TOAR database python utilities

(status 22 April 2016)

Summary

The TOAR database has assembled the world’s largest collection of surface ozone observations ever achieved. It contains more than 9,500 records of multi-year datasets from 84 countries around the world. The database stores relatively comprehensive metadata to describe stations and the data series measured at these stations. In order to facilitate use of the database and allow for sophisticated analysis of this vast data collection, various software tools were developed in python. Due to data access restrictions agreed with the data owners, it is only possible to use these tools from within the Forschungszentrum Jülich, and only a very limited number of people can get access to these tools. As a consequence, the final data processing of surface ozone observations for the TOAR report will have to be done by Jülich staff. We will run the scripts you provide to us and assist you in the development and testing of your code.

The python scripts use the same code base that is also used in the JOIN web interface (https://join.fz-juelich.de), so it is generally save to explore certain things in JOIN (for example site selection) before translating this into python programs. The JOIN web interface is available to any one after registration (simply click on the registration link on the top right to create your own account).

The two main tasks which will need to be performed over and over again when analyzing TOAR surface ozone data are database queries for metadata on stations and parameter_series and database queries to extract actual data values. In addition, statistics and metrics need to be calculated and results must be plotted or summarized in tables. With the toar_utils.py code, some high-level interfaces are provided to access and process TOAR data and metadata. It is the purpose of this document to describe these high-level routines and explain some details about their inner workings so that you will be able to develop own code for the analysis you wish to do in TOAR (and which we can then later run for you). Additional documentation on the statistics and metrics that have been implemented is given in the TOAR surface ozone statistics and metrics.pdf document, and the actual database structure is described in Obs_surface_stations_metadata.pdf.

Package structure

All python tools that are necessary to perform the analysis of data from the TOAR database are contained in the workshop package. Individual users may find personalized versions (e.g., workshop_owen) in their home directories. Personalizing your version will help us later to re-use your codes for the final processing.

The directory structure is as follows:
The sub directories contain either the code that actually performs most of the database access and number crunching tasks (cdm, qa_plots, and statistics), or documentation (docs; including this document), or the output which you generate (output).

There are two demonstration programs (demo_get_data.py and demo_metadata.py). These show how the database can be accessed and data can be loaded and processed. The plot_utils.py is very rudimentary at present and shall contain a collection of useful plotting routines that can be reused throughout the TOAR report. Test_data_capture.py is a sample application which runs through a set of 3 statistics for ~400 sites and evaluates how the choice of the data_capture threshold in the TOAR statistics influences the availability of years that can be analyzed. The toar_dataset_groups.csv table contains the information which dataseries shall be replaced or combined with others in case there are duplicate data submissions from one station or consecutive time series, etc. Note that this table is still preliminary and shall be reviewed again before the final TOAR data processing. More on the merging concept can be found in section “A note on data series merging” below. Toar_merger.py defines a class that performs the data series merging, and toar_utils.py is a package with various high-level routines to facilitate your analysis tasks.

Besides the two demonstration programs and test_data_capture.py which you can use as blueprints for your own programs, you will most likely only need to work with standard python libraries such as pandas, numpy, scipy, datetime, matplotlib, etc. However, as shown in demo_metadata.py, any operations that involve station or parameter_series metadata will most likely make direct use of the search method of the database interface object (ObsStationsDatabase, defined in cdm/obs_stations_database.py). This document therefore contains a section that describes this method and also the even more generic query method. If you wish to understand more details about how this all works, or, in particular, how the TOAR statistics and metrics are evaluated, please feel free to browse through all subdirectories and look at the code. Usually, all functions, classes, and methods are reasonably documented. Where they are not, feedback is welcome.

Finally, there are a few constants defined in a few places that may be worth knowing in order to facilitate your work, make things more human-readable, or harmonize the analyses across the different TOAR chapters. These are described in section “Useful predefined constants” at the end of this document.
How to access stations and parameter_series metadata

There are several use cases which will make it necessary to access and process the metadata on stations or the metadata on parameter_series in the TOAR database. The simplest case is that you wish to return information about a given station or ozone series. However, you may also wish to find out how many or which stations are located in a given region (identified either as geographic boundary, by country, by climatic zone, or as identified by other proxy data), or which stations have measured ozone during a specific period, etc. Practically every information about stations or ozone time series that is available in the database can be queried very easily with one single method: the search() method in cdm/obs_stations_database.py.

For example, to find all ozone monitoring stations in the United States of America and return their station_id and the network they belong to, you simply write:

```
recs = db.search(columns="network_name, station_id",
                 parameter_name="o3",
                 station_country="United States of America")
```

Other examples are provided in demo_metadata.py.

Here, db is an instance of the ObsStationsDatabase object, and search(...) is the call of the search method (nothing different from a normal python function, except that this function "belongs" to the object). The result will be a list of so-called namedtuples (see https://docs.python.org/2/library/collections.html), and these tuples contain the requested information. To access the network_name and station_id from the 5th US station, simply write

```
print recs[4].network_name, recs[4].station_id
```

After a bit of practice you will recognize that the field names of the namedtuples correspond to the column names in the database tables stations and parameter_series, respectively (see docs/Obs_surface_stations_metadata.pdf for details). With few exceptions explained below, the keywords which you can use in the search function are also identical to these column names. This makes searches and working with search results generally easy and straightforward.

