weatherDataHarmonizer

Release 1.2.0

Michael Wagner

Apr 26, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The weatherDataHarmonizer project facilitates the harmonizing of different weather data of the German Weatherservice (DWD). It incorporates recent data as well as forecast data from RadolanRW, RadvorRQ, RadolanRV, Icon-D2, Icon-D2-EPS, Icon-EU, and Icon-EU-EPS products.

This chapter includes examples for typical usages. The comprehensive technical description follows in the next chapter.

1.1 Content and usage for fast readers

The weatherDataHarmonizer ist parted into different packages (e.g. `met_entities`, `read_radolan`, `read_icon`), which can be imported into own projects. Various example implementations are given in https://gitlab.hrz.tu-chemnitz.de/ihmtud/weatherdataharmonizer/-/blob/main/eval_met_packages.py.

The code examples below show low level access using modules in `read_radolan`, and high level access for advanced reading and harmonizing features.

1.2 Example: low level access of a raw dataset

Using the low level access variant, it is possible to directly load a raw file of the appropriate data type. Currently supported are data of type Radolan, Icon, and Cosmo.

The following piece of code reads in a Radolan type data file and stores the data in a `RadolanData` class instance. Further, the longitudes and latitudes of all centerpoints of the original 900 x 900 grid are given in a tuple of `LonLatTime` class instances. The `readRadolan.read_radolan` method allows further attributes, e.g. `scale_factor` or `fill_value` to further influence the file reading.

```
from read_radolan import readRadolan as readRadolan

data1 = readRadolan.read_radolan('path/to/radolan/data_file')
lonlat = readRadolan.get_lonlat(format_version=4, grid_variant='radolanrx')
```

1.3 Example: read/export one raw dataset

This code example defines a target grid, reads in one time step of RadolanRW data, regrids the data using IDW method with the nearest three neighbors, and exports the resulting grid to a netcdf file.

```
import datetime as dt
import numpy as np
from met_entities.RadolanRW import RadolanRW
from met_entities.LonLatTime import LonLatTime
```

(continues on next page)

(continued from previous page)

```
# define target grid
lon = np.arange(4, 15, .3)
lat = np.arange(54, 48, -.3)
lon_target, lat_target = np.meshgrid(lon, lat)
lon_target = LonLatTime(data=lon_target)
lat_target = LonLatTime(data=lat_target)

# read, regrid, and export data
start_datetime = dt.datetime(2022, 10, 25, 6, 50)
radrw = RadolanRW()
radrw.read_file(start_datetime=start_datetime, directory='path/to/data')
radrw.regrid(lon_target=lon_target, lat_target=lat_target)
radrw.export_netcdf('path/to/netcdf.nc')
```

1.4 Example: read/export multiple raw datasets

The next example shows the collection of data from multiple time steps. Here, only IconD2 data is read and exported/appended to a netcdf file. At the first time step the netcdf file is created. All other steps will append data to this file along the time dimension. The data is packed via `data_format='i2'` and `scale_factor_nc=0.01` (explanation see below in *Example: read/harmonize/export multiple datasets of different types*).

```
import datetime as dt
import numpy as np
from met_entities.LonLatTime import LonLatTime
from met_entities.IconD2 import IconD2

# define target grid
lon = np.arange(4, 15, .3)
lat = np.arange(54, 48, -.3)
lon_target, lat_target = np.meshgrid(lon, lat)
lon_target = LonLatTime(data=lon_target)
lat_target = LonLatTime(data=lat_target)

# define forecast time steps to be read in
start_datetime = dt.datetime(2022, 12, 14, 9)
end_datetime = dt.datetime(2022, 12, 14, 15)
period_step = dt.timedelta(hours=3)
periods = [start_datetime]
ct = 0
while periods[-1] < end_datetime:
    ct = ct + 1
    periods.append(start_datetime + ct * period_step)

# read, regrid, and export/append data
nc_file = 'path/to/netcdf.nc'
mode = 'create'
for period in periods:
    print(period)
    icond2 = IconD2()
    icond2.read_file(period, directory='path/to/data', forecast_hours=2, fill_value=-
↪1)
    icond2.regrid(lon_target=lon_target, lat_target=lat_target, file_nearest='path/to/
↪regridding/rule.npz')
    if mode == 'create':
```

(continues on next page)

(continued from previous page)

```
        icond2.export_netcdf(nc_file, data_format='i2', scale_factor_nc=0.01)
    else:
        icond2.export_netcdf_append(nc_file)
    mode = 'append'
```

1.5 Example: read/harmonize/export multiple datasets of different types

The class WeatherData is developed for the harmonizing of different data resources (the same temporal and spatial resolution, the same scaling and the same filling of missing values). For instance, we start with measured RadolanRW data from 2 hours ago until the most recent time step, include IconD2 forecast data and afterwards replace the nearest forecast by nowcast products like RadolanRV. The latter two products are assumed to be stored in monthly separated directories (e.g. .../icond2/202302/...). All summarized data shall be exported as a netcdf file with a filename reflecting the actual datetime. The following code shows this example.

```
import numpy as np
import datetime as dt

from met_entities.VariableDescription import RegridDescription
from met_entities.LonLatTime import LonLatTime
from met_entities.WeatherData import WeatherData

def main_combined_data():
  # define an arbitrary target grid
  lon = np.arange(4, 15, .3)
  lat = np.arange(54, 48, -.3)
  lon_target, lat_target = np.meshgrid(lon, lat)
  lon_target = LonLatTime(data=lon_target)
  lat_target = LonLatTime(data=lat_target)

  # define the time of the supposed last observation
  time_now = dt.datetime.now(tz=dt.timezone.utc)
  time_pivot = dt.datetime(time_now.year, time_now.month, time_now.day, time_now.hour,
↪ tzinfo=dt.timezone.utc)

  # define regridding descriptions for all used products
  regrid_description = {'radolanrw': RegridDescription(lon_target=lon_target, lat_
↪target=lat_target,
                                                       file_nearest='data/radrw_
↪regridding.npz'),
                        'radolanrv': RegridDescription(lon_target=lon_target, lat_
↪target=lat_target,
                                                       file_nearest='data/radrv_
↪regridding.npz'),
                        'icond2': RegridDescription(lon_target=lon_target, lat_
↪target=lat_target,
                                                    file_nearest='data/icond2_
↪regridding.npz')}

  # instantiate WeatherData class with central specifications (temporal/spatial␣
↪resolution, filling, scaling, usage of
  # np.int16 type for memory saving)
  wd = WeatherData(time_now=time_pivot, delta_t=dt.timedelta(minutes=15), fill_value=-
```

(continues on next page)

(continued from previous page)

```
↪1, scale_factor=0.01,
                  regrid_description=regrid_description, short='int16')

 # collect all data, RadolanRW starting two hours ago
 wd.collect_radolanrw(time_start=time_pivot - dt.timedelta(hours=2), directory='path/
↪to/radolanrw')
 wd.collect_icond2(latest_event=time_now, directory='path/to/icond2', dir_time_
↪descriptor=['%Y%m'])
 wd.collect_radolanrv(latest_event=time_now, directory='/mnt/08_hwstore/RadolanRV',␣
↪dir_time_descriptor=['%Y%m'])

 # export to netcdf with packed data (type i2: short integer, internal netcdf␣
↪scaling of 0.01, undo the scaling from
 # above); the large data variables are compressed with zlib and compression level 4
 filename_nc = f'data/precipitation_data_example_{wd.time_now.strftime("%Y%m%d%H%M")}
↪.nc'
 wd.export_netcdf(filename=filename_nc, institution='Institution as global attribute
↪', data_format='i2',
                  scale_factor_nc=0.01, scale_undo=True, data_kwargs={'compression':
↪'zlib', 'complevel': 4})
```

Please note the `scale_factor=0.01` in the WeatherData instance in combination with `short='int16'`. This combination guarantees a precision of two floating points with a possible maximum of 327 (reasonable for 15 min precipitation). The data could be left in original float precision (simply omit `scale_factor` and `short`) if memory consumption is not limiting. The possibility of `short` is specifically included for IconD2EPS data that consumes a lot more memory if used for the whole prediction region and time.

Please also note the exporting to netcdf method with `data_format='i2'`, `scale_factor_nc=0.01`, and `scale_undo=True`. This combination takes back the scaling from data import and defines an internal scaling for netcdf and short integer type for the data. The resulting netcdf file will be much smaller but ensures the same precision of two floating points. This method is known as packing data values. All typical libraries to access netcdf content consider the `scale_factor` variable attribute and de-pack the data automatically. The accompanying `add_offset` attribute for packing is not supported here as the data is typically extended over time and a perfect packing (recommendations here: https://docs.unidata.ucar.edu/nug/current/best_practices.html) cannot be done. Moreover, it would introduce more complexity with missing values.

