National Centre for Atmospheric Science

NATURAL ENVIRONMENT RESEARCH COUNCIL

ncas-amof-netcdf-template documentation

Version 2.3.1

Contents

1 ncas_amof_netcdf_template documentation

ncas_amof_netcdf_template is a Python module to help NCAS instrument scientists create netCDF files that meet the NCAS-GENERAL netCDF metadata standard, starting with version 2.0 of the standard. This standard is based on version 1.6 of the CF metadata conventions, with additional information required regarding the instrument used and the state of its deployment.

2 Installation

2.1 Conda

The latest release version of the module, along with any dependencies, can be downloaded and installed using conda:

```
conda install -c conda-forge ncas-amof-netcdf-template
```

2.2 Pip

The latest release version is also availalbe through pip:

```
pip install ncas-amof-netcdf-template
```

2.3 GitHub

Alternatively, the latest version of the module can be downloaded from GitHub:

```
git clone https://github.com/joshua-hampton/ncas_amof_netcdf_template.git
cd ncas_amof_netcdf_template
pip install .
```

Previous release versions can also be found through GitHub.

3 How to use

Unless otherwise stated, all examples will use the `ncas-ceilometer-3` instrument as an example.

3.1 Create netCDF file

In its very simplest form:

```
import ncas_amof_netcdf_template as nant
ncs = nant.create_netcdf.main('ncas-ceilometer-3')
```

This will create a netCDF file with today's date for all data products availalbe for the given instrument. If files for multiple products are made, the returned object will be a list containing all objects; if only a single file then just that netCDF file object is returned.

A netCDF file can also be created for a specific data product, rather than by a specific instrument.

```
nc = nant.create_netcdf.make_product_netcdf('surface-met', 'my-home-weather-station')
```

The file created in this example uses the `surface-met` data product definition, and requires the instrument name `my-home-weather-station` for the file name.

3.1.1 Dimensions

Dimension sizes need to be defined when creating a netCDF file. Dimension lengths can be provided to the `create_netcdf.main` function as a dictionary:

```
ncs = nant.create_netcdf.main('ncas-ceilometer-3', dimension_lengths = {'time':96, 'altitude':45, 'layer_index':4})
```

If dimensions aren't given, Python asks for the dimension lengths to be given:

```
ncs = nant.create_netcdf.main('ncas-ceilometer-3')
Enter length for dimension time: 96
Enter length for dimension altitude: 45
Enter length for dimension layer_index: 4
```

3.1.2 Platform

The deployment platform, where the instrument was located while measuring data, is recorded in both the file name and as a global attribute. The platform names are controlled by CEDA. NCAS instruments that are primarily based at an NCAS observatory have the relevant platform associated with the instrument, but mobile instruments have the phrase "mobile" listed as the default platform, which needs to be changed. Both the `create_netcdf.main` and `create_netcdf.make_product_netcdf` functions can take a `platform` argument, which sets or overrides the platform name to use:

```
nc = nant.create_netcdf.main("ncas-ceilometer-3", platform = "cao")
```

3.1.3 Date

The file-naming convention for the NCAS-GENERAL standard includes the date, and sometimes time, which the data within the file represents. By default, today's date will be used in the file name. This behaviour can be overridden by giving the date to the `create_netcdf.main` function:

```
ncs = nant.create_netcdf.main('ncas-ceilometer-3', date='20220214')
```

3.1.4 Data Products

List available data products for an instrument:

```
nant.create_netcdf.list_products('ncas-ceilometer-3')
```

Alternatively, all possible data products can be listed if no instrument name is given.