Preparing a search query = creating a db instance

In order to perform a metadata query, you first need to create an instance of the ObsStationsDatabase object. In essence this will open a database connection and return a pointer to this connection which you can then use to access information (and data) from the database. Of course, we first need to import the module where the ObsStationsDatabase class is defined:

```
from cdm.obs_stations_database import ObsStationsDatabase
```

You will also need a valid user name and password. This can be obtained by importing the relevant fields from the cdm_settings.py module:
from cdm.cdm_settings import SURFACES_STATION_DB_USER, SURFACE_STATION_DB_PASSWORD

Both of these modules are contained in the cdm folder. This is why you prepend a cdm. In front of the module name.

The preferred way then to open a database connection is per python’s with statement as follows:

```python
with ObsStationsDatabase(SURFACE_STATION_DB_USER, SURFACE_STATION_DB_PASSWORD) as db:
    # your code here
```

This opens a code block (similarly to an if statement, a for loop or a while loop). Your database connection will live inside this code block under the name of `db`. The big advantage of this method is that the with statement automatically ensures that the database connection will be closed again when you leave the code block – even if this happens as a result of an error (i.e. when an exception is raised). Thus, unless you want to perform interactive queries on a python shell (in this case you would open the database as `db = ObsStationsDatabase(...)`, you should always encapsulate your database work inside a with block.

**Search keywords and their use**

As mentioned above, most of the search options (i.e. the keywords to the search method) are identical to the column names of the stations and parameter_series tables in the TOAR database. Here is a complete list of all allowed keywords to the search method of the ObsStationsDatabase object and their meaning. The first three keywords control the output (i.e. which fields shall be returned and how many records you wish to retrieve):

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>columns</code></td>
<td>Specify the columns of the stations table which shall be included in the result. Column names must be valid names of the database table columns. There are two exceptions: you can also include &quot;parameter_label&quot; which automatically combines the parameter_name, parameter_attribute, and parameter_contributor_shortname into one string, and you can ask for &quot;join_parameter_url&quot; which will generate a suitable link to load the respective parameter_data in the Juelich Open Web Interface JOIN. Columns may be given either as one comma-separated string or as a list of strings. Default is to return all columns (i.e. columns = &quot;*&quot;).</td>
</tr>
<tr>
<td><code>limit</code></td>
<td>Set a limit for the maximum number of search results. Default: None.</td>
</tr>
<tr>
<td><code>join</code></td>
<td>If True always force a table join (i.e. merge the information from the stations and the parameter_series tables) in order to return also the parameter_series metadata associated with a station, even if no specific parameter information is requested in columns nor included in the search_terms. Default is False, i.e. join only when necessary.</td>
</tr>
</tbody>
</table>
All other keywords can be used to narrow your search to identify either a specific set of stations or parameter_series:

