **grid_str** – The given str or the read str.

Return type str

Raises `TypeError` – If path_str is not a str type.

`pymepps.loader.`**`open_model_dataset`**(*data_path*, *file_type=None*, *grid=None*, *processes=1*)

`pymepps.loader.`**`open_station_dataset`**(*data_path*, *file_type=None*, *lonlat=None*, *processes=1*)

class `pymepps.utilities.`**`PathEncoder`**(*base_path*, *date=None*, *undet_numbers=None*)

Bases: `object`

`get_encoded`()

Encode the path with given data.

Returns List with encoded paths.

Return type list of str

`get_file_number`()

class `pymepps.utilities.`**`MultiThread`**(*processes*, *threads=True*)

Bases: `object`

`processes`

Python Module Index

Index