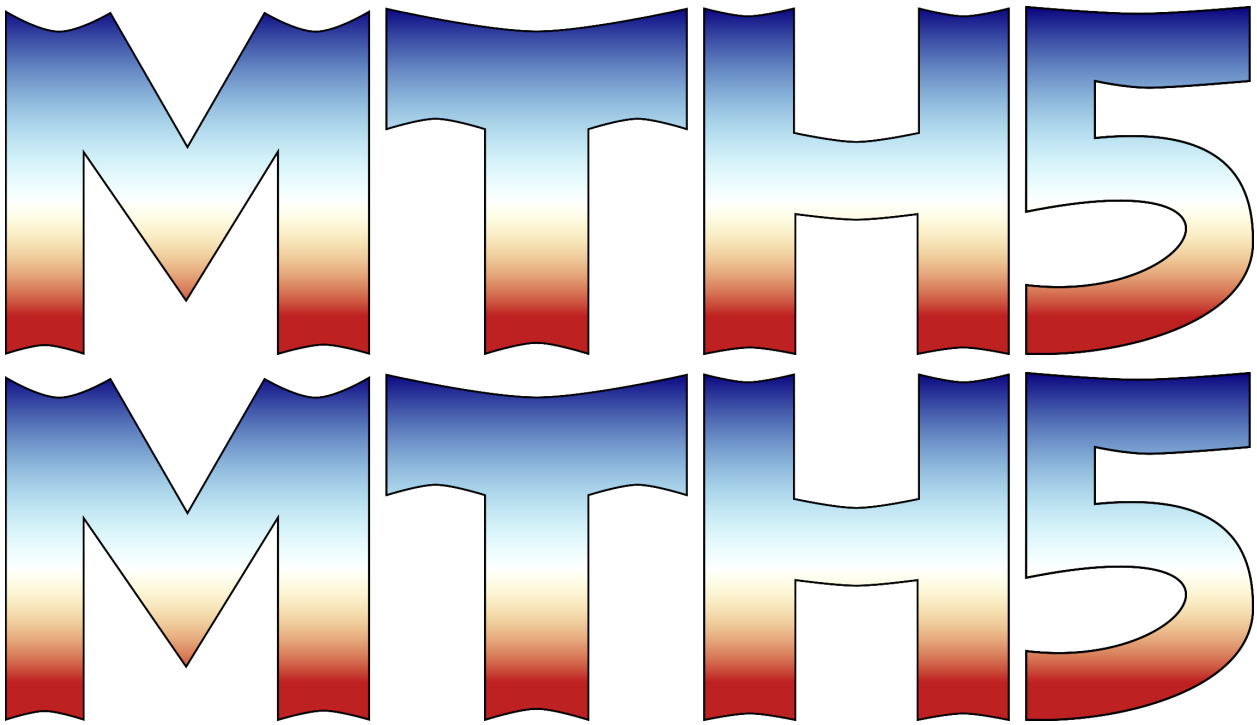
MTH5 Documentation

Release 0.2.0

Jared Peacock

Jun 27, 2021

CONTENTS:



CHAPTER

ONE

INTRODUCTION

The goal of **MTH5** is to develop a standard format and tools for archiving magnetotelluric (MT) time series data.

The preferred format is HDF5 and has been adopted to conform to MT data, something that has been needed in the EM community for some time. The module mth5 contains reading/writing capabilities and will contain tools for retrieving data in useful ways to work with processing codes.

The metadata follows the standards proposed by the IRIS-PASSCAL MT Software working group and documented in MT Metadata Standards.