Further the data variable is compressed to save storage.

1.6 Example: Download and collect Radolan data

To facilitate retrieving the data a download helper is incorporated in the small package `download_data`. It enables the download of recent data of currently RadolanRW, RadvorRQ, RadolanRV, IconD2, IconD2EPS, IconEU, and IconEUEPS data. Please note, that nowcast and forecast data in particular typically do not stay on the website longer than 24 hours.

In the code block below the downloader is used to obtain the data from https://opendata.dwd.de/. Further a the data is read in, cropped, and exported to netCDF. All time steps after the first one are appended to the existing netCDF file.

```
import datetime as dt
from download_data.DownloadJob import DownloadJob
from met_entities.RadolanRW import RadolanRW

start_datetime = dt.datetime(2023, 4, 28, 2, tzinfo=dt.timezone.utc)
end_datetime = dt.datetime(2023, 4, 28, 12, tzinfo=dt.timezone.utc)
```

(continues on next page)

(continued from previous page)

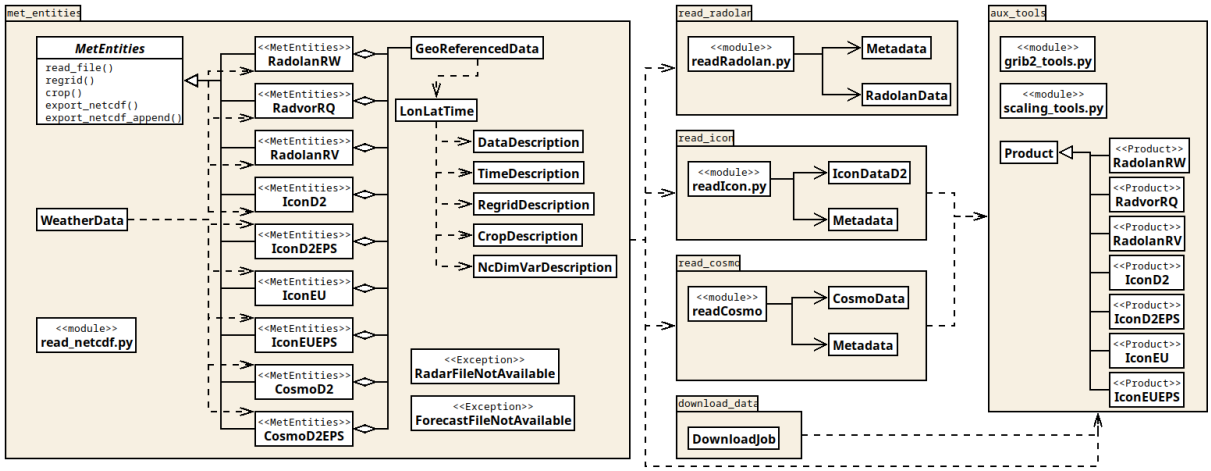
```
dj = DownloadJob(product='RadolanRW', directory='tmp', date_start=start_datetime,␣
↪date_end=end_datetime)
dj.download_files()

period_step = dt.timedelta(hours=1)
periods = [start_datetime]
ct = 0
while periods[-1] < end_datetime:
    ct = ct + 1
    periods.append(start_datetime + ct * period_step)
mode = 'create'
for period in periods:
    radrw = RadolanRW()
    radrw.read_file(start_datetime=period, directory='tmp', fill_value=-1)
    radrw.crop(lon_west=11.7, lon_east=15.2, lat_south=50.1, lat_north=51.8)
    if mode == 'create':
        radrw.export_netcdf('radrw_example.nc', data_format='i2', scale_factor_nc=0.1)
    else:
        radrw.export_netcdf_append('radrw_example.nc')
    mode = 'append'

dj.delete_files()
```

1.7 Structure

In the image below the UML class diagram of the weatherDataHarmonizer is depicted. A detailed version can be found in https://gitlab.hrz.tu-chemnitz.de/ihmtud/weatherdataharmonizer/-/blob/main/weatherDataHarmonizer_UML.png.



1.8 License

The weatherDataHarmonizer is licensed under Apache-2.0. You may obtain a copy of the License in the project's LICENSE at https://gitlab.hrz.tu-chemnitz.de/ihmtud/weatherdataharmonizer/-/blob/main/LICENSE or at http://www.apache.org/licenses/LICENSE-2.0.

1.9 Get Code and Contact

The most recent version of weatherDataHarmonizer package can be obtained at https://gitlab.hrz.tu-chemnitz.de/ihmtud/weatherdataharmonizer/. The author can be contacted via email: michael.wagner@tu-dresden.de.

CHAPTER

TWO

READ_RADOLAN PACKAGE

The read_radolan package provides access to radolan formatted data from the german weatherservice (DWD).

The module readRadolan.py includes reading compressed and uncompressed DWD binaries. Further it provides a function to get the coordinates (lon/lat) of sphere earth model (older version) and the newer WGS84 ellipsoid earth model.

Metadata module has Metadata class with relevant metadata of radolan formatted data.

RadolanData module has RadolanData class with relevant data of radolan formatted data.

2.1 read_radolan.Metadata module

class read_radolan.Metadata.**Metadata**(*product=None*, *datum=None*, *datum_iso=None*, *product_length=None*, *format_version=None*, *version=None*, *interval_length=None*, *ncols=None*, *nrows=None*, *coord_ll=None*, *prediction_time=None*, *module_flags=None*, *quantification_kind=None*, *sites=None*)

Bases: object

Metadata class contains relevant metadata of radolan formatted data.

2.2 read_radolan.RadolanData module

class read_radolan.RadolanData.**RadolanData**(*data=None*, *metadata=None*, *idx_clutter=None*, *idx_nan=None*)

Bases: object

RadolanData class contains all relevant data of radolan formatted data.

2.3 read_radolan.readRadolan module

read_radolan.readRadolan.**get_lonlat**(*format_version*, *grid_variant=None*, *num_rows=None*, *target_epsg=4326*, *corners=False*)

Calculate longitudes and latitudes for centerpoints of a given Radolan raster. The format version (identifier VS in radolan binary files) must be given, as until 4 a sphere earth model is used and from 5 on the WGS84 ellipsoid as earth model is applied.

Parameters

- **format_version** (*int*) – version from VS identifier in radolan binary files; relevant is only <=4 or >=5

- **grid_variant** (*str, optional*) – either radolanrx, de1200, de4800, or europecomposite (case-insensitive); either grid variant or num_rows must be given
- **num_rows** (*int, optional*) – either 900, 1200, 4800, or 2400; either grid variant or num_rows must be given
- **target_epsg** (*int, optional*) – epsg number of target crs (default: 4326)
- **corners** (*bool, optional*) – if true, the outer coordinates of all four corners are returned (default: False)

Returns
longitude and latitude objects

Return type
met_entities.LonLatTime.LonLatTime, *met_entities.LonLatTime.LonLatTime*

read_radolan.readRadolan.**get_lonlat_sphere**(*num_rows*, *corners=False*)

Calculate longitudes and latitudes for centerpoints of the currently used Radolan rasters (either 900 or 1100 rows long). The eastbound shift of 1100 rows products is respected.

Parameters

- **num_rows** (*int*) – number of rows (900 or 1100)
- **corners** (*bool, optional*) – if true, the outer coordinates of all four corners are returned (default: False)

Returns
longitude and latitude objects

Return type
met_entities.LonLatTime.LonLatTime, *met_entities.LonLatTime.LonLatTime*

read_radolan.readRadolan.**read_radolan**(*filename*, *scale_factor=1*, *fill_value=nan*, *metadata_only=False*, *short=None*)

Read in radolan formatted data as bz2 or gz compressed data or as binary.

Parameters

- **filename** (*str*) – name of file; distinction is made upon the file extension (.bz2, .gz, or nothing)
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **metadata_only** (*bool, optional*) – if True, only the metadata of the file is extracted
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Returns
RadolanData class or Metadata class with corresponding content

Return type
read_radolan.RadolanData.RadolanData, read_radolan.RadolanData.Metadata

read_radolan.readRadolan.**read_radolan_binary**(*filename*, *scale_factor=1*, *fill_value=nan*, *metadata_only=False*, *short=None*)

Read in radolan formatted binary data.

Parameters

- **filename** (*str*) – name of file
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value

- **metadata_only** (*bool, optional*) – if True, only the metadata of the file is extracted
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Returns
RadolanData class or Metadata class with corresponding content

Return type
read_radolan.RadolanData.RadolanData, read_radolan.RadolanData.Metadata

read_radolan.readRadolan.**read_radolan_bz2**(*filename*, *scale_factor=1*, *fill_value=nan*, *metadata_only=False*, *short=None*)

Read in radolan formatted and bzip2 compressed data.