<table>
<thead>
<tr>
<th><strong>Station metadata</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>network_name</td>
<td>the network_name; any network if None. Argument can be a single network_name or a list of names.</td>
</tr>
<tr>
<td>station_id</td>
<td>a station identifier or the beginning of a station identifier. Argument can be a single station_id or a list.</td>
</tr>
<tr>
<td>station_name</td>
<td>a station name or a regular expression matching parts of station names. Argument can be a single string or a list.</td>
</tr>
<tr>
<td>station_numid</td>
<td>the (internal) numeric id of the station or a list of values.</td>
</tr>
<tr>
<td>longitude</td>
<td>either a single value or a pair of values. A single value will be expanded to [value-htol, value+htol], the htol keyword defaults to 0.1 degrees. If None, no selection by longitude will take place. Note that the database column name is station_lon instead of longitude.</td>
</tr>
<tr>
<td>latitude</td>
<td>same as for longitude. Note that the database column name is station_lat.</td>
</tr>
<tr>
<td>boundingbox</td>
<td>alternative of specifying longitude and latitude pairs. The boundingbox is defined as &quot;lower left, upper right&quot; with lon0, lat0, lon1, lat1. Note that longitude values in the bounding box are limited to -180..+180.</td>
</tr>
<tr>
<td>altitude</td>
<td>same as for longitude, except that the tolerance is determined by the vtol keyword which defaults to 100 metres. Note that the database column name is station_alt.</td>
</tr>
<tr>
<td>htol</td>
<td>set the horizontal tolerance limit for queries with single longitude, latitude values. The same tolerance will be used for both. Htol is expressed in degrees. Default is 0.1 degrees.</td>
</tr>
<tr>
<td>vtol</td>
<td>set the vertical tolerance limit for queries with a single altitude value. Vtol is expressed in metres and defaults to 100 metres.</td>
</tr>
<tr>
<td>station_local_id</td>
<td>an alternative station_id for local use within a network. Rarely used.</td>
</tr>
<tr>
<td>station_type</td>
<td>a characterization of the station type according to the EEA Airbase classification. Airbase distinguishes between “background”, “industrial”, “traffic”, and “other”. The TOAR database also has “unknown” as a potential value if no metadata information on station type was provided. Note that, although there are rules for designating a station type within Airbase, this information is somewhat ad-hoc and has to be used with caution.</td>
</tr>
<tr>
<td>station_type_of_area</td>
<td>a characterisation of the station’s surrounding according to the EEA Airbase classification scheme. Airbase distinguishes between “rural”, “suburban”, and “urban”. We added “remote” and “unknown” as additional values. The same caveat as for station_type also applies here.</td>
</tr>
<tr>
<td>station_category</td>
<td>originally intended to save the GAW labels of “global”, “regional”, and “contributing” stations, this field is now primarily used to describe the station surroundings in terms of land-use. However, these terms have not been systematically assessed and there is no control over the vocabulary in use. For example, the following terms describe</td>
</tr>
</tbody>
</table>
stations that are in some proximity to agricultural activities: AgricFlat, AgricMountaintop, AgricRolling, Agricultural, Pasture, RangeComplex, RangeFlat, RangeRolling, RuralAgricultural, Urban/AgricRolling, Vinyards orchards and paddocks, agricultural, agriculture, agriculture-forest, local located in an agricultural area, rural background.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>station_country</td>
<td>the country to which the station belongs, and in most cases where the station resides. Country names are standardized throughout the database, but they do not strictly follow the ISO naming. Be aware that, for example, stations in Antarctica are associated with different countries.</td>
</tr>
<tr>
<td>station_state</td>
<td>the state (or province) in which a station resides. This information was obtained from the python module geocoder based on the longitude and latitude information of the station.</td>
</tr>
<tr>
<td>station_timezone</td>
<td>the time zone in which the station resides. This information was obtained from the python module geocoder based on the longitude and latitude of the station location. Time zones are specified as strings (example 'America/Chicago').</td>
</tr>
<tr>
<td>station_population_density_5km</td>
<td>population density extracted from the gridded CIESIN product (<a href="http://sedac.ciesin.columbia.edu/gpw">http://sedac.ciesin.columbia.edu/gpw</a>) at 5 km resolution. Values are of type integer and range from 0 to 48082 people per km^2. This keyword accepts a single value or a range (list) of two values specifying the minimum and maximum value for which metadata shall be extracted. If no value has been assigned for a station, the value of -1 is returned.</td>
</tr>
<tr>
<td>station_population_density_02deg</td>
<td>same as station_population_density_5km but from a gridded product at 0.2 degrees resolution. The maximum value stored in the database is 16126.</td>
</tr>
<tr>
<td>station_nightlight_5km</td>
<td>nighttime lights extracted from the NOAA DMSP product () at 5km grid resolution. Values are integers from 0 to 63. This keyword accepts a single value or a range (list) of two values specifying the minimum and maximum value for which metadata shall be extracted. If no value has been assigned for a station, the value of -1 is returned.</td>
</tr>
<tr>
<td>station_nightlight_02deg</td>
<td>same as station_nightlight_5km but from a product with grid resolution 0.2 degrees</td>
</tr>
<tr>
<td>station_climatic_zone</td>
<td>integer value indicating the climatic zone of the station location according to <a href="http://eusoils.jrc.ec.europa.eu/projects/RenewableEnergy">http://eusoils.jrc.ec.europa.eu/projects/RenewableEnergy</a>. The argument can be given as single value or a list of values. The codes are: -1 unknown (no value assigned) 0 unclassified 1 Warm Temperate Moist 2 Warm Temperate Dry 3 Cool Temperate Moist 4 Cool Temperate Dry 5 Polar Moist 6 Polar Dry 7 Boreal Moist 8 Boreal Dry 9 Tropical Montane</td>
</tr>
<tr>
<td>Station Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>station_wheat_production</td>
<td>annual wheat production of the year 2000 extracted from <a href="http://gaez.fao.org/Main.html">http://gaez.fao.org/Main.html</a>. The result is a float value in units of thousand tonnes per year. This keyword accepts a single value or a range (list) of two values specifying the minimum and maximum value for which metadata shall be extracted. If no value has been assigned for a station, the value of -999. is returned. The maximum value stored in the database is 22.03.</td>
</tr>
<tr>
<td>station_rice_production</td>
<td>as station_wheat_production but for annual rice production. The maximum value stored in the database is 39.35.</td>
</tr>
<tr>
<td>station_nox_emissions</td>
<td>annual NOx emissions of the year 2010 from EDGAR HTAP inventory V2 (gridded data in units of grams of NO\textsubscript{2} m-2 yr-1 obtained from <a href="http://edgar.jrc.ec.europa.eu/htap_v2/index.php?SECURE=123">http://edgar.jrc.ec.europa.eu/htap_v2/index.php?SECURE=123</a>. This keyword accepts a single value or a range (list) of two values specifying the minimum and maximum value for which metadata shall be extracted. The result is a float value in the range of 0 to 688.55. If no value has been assigned for a station, the value of -999. is returned.</td>
</tr>
<tr>
<td>station_omi_no2_column</td>
<td>5-year average (2011-2015) high-resolution NO\textsubscript{2} column value from the OMI instrument in units of 10\textsuperscript{15} molec cm\textsuperscript{-2}. The product has been obtained from Chris McLinden at Environment Canada. This keyword accepts a single value or a range (list) of two values specifying the minimum and maximum value for which metadata shall be extracted. The result is a float value in the range of 0 to 20.80. If no value has been assigned for a station, the value of -999. is returned.</td>
</tr>
</tbody>
</table>

All station proxy data derived from gridded products were provided by Owen Cooper, NOAA, Boulder, CO, USA.