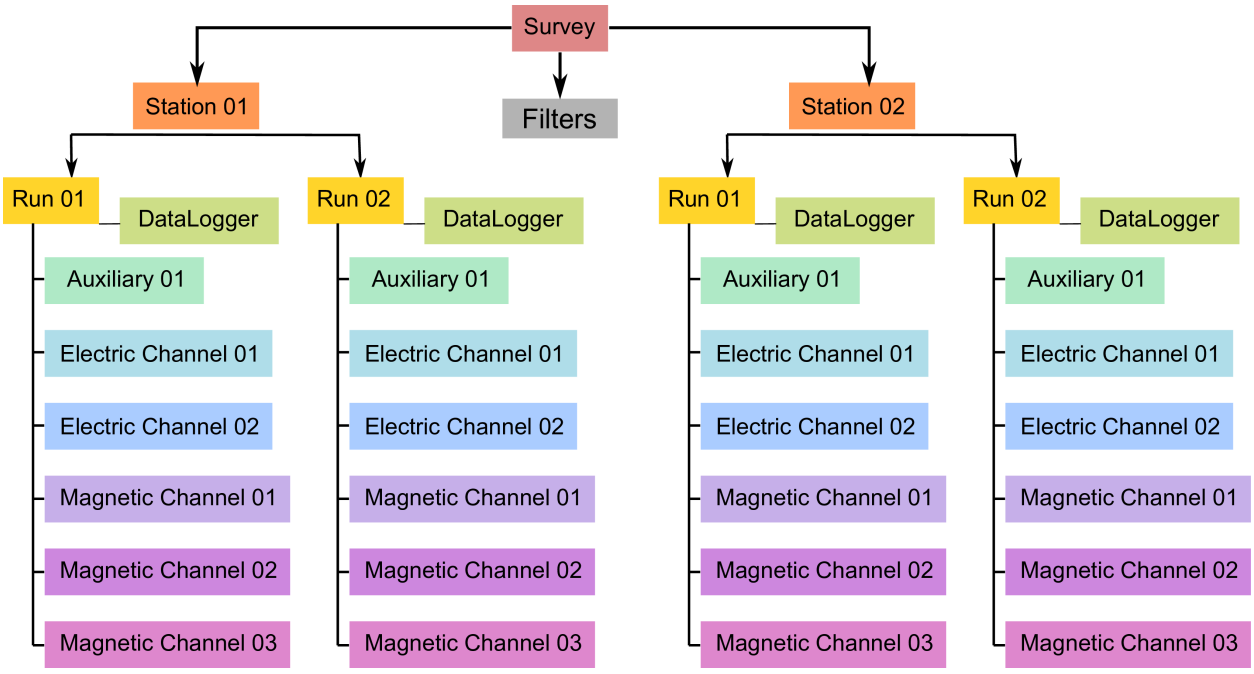
Note: This is a work in progress. Feel free to comment or send me a message at jpeacock@usgs.gov on the data format.

CHAPTER

TWO

MTH5 FORMAT

- The basic format of MTH5 is illustrated below, where metadata is attached at each level.



CHAPTER

THREE

INSTALLATION

3.1 Stable release

To install MTH5, run this command in your terminal:

```
$ pip install mth5
```

This is the preferred method to install MTH5, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

Note: This will not currently work, only a source download is available.

3.2 From sources

The sources for MTH5 can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/kujaku11/mth5
```

Or download the tarball:

```
$ curl -OJL https://github.com/kujaku11/mth5/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

FOUR

USAGE

MTH5 is written to make read/writing an *.mth5* file easier.

Hint: MTH5 is comprehensively logged, therefore if any problems arise you can always check the mth5_debug.log and the mth5_error.log, which will be written to your current working directory.

Each MTH5 file has default groups. A 'group' is basically like a folder that can contain other groups or datasets. These are:

- **Survey** –> The master or root group of the HDF5 file
- **Filters** –> Holds all filters and filter information
- **Reports** –> Holds any reports relevant to the survey
- **Standards** –> A summary of metadata standards used
- **Stations** –> Holds all the stations an subsequent data

Each group also has a summary table to make it easier to search and access different parts of the file. Each entry in the table will have an HDF5 reference that can be directly used to get the appropriate group or dataset without using the path.

4.1 Opening and Closing Files

To open a new *.mth5* file:

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open(r"path/to/file.mth5", mode="w")
```

To open an exiting *.mth5* file:

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open(r"path/to/file.mth5", mode="a")
```

Note: If 'w' is used for the mode, it will overwrite any file of the same name, so be careful you don't overwrite any files. Using 'a' for the mode is safer as this will open an existing file of the same name and will give you write privilages.

To close a file:

```
>>> mth5_obj.close_mth5()
2020-06-26T15:01:05 - mth5.mth5.MTH5.close_mth5 - INFO - Flushed and
closed example_02.mth5
```

Note: Once a MTH5 file is closed any data contained within cannot be accessed. All groups are weakly referenced, therefore once the file closes the group can no longer access the HDF5 group and you will get a similar message as below. This is to remove any lingering references to the HDF5 file which will be important for parallel computing.

```
>>> 2020-06-26T15:21:47 - mth5.groups.Station.__str__ - WARNING - MTH5 file is closed␣
↪and cannot be accessed. MTH5 file is closed and cannot be accessed.
```

A MTH5 object is represented by the file structure and can be displayed at anytime from the command line.

```
>>> mth5_obj
/:
====================
        |- Group: Survey
        ----------------
                |- Group: Filters
                -----------------
                        --> Dataset: Summary
                        ......................
                |- Group: Reports
                -----------------
                        --> Dataset: Summary
                        ......................
                |- Group: Standards
                -------------------
                        --> Dataset: Summary
                        ......................
                |- Group: Stations
```

(continues on next page)

(continued from previous page)

```
                ------------------
                        |- Group: MT001
                        ---------------
                                --> Dataset: Summary
                                ......................
                        --> Dataset: Summary
                        ......................
```

This file does not contain a lot of stations, but this can get verbose if there are a lot of stations and filters. If you want to check what stations are in the current file.

```
>>> mth5_obj.station_list
['Summary', 'MT001']
```

Each group has a property attribute with an appropriate container including convenience methods. Each group has a property attribute called *group_list* that lists all groups the next level down.