A data product can be defined in the call to create the netCDF file:

```
nc = nant.create_netcdf.main('ncas-ceilometer-3', products = 'aerosol-backscatter')
```

Or multiple products can be defined by using a list:

```
ncs = nant.create_netcdf.main('ncas-ceilometer-3', products = ['cloud-base','cloud-coverage'])
```

3.1.5 Deployment Modes

NCAS instruments can be deployed in one of four deployment modes - land, sea, air, or trajectory. Each of these modes requires different dimensions and variables, and the deployment mode is recorded as a global attribute in the netCDF file. The default deployment mode is `'land'`; however, an alternative deployment mode can be selected using the `loc` keyword:

```
ncs = nant.create_netcdf.main('ncas-ceilometer-3', loc = 'sea')
```

3.1.6 Output Location

The netCDF file will be written to the current working directory by default. To specify an alternative location, the `'file_location'` keyword can be used:

```
ncs = nant.create_netcdf.main('ncas-ceilometer-3', file_location = '/path/to/save/location')
```

3.1.7 Offline Use

The information needed to create these netCDF files are stored in the AMF_CVs GitHub repository, and this package reads data from this repository when it is used. If the package will need to be used offline, the tsv product-definitions folder should be downloaded onto the computer, and the option `use_local_files` can be passed to functions such as `create_netcdf.main` with the path to the product definitions as the argument.

3.1.8 Other Options

All available options for this function can be found on this API page.

3.2 Add Data

After the netCDF file is created, the file then needs to be opened in append mode, and data can then be added to the file:

```
import ncas_amof_netcdf_template as nant
from netCDF4 import Dataset

# Read raw data into python
# ...
# backscatter_data = ...
```

```
nc = nant.create_netcdf.main('ncas-ceilometer-3', date='20221117', product = 'aerosol-backscatter')

nant.util.update_variable(nc, 'attenuated_aerosol_backscatter_coefficient', backscatter_data)
```

where `'attenuated_aerosol_backscatter_coefficient'` is the name of the variable in the netCDF file, and `'backscatter_data'` is an array containing the data. This will also update the `valid_min` and `valid_max` attributes for each variable where applicable.

3.2.1 Quality Control Flag Data

Quality control flags in the NCAS-GENERAL standard use flag_values and flag_meanings to convey the quality of the data. When adding data to a quality control variable, an error is raised if that data includes values not in the flag_values attribute.

3.3 Time

netCDF files that follow the NCAS-GENERAL metadata standard require a number of variables that correspond to time, or a portion of it, including (but not limited to) UNIX time, year, month and day. This module includes a function that will take a list of datetime objects and return the times in all the required formats.

```
import ncas_amof_netcdf_template as nant
import datetime as dt

# generate some times for this example
t1 = dt.datetime.strptime('20221117T120000','%Y%m%dT%H%M%S')
t2 = dt.datetime.strptime('20221117T120500','%Y%m%dT%H%M%S')
times = [t1,t2]

unix_times, day_of_year, years, months, days, hours, minutes, seconds, \
  time_coverage_start_unix, time_coverage_end_unix, file_date = nant.util.get_times(times)
```

This returns 8 lists with the time formatted as needed for variables in the netCDF file, as well as the first and last UNIX time stamp which can be used for the time coverage start and end metadata fields, and the date/time with the correct precision which, if required, could be used for the date in the `create_netcdf.main` function (e.g. in the example above it would return `'20221117-12'`).

3.4 Metadata

While all required metadata fields are added to the global attributes of the netCDF file, and in some cases the defined values are directly inserted, it is necessary to add further metadata values to the netCDF file, for example `creator_name`. Fields that need metadata adding to them are initially given placeholder text which starts with the word "CHANGE" - simple interrogation of the created netCDF file will reveal which attributes need specifying.

The contents of a CSV file containing metadata can then be added to the netCDF file

```
nant.util.add_metadata_to_netcdf(nc, 'metadata.csv')
```

Metadata can be supplied in CSV, JSON, YAML or XML formats; see the metadata formats page for more details. The `add_metadata_to_netcdf` function will attempt to detect the format type based on the file extension. If this detection fails, the `file_format` argument can be used, e.g.

```
nant.util.add_metadata_to_netcdf(nc, 'metadata_file', file_format = 'csv')
```

If detection fails and `file_format` is not given, the function will attempt to read the file as a CSV.