**Parameter_series metadata**

- parameter_name: (lower case) name of a database parameter. For use in TOAR, choose “o3”.
- parameter_attribute: a string that can be used to distinguish two or more time series of the same parameter at the same station. For example “filtered” and “unfiltered”, or “V1” and “V2”, etc. Most data series have an empty string as parameter_attribute. The argument to search can be a single string or a list of strings.
- parameter_label: a single parameter_label or a list of parameter_labels. A parameter_label consists of the name, attribute and contributor_shortname of a parameter, concatenated with “-”. If a parameter exists with different sampling intervals, a “:T” will be added (where “T” stands for “hourly”, “monthly”, etc.). Examples are described in the Obs_surface_stations_metadata.pdf document.
- parameter_contributor: name of the entity (organization) that provided the data set.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter_contributor_shortname</td>
<td>abbreviation of the parameter_contributor. As with most other string keywords, you can pass a single value or a list of values to the search method.</td>
</tr>
<tr>
<td>parameter_contributor_country</td>
<td>the name of the country where the contributing organization resides. Note that this may differ from the station_country (for example in the case of data contributed by NOAA from their network of global stations).</td>
</tr>
<tr>
<td>parameter_status</td>
<td>integer flag value indicating the status of a data series. The only flag value that is currently in use is the value of 1 for embargoed data.</td>
</tr>
<tr>
<td>data_before</td>
<td>a datetime value or string formatted as &quot;YYYY-mm-dd [HH:MM:SS]&quot; specifying the latest data_start_date of a parameter_series. Use this together with data_after to ensure that all stations which you want to analyze contain at least some data within the period that is of interest to you.</td>
</tr>
<tr>
<td>data_after</td>
<td>similar to data_before, but specifying the earliest data_end_date.</td>
</tr>
<tr>
<td>min_data_length</td>
<td>data_end_date - data_start_date must be of at least this interval. Use the PostgreSQL syntax of interval, for example &quot;5 years&quot;. Also accepts an integer or float value which is then interpreted as &quot;years&quot;.</td>
</tr>
<tr>
<td>max_data_length</td>
<td>similar to min_data_length, but maximum difference between data_end_date and data_start_date. This keyword is primarily used for data quality control.</td>
</tr>
<tr>
<td>created_before</td>
<td>a datetime value or string formatted as &quot;YYYY-mm-dd [HH:MM:SS]&quot;. Only those parameter_series will be returned which were created before or on this date. You can use this keyword together with created_after to define a range of creation dates. Note that this date may change if a data series is overwritten with a new version of the same dataset.</td>
</tr>
<tr>
<td>created_after</td>
<td>same as created_before, but signalling the earliest creation date of a parameter_series.</td>
</tr>
<tr>
<td>modified_before</td>
<td>same as created_before, but inspecting the modification_date instead of the creation_date.</td>
</tr>
<tr>
<td>modified_after</td>
<td>same as created_after, but inspecting the modification_date instead of the creation_date.</td>
</tr>
<tr>
<td>parameter_sampling_type</td>
<td>an attribute that indicates how the measurement was performed. Currently one of “continuous”, “flask”, “remote sensing”.</td>
</tr>
<tr>
<td>parameter_measurement_method</td>
<td>a not yet standardized description of the measurement principle with which the measurement was performed.</td>
</tr>
<tr>
<td>parameter_original_units</td>
<td>the units in which the data were originally reported to the TOAR database. Note that these units may differ from the original reporting units to a data provider.</td>
</tr>
</tbody>
</table>
Examples of database search queries

1. return network_name and station_id of all stations:
   res = db.search(columns="network_name, station_id")
   or:
   res = db.search(columns=["network_name", "station_id"])

2. return the station_name and station_id of all UBA stations which have an O3 record:
   res = db.search(columns="station_name, station_id",
                   network_name="UBA", parameter_name="o3")

3. as before, but limit to stations which have O3 data before 1991, and also obtain the
   station_state ("Bundesland") and the start_date and end_date of the measurements:
   res = db.search(columns="station_name, station_state, " + \n       "station_id, data_start_date, data_end_date",
                   network_name="UBA", parameter_name="o3",
                   data_after="1991-01-01")

4. retrieve the complete metadata of the three stations from the GAW network whose
   station_id begins with "CGO", "CPT", and "USH", respectively:
   res = db.search(network_name="GAW",
                   station_id=['CGO', "CPT", "USH"])