See also:

mth5.groups and `mth5.metadata` for more information.

4.2 Metadata

Each group object has a container called *metadata* that holds the appropriate metadata (`mth5.metadata`) data according to the standards defined at MT Metadata Standards. The exceptions are the HDF5 file object which has metadata that describes the file type and is not part of the standards, and the stations_group, which is just a container to hold a collection of stations.

Input metadata will be validated against the standards and if it does not conform will throw an error.

The basic Python type used to store metadata is a dictionary, but there are three ways to input/output the metadata, dictionary, JSON, and XML. Many people have their own way of storing metadata so this should accommodate most everyone. If you store your metadata as JSON or XML you will need to read in the file first and input the appropriate element to the metadata.

4.2.1 Setting Attributes

Metadata can be input either manually by setting the appropriate attribute:

```
>>> existing_station = mth5_obj.get_station('MT001')
>>> existing_station.metadata.archive_id = 'MT010'
```

Hint: Currently, if you change any *metadata* attribute you will need to mannually update the attribute in the HDF5 group:

```
>>> existing_station.write_metadata()
```

4.2.2 Metadata Help

To get help with any metadata attribute you can use:

```
>>> existing_station.metadata.attribute_information('archive_id')
```

archive_id: alias: [] description: station name that is archived {a-z;A-Z;0-9} example: MT201 options: [] required: True style: alpha numeric type: string units: None

If no argument is given information for all metadata attributes will be printed.

4.2.3 Creating New Attributes

If you want to add new standard attributes to the metadata you can do this through **:function:`mth5.metadata.Base.add_base_attribute method`**

```
>>> extra = {'type': str,
...          'style': 'controlled vocabulary',
...          'required': False,
...          'units': 'celsius',
...          'description': 'local temperature',
...          'alias': ['temp'],
...          'options': [ 'ambient', 'air', 'other'],
...          'example': 'ambient'}
>>> existing_station.metadata.add_base_attribute('temperature', 'ambient', extra)
```

4.2.4 Dictionary Input/Output

You can input a dictionary of attributes

Note: The dictionary must be of the form {'level': {'key': 'value'}}, where 'level' is either ['survey' | 'station' | 'run' | 'channel' | 'filter']

```
>>> meta_dict = {'station': {'archive_id': 'MT010'}}
>>> existing_station.metadata.from_dict(meta_dict)
>>> exiting_station.metadata.to_dict()
{'station': OrderedDict([('acquired_by.author', None),
      ('acquired_by.comments', None),
      ('archive_id', 'MT010'),
      ('channel_layout', 'X'),
      ('channels_recorded', ['Hx', 'Hy', 'Hz', 'Ex', 'Ey']),
      ('comments', None),
      ('data_type', 'BB, LP'),
      ('geographic_name', 'Beachy Keen, FL, USA'),
      ('hdf5_reference', '<HDF5 object reference>'),
      ('id', 'FL001'),
      ('location.declination.comments',
       'Declination obtained from the instrument GNSS NMEA sequence'),
      ('location.declination.model', 'Unknown'),
      ('location.declination.value', -4.1),
```

(continues on next page)

(continued from previous page)

```
      ('location.elevation', 0.0),
      ('location.latitude', 29.7203555),
      ('location.longitude', -83.4854715),
      ('mth5_type', 'Station'),
      ('orientation.method', 'compass'),
      ('orientation.reference_frame', 'geographic'),
      ('provenance.comments', None),
      ('provenance.creation_time', '2020-05-29T21:08:40+00:00'),
      ('provenance.log', None),
      ('provenance.software.author', 'Anna Kelbert, USGS'),
      ('provenance.software.name', 'mth5_metadata.m'),
      ('provenance.software.version', '2020-05-29'),
      ('provenance.submitter.author', 'Anna Kelbert, USGS'),
      ('provenance.submitter.email', 'akelbert@usgs.gov'),
      ('provenance.submitter.organization',
       'USGS Geomagnetism Program'),
      ('time_period.end', '2015-01-29T16:18:14+00:00'),
      ('time_period.start', '2015-01-08T19:49:15+00:00')])}
```