Parameters

- **filename** (*str*) – name of file
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **metadata_only** (*bool, optional*) – if True, only the metadata of the file is extracted
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Returns
RadolanData class or Metadata class with corresponding content

Return type
read_radolan.RadolanData.RadolanData, read_radolan.RadolanData.Metadata

read_radolan.readRadolan.**read_radolan_data**(*stream*, *scale_factor=1*, *fill_value=nan*, *metadata_only=False*, *short=None*)

Read radolan formatted data from stream.

Parameters

- **stream** (*bytes*) – the data stream
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **metadata_only** (*bool, optional*) – if True, only the metadata of the file is extracted
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Returns
RadolanData class or Metadata class with corresponding content

Return type
read_radolan.RadolanData.RadolanData, read_radolan.RadolanData.Metadata

read_radolan.readRadolan.**read_radolan_gz**(*filename*, *scale_factor=1*, *fill_value=nan*, *metadata_only=False*, *short=None*)

Read in radolan formatted and gzip compressed data.

Parameters

- **filename** (*str*) – name of file
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value

- **fill_value** (*float, optional*) – missing data is filled with that value
- **metadata_only** (*bool, optional*) – if True, only the metadata of the file is extracted
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Returns
RadolanData class or Metadata class with corresponding content

Return type
read_radolan.RadolanData.RadolanData, read_radolan.RadolanData.Metadata

CHAPTER

THREE

READ_ICON PACKAGE

The read_icon package provides access to Icon formatted data from the german weather service (DWD).

The module readIcon.py provides a function to read in typical Icon data, either as bz2 compressed or uncompressed grib data. In the case of compressed data a temporary file is written (in OS specific temporary folder) and deleted afterwards. Further a function is incorporated to return coordinates (lon/lat).

Metadata module has Metadata class with relevant metadata of Icon formatted data.

IconData module has IconData class with relevant data of Icon formatted data.

3.1 read_icon.IconData module

class read_icon.IconData.**IconData**(*data=None*, *metadata=None*)

Bases: object

IconData class contains all relevant data and metadata of Icon formatted data.

3.2 read_icon.Metadata module

class read_icon.Metadata.**Metadata**(*centre=None*, *centre_description=None*, *datum=None*, *datum_end_of_overall_time_interval=None*, *datum_iso=None*, *prediction_time=None*, *step_type=None*, *grid_type=None*, *name=None*, *units=None*, *missing_value=None*, *number_of_missing=None*, *number_of_data_points=None*, *fill_value=None*)

Bases: object

Metadata class contains relevant metadata of icon formatted data.

3.3 read_icon.readIcon module

read_icon.readIcon.**get_lonlat**(*variant*, *clon_file=None*, *clat_file=None*)

Read longitudes and latitudes from clon/clat files (either grib2 files or bz2 compressed files).

Parameters

- **variant** (*str*) – variant of Icon data (case-insensitive), currently supported: Icon-D2 - 'd2', Icon-D2-EPS - 'd2eps', Icon-EU - 'eu', Icon-EU-EPS - 'eueps'
- **clon_file** (*str, optional*) – grib2 or bz2 compressed grib2 file with center longitudes of Icon; if not given the file from the resources directory in this package is used
- **clat_file** (*str, optional*) – grib2 or bz2 compressed grib2 file with center latitudes of Icon; if not given the file from the resources directory in this package is used

Returns
center longitudes and latitudes

Return type
met_entities.LonLatTime.LonLatTime, *met_entities.LonLatTime.LonLatTime*

read_icon.readIcon.**read_icon_d2**(*filename*, *scale_factor=1*, *fill_value=-999*, *variant='d2'*, *short=None*)

Read in IconD2 or IconEU data from file, either as grib2 file or as bz2 compressed grib2 file. Currently, IconD2, IconD2EPS, IconEU, and IconEUEPS are supported.

Parameters

- **filename** (*str*) – name of file
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **variant** (*str, optional*) – specify variant, either 'd2' (default), 'd2eps', 'eu', or 'eueps'
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Returns
Icon data of four 15 min values (for some older data also one 1 h value possible) as in the content of the grib2 file

Return type
list

CHAPTER

FOUR

READ_COSMO PACKAGE

The read_cosmo package provides access to Cosmo formatted data from the german weather service (DWD).

The module readCosmo.py provides a function to read in typical Cosmo data, either as bz2 compressed or uncompressed grib data. In the case of compressed data a temporary file is written (in OS specific temporary folder) and deleted afterwards. Further a function is incorporated to return coordinates (lon/lat).

Metadata module has Metadata class with relevant metadata of Cosmo formatted data.

CosmoData module has CosmoData class with relevant data of Cosmo formatted data.

4.1 read_cosmo.CosmoData module

class read_cosmo.CosmoData.**CosmoData**(*data=None*, *metadata=None*)

Bases: object

CosmoData class contains all relevant data and metadata of Cosmo formatted data.

4.2 read_cosmo.Metadata module

class read_cosmo.Metadata.**Metadata**(*centre=None*, *centre_description=None*, *datum=None*, *datum_end_of_overall_time_interval=None*, *datum_iso=None*, *prediction_time=None*, *step_type=None*, *grid_type=None*, *name=None*, *units=None*, *missing_value=None*, *number_of_missing=None*, *number_of_data_points=None*, *fill_value=None*)

Bases: object

Metadata class contains relevant metadata of cosmo formatted data.

4.3 read_cosmo.readCosmo module

read_cosmo.readCosmo.**get_lonlat**(*corners=False*)

Calculate longitudes and latitudes for centerpoints of the last Cosmo-D2 rasters (716 rows, 651 columns).

Parameters

corners (*bool, optional*) – if true, the coordinates of all four corner points are returned (default: False)

Returns

longitude and latitude objects

Return type

met_entities.LonLatTime.LonLatTime, *met_entities.LonLatTime.LonLatTime*

read_cosmo.readCosmo.**read_cosmo_d2**(*filename*, *scale_factor=1*, *fill_value=-999*, *variant='d2'*, *short=None*)

Read in CosmoD2 data from file, either as grib2 file or as bz2 compressed grib2 file. Currently, both variants, CosmoD2 and CosmoD2EPS are supported.

Parameters

- **filename** (*str*) – name of file
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **variant** (*str, optional*) – specify variant, either 'd2' (default) or 'd2eps'
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Returns

Cosmo data of four 15 min values as in the content of the grib2 file

Return type

list

CHAPTER

FIVE

MET_ENTITIES PACKAGE

The package met_entities consists of different classes to handle different weather data formats of the german weather service (DWD). It supports RadolanRW, RadvorRQ, IconD2, IconD2EPS datasets.

MetEntities.py module contains MetEntities class as an abstract class that defines methods for handling meteorological data.

RadolanRW.py module contains RadolanRW class that provides all relevant data of RadolanRW data from DWD.

RadvorRQ.py: dito for RadvorRQ data

RadolanRV.py: dito for RadolanRV data

IconD2.py: dito for Icon-D2 data

IconD2EPS: dito for Icon-D2-EPS data

IconEU.py: dito for Icon-EU data

IconEUEPS.py: dito for Icon-EU-EPS data

CosmoD2.py: dito for Cosmo-D2 data

CosmoD2EPS.py: dito for Cosmo-D2-EPS data

GeoReferencedData.py module contains GeoReferencedData class that serves as a container for spatial data including coordinates, data and their description; further elementary functions are implemented (regridding, cropping).

LonLatTime.py module contains LonLatTime class that is used as a container for coordinate- or time-related data to facilitate the export to netCDF.

VariableDescription.py module contains classes DataDescription (further description of climate data), TimeDescription (description of time aspects of climate data), RegridDescription (attributes necessary for regridding), CropDescription (attribute necessary for cropping), and NcDimVarDescription (describes the dimension and variable names of a netcdf file in detail).

Exceptions.py module contains typical exceptions raised in met_entities.

WeatherData.py module contains WeatherData class that provides collection and harmonizing routines for all types of above-mentioned supported data.

5.1 met_entities.CosmoD2 module

class met_entities.CosmoD2.**CosmoD2**(*time_value:* LonLatTime | *None* = *None*, *forecast_value:* LonLatTime | *None* = *None*, *gr_data:* GeoReferencedData | *None* = *None*, *short: str* | *None* = *None*)

Bases: *MetEntities*

CosmoD2 class provides all relevant data of CosmoD2 data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*)

Cropping of data of this CosmoD2 class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (*VariableDescription.CropDescription, optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north, idx_array variables
- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit

export_netcdf(*filename*, *data_format='f8'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this CosmoD2 class to a new netcdf file.