5. find the stations which have an O3 record of at least 30 years:
   res = db.search(columns="network_name, station_id",
                   parameter_name="o3",
                   min_data_length="30 years")

6. get all parameter metadata of all parameters measured at the German UBA station DEUB005
   (Waldhof). Note: this also contains (duplicate) information on the station. The join keyword
   must be set to True in this case in order to force the SQL inner join command.
   res = db.search(network_name="UBA", station_id="DEUB005",
                   join=True)

7. get complete station metadata of all stations with "Bad " in their name:
   res = db.search(station_name="Bad ")

8. likewise for all stations whose name begins with "Bad ":
   res = db.search(station_name="^Bad ")

9. obtain network_name and station_id of all stations within the European region (specified as
    bounding box), which measure CO:
   res = db.search(columns="network_name, station_id",
                   boundingbox=[-12.0,33.5,33.8,74.6],
                   parameter_name="co")

10. find all UBA stations of type "background" the data of which were modified after 2014-10-01.
    Note that there may be multiple records from the same station in the result. Use res =
    list(set(list)) to remove duplicates.
    res = db.search(columns="network_name, station_id",
                     station_type="background",
                     modified_after="2014-10-01")

11. find all French mountain stations (located at altitudes above 2000 m):
    res = db.search(columns=["network_name", "station_id",
                             "station_lon", "station_lat",
                             "station_alt", "station_name"]),
station_country="France",
altitude=[2000., 10000.])

12. compare the location information and lengths of O3 timeseries of the different records from
the German station Westerland:
columns="network_name, station_id, station_lon, station_lat,"
  "station_alt, id, data_start_date, data_end_date"
res = db.search(columns=columns, station_name="Westerland",
  parameter_name="o3")

13. find parameter_series with specific labels
res = db.search(columns="parameter_label, id, station_name",
  parameter_label= ["O3-ALL","O3-FILTERED"])

Advanced database queries (custom queries)

If you are familiar with SQL (PostgreSQL) and the TOAR database structure, you can also formulate
more complex queries via the ObsStationsDatabase object. This is done via the query method as
follows:

with db = ObsStationsDatabase(...) as db:
  q = "SELECT max(station_wheat_production) FROM stations"
columns, result = db.query(q)
print result

In this example, we ask the database to return the maximum value that is stored for the
station_wheat_production metadata field. This is of course much faster than loading all metadata in
python (as per db.search(columns="station_wheat_production",
station_wheat_production = [0., 1.e6])) and then use python to obtain the
maximum value.

The query method returns a list of column names and a list of query results, which are essentially
tuples with the values of the individual column fields. For example, the above “max” query will return
columns = ['max'] and result = [(22.0269,)]. Note that single value tuples in python have to end with a
,.

Another use case for a custom query is the request to return all unique values of a certain database
field. For example, in order to obtain a list of all countries which provided ozone data to TOAR, you
can write:

q = "SELECT DISTINCT parameter_contributor_country FROM " + 
  "parameter_series WHERE parameter_name = 'o3'"
columns, result = db.query(q)
nice_result = sorted([str(x[0]).strip() for x in result])
print nice_result

In principle you could also use this method to directly compute monthly mean values etc. from the
hourly data. However, we advise against this, because the TOAR statistics routines make use of the
agreed data capture thresholds, while this would be rather complicated to factor into an SQL query.
Content of toar_utils.py

The toar_utils.py module includes the following user functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_toar_data</td>
<td>general data extraction routine including search for stations/timeseries, merging of data, and evaluation of statistics/metrics</td>
</tr>
<tr>
<td>get_idlist</td>
<td>utility to obtain (and print) the merged id list corresponding to a database query set of stations. This idlist can then be re-used in get_toar_data if you want to perform multiple analyses on the same set of stations -- or you can use it to obtain the series ids for information (for example to use them in get_toar_metadata)</td>
</tr>
<tr>
<td>plot_data_summary</td>
<td>generate a data summary plot from a merged TOAR data series or from an individual series id</td>
</tr>
</tbody>
</table>

Occasionally you may also need one of the following utilities:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_data_from_list</td>
<td>read and concatenate data from all data_extraction_tuples which belong together. The result will be a DataSlice object which can be passed to the statistical processor</td>
</tr>
<tr>
<td>get_time_offset</td>
<td>return the appropriate timedelta to convert UTC data to local or solar time (note that all data are stored in the database as UTC)</td>
</tr>
<tr>
<td>merge_metadata</td>
<td>combine metadata information from a list of merged records into one metadata record.</td>
</tr>
<tr>
<td>get_toar_metadata</td>
<td>utility to obtain (and print) the complete metadata for a given data series</td>
</tr>
<tr>
<td>merge_ozone_series</td>
<td>takes a list of ReclInfo elements (output from db.search with columns='id') as input and generates an idlist which can be used directly in the get_toar_data routine</td>
</tr>
</tbody>
</table>

The get_toar_data function

This function encapsulates several processing steps which are needed to prepare TOAR statistics or TOAR metrics for plotting or further analysis:

1. Search and identify all (ozone) data sets which match your search criteria (see description of the ObsStationsDatabase search method above),
2. Identify duplicate series and combine data series in case of disrupted measurements at the same station, or when gap filling from one network’s data holding can complement the data from another datacenters holding (the merge process),
3. Loop over the merged series, load all requested data (optionally within a given daterange), and compute a list of specified statistics or metrics (see TOAR surface ozone statistics and metrics.pdf for the complete description of metrics and the allowed sampling intervals and season definitions).
Get_toar_data is very powerful, but can also easily block your computer for hours if you are not careful about formulating your request. There are some security mechanisms built-in to protect you from excessive coffee consumption during the waiting time.