4.2.5 JSON Input/Output

JSON input is as a string, therefore you will need to read the file first.

```
>>> json_string = '{"station": {"archive_id": "MT010"}}
>>> existing_station.metadata.from_json(json_string)
>>> print(existing_station.metadata.to_json(nested=True))
{
        "station": {
                "acquired_by": {
                        "author": null,
                        "comments": null
                },
                "archive_id": "FL001",
                "channel_layout": "X",
                "channels_recorded": [
                        "Hx",
                        "Hy",
                        "Hz",
                        "Ex",
                        "Ey"
                ],
                "comments": null,
                "data_type": "BB, LP",
                "geographic_name": "Beachy Keen, FL, USA",
                "hdf5_reference": "<HDF5 object reference>",
                "id": "MT010",
                "location": {
                        "latitude": 29.7203555,
                        "longitude": -83.4854715,
                        "elevation": 0.0,
                        "declination": {
```

(continues on next page)

(continued from previous page)


```
                                "comments": "Declination obtained from the instrument␣
↪GNSS NMEA sequence",
                                "model": "Unknown",
                                "value": -4.1
                        }
                },
                "mth5_type": "Station",
                "orientation": {
                        "method": "compass",
                        "reference_frame": "geographic"
                },
                "provenance": {
                        "creation_time": "2020-05-29T21:08:40+00:00",
                        "comments": null,
                        "log": null,
                        "software": {
                                "author": "Anna Kelbert, USGS",
                                "version": "2020-05-29",
                                "name": "mth5_metadata.m"
                        },
                        "submitter": {
                                "author": "Anna Kelbert, USGS",
                                "organization": "USGS Geomagnetism Program",
                                "email": "akelbert@usgs.gov"
                        }
                },
                "time_period": {
                        "end": "2015-01-29T16:18:14+00:00",
                        "start": "2015-01-08T19:49:15+00:00"
                }
        }
}
```


4.2.6 XML Input/Output

You can input as a XML element following the form previously mentioned. If you store your metadata in XML files you will need to read the and input the appropriate element into the metadata.

```
>>> from xml.etree import cElementTree as et
>>> root = et.Element('station')
>>> et.SubElement(root, 'archive_id', {'text': 'MT010'})
>>> existing_station.from_xml(root)
>>> print(existing_station.to_xml(string=True)
<?xml version="1.0" ?>
<station>
        <acquired_by>
                <author>None</author>
                <comments>None</comments>
        </acquired_by>
        <archive_id>MT010</archive_id>
        <channel_layout>X</channel_layout>
```

(continues on next page)

(continued from previous page)

```
        <channels_recorded>
                <item>Hx</item>
                <item>Hy</item>
                <item>Hz</item>
                <item>Ex</item>
                <item>Ey</item>
        </channels_recorded>
        <comments>None</comments>
        <data_type>BB, LP</data_type>
        <geographic_name>Beachy Keen, FL, USA</geographic_name>
        <hdf5_reference type="h5py_reference">&lt;HDF5 object reference&gt;</hdf5_
↪reference>
        <id>FL001</id>
        <location>
                <latitude type="float" units="degrees">29.7203555</latitude>
                <longitude type="float" units="degrees">-83.4854715</longitude>
                <elevation type="float" units="degrees">0.0</elevation>
                <declination>
                        <comments>Declination obtained from the instrument GNSS NMEA␣
↪sequence</comments>
                        <model>Unknown</model>
                        <value type="float" units="degrees">-4.1</value>
                </declination>
        </location>
        <mth5_type>Station</mth5_type>
        <orientation>
                <method>compass</method>
                <reference_frame>geographic</reference_frame>
        </orientation>
        <provenance>
                <creation_time>2020-05-29T21:08:40+00:00</creation_time>
                <comments>None</comments>
                <log>None</log>
                <software>
                        <author>Anna Kelbert, USGS</author>
                        <version>2020-05-29</version>
                        <name>mth5_metadata.m</name>
                </software>
                <submitter>
                        <author>Anna Kelbert, USGS</author>
                        <organization>USGS Geomagnetism Program</organization>
                        <email>akelbert@usgs.gov</email>
                </submitter>
        </provenance>
        <time_period>
                <end>2015-01-29T16:18:14+00:00</end>
                <start>2015-01-08T19:49:15+00:00</start>
        </time_period>
</station>
```

See also:

`mth5.metadata` for more information.

4.3 GOTCHAS

There are some gotchas or things you should understand when using HDF5 files as well as MTH5

4.3.1 Compression

Compression can slow down making a MTH5 file, so you should understand the compression parameters. See <https://pythonhosted.org/hdf5storage/compression.html#>`__ and <https://docs.h5py.org/en/stable/high/dataset.html>`__ for more information.