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this CosmoD2 class to an existing netcdf file.

Parameters

filename (*str*) – filename of netcdf file

read_file(*start_datetime*, *directory='./'*, *dir_time_descriptor=None*, *forecast_hours=27*, *scale_factor=1*, *fill_value=nan*, *short=None*)

Reading in all available files of an CosmoD2 dataset. If a file for a specific forecast time is not available yet, the function automatically stops reading in and delivers a proper datastructure.

Parameters

- **start_datetime** (*datetime.datetime*) – time to start reading in CosmoD2 files, it is used to build the filenames following the convention of the german weather service (DWD)
- **directory** (*str, optional*) – directory with all CosmoD2 files from start_datetime
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **forecast_hours** (*int, optional*) – the number of forecast hours to be read in, default is 27 (normal max for CosmoD2)
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables
- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.2 met_entities.CosmoD2EPS module

class met_entities.CosmoD2EPS.**CosmoD2EPS**(*time_value:* LonLatTime | *None* = *None*, *forecast_value:* LonLatTime | *None* = *None*, *gr_data=None*, *eps_member: list* | *None* = *None*, *short: str* | *None* = *None*)

Bases: *MetEntities*

CosmoD2EPS class provides all relevant data of CosmoD2EPS data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*)

Cropping of data of this CosmoD2 class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (VariableDescription.CropDescription, *optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north, idx_array variables
- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit

export_netcdf(*filename*, *data_format='f8'*, *version='separated'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this CosmoD2EPS class to a new netcdf file. The function expects the same coordinates for all realisations (i.e. the same regridding/cropping for each realisation).

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **version** (*str, optional*) – the netcdf output is enabled in two different versions: 1. 'separated' (default) with an own variable for each realisation, 2. 'integrated' with one large data matrix with eps as the last dimension
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this CosmoD2EPS class to an existing netcdf file.

Parameters

filename (*str*) – filename of netcdf file

read_file(*start_datetime*, *directory='./'*, *dir_time_descriptor=None*, *forecast_hours=27*, *eps_member=None*, *scale_factor=1*, *fill_value=nan*, *short=None*)

Reading in all available files of a CosmoD2EPS dataset. If a file for a specific forecast time is not available yet, the function automatically stops reading in and delivers a proper datastructure.

Parameters

- **start_datetime** (*datetime.datetime*) – time to start reading in CosmoD2EPS files, it is used to build the filenames following the convention of the german weather service (DWD)
- **directory** (*str, optional*) – directory with all CosmoD2EPS files from start_datetime
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **forecast_hours** (*int, optional*) – the number of forecast hours to be read in, default is 27 (max for CosmoD2EPS)
- **eps_member** (*list, optional*) – list with numbers of eps members to read (0 to 19); if not given, take all 20 realisations
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables
- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.3 met_entities.Exceptions module

exception met_entities.Exceptions.**ForecastFileNotAvailable**(*message*)

Bases: Exception

Raised if already the first file of forecast dataset is not found.

exception met_entities.Exceptions.**RadarFileNotAvailable**(*message*)

Bases: Exception

Raised if a radar observation file is not found.

5.4 met_entities.GeoReferencedData module

class met_entities.GeoReferencedData.**GeoReferencedData**(*lon: <module 'met_entities.LonLatTime' from '/home/mwagner/Hydrologie/metData/weatherDataHarmo = None*, *lat: <module 'met_entities.LonLatTime' from '/home/mwagner/Hydrologie/metData/weatherDataHarmo = None*, *data=None*, *data_description=None*, *regridded=False*, *cropped=False*)

Bases: object

GeoReferencedData serves as a container for spatial data including coordinates, data and their description.

crop(*lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*, *idx_array=None*)

Cropping of data of this GeoReferencedData class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. The function handles 1D/2D data without forecast as well as 2D/3D data including forecasts.

Parameters

- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit
- **idx_array** (*np.ndarray, optional*) – index for 1D array in lon and lat (e.g. original icond2 data)

find_nearest(*lon_target*, *lat_target*, *neighbors*)

Find an arbitrary number of the nearest points from the source coordinates for the target grid. It works with source coordinates given as 2D matrix and given as 1D array.

Parameters

- **lon_target** (LonLatTime.LonLatTime) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int*) – number of neighbors for IDW

Returns

(1) matrix with indexes of the nearest points for source coordinates given as matrix with shape (num_lat, num_lon, 2, neighbors), whereas the third dimension differentiates between index of row and index of col or for source coordinates given as array with shape (num_lat, num_lon, neighbors); (2) matrix with distances of the nearest points with shape (num_lat, num_lon, neighbors)

Return type

numpy.ndarray, numpy.ndarray

find_nearest_slower(*lon_target*, *lat_target*, *neighbors*)

Find an arbitrary number of the nearest points from the source grid for the target grid. The algorithm is somewhat easier to understand, but significantly slower than find_nearest. The code is not extended to work with source coordinates given as an array (not a matrix).

Parameters

- **lon_target** (LonLatTime.LonLatTime) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int*) – number of neighbors for IDW

Returns

(1) matrix with indexes of nearest points with shape (num_lat, num_lon, 2, neighbors), whereas the third dimension differentiates between index of row and index of col; (2) matrix with distances of the nearest points with shape (num_lat, num_lon, neighbors)

Return type

numpy.ndarray, numpy.ndarray

regrid_idw(*lon_target*, *lat_target*, *neighbors=3*, *file_nearest=None*, *short=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. The original class instance is changed. The function handles 1D/2D data without forecast as well as 2D/3D data including forecasts.

Parameters

- **lon_target** (LonLatTime.LonLatTime) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup; if the file does not exist, it is built, otherwise the content of the file is checked if it is suitable for the actual data
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

5.5 met_entities.IconD2 module

class met_entities.IconD2.**IconD2**(*time_value:* LonLatTime | *None* = *None*, *forecast_value:* LonLatTime | *None* = *None*, *gr_data:* GeoReferencedData | *None* = *None*, *short: str* | *None* = *None*)

Bases: *MetEntities*

IconD2 class provides all relevant data of IconD2 data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*, *idx_array=None*)

Cropping of data of this IconD2 class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (VariableDescription.CropDescription, *optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north, idx_array variables
- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit
- **idx_array** (*np.ndarray, optional*) – index for 1D array in lon and lat (e.g. original icond2 data)

export_netcdf(*filename*, *data_format='f8'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this IconD2 class to a new netcdf file.

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this IconD2 class to an existing netcdf file.

Parameters

filename (*str*) – filename of netcdf file

read_file(*start_datetime*, *directory='./'*, *dir_time_descriptor=None*, *forecast_hours=48*, *scale_factor=1*, *fill_value=nan*, *short=None*)

Reading in all available files of an IconD2 dataset. If a file for a specific forecast time is not available yet, the function automatically stops reading in and delivers a proper datastructure.

Parameters

- **start_datetime** (*datetime.datetime*) – time to start reading in IconD2 files, it is used to build the filenames following the convention of the german weather service (DWD)
- **directory** (*str, optional*) – directory with all IconD2 files from start_datetime

- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **forecast_hours** (*int, optional*) – the number of forecast hours to be read in, default is 48 (max for IconD2)
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables
- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.6 met_entities.IconD2EPS module

class met_entities.IconD2EPS.**IconD2EPS**(*time_value:* LonLatTime | *None* = *None*, *forecast_value:* LonLatTime | *None* = *None*, *gr_data=None*, *eps_member:* *list* | *None* = *None*, *short: str* | *None* = *None*)

Bases: *MetEntities*

IconD2EPS class provides all relevant data of IconD2EPS data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*, *idx_array=None*)

Cropping of data of this IconD2 class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (VariableDescription.CropDescription, *optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north, idx_array variables
- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit

- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit
- **idx_array** (*np.ndarray, optional*) – index for 1D array in lon and lat (e.g. original icond2 data)

export_netcdf(*filename*, *data_format='f8'*, *version='separated'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this IconD2EPS class to a new netcdf file. The function expects the same coordinates for all realisations (i.e. the same regridding/cropping for each realisation).

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **version** (*str, optional*) – the netcdf output is enabled in two different versions: 1. 'separated' (default) with an own variable for each realisation, 2. 'integrated' with one large data matrix with eps as the last dimension
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this IconD2EPS class to an existing netcdf file.

Parameters

filename (*str*) – filename of netcdf file

read_file(*start_datetime*, *directory='./'*, *dir_time_descriptor=None*, *forecast_hours=48*, *eps_member=None*, *scale_factor=1*, *fill_value=nan*, *short=None*)

Reading in all available files of an IconD2EPS dataset. If a file for a specific forecast time is not available yet, the function automatically stops reading in and delivers a proper datastructure.