The arguments to get_toar_data are:

<table>
<thead>
<tr>
<th>Keyword name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>daterange</td>
<td>a list with two string elements of the form 'YYYY-MM-DD hh:mm'. If None, the entire length of all timeseries will be extracted. For testing purposes, daterange should be set to shorter intervals.</td>
</tr>
<tr>
<td>region</td>
<td>a string with a region name, i.e. one of the keys defined in the TOAR_REGION dictionary (see documentation of constants below). If present, data will be extracted for this region only (based on the boundingbox specification in TOAR_REGION).</td>
</tr>
<tr>
<td>sampling</td>
<td>the output time interval for statistics, i.e. one of 'daily', 'monthly', 'seasonal', 'vegseason', 'summer', 'annual'. If you specify 'vegseason', you also need to specify crop ('wheat' or 'rice').</td>
</tr>
<tr>
<td>crop</td>
<td>the crop name for specification of 'vegseason' as sampling type.</td>
</tr>
<tr>
<td>statistics</td>
<td>a list of the statistics/metrics that shall be computed (see statistics documentation). Default is calculation of mean values.</td>
</tr>
<tr>
<td>data_capture</td>
<td>the required fractional data capture for statistics. Default is 0.75 as per the TOAR metrics definitions.</td>
</tr>
<tr>
<td>merge</td>
<td>a boolean keyword to activate the automatic merging of time series. If True (default), the station_groups.csv file will be used to determine which portions of which parameter_series shall be used for a given request. Note that the station_groups table had to be constructed manually and there is a small risk for incomplete information or errors.</td>
</tr>
<tr>
<td>idlist</td>
<td>a previously calculated list of lists of data_extraction tuples, i.e. the result of a get_idlist or merge_ozone_series() call. If idlist is not None, the database search and merging procedures will be bypassed and the data will be processed directly. For explanation of the data_extraction tuples see below.</td>
</tr>
<tr>
<td>test</td>
<td>a boolean flag to switch on the testing mode. In testing mode, the data extraction will be limited to the first 10 stations/series. Test defaults to True. This prevents you from unwillingly extracting far too much data only to find out that the rest of your script is not working. Once you are certain that everything is OK, you must remember to set test to False, or every TOAR plot will consist of maximum 10 data series.</td>
</tr>
<tr>
<td>**kwargs</td>
<td>any of the database search keywords as defined in the search method from the ObsStationsDatabase object in obs_stations_database.py. See extensive documentation and examples above.</td>
</tr>
</tbody>
</table>

The output from this routine will be a nested list. Each list element is a list of so-called DataSlices containing the requested statistics/metrics for one series that matches the search conditions.

Get_toar_data is very flexible and powerful, but it has the disadvantage that it will process your entire request in one block. If you encounter an error during processing of many datasets, say at station number 2345, you need to repeat everything. Therefore, it is strongly recommended that you use a combination of get_idlist (see below) and get_toar_data when you wish to process large amounts of data.
**Processing output of get_toar_data**

In order to develop your final processing and plotting routines, you may need to understand how the output of the get_toar_data routine is structured. Let us walk through this by example.

We request various percentiles from all stations whose name begins with “Cape” (hypothesizing these are coastal sites with some similar characteristics), and evaluate these during the “summer” (i.e. the 6-month period from April-September in the Northern Hemisphere and from October-March in the Southern Hemisphere). The get_toar_data request is thus:

```python
data = get_toar_data(daterange=['2007-01-01 00:00',
                                 '2010-01-01 00:00'],
                     station_name="^Cape",
                     sampling="summer",
                     statistics=['percentiles2'])
```

(Note the little caret in front of “Cape” – this is regular expression syntax and denotes the beginning of a string; without it, the search would also return stations which have “Cape” somewhere in the middle of their name)

If you inspect the result via print data, you will see a lot of output which at first may look garbled to you. What get_toar_data returns is a list of results, one result for each series. Each result is itself a tuple that consists of a list of data slices, and a RecInfo structure with the complete metadata information about the station and parameter_series. In our example, there are 8 data slices per data series (i.e. per result), because we requested “percentiles2” as statistics, which returns the 5-, 10-, 25-, 50-, 75-, 90-, 95-, and 98-percentile values (note that the 98-percentile is not included in monthly sampling). Thus, we have a structure like (abbreviated form):

```
[ ([<cdm.data_slice.DataSlice object...>, <cdm.data_slice.DataSlice object...>, ...],
    RecInfo(numid=4596, network_name=u'GAW', station_id=u'GPT134S00', ...)),
  ([<cdm.data_slice.DataSlice object...>, <cdm.data_slice.DataSlice object...>, ...],
    RecInfo(numid=4628, network_name=u'GAW', station_id=u'CVO116N00', ...))
```