Compression is set in MTH5 when you instatiate an MTH5 object

```
>>> m = MTH5(shuffle=None, fletcher32=None, compression=None, compression_opts=None)
```

The compression parameters will be validated using *mth5.helpers.validate_compression*

Datasets can use chunks, which by default is set to True, which lets h5py pick the most efficient way to chunk the data.

Lossless compression filters

GZIP filter (`"gzip"`) Available with every installation of HDF5, so it's best where portability is required. Good compression, moderate speed. `compression_opts` sets the compression level and may be an integer from 0 to 9, default is 4.

LZF filter (`"lzf"`) Available with every installation of h5py (C source code also available). Low to moderate compression, very fast. No options.

SZIP filter (`"szip"`) Patent-encumbered filter used in the NASA community. Not available with all installations of HDF5 due to legal reasons. Consult the HDF5 docs for filter options.

4.3.2 Logging

Logging is great, but can have dramatic effects on performance, mainly because I'm new to logging and probably haven't written them most efficiently. By default the logging level is set to INFO. This seems to run as you might expect with slight overhead. If you change the logging level to DEBUG expect a slow down. You should only do this if you are a developer or are curious as to why something looks weird.

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/kujaku11/mth5/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

MTH5 could always use more documentation, whether as part of the official MTH5 docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/kujaku11/mth5/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *mth5* for local development.

1. Fork the *mth5* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/mth5.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv mth5
$ cd mth5/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 mth5 tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy. Check the Actions report to make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ pytest tests.test_mth5
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

GitActions will then deploy to PyPI if tests pass.

CHAPTER

SIX

CREDITS

This project is a joint effor between IRIS-PASSCAL and USGS.

Partial funding is provided by the Community for Data Integration at the U.S. Geological Survey.

6.1 Development Lead

- Jared Peacock <jpeacock@usgs.gov>

6.2 Contributors

- Karl Kappler
- Lindsey Heagy
- Tim Ronan

CHAPTER

SEVEN

HISTORY

7.1 0.1.0 (2021-06-30)

- First release on PyPI.

CHAPTER

EIGHT

MTH5 STRUCTURE

8.1 Metadata Validation

Metadata validation can be done in many different ways. After writing most of the code I learned about JSON and XML validation, but since I was naive to those I developed a structure that seems to work. That structure is this, all metadata key words have certain attributes that describe how the metadata value should be represented and validated. Those are:

- **type** - How the value should be represented based on very basic types
  - *string*
  - *number* (float or integer)
  - *boolean*
- **required** - A boolean (True or False) denoting whether the metadata key word required to represent the data.
- **style** - How the value should be represented within the type. For instance is the value a controlled string where there are only a few options, or is the value a controlled naming convention where only a 5 character alpha-numeric string is allowed. The styles are
  - *Alpha Numeric* a string with alphabetic and numberic characters
  - *Free Form* a free form string
  - *Controlled Vocabulary* only certain values are allowed according to **options**
  - *Date* a date and/or time string in ISO format
  - *Number* a float or integer
  - *Boolean* the value can only be True or False
- **units** - Units of the value
- **description** - Full description of what the metadata key is meant to convey.
- **options** - Any options of a **Controlled Vocabulary** style.

- **alias** - Any aliases that may represent the same metadata key.
- **example** - An example value to inform the user.

Each metadata key word has these attributes and are stored in CSV files under */mth5/standards/* in files for each basic and more complex levels. Therefore if the standards change or are updated these are the files that will need to be adjusted.

When MTH5 is initiated these CSV files are read in to `mth5.standards.Schema.Standards` as the basis to validate new values against. `mth5.standards.Schema.Standards` has property values that are `mth5.standards.Schema.BaseDict` objects for each of the basic and more complex metadata structures. The basic structures are metadata that describe a single entity, like *location* or *person* or *instrument*. These are then collected by more complex structures like **citation** or **survey** or **station**.

Each of the aforementioned metadata attributes are used in a validation function. For example **type** has is validated and if the input is not the specified type, the function will try to put it into standards type. If the **type** is *integer* and the value 10.9 is given, the output of the validator will be 10. Or if the **type** is a string and 10.9 is given the output will be "10.9".