Parameters

- **start_datetime** (*datetime.datetime*) – time to start reading in IconD2EPS files, it is used to build the filenames following the convention of the german weather service (DWD)
- **directory** (*str, optional*) – directory with all IconD2EPS files from start_datetime

- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **forecast_hours** (*int, optional*) – the number of forecast hours to be read in, default is 48 (max for IconD2EPS)
- **eps_member** (*list, optional*) – list with numbers of eps members to read (0 to 19); if not given, take all 20 realisations
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables
- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.7 met_entities.IconEU module

class met_entities.IconEU.**IconEU**(*time_value:* LonLatTime | *None* = *None*, *forecast_value:* LonLatTime | *None* = *None*, *gr_data:* GeoReferencedData | *None* = *None*, *short: str* | *None* = *None*)

Bases: *MetEntities*

IconEU class provides all relevant data of IconEU data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*, *idx_array=None*)

Cropping of data of this IconD2 class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (VariableDescription.CropDescription, *optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north, idx_array variables
- **lon_west** (*float, optional*) – western longitude limit

- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit
- **idx_array** (*np.ndarray, optional*) – index for 1D array in lon and lat (e.g. original icond2 data)

export_netcdf(*filename*, *data_format='f8'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this IconEU class to a new netcdf file.

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this IconEU class to an existing netcdf file.

Parameters

filename (*str*) – filename of netcdf file

read_file(*start_datetime*, *directory='./'*, *dir_time_descriptor=None*, *forecast_hours=120*, *scale_factor=1*, *fill_value=nan*, *short=None*, *harmonize_dt=True*)

Reading in all available files of an IconEU dataset. If a file for a specific forecast time is not available yet, the function automatically stops reading in and delivers a proper datastructure.

Parameters

- **start_datetime** (*datetime.datetime*) – time to start reading in IconEU files, it is used to build the filenames following the convention of the german weather service (DWD)
- **directory** (*str, optional*) – directory with all IconEU files from start_datetime
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename

- **forecast_hours** (*int, optional*) – the number of forecast hours to be read in, default is 120 (max for IconEU)
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision
- **harmonize_dt** (*bool*) – if True (default), harmonize possibly changing dt to one final dt equal to the time duration between the first two IconEU data matrices; the data at longer dt are evenly distributed to the shorter target dt

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables
- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.8 met_entities.IconEUEPS module

class met_entities.IconEUEPS.**IconEUEPS**(*time_value: LonLatTime | None = None*, *forecast_value: LonLatTime | None = None*, *gr_data=None*, *eps_member: list | None = None*, *short: str | None = None*)

Bases: *MetEntities*

IconEUEPS class provides all relevant data of IconEUEPS data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*, *idx_array=None*)

Cropping of data of this IconD2 class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (VariableDescription.CropDescription, *optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north, idx_array variables
- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit

- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit
- **idx_array** (*np.ndarray, optional*) – index for 1D array in lon and lat (e.g. original icond2 data)

export_netcdf(*filename*, *data_format='f8'*, *version='separated'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this IconEUEPS class to a new netcdf file. The function expects the same coordinates for all realisations (i.e. the same regridding/cropping for each realisation).

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **version** (*str, optional*) – the netcdf output is enabled in two different versions: 1. 'separated' (default) with an own variable for each realisation, 2. 'integrated' with one large data matrix with eps as the last dimension
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this IconEUEPS class to an existing netcdf file.

Parameters

filename (*str*) – filename of netcdf file

read_file(*start_datetime*, *directory='./'*, *dir_time_descriptor=None*, *forecast_hours=120*, *eps_member=None*, *scale_factor=1*, *fill_value=nan*, *short=None*, *harmonize_dt=True*)

Reading in all available files of an IconEUEPS dataset. If a file for a specific forecast time is not available yet, the function automatically stops reading in and delivers a proper datastructure.

Parameters

- **start_datetime** (*datetime.datetime*) – time to start reading in IconEUEPS files, it is used to build the filenames following the convention of the german weather service (DWD)
- **directory** (*str, optional*) – directory with all IconEUEPS files from start_datetime

- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **forecast_hours** (*int, optional*) – the number of forecast hours to be read in, default is 120 (max for IconEUEPS)
- **eps_member** (*list, optional*) – list with numbers of eps members to read (0 to 39); if not given, take all 40 realisations
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision
- **harmonize_dt** (*bool*) – if True (default), harmonize possibly changing dt to one final dt equal to the time duration between the first two IconEU data matrices; the data at longer dt are evenly distributed to the shorter target dt

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables
- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.9 met_entities.LonLatTime module

class met_entities.LonLatTime.**LonLatTime**(*data=None*, *data_description=None*)

Bases: object

LonLatTime class is used as a container for coordinate- or time-related data to facilitate the export to netCDF.

5.10 met_entities.MetEntities module

class met_entities.MetEntities.**MetEntities**(*time_value:* LonLatTime | *None* = *None*, *forecast_value:* LonLatTime | *None* = *None*, *gr_data=None*, *eps_member: list* | *None* = *None*, *short: str* = *False*)

Bases: ABC

MetEntities is an abstract class that defines methods for handling meteorological data.

abstract crop(***kwargs*)

abstract export_netcdf(*filename*, ***kwargs*)

abstract export_netcdf_append(*filename*)

abstract read_file(***kwargs*)

abstract regrid(***kwargs*)

5.11 met_entities.RadolanRV module

class met_entities.RadolanRV.**RadolanRV**(*time_value:* LonLatTime | *None* = *None*, *forecast_value:* LonLatTime | *None* = *None*, *gr_data:* GeoReferencedData | *None* = *None*, *short: str* | *None* = *None*)

Bases: *MetEntities*

RadolanRV class provides all relevant data of RadolanRV data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*)

Cropping of data of this RadvorRQ class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (VariableDescription.CropDescription, *optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north variables
- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit

export_netcdf(*filename*, *data_format='f8'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this RadolanRV class to a new netcdf file.

Parameters

- **filename** (*str*) – filename of netcdf file

- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this RadolanRV class to an existing netcdf file.

Parameters
filename (*str*) – filename of netcdf file

read_file(*filename=None*, *start_datetime=None*, *directory='./'*, *dir_time_descriptor=None*, *scale_factor=1*, *fill_value=nan*, *short=None*)

Reading in a compressed file of RadolanRV data: at 0, +5,..., +120 minutes.

Parameters

- **filename** (*str, optional*) – RV filename typically in the form DE1200_RVyymmddHHMM.tar.bz2
- **start_datetime** (*datetime.datetime, optional*) – time to start reading in RadolanRV files, it is used to build the filenames following the convention of the german weather service (DWD)
- **directory** (*str, optional*) – directory with the RadolanRW file at start_datetime
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables

- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.12 met_entities.RadolanRW module

class met_entities.RadolanRW.**RadolanRW**(*time_value: <module 'met_entities.LonLatTime' from '/home/mwagner/Hydrologie/metData/weatherDataHarmonizer/met_entities/Lo = None*, *gr_data: ~met_entities.GeoReferencedData.GeoReferencedData = None*, *short: str = None*)

Bases: *MetEntities*

RadolanRW class provides all relevant data of RadolanRW data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*)

Cropping of data of this RadvorRQ class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (VariableDescription.CropDescription, *optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north variables
- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit

export_netcdf(*filename*, *data_format='f8'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this RadolanRW class to a new netcdf file.

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'

- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this RadolanRW class to an existing netcdf file.

Parameters

filename (*str*) – filename of netcdf file

import_netcdf(*filename*, *nc_desc=None*)

Import a netcdf file to instantiate a RadolanRW object.

Parameters

- **filename** (*str*) – name of the netcdf file.
- **nc_desc** (*vd.NcDimVarDescription, optional*) – detailed description of dimension and variable names in the netcdf file; if omitted the standard is taken (see constructor __init__)

read_file(*filename=None*, *start_datetime=None*, *directory='./'*, *dir_time_descriptor=None*, *scale_factor=1*, *fill_value=nan*, *short=None*)

Reading in a RadolanRW file.