Typically, you will want to loop over your data series like so:

```python
# outer loop over data series
for thisdata, thismetadata in data:
    # do something that is needed for all percentiles
    print thismetadata.station_id
    # inner loop over individual statistics
    for stat in thisdata:
        # do something with the statistical result (the data slice)
        print stat.x, stat.y
```

If you wish to directly access a specific result, for example the 75-percentiles from the 5th data series, you can of course use indexing:

```python
P75_5 = data[4][0][4].y
```
This extracts the 5\textsuperscript{th} result (python indexing always starts with zero), from this it takes the first tuple element (the list of data slices), then it indexes the 5\textsuperscript{th} statistics (in this case the 75-percentiles), and finally extracts the y values from this data slice. The result will be:

\[
\text{array([36.667, 33.13, 35.22765, nan])}.
\]

The data slice object (actually DataSlice and defined in cdm/data_slice.py) is a convenient structure to store data in a particular “view”. Here, we will only get “timeseries” views, which means that the DataSlice “knows” that \( x \) values are datetime values, and \( y \) values are the data values. You can easily access the data via \( dslice.x \) and \( dslice.y \). The DataSlice also comes with ample metadata. It has a gattr attribute, which will contain the station information, and a varattr attribute (you can also use yattr), which contains the metadata on the parameter and the statistical processing.

From the example above: if we set \( dslice = data[4][0][4] \), then print \( dslice.gattr \) will yield a dictionary:

\[
\{'station_state': u'S\xe3o Vicente', 'station_category': u'Global', \ldots\}
\]

so will print \( dslice.varattr \):

\[
\{'status': 0, 'units': '', 'name': u'o3', \ldots\}
\]

Note in particular that varattr contains a \textit{long_name} attribute which provides a nice title for your plots:

\[
>>> \text{print dslice.varattr['long_name']}
\]

\textit{Summer 75\%-iles of o3-NH-Summer}

This long\_name is composed of the sampling interval ("summer"), the statistics ("75\%-iles"), the parameter\_name ("o3"), and (if needed) the season name (here "NH-Summer"). For stations in the Southern hemisphere, the “summer” statistics will automatically switch to “SH-Summer” instead.

The DataSlice object also has a filename attribute, which in the example is set to \texttt{GAW\_CVO116N00\_O3}. With two lines it is possible to save the DataSlice into a netcdf file:

\[
dset=dslice.to_dataset()
dset.to_file('output/'+dslice.filename)
\]

\textbf{A note on data series merging}

Some ozone series consist of two or more partial series, and in other cases, stations were relocated within a few hundred metres so that the combined series could be regarded as continuous, but the parts are stored individually in the database. Yet, in other cases, a station reported data to different networks (in Europe you can find data from the same station reported up to four times, namely as UBA dataset, EMEP dataset, Airbase dataset, and finally as GAW dataset). For example, ozone data from the station “Westerland, Sylt” are available under station\_id “DEUB001” (UBA and Airbase), “DE0001R” (EMEP), and “WES654N00” (GAW). Each of these series has different start and end dates:
We visually inspected these data series and identified the most suitable series for use during the different periods. The results of this visual inspection are coded in the toar_dataset_groups.csv table, which contains the necessary information how the data series shall be merged in a machine-readable form. For the example of Westerland, you will find the following two lines in this table:

```
[16601; 25758; 21935; 23065],19,DE0001R,EMEP,25758,29.02.1984 23:00,31.12.1989 23:00
[16601; 25758; 21935; 23065],19,DEUB001,UBA,16601,01.01.1990 01:00,03.03.2016 06:00
```

The first column contains a list of data series ids (unambiguous numerical identifiers of a data series) to which the following merge statement applies. Here, we have the ids for the Westerland series from UBA, EMEP, GAW, and Airbase (in this order). The next column is a “statement number” – all lines with the same statement number belong together. Then we see the station_id and network name of the data series that shall be used during one period of the time series. This is purely for human readability and not evaluated by the merging code. Finally, we see the data series id that shall be used (corresponding to the station_id and network_name listed before), and the date range in which this series shall be used.

In this example, the result of any query asking for data from Westerland will always return EMEP data during the time period from February, 29th, 1984 to December, 31st, 1989, and UBA data beginning with January 1st, 1990. So, even if you ask for Airbase data from Westerland, you will get the EMEP and UBA data series instead, unless you set the merge keyword to False in either the get_toar_data or get_idlist function calls.

As mentioned above, the merging table has been constructed by hand, so please be aware that there may be errors. We will appreciate any feedback on series that are merged while they shouldn’t be or series that are not merged although they should be.