One additional parameters can be supplied in the metadata file with each individual attributes:

- `type` - what data type the value of the attribute should take, e.g. `integer` or `string`. Default if absent is `string`.

3.4.1 Latitude, Longitude, and Geospatial Bounds

Although latitude and longitude are variables in the netCDF file, single value latitude and longitude values, with units "degrees North" and "degrees East" respectively can be included in the metadata file, for example if using a CSV meatadata file

```
latitude,53.801277
longitude,-1.548567
```

The `geospatial_bounds` global attribute can also be defined directly in the metadata file, or calculated from the latitude and longitude values:

```
nant.util.add_metadata_to_netcdf(nc, 'metadata.csv')
geobounds = f"{ncfile.variables['latitude'][0]}N, {ncfile.variables['longitude'][0]}E"
nc.setncattr('geospatial_bounds', geobounds)
```

3.4.2 Time Coverage Start and End

As mentioned above, the `time_coverage_start` and `time_coverage_end` global attribute values can be obtained using the get_times function. The returns from this function include the first and last times as UNIX time stamps, which can be converted into the correct format for the global attribute values:

```
dt.datetime.fromtimestamp(time_coverage_start_unix, dt.timezone.utc).strftime("%Y-%m-%dT%H:%M:%S")
```

3.5 Remove Empty Variables

The NCAS-GENERAL metadata standard can be seen as two parts: the first being "common" attributes, dimensions and variables that are required in all files, the second is "product-specific" information, for example the `aerosol-backscatter` product has variables `attenuated_aerosol_backscatter_coefficient` and `range_squared_corrected_backscatter_power` which are not in the `cloud-base` product. However, there may be cases where the instrument does not measure one or more of these product-specific variables. These empty product-specific variables should not be included in the final netCDF file.

```
nant.remove_empty_variables.main('./ncas-ceilometer-3_iao_20221117_aerosol-backscatter_v1.0.nc')
```

The netCDF file needs to be closed before this can be done, using `nc.close()`.

3.6 Full Example

An example of a full work flow using `ncas_amof_netcdf_template` to create the netCDF file, where is is assumed the actual reading of the raw data is handled by a function called `read_data_from_raw_files`, and metadata is stored in a file called `metadata.csv`.

```
import ncas_amof_netcdf_template as nant
import datetime as dt
from netCDF4 import Dataset

# Read the raw data with user-written function, with times returning data in datetime format
# In this example, `time` and `altitude` are the only dimensions
backscatter_data, times, altitudes, other variables = read_data_from_raw_files()

# Get all the time formats
```

```
unix_times, day_of_year, years, months, days, hours, minutes, seconds, \
  time_coverage_start_unix, time_coverage_end_unix, file_date = nant.util.get_times(times)

# Create netCDF file and read it back into the script in append mode
nc = nant.create_netcdf.main('ncas-ceilometer-3', date = file_date,
                             dimension_lengths = {'time':len(times), 'altitude':len(altitudes)},
                             loc = 'land', products = 'aerosol-backscatter',
                             file_location = ncfile_location)

# Add variable data to netCDF file
nant.util.update_variable(nc, 'altitude', altitudes)
nant.util.update_variable(nc, 'attenuated_aerosol_backscatter_coefficient',
                          backscatter_data)
nant.util.update_variable(nc, 'time', unix_times)
nant.util.update_variable(nc, 'day_of_year', day_of_year)
nant.util.update_variable(nc, 'year', years)
# and so on for each time format

# Add metadata from file
nant.util.add_metadata_to_netcdf(nc, 'metadata.csv')

# Add time_coverage_start and time_coverage_end metadata using data from get_times
nc.setncattr('time_coverage_start',
             dt.datetime.fromtimestamp(time_coverage_start_unix, dt.timezone.utc).strftime("%Y-%m-%dT%H:%M:%S"))
nc.setncattr('time_coverage_end',
             dt.datetime.fromtimestamp(time_coverage_end_unix, dt.timezone.utc).strftime("%Y-%m-%dT%H:%M:%S"))

# Look to see if latitude and longitude values have been added, and
# geospatial_bounds NOT added, through the metadata file
lat_masked = nc.variables['latitude'][0].mask
lon_masked = nc.variables['longitude'][0].mask
geospatial_attr_changed = "CHANGE" in nc.getncattr('geospatial_bounds')
if geospatial_attr_changed and not lat_masked and not lon_masked:
    geobounds = f"{nc.variables['latitude'][0]}N, {nc.variables['longitude'][0]}E"
    nc.setncattr('geospatial_bounds', geobounds)

# Close file
nc.close()

# Check for empty variables and remove if necessary
nant.remove_empty_variables.main(f'{ncfile_location}/ncas-ceilometer-3_iao_{file_date}_aerosol-backscatter_v1.0.nc')
```