Parameters

- **filename** (*str, optional*) – RW file, typically in the form raa01-rw_10000-yymmddHHMM-dwd—bin.bz2; if not given start_datetime and directory elements must be provided
- **start_datetime** (*datetime.datetime, optional*) – time to start reading in RadolanRW files, it is used to build the filenames following the convention of the german weather service (DWD); can be given in xx:50 or xx:00 (full hours), in the second case a -10 minutes shift is done internally to meet the RadolanRW delivering convention
- **directory** (*str, optional*) – directory with the RadolanRW file at start_datetime
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables
- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW
- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.13 met_entities.RadvorRQ module

class met_entities.RadvorRQ.**RadvorRQ**(*time_value:* LonLatTime | *None* = *None*, *forecast_value:* LonLatTime | *None* = *None*, *gr_data:* GeoReferencedData | *None* = *None*, *short: str* | *None* = *None*)

Bases: *MetEntities*

RadvorRQ class provides all relevant data of RadvorRQ data from DWD.

crop(*crop_description=None*, *lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*)

Cropping of data of this RadvorRQ class. Usage of indexes directly if given. Otherwise, use lon/lat with the guarantee that the whole requested area is within the cropped region. If crop_description is given it is prioritized.

Parameters

- **crop_description** (VariableDescription.CropDescription, *optional*) – crop description with some of lon_west, lon_east, lat_south, lat_north, idx_west, idx_east, idx_south, idx_north variables
- **lon_west** (*float, optional*) – western longitude limit
- **lon_east** (*float, optional*) – eastern longitude limit
- **lat_south** (*float, optional*) – southern latitude limit
- **lat_north** (*float, optional*) – northern latitude limit
- **idx_west** (*int, optional*) – western index limit
- **idx_east** (*int, optional*) – eastern index limit
- **idx_south** (*int, optional*) – southern index limit
- **idx_north** (*int, optional*) – northern index limit

export_netcdf(*filename*, *data_format='f8'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this RadvorRQ class to a new netcdf file.

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: ‘f8’, ‘f4’, ‘i8’, ‘i4’, ‘i2’, ‘i1’, ‘u8’, ‘u4’, ‘u2’, ‘u1’, ‘S1’ (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)

- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append the relevant content of this RadvorRQ class to an existing netcdf file.

Parameters

filename (*str*) – filename of netcdf file

read_file(*start_datetime*, *directory='./'*, *dir_time_descriptor=None*, *scale_factor=1*, *fill_value=nan*, *short=None*)

Reading in three files of RadvorRQ data: at 0, +60, +120 minutes.

Parameters

- **start_datetime** (*datetime.datetime*) – time to start reading in RadvorRQ files, it is used to build the filenames following the convention of the german weather service (DWD)
- **directory** (*str, optional*) – directory with all RadvorRQ files from start_datetime
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

regrid(*regrid_description=None*, *lon_target=None*, *lat_target=None*, *neighbors=3*, *file_nearest=None*)

Interpolate the gridded data onto a new raster in the same coordinate system. It uses an inverse distance weighted (IDW) method with an arbitrary number of neighbors. If regrid_description is given it is prioritized.

Parameters

- **regrid_description** (VariableDescription.RegridDescription, *optional*) – regrid description with some of lon_target, lat_target, neighbors, file_nearest variables
- **lon_target** (LonLatTime.LonLatTime, *optional*) – longitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **lat_target** (LonLatTime.LonLatTime, *optional*) – latitudes of the target raster, given as raster with shape (num_lat, num_lon)
- **neighbors** (*int, optional*) – number of neighbors for IDW

- **file_nearest** (*str, optional*) – npz file with indexes and lengths for the current setup

5.14 met_entities.VariableDescription module

class met_entities.VariableDescription.**CropDescription**(*lon_west=None*, *lon_east=None*, *lat_south=None*, *lat_north=None*, *idx_west=None*, *idx_east=None*, *idx_south=None*, *idx_north=None*, *idx_array=None*)

Bases: object

CropDescription class contains variables to describe the cropping of weather data.

class met_entities.VariableDescription.**DataDescription**(*fill_value=None*, *units=None*, *scale_factor=None*, *coordinate_system=None*, *long_name=None*, *standard_name=None*, *time_note=None*, *eps_note=None*)

Bases: object

DataDescription class contains metadata to further describe climate data to facilitate the export to netCDF.

class met_entities.VariableDescription.**NcDimVarDescription**(*dim_time=None*, *dim_forecast=None*, *dim_lon=None*, *dim_lat=None*, *dim_coord=None*, *dim_eps=None*, *var_time=None*, *var_forecast=None*, *var_lon=None*, *var_lat=None*, *var_eps=None*, *var_data=None*)

Bases: object

NcVarDimDescription class describes detailed the dimension and variable names of a netcdf file.

class met_entities.VariableDescription.**RegridDescription**(*lon_target*, *lat_target*, *neighbors=3*, *file_nearest=None*)

Bases: object

RegridDescription class contains variables do describe the regridding of weather data.

class met_entities.VariableDescription.**TimeDescription**(*calendar=None*, *units=None*)

Bases: object

TimeDescription class contains variables to further describe time aspects of climate data to facilitate the export to netCDF.

5.15 met_entities.WeatherData module

class met_entities.WeatherData.**WeatherData**(*time_now: datetime*, *delta_t: timedelta*, *data=None*, *scale_factor=1*, *fill_value=-999*, *short=None*, *regrid_description: dict | None = None*, *crop_description:* CropDescription *| None = None*)

Bases: object

WeatherData class is a container for mixed weather data from radolan, radvor, icond2, icond2eps, iconeu, iconeueps data sources. It comprises methods to sort the date in a chronological order from past to future.

amend_time_data(*time*, *data*)

Amend the time vector in self.time from existing datetimes until latest_time. self.data is extended and filled accordingly. If (single or all) datetime already available, the existing forecast data is overwritten. This can be used to load a longer forecasting product and overwrite some time steps with a (potentially better) short-time forecast product. Time steps which are in self.time_protected (typically observed data) are never changed.

Parameters

- **time** (*list*) – the latest datetime to be included in self.time
- **data** (*list*) – list of data as GeoreferencedData instances

collect_icond2(*latest_event*, *directory=None*, *dir_time_descriptor=None*, *forecast_hours=48*)

Collect all IconD2 data into the variable data and construct the according time vector. collect_radolanrw must be invoked before collecting prediction data.

Parameters

- **latest_event** (*datetime.datetime*) – datetime of the latest forecast start to be tried to load
- **directory** (*str, optional*) – directory with IconD2 data
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **forecast_hours** (*int, optional*) – try to load this number of hours at maximum

collect_icond2eps(*latest_event*, *directory=None*, *dir_time_descriptor=None*, *forecast_hours=48*, *eps_member=range(0, 20)*)

Collect all IconD2EPS data for one ensemble member into the variable data and construct the according time vector. collect_radolanrw must be invoked before collecting prediction data.

Parameters

- **latest_event** (*datetime.datetime*) – datetime of the latest forecast start to be tried to load
- **directory** (*str, optional*) – directory with IconD2EPS data
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **forecast_hours** (*int, optional*) – try to load this number of hours at maximum
- **eps_member** (*list, optional*) – list of eps members to load (0 to 19)

collect_iconeu(*latest_event*, *directory=None*, *dir_time_descriptor=None*, *forecast_hours=120*)

Collect all IconEU data into the variable data and construct the according time vector. collect_radolanrw must be invoked before collecting prediction data.

Parameters

- **latest_event** (*datetime.datetime*) – datetime of the latest forecast start to be tried to load
- **directory** (*str, optional*) – directory with IconEU data
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename

- **forecast_hours** (*int, optional*) – try to load this number of hours at maximum

collect_iconeueps(*latest_event*, *directory=None*, *dir_time_descriptor=None*, *forecast_hours=120*, *eps_member=range(0, 40)*)

Collect all IconEUEPS data for one ensemble member into the variable data and construct the according time vector. collect_radolanrw must be invoked before collecting prediction data.

Parameters

- **latest_event** (*datetime.datetime*) – datetime of the latest forecast start to be tried to load
- **directory** (*str, optional*) – directory with IconEUEPS data
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **forecast_hours** (*int, optional*) – try to load this number of hours at maximum
- **eps_member** (*list, optional*) – list of eps members to load (0 to 39)

collect_radolanrv(*latest_event*, *directory=None*, *dir_time_descriptor=None*)

Collect all RadolanRV data into the variable data and construct the according time vector. collect_radolanrw must be invoked before collecting prediction data.

Parameters

- **latest_event** (*datetime.datetime*) – datetime of the latest forecast start to be tried to load
- **directory** (*str, optional*) – directory with RadolanRV data
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename

collect_radolanrw(*time_start*, *directory=None*, *dir_time_descriptor=None*)

Collect all RadolanRW data into the variable data and construct the according time vector.

Parameters

- **time_start** (*datetime.datetime*) – time to start with RadolanRW data; it must avoid xx:50, instead give full hours; the time shift is done function-internal
- **directory** (*str, optional*) – directory with RadolanRW data
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename

collect_radvorrq(*latest_event*, *directory=None*, *dir_time_descriptor=None*)

Collect all RadvorRQ data into the variable data and construct the according time vector. collect_radolanrw must be invoked before collecting prediction data.