The information that is extracted from the merging table is saved as so-called data_extraction_tuples. Such a tuple is nothing else than the combination of the series id to be used, the starting date for its use and the end date for its use. If either the staring date or end date are undefined, None can be used instead.

Example:

Suppose you wish to construct a data_extraction_tuple that will trigger reading of the Airbase series from Westerland. This might be `(23056, None, None)` if we want to use Airbase data during the entire period for which we request data. Note that get_toar_data always requires the data_extraction_tuples to be given as list (the idlist keyword). Hence, you would have to pass

```
idlist = [(23056, None, None)]
```

in this case. Note also that in this case it would be easier to set merge=False and search for the Airbase dataset from Westerland by specifying `network_name="AIRBASE" and station_name="Westerland"`. 


Split your task with get_idlist when you shoot for big targets

When you plan to do any major analysis of TOAR data involving many stations and possibly several statistics together, you should break your job into pieces and follow the program structure of the test_data_capture.py program. In particular, it is strongly recommended that you first establish a list of data_extraction_tuples with the get_idlist function from toar_utils.py and then loop over the list entries to process each merged data series one after the other. This allows for much better control of errors and you will not need to reprocess everything but can resume your work at the point where the program failed.

The get_idlist functions accepts the same arguments as get_toar_data, except for those arguments that are related to the definition of the statistics you want to compute, the output sampling frequency, and the data_capture. When you then call get_toar_data with individual idlist elements, make sure to pass these as list, and pass the keyword arguments that are related to the statistical processing. Also, if you wish to use a daterange, make sure to include this (also) in get_toar_data.

An example sequence looks like the following:

daterange = ['2000-01-01 00:00', '2010-01-01 00:00']
idlist = get_idlist(merge=True, region='East Asia', test=False)
statistics = ['data_capture', 'percentiles1']
for series in idlist:
data = get_toar_data(idlist=[series], daterange=daterange,
sampling='monthly', statistics=statistics)
    # do something with the results

Once again, note that you should use test=True initially to develop your program and design the output you wish to generate. Only when you are sure that everything works as expected, set test=False and start the big number crunching job.

Useful predefined constants

The TOAR python tools contain a number of useful constants that are defined at various places in the code. This section describes the most relevant of these:

**TOAR_REGION** (in toar_utils.py):

A simple dictionary with the following strings as key names (bounding boxes given in square brackets):

TOAR_REGION['Europe'] = [-28., 36., 41., 85.]
TOAR_REGION['North America'] = [-169., 24.,-52., 85.]
TOAR_REGION['South and Central America'] = [-107., -57., -31., 23.999]
TOAR_REGION['Australia and Oceania'] = [93., -50., 180., 19.999]
TOAR_REGION['Africa and Middle East'] = [-26.,-39.,59.999,35.999]
TOAR_REGION['East Asia'] = [93.,20.,180.,65.]
TOAR_REGION['Central and Southern Asia'] = [60., 0.,92.999,65.]
TOAR_REGION['Antarctic'] = [-180., -90., 180., -60.]
TOAR_REGION['SH'] = [-180., -90., 180., 0.]
TOAR_REGION['NH_NA'] = [-180., 0., -40., 90.]

**CLIMATIC_ZONE** (in cdm/obs_stations_database.py):
An ordered dictionary linking the climatic_zone keys to the string definitions:

CLIMATIC_ZONE[-1] = "undefined"
CLIMATIC_ZONE[0] = "unclassified"
CLIMATIC_ZONE[1] = "warm temperate moist"
CLIMATIC_ZONE[2] = "warm temperate dry"
CLIMATIC_ZONE[3] = "cool temperate moist"
CLIMATIC_ZONE[4] = "cool temperate dry"
CLIMATIC_ZONE[5] = "polar moist"
CLIMATIC_ZONE[6] = "polar dry"
CLIMATIC_ZONE[7] = "boreal moist"
CLIMATIC_ZONE[8] = "boreal dry"
CLIMATIC_ZONE[9] = "tropical montane"
CLIMATIC_ZONE[10] = "tropical wet"
CLIMATIC_ZONE[12] = "tropical dry"

If you wish to return the index number for a given climatic_zone string, you can do the following:

```python
idict = { val: key for key, val in CLIMATIC_ZONE.items() }
cindex = idict['warm temperate dry']
```

This creates an “inverse dictionary” idict which has the former integer indices as values and the former string values as indices.

**WMO_QUALITY_FLAG** (in cdm/obs_stations_database.py):
Definition of the data quality flags that accompany each value in the database. Default is to only extract data with flag values of 0 (“OK”) or 4 (“not checked”). Note that the TOAR database does not store missing values. These are simply deleted from the time series.

WMO_QUALITY_FLAG["OK"] = 0
WMO_QUALITY_FLAG["inconsistent"] = 1
WMO_QUALITY_FLAG["doubtful"] = 2
WMO_QUALITY_FLAG["wrong"] = 3
WMO_QUALITY_FLAG["not_checked"] = 4
WMO_QUALITY_FLAG["changed"] = 5
WMO_QUALITY_FLAG["estimated"] = 6
WMO_QUALITY_FLAG["missing_value"] = 7