4 Metadata File Formats

Metadata can be provided in one (or more if so desired) of four different formats - CSV, JSON, YAML and XML. This page describes the required layout of each of these formats.

4.1 CSV

Metadata in CSV files must be formatted with one attribute per line, starting with the name of the attribute, followed by its value (which can have commas in it), optionally followed by `type=`, for example:

```
attribute_name1,attribute_value1
attribute_name2,153,type=integer
attribute_name3,attribute_value3, value3 continued, type=string
```

This will write to three attributes into the netCDF file:

- `attribute_name1` will be written with the value `attribute_value1`.
- `attribute_name2` will be written with the value `153` as an integer.
- `attribute_name3` will be written with the value `attribute_value3, value3 continued` as a string.

4.2 JSON

The following example will produce the same end result as above:

```
{
    "attribute_name1": "attribute_value1",
    "attribute_name2": {
        "value": 153,
        "type": "int"
    },
    "attribute_name3": {
        "value": "attribute_value3, value3 continued",
        "type": "string"
    }
}
```

In the JSON format, if the value given to the attribute name is a single value (as in `attribute_name1`), then that will be the attributes value, with the default options for `type` applied. If the value given is a dictionary (as in `attribute_name2` and `attribute_name3`, then it must have a key `value` for the value of the attribute, and can optionally have `type` given.

4.3 YAML

The same information as shown in the previous examples can be given in a YAML file:

```
attribute_name1: attribute_value1
attribute_name2:
  value: 153
  type: int
attribute_name3:
  value: attribute_value3, value3 continued
  type: string
```