Parameters

- **latest_event** (*datetime.datetime*) – datetime of the latest forecast start to be tried to load
- **directory** (*str, optional*) – directory with RadvorRQ data
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data,

e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename

export_netcdf(*filename*, *data_format='f8'*, *institution=None*, *scale_factor_nc=1*, *scale_undo=False*, *data_kwargs=None*)

Export the relevant content of this WeatherData class to a new netcdf file.

Parameters

- **filename** (*str*) – filename of netcdf file
- **data_format** (*str, optional*) – format description of resulting data field in netcdf, the following specifiers are allowed: 'f8', 'f4', 'i8', 'i4', 'i2', 'i1', 'u8', 'u4', 'u2', 'u1', 'S1' (f - float, i - integer, u - unsigned integer, S1 - single character string; the number specifies the number of bytes)
- **institution** (*str*) – description of the institution generating the netcdf file; will be used for global netcdf attribute 'institution'
- **scale_factor_nc** (*float, optional*) – is used as scale_factor for netcdf file and can be used for storage saving purposes in conjunction with data_format (e.g. 'i2'); do not mix it up with scale_factor_internal
- **scale_undo** (*bool, optional*) – if True, the original internal scale_factor is taken back in order to get the original values.
- **data_kwargs** (*dict, optional*) – keyword arguments that are passed to netCDF4.createVariable for data variables, for supported arguments refer to https://unidata.github.io/netcdf4-python/#Dataset.createVariable; e.g. {'compression': 'zlib'} compresses the data with zlib alorithm and default complevel=4

export_netcdf_append(*filename*)

Append or replace the relevant content of this WeatherData class to an existing netcdf file. It is possible to broadcast one (here) to many (netcdf), but it is not allowed to store many (here) to one (netcdf). In the latter case the standard variable name in the netcdf would have to be removed. That is not supported in netcdf4 standard.

Newer times (here) can be amended to netcdf, extending older time is not supported.

New precipitation data (here) will overwrite existing data in netcdf with the same datetime. If multiple ensemble members exist in netcdf, newer data without ensemble members will overwrite all ensemble members at the appropriate times.

Several checks are conducted to ensure the compatibility of the netcdf with the data to be amended before the netcdf file is changed.

Parameters

filename (*str*) – filename of an existing netcdf file with the same general properties

met_entities.WeatherData.**collect_fc**(*product*, *directory*, *time_now*, *delta_t*, *latest_event*, *update_time*, *dir_time_descriptor=None*, *fc_max_time=datetime.timedelta(0)*, *regrid_description=None*, *crop_description=None*, *scale_factor=1*, *fill_value=nan*, *eps_member=range(0, 20)*, *short=None*)

Collection of available forecast data including loading, aggregation/disaggregation as needed, and a consistent time vector.

Parameters

- **product** (*str*) – product name; currently supported: radvorrq, radolanrv, icond2, icond2eps, iconeu, iconeueps
- **directory** (*str*) – directory with the forecast data
- **time_now** (*datetime.datetime*) – current time step (ergo start of forecast time)
- **delta_t** – required temporal resolution

- **latest_event** (*datetime.datetime*) – time of the latest forecast start event that shall potentially be included; must be a time that is potentially available (e.g. 15:00 for icond2, or 10:15 for radvorrq)
- **fc_max_time** (*datetime.timedelta*) – potential maximum forecast time for the specific forecast product
- **update_time** (*datetime.timedelta*) – time between updates of the forecast product
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename
- **regrid_description** (*dict*) – dictionary with product key and a description as VariableDescription.RegridDescription instance
- **crop_description** (VariableDescription.CropDescription) – crop description
- **scale_factor** (*float, optional*) – the final data has to be multiplied with this value
- **fill_value** (*float, optional*) – missing data is filled with that value
- **eps_member** (*list, optional*) – list of eps members to load (e.g. in IconD2EPS max possible: 0 to 19 - default, in IconEUEPS: 0 to 39)
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Type
delta_t: datetime.timedelta

Returns
(i) list of GeoReferencedData with the most recent forecast value for each time step, (ii) list of time steps that belong to the data

Return type
list, list

met_entities.WeatherData.**create_transient_data_matrix**(*data_list*, *fc_delta_t*, *delta_t*, *time_now*, *short=None*)

Creation of a transient data matrix with target temporal resolution. Aggregation/disaggregation is done as needed. Currently only works for two-dimensional forecast data and a third dimension with the forecast time.

Parameters

- **data_list** (*list*) – list of forecast data
- **fc_delta_t** (*datetime.timedelta*) – temporal resolution of forecast data
- **delta_t** (*datetime.timedelta*) – temporal resolution of target data
- **time_now** (*datetime.datetime*) – last time step of observations, afterwards forecast starts; only used for aggregation
- **short** (*str, optional*) – if int16 or int32, the large data variables are cast to numpy.int16/int32 to minimize memory usage; the user must pay attention to the scale_factor to include the necessary precision

Returns
transient list of forecast data and time vectors

Return type
list, list

met_entities.WeatherData.**data_list_time_interpretation**(*data_list*, *time_now*, *fc_max_time*, *update_time*)

Interpretation of time aspects of data in a list regarding full coverage until time_now and the potential maximum forecast time. If the beginning of the first forecast data is equal to or earlier than last observation, further data loading is advised. Further data loading is also advised if the newest forecast time plus its maximum forecast time is older than a potential forecast with its maximum forecast time one update step earlier than already loaded. The second case appears if, e.g. the actual (newest) forecast is not completely provided by DWD, yet.

Loading observed data is always advised.

Parameters

- **data_list** (*list*) – list of forecast data
- **time_now** (*datetime.datetime*) – actual time when forecast starts
- **fc_max_time** (*datetime.timedelta*) – potential maximum forecast time for the specific forecast product
- **update_time** (*datetime.timedelta*) – time between updates of the forecast product

Returns
logical value whether further loading data is advised (True) or no not necessary (False)

Return type
bool

met_entities.WeatherData.**get_delta_t_val**(*data*, *fc_delta_t*, *divider*)

Get delta_t_val, the delta_t of the values within data object.

Parameters

- **data** (MetEntities.MetEntities) – entity with meteorological data
- **fc_delta_t** (*datetime.timedelta*) – delta_t of the data
- **divider** (*float*) – division of the original data, > 1 means a disaggregation, < 1 an aggregation

Returns
the new delta_t for resulting values

Return type
float

met_entities.WeatherData.**get_potential_netcdf_filename**(*product_class*, *time_step*, *directory*, *dir_time_descriptor*)

Build path string with a potentially existing netcdf containing the requested data.

Parameters

- **product_class** (aux_tools.Product.Product) – product object
- **time_step** (*datetime.datetime*) – time object of requested data
- **directory** (*str, optional*) – directory with all IconD2 files from start_datetime
- **dir_time_descriptor** (*list, optional*) – list of datetime.strftime time descriptors for an arbitrary number of additional time dependent folders with the data, e.g. ['%Y', '%Y%m%d'] for */yyyy/yyyymmdd/*; the final path is built to directory/dir_time_directory[0]/../dir_time_directory[n]/filename

Returns
path string

Return type
str

met_entities.WeatherData.**get_time_value_data**(*time_value_data_datetime*, *time_now*, *delta_t*)

Find indexes in timestamps which are met from time_now with a multiple delta_t distance, e.g. in a list 13:30, 14:00, 14:30, 15:00, time_now = 13:00, delta_t = 1 h, find indexes 1 and 3. It can be used for aggregation.

Parameters

- **time_value_data_datetime** (*list*) – timestamps which shall be scanned through
- **time_now** (*datetime.datetime*) – time from which starts the multiple delta_t distances
- **delta_t** (*datetime.timedelta*) – target delta t

Returns
Indexes compliant to target resolution

Return type
list

met_entities.WeatherData.**get_timestamps**(*data*)

Get datetime timestamps in time_data as a combination of time unit description and float value as distance.

Parameters
data (MetEntities.MetEntities) – object with data; if attribute forecast_value exists this is taken, otherwise take attribute time_value.

Returns
timestamps of the time object

Return type
list

5.16 met_entities.read_netcdf module

met_entities.read_netcdf.**read_netcdf**(*filename*, *scale_factor_result=None*, *fill_value_result=None*, *short_result=None*)

Read in data from netcdf file formerly created with export_netcdf in instances of MetEntities subclasses. The type is derived automatically from variable naming in the netcdf file. It results in a MetEntities subclass (e.g. RadolanRW, IconD2EPS or others) with all variables and descriptions as in the class content if loaded from raw data. The scale_factor in netcdf is multiplied with scale_factor_internal to ensure precision preservation. Explained: The scale_factor is typically only used for packing the data in netcdf to save hard disk storage. Additionally, the datatype from the netcdf file is also taken as internal python datatype. Without multiplication of scale_factor the data could lose precision in case of integer datatypes (which are used to save memory).