8.2 Metadata Structure

Each metadata class is built on `mth5.metadata.Base` where the main chage is into the '_attr_dict' attribute which is a `mth5.standards.Schema.BaseDict`. The attributes of the metadata class are given, if the attribute is a basic metadata object that is specified. The purpose of this structure is that so a user can access attributes of the metadata in a Pythonic way. For example:

```
>>> from mth5 import metadata
>>> station = metadata.Station()
>>> station.location.latitude = 10.9
```

The basic representation of `mth5.metadata.Base` is a dictionary:

```
>>> from mth5 import metadata
>>> station = metadata.Station()
>>> station.location.latitude = 10.9
>>> station
{
        "station": {
                "acquired_by.author": null,
                "channels_recorded": [],
                "data_type": null,
                "geographic_name": null,
                "id": null,
                "location.declination.model": null,
                "location.declination.value": null,
                "location.elevation": 0.0,
                "location.latitude": 10.9,
                "location.longitude": 0.0,
                "orientation.method": null,
                "orientation.reference_frame": "geographic",
                "provenance.creation_time": "2020-10-23T00:32:04.780657+00:00",
                "provenance.software.author": null,
                "provenance.software.name": null,
                "provenance.software.version": null,
```

(continues on next page)

(continued from previous page)

```
                "provenance.submitter.author": null,
                "provenance.submitter.email": null,
                "provenance.submitter.organization": null,
                "time_period.end": "1980-01-01T00:00:00+00:00",
                "time_period.start": "1980-01-01T00:00:00+00:00"
        }
}
```

The metadata can be output as XML or JSON as well with functions *to_json* and *to_xml*. JSON can be structured such that each level is parsed out:

```
>>> print(s.to_json(nested=True))
{
        "station": {
                "acquired_by": {
                        "author": null
                },
                "channels_recorded": [],
                "data_type": null,
                "geographic_name": null,
                "id": null,
                "location": {
                        "latitude": 10.9,
                        "longitude": 0.0,
                        "elevation": 0.0,
                        "declination": {
                                "model": null,
                                "value": null
                        }
                },
                "orientation": {
                        "method": null,
                        "reference_frame": "geographic"
                },
                "provenance": {
                        "creation_time": "2020-10-23T00:32:04.780657+00:00",
                        "software": {
                                "author": null,
                                "version": null,
                                "name": null
                        },
                        "submitter": {
                                "author": null,
                                "organization": null,
                                "email": null
                        }
                },
                "time_period": {
                        "end": "1980-01-01T00:00:00+00:00",
                        "start": "1980-01-01T00:00:00+00:00"
                }
        }
```

(continues on next page)

(continued from previous page)

```
}
```

Similarly, the `mth5.metadata.Base` can read in an XML or JSON string. The benefit of this is that it is flexible to what people are more familiar with.

8.3 MTH5 Group Objects

Each group within an MTH5 file has a corresponding group object that provides convenience functions to interact with the HDF5 file and add, get, and remove groups or datasets to that group. For example a station has a corresponding *mth5.groups.StationGroup* object that provides functions to add, get, and remove a run. These are meant to make it easier for the user to interact with the HDF5 file without too much trouble.

Note: Each group contains a weak reference to the HDF5 group, this way when a file is closed there are no lingering references to a closed HDF5 file.

8.4 Time Series Objects

All datasets at the *Channel* level are represented by *mth5.timeseries.ChannelTS* objects. The *mth5.timeseries.ChannelTS* are based on `xarray.DataArray` objects. This way memory usage is minimal because xarray is lazy and only uses what is called for. Another benefit is that metadata can directly accompany the data. Currently the model is that all metadata are input into a `mth5.metadata.Base` object to be validated first and then the `xarray.DataArray` can be updated. This is not automated at this point so the user just needs to use the function update_xarray_metadata when metadata values are changed. Another advantage of using xarray is that the time series data are indexed by time making it easier to align, trim, extract, sort, etc.

All run datasets are represented by *mth5.timeseries.RunTS* objects, which are based on `xarray.DataSet` which is a collection of `xarray.DataArray` objects. The benefits of using xarray are that many of the methods such as aligning, indexing, sorting are already developed and are robust. Therefore the useability is easier without more coding.

Another reason why xarray was picked as the basis for representing the data is that it works seamlessly with other programs like Dask for parallel computing, and plotting tools like hvplot.

CHAPTER

NINE

STATIONS

Stations are the top level for an MT sounding. There are 2 station containers `mth5.groups.MasterStationsGroup` and *mth5.groups.StationGroup*.

9.1 Master Stations Group

`mth5.groups.MasterStationsGroup` is an umbrella container that holds a collection of *mth5.groups.StationGroup* objects and contains a summary table that summarizes all stations within the survey. Use `mth5.groups.MasterStationsGroup` to add/get/remove stations.

No metadata currently accompanies `mth5.groups.MasterStationsGroup`. There will soon be a list of *mth5.groups.StationGroup* objects for all stations.

There are 2 ways to add/remove/get stations. Add/get will return a *mth5.groups.StationGroup*. If adding a station that has the same name as an existing station the *mth5.groups.StationGroup* returned will be of the existing station and no station will be added. Change the name or update the existing staiton. If getting a station that does not exist a *mth5.utils.exceptions.MTH5Error* will be raised.