4.4 XML

These metadata can also be given in an XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<attributes>
    <attribute_name1>
        <value>attribute_value1</value>
    </attribute_name1>
    <attribute_name2>
        <value>153</value>
        <type>integer</type>
    </attribute_name2>
    <attribute_name3>
        <value>attribute_value3, value3 continued</value>
        <type>string</value>
    </attribute_name3>
</attributes>
```

5 History and Deprecations

5.1 Revision History

Important changes of note with each release:

5.1.1 2.4.0

- Added option to overwrite platform used in file name and global attribute - `platform` attribute to `create_netcdf.main` and `create_netcdf.make_product_netcdf`.
- Deprecating use of `instrument_loc` in `tsv2dict.product_dict` and `create_netcdf.make_product_netcdf` - use `platform` instead. `instrument_loc` will be removed in version 2.6.0.
- Added options for metadata files in different file formats, including the option to specify the data type of the value of the attribute - this supercedes the update to numbers in metadata in version 2.3.2.
- Updated default for `return_open` argument to `True` (see revision in version 2.3.0 for more information).

5.1.2 2.3.2

- Corrected how microseconds were being treated by `util.get_times`
- Numbers in metadata can be neatly added as strings to global attributes by surrounding them with single quotes, for example *'1.2'* in the metadata CSV file. Numbers not surrounded by single quotes are still treated as integers or floats.

5.1.3 2.3.1

- Added package version number to text in the history global attribute.
- `util.get_times` returns `day_of_year` as a list rather than an array, in line with other times returned.

5.1.4 2.3.0

- Dropped support for Python 3.7, added support for Python 3.12
- Added History and Deprecations page to documentation.
- Error raised (with option for warning instead) when using `util.update_variable` to add data to Quality Control variable if that data includes values not in the `flag_values` variable attribute.
- Added deprecation to `create_netcdf.main`, `create_netcdf.make_netcdf`, and `create_netcdf.make_product_netcdf`, for closing the netCDF file after initial creation and population. As of version 2.5.0, these functions will all return an open netCDF file, or a list containing open netCDF files in the case of the function creating multiple files, e.g. multiple data products. This behaviour can be used from version 2.3.0 by passing `return_open=True` to these functions. As of version 2.4.0, `return_open=True` will be the default option, with the previous behaviour available by passing `return_open=False`. In version 2.5.0, the behaviour of `return_open=False` will be removed.
- Added option to use locally saved tsv files rather than reading from GitHub.

5.2 Deprecation Policy

Through the life of software, it is very likely parts of the package will have to be changed and altered in such a way the user will have to make small changes to their work in order to keep up with changes in the software. While these deprecations are kept to a minimum, they cannot be altogether avoided. This policy states how this package aims to deal with changes and deprecation of code.

If something in the package is to be deprecated and replaced:

1. A `DeprecationWarning` must be raised when the code that will be removed will be executed, which must mention in which version of the package that thing will no longer work or be available. That version must be at least 2 minor versions later (i.e. code that raises a deprecation warning added in 2.3.x cannot be removed until 2.5.0 at the earliest).
2. If possible, a way of using the new code should be made available simultaneously. In this case, the deprecation warning must include information to the user on how to use the new code. If this is not possible, the deprecation warning should last for at least 3 minor versions (i.e. if first raised in 2.3.x, it should not be removed until 2.6.0), with beta versions of the package published with the new code.
3. If the change in the code can be contained within a boolean argument to a function, then that argument must default to the original code when the deprecation warning is first added, and removed in the version of the package where the code is deprecated, but it is allowed to change the default option to the new code in an intermediate release while the old version of the code is still available within the function. This change of default option must happen in the next minor version release at the earliest, with the code removed at least one further minor release later (i.e. if a deprecation is added in 2.3.x and is covered by a boolean argument, in 2.3.x that boolean argument must default to the original code, in 2.4.x it can default to original or new, and then the argument and original code can be removed from 2.5.0). If this default option is changed before the deprecated code is removed, this must be mentioned in the deprecation warning before the change happens (i.e. in the example above, in version 2.3.x), and in the deprecation warning afterwards (i.e. in 2.4.x) it must state how to use the original code.
4. Deprecations must be added to the documentation, including what is being deprecated, which version the deprecation warning was first introduced, in which version the original code will be removed, and all information on how to use both original and new versions of the code in the intervening releases.
5. All references to minor release versions must be superseded by a major release version, for example if a deprecation warning introduced in 2.3.x states something will be removed in 2.5.0, then that thing must be removed in 3.0.0, even if it comes out before 2.5.0 (which presumably would then not be published).

6 Suggestions and Further Help

6.1 Code layout

ncas-ceilometer-3-software is an example of this module in use for creating NCAS-GENERAL netCDF files. There are four main components to this repo:

1. `read_ceilometer.py` - this script contains functions to read the raw data files from the instrument and returns all of the variables and data in a Python-friendly format.
2. `process_ceilometer.py` - this uses the functions in `read_ceilometer.py` to read the raw data, and uses the functions from this module to actually create the netCDF file.
3. `metadata.csv` - CSV file which has relevant metadata for this instrument, used as described here.
4. `scripts` - folder that contains Bash scripts that can be called in a crontab/scheduler to automatically create netCDF files on a regular basis.

6.2 Problems?

The best way to get help is through issues on GitHub. Here you'll be able to see if anyone else has had the same problem, and any fixes or solutions that may have been found, or raise an issue to highlight a new problem.

If you find a problem and also work out a solution to the issue, feel free to fork the GitHub repo and create a pull request with your solution. This will then be reviewed before being accepted into a future release.