Additionally, a scale_factor_result can be given to request a specific scale_factor that shall be valid in the end.

Parameters

- **filename** (*str*) – filename of the netcdf file that has the format used in MetEntities.export_netcdf.
- **scale_factor_result** (*float, optional*) – the final data has to be multiplied with this value; if None the applied scale_factor is the multiplication of scale_factor in netcdf and scale_factor_internal (see description above)
- **fill_value_result** (*float, optional*) – fill value of the final result; if given, the netcdf internally used fill value is replaced with this one
- **short_result** (*str, optional*) – if int16, int32 or None, the resulting netcdf variables are cast to numpy.int16/int32/float no matter the data type in netcdf; the user must pay attention to the scale_factor_result to ensure the necessary precision

Returns
instance of a MetEntities subclass

Return type
met_entities.MetEntities.MetEntities

CHAPTER

SIX

AUX_TOOLS PACKAGE

The package aux_tools is a container for various helper tools used in weatherDataHarmonizer.

Product.py module contains the Product class that provides all relevant technical data for the download of files.

products.py module comprises inherited Product class instances for the supported DWD data.

grib2_tools.py module contains various helper tools for grib2 handling.

scaling_tools.py module provides functions for the handling of missing values in netcdf and a potential netcdf data packing.

6.1 aux_tools.Product module

class aux_tools.Product.**Product**(*update_time: timedelta | None = None*, *base_time: timedelta = datetime.timedelta(0)*, *url_folder: str | None = None*, *url_time_subfolder: str | None = None*, *url_subfolder: str | None = None*, *file_prefix: str | None = None*, *file_mid: str | None = None*, *file_mid_aux: list | None = None*, *file_suffix: str | None = None*, *file_extension: str | None = None*)

Bases: `object`

Product class contains all relevant technical data for download of RadolanRW, RadvorRQ, RadolanRV, IconD2, IconD2EPS files (from the German Weather Service - DWD).

6.2 aux_tools.grib2_tools module

aux_tools.grib2_tools.**accum_to_instantaneous_flattened**(*data_list*, *fill_value*)

Convert accumulated forecast data series, e.g. from IconD2, to a flattened list with instantaneous values. All 15 minutes forecasts are treated equally. Occasionally occurring negative precipitation values are corrected to zero.

Parameters

- **data_list** (*list*) – list of accumulated data (e.g. IconData or CosmoData class)
- **fill_value** (*float*) – missing data is filled with that value

Returns

1D list of forecast data content from grib files (all 15 minutes datasets in a row, e.g. in IconD2)

Type

list

aux_tools.grib2_tools.**average_to_instantaneous_flattened**(*data_list*, *fill_value*, *start_time_step*)

Convert averaged forecast data series, e.g. from IconD2, to a flattened list with instantaneous values. All 15 minutes forecasts are treated equally. Occasionally occurring negative radiation values are corrected to zero. The de-averaging is built like proposed in the Icon description (Reinert et al.: DWD Database Reference for Global and Regional ICON and ICON-EPS Forecasting System. p. 47, Version 2.2.0, 2022).

Parameters

- **data_list** (*list*) – list of accumulated data (e.g. IconData class)
- **fill_value** (*float*) – missing data is filled with that value
- **start_time_step** (*float*) – defines the time step at the start of the database (zero based); especially necessary, if an intermediate dataset with 15 min values is given to the function

Returns

1D list of forecast data content from grib files (all 15 minutes datasets in a row, e.g. in IconD2)

Type

list

aux_tools.grib2_tools.**get_keys**(*filename*, *message_number=1*)

Get keys and corresponding values of the first message in the grib2 file.

Parameters

- **filename** (*str*) – name of grib2 or bz2 compressed grib file
- **message_number** (*int, optional*) – number of grib message

Returns

keys and values

Return type

dict

aux_tools.grib2_tools.**harmonize_time_step**(*data_list*, *fill_value*, *forecast_time*)

Harmonize time step (dt) in data_list. The target dt is taken from the first two time steps in forecast_time. A changing dt is allowed at an arbitrary number of elements in data_list, as far they are at the end of that list and have an integer divider compared to the first dt. The data disaggregation uses a simple block design and fills all values evenly with the sum_disaggregated = sum_original/divider. If dt is consistent, the original data is returned.

Parameters

- **data_list** (*list*) – list of various data in met entities classes (e.g. IconEU)
- **fill_value** (*float*) – missing data is filled with that value
- **forecast_time** (*list*) – list of values for forecast times

Returns

new data_list and forecast_time objects

Type

list, list

aux_tools.grib2_tools.**write_temporary_grib_file**(*filename*)

Write a temporary grib2 file from a bz2 compressed file. It is saved in OS dependent directory for temporary files and can be removed after use with the filename.

Parameters

filename (*str*) – bz2 compressed grib file

Returns

filename of temporary file

Return type
str

6.3 aux_tools.products module

class aux_tools.products.**IconD2EPSProp**
Bases: *Product*

Instantiate Product class with data for IconD2EPS.

class aux_tools.products.**IconD2Prop**
Bases: *Product*

Instantiate Product class with data for IconD2.

class aux_tools.products.**IconEUEPSProp**
Bases: *Product*

Instantiate Product class with data for IconEUEPS.

class aux_tools.products.**IconEUProp**
Bases: *Product*

Instantiate Product class with data for IconEU.

class aux_tools.products.**RadolanRVProp**
Bases: *Product*

Instantiate Product class with data for RadolanRV.

class aux_tools.products.**RadolanRWProp**
Bases: *Product*

Instantiate Product class with data for RadolanRW.

class aux_tools.products.**RadvorRQProp**
Bases: *Product*

Instantiate Product class with data for RadvorRQ.

6.4 aux_tools.scaling_tools module

aux_tools.scaling_tools.**get_fill_value_unpack**(*gr_data*, *scale_factor_nc*)
Include the scale factor for missing values only in the geodata. This is used for packed netcdf export.

Parameters

- **gr_data** (met_entities.GeoReferencedData.GeoReferencedData) – one set of geodata
- **scale_factor_nc** (*float*) – scale_factor for netcdf writing; leads to the variable attribute scale_factor

Returns
nd-matrix with data from gr_data and missing values multiplied with scale_factor_nc

Return type
numpy.ndarray

aux_tools.scaling_tools.**gr_data_scaling**(*gr_data*, *scale_undo*, *scale_factor*, *scale_factor_nc*)

Prepare geodata for export to netcdf. The internal scaling can be rolled back. If the netcdf variable shall be packed the missing values have to be multiplied with the according scale factor first. This is necessary as the packing ist done before writing to netcdf and the final values in netcdf raw data are compared to the attribute _FillValue to mark missing values.

Parameters

- **gr_data** (met_entities.GeoReferencedData.GeoReferencedData) – one set of geodata
- **scale_undo** (*bool*) – if True the internal scale factor is taken back
- **scale_factor** (*float*) – internal scale factor; e.g. used at reading in large data to save memory; leads to the variable attribute scale_factor_internal
- **scale_factor_nc** (*float*) – scale_factor for netcdf writing; leads to the variable attribute scale_factor

Returns

nd-matrix with back scaled data from gr_data and missing values multiplied with scale_factor_nc

Return type

numpy.ndarray

CHAPTER

SEVEN

DOWNLOAD_DATA PACKAGE

The package download_data implements a download helper for RadolanRW, RadvorRQ, RadolanRV, IconD2, IconD2EPS, IconEU, IconEUEPS data from the German Weather Service (DWD) via https://opendata.dwd.de/.

DownloadJob.py module provides the DownloadJob class. It allows the download of files and their deleting.

7.1 download_data.DownloadJob module

class download_data.DownloadJob.**DownloadJob**(*product: str*, *directory: str*, *date_start: datetime | None = None*, *date_end: datetime = datetime.datetime(2023, 4, 26, 17, 50, 55, 772621, tzinfo=datetime.timezone.utc)*)

Bases: `object`

DownloadJob class contains all data and methods to download RadolanRW, RadvorRQ, RadolanRV, IconD2, IconD2EPS, IconEU, IconEUEPS data from https://opendata.dwd.de/.

delete_files()

Delete all downloaded files from the current DownloadJob instance.

download_files()

Downloads the files. If the start or the end date do not match dates of delivery at DWD the dates are floored to the next provision date.

PYTHON MODULE INDEX

INDEX

F

G

H

I

L

M

N

P

R

T

W