9.1.1 1) Using *stations_group*

The first way to add/get/remove stations is from the **:attribute:`mth5.MTH5.stations_group`** which is a `mth5.groups.MasterStationsGroup` object.

```
>>> stations = mth5_obj.stations_group
>>> type(stations)
mth5.groups.MasterStationsGroup
>>> stations
/Survey/Stations:
```

(continues on next page)

(continued from previous page)

```
====================
        |- Group: MT001
        ---------------
                |- Group: MT001a
                ----------------
                        --> Dataset: Ex
                        ..................
                        --> Dataset: Ey
                        ..................
                        --> Dataset: Hx
                        ..................
                        --> Dataset: Hy
                        ..................
                        --> Dataset: Hz
                        ..................
                        --> Dataset: Summary
                        ......................
```

From the *stations_group* you can add/remove/get a station.

To add a station:

```
>>> new_station = stations.add_station('MT002')
>>> print(type(new_station))
mth5.groups.StationGroup
>>> new_station
/Survey/Stations/MT002:
====================
--> Dataset: Summary
......................
```

To get an existing station:

```
>>> existing_station = stations.get_station('MT001')
```

To remove an existing station:

```
>>> stations.remove_station('MT002')
>>> stations.group_list
['Summary', 'MT001']
```

9.1.2 2) Using Covnenience methods

The second way to add/remove/get stations is from the convenience functions in `mth5.MTH5`. These use the same methods as the `mth5.groups.MasterStationsGroup` but can be accessed directly.

To add a station:

```
>>> new_station = mth5_obj.add_station('MT002')
>>> mth5_obj
/:
====================
```

(continues on next page)

(continued from previous page)

```
        |- Group: Survey
        ----------------
                |- Group: Filters
                -----------------
                        --> Dataset: Summary
                        ......................
                |- Group: Reports
                -----------------
                        --> Dataset: Summary
                        ......................
                |- Group: Standards
                -------------------
                        --> Dataset: Summary
                        ......................
                |- Group: Stations
                ------------------
                        |- Group: MT001
                        ---------------
                                --> Dataset: Summary
                                ......................
                        |- Group: MT002
                        ---------------
                                --> Dataset: Summary
                                ......................
                        --> Dataset: Summary
                        ......................
```

To get an existing station:

```
>>> existing_station = mth5_obj.get_station('MT002')
```

To remove an existing station:

```
>>> mth5_obj.remove_station('MT002')
```

9.1.3 Summary Table

Column	Description
archive_id	Station archive name
start	Start time of the station (ISO format)
end	End time of the station (ISO format)
components	All components measured by the station
measurement_type	All measurement types collected by the station
location.latitude	Station latitude (decimal degrees)
location.longitude	Station longitude (decimal degrees)
location.elevation	Station elevation (meters)
hdf5_reference	Internal HDF5 reference

9.2 Station Group

A single station is contained within a *mth5.groups.StationGroup* object, which has the appropriate metadata for a single station. *mth5.groups.StationGroup* contains all the runs for that station.

9.2.1 Summary Table

The summary table in *mth5.groups.StationGroup* summarizes all runs for that station.

Column	Description
id	Run ID
start	Start time of the run (ISO format)
end	End time of the run (ISO format)
components	All components measured for that run
measurement_type	Type of measurement for that run
sample_rate	Sample rate of the run (samples/second)
hdf5_reference	Internal HDF5 reference

9.2.2 Metadata

Metadata is accessed through the *metadata* property, which is a `mth5.metadata.Station` object.

```
>>> type(new_station.metadata)
mth5.metadata.Station
>>> new_station.metadata
{
        "station": {
                "acquired_by.author": null,
                "acquired_by.comments": null,
                "archive_id": "FL001",
                "channel_layout": "X",
                "channels_recorded": [
                        "Hx",
                        "Hy",
                        "Hz",
                        "Ex",
                        "Ey"
                ],
                "comments": null,
                "data_type": "BB, LP",
                "geographic_name": "Beachy Keen, FL, USA",
                "hdf5_reference": "<HDF5 object reference>",
                "id": "FL001",
                "location.declination.comments": "Declination obtained from the␣
↪instrument GNSS NMEA sequence",
                "location.declination.model": "Unknown",
                "location.declination.value": -4.1,
                "location.elevation": 0.0,
                "location.latitude": 29.7203555,
                "location.longitude": -83.4854715,
```

(continues on next page)

(continued from previous page)


```
                "mth5_type": "Station",
                "orientation.method": "compass",
                "orientation.reference_frame": "geographic",
                "provenance.comments": null,
                "provenance.creation_time": "2020-05-29T21:08:40+00:00",
                "provenance.log": null,
                "provenance.software.author": "Anna Kelbert, USGS",
                "provenance.software.name": "mth5_metadata.m",
                "provenance.software.version": "2020-05-29",
                "provenance.submitter.author": "Anna Kelbert, USGS",
                "provenance.submitter.email": "akelbert@usgs.gov",
                "provenance.submitter.organization": "USGS Geomagnetism Program",
                "time_period.end": "2015-01-29T16:18:14+00:00",
                "time_period.start": "2015-01-08T19:49:15+00:00"
        }
}
```


See also:

mth5.groups.StationGroup

CHAPTER

TEN

RUNS

A run is a collection of channels that recorded at similar start and end times at the same sample rate for a given station. A run is contained within a *mth5.groups.RunGroup* object. A run is the next level down from a station.

The main way to add/remove/get a run object is through a *mth5.groups.StationGroup* object

10.1 Accessing through StationGroup

You can get a *mth5.groups.StationGroup* using either method in the previous section.

```
>>> new_station = mth5_obj.add_station('MT003')
```

or

```
>>> new_station = mth5_obj.stations_group.add_station('MT003')
```

10.1.1 Add Run

```
>>> # if you don't already have a run name one can be assigned based on existing runs
>>> new_run_name = new_station.make_run_name()
>>> new_run = new_station.add_run(new_run_name)
```

Or

```
>>> new_run = mth5_obj.add_run('MT003', 'MT003a')
```

10.1.2 Get Run

Similar methods for get/remove a run

```
>>> existing_run = new_station.get_run('MT003a')
```

or

```
>>> existing_run = mth5_obj.get_run('MT003', 'MT003a')
```

10.1.3 Remove Run

```
>>> new_station.remove_run('MT003a')
```

or

```
>>> mth5_obj.remove_run('MT003', 'MT003a')
```

10.2 Summary Table

The summary table summarizes all channels for that run.

Column	Description
component	Component name
start	Start time of the channel (ISO format)
end	End time of the channel (ISO format0
n_samples	Number of samples for the channel
measurement_type	Measuremnt type of the channel
units	Units of the channel data
hdf5_reference	HDF5 internal reference

10.3 Metadata

Metadata is accessed through the *metadata* property, which is a `mth5.metadata.Run` object.

```
>>> type(new_run)
mth5.metadata.Run
>>> new_run.metadata
{
        "run": {
                "acquired_by.author": "BB",
                "acquired_by.comments": "it's cold in florida",
                "channels_recorded_auxiliary": null,
                "channels_recorded_electric": null,
                "channels_recorded_magnetic": null,
                "comments": null,
                "data_logger.firmware.author": "Barry Narod",
                "data_logger.firmware.name": null,
```

(continues on next page)

(continued from previous page)


```
                "data_logger.firmware.version": null,
                "data_logger.id": "1305-1",
                "data_logger.manufacturer": "Barry Narod",
                "data_logger.model": "NIMS",
                "data_logger.power_source.comments": "voltage measurements not recorded",
                "data_logger.power_source.id": null,
                "data_logger.power_source.type": "battery",
                "data_logger.power_source.voltage.end": null,
                "data_logger.power_source.voltage.start": null,
                "data_logger.timing_system.comments": null,
                "data_logger.timing_system.drift": 0.0,
                "data_logger.timing_system.type": "GPS",
                "data_logger.timing_system.uncertainty": 1.0,
                "data_logger.type": null,
                "data_type": "BB, LP",
                "hdf5_reference": "<HDF5 object reference>",
                "id": "MT003a",
                "metadata_by.author": "Anna Kelbert; Paul Bedrosian",
                "metadata_by.comments": "Paul Bedrosian: Ey, electrode dug up",
                "mth5_type": "Run",
                "provenance.comments": null,
                "provenance.log": null,
                "sample_rate": 8.0,
                "time_period.end": "2015-01-19T14:54:54+00:00",
                "time_period.start": "2015-01-08T19:49:15+00:00"
        }
}
```


See also:

mth5.groups.RunGroup and `mth5.metadata.Run` for more information.

CHAPTER

ELEVEN

FILE READERS

11.1 Basics

The file readers have been setup to be like plugins, well hacked so it is setup like plugins. Further work needs to be done to fully set it up as Python plugins, but for now it works.

There is a generic reader that is loaded when MTH5 is imported called *read_file*. It will pick the correct reader based on the file extension or if the extension is ambiguous the user can input the specific file type.

```
>>> import mth5
>>> run_obj = mth5.read_file(r"/home/mt_data/mt001.bin")
>>> run_obj = mth5.read_file(r"/home/mt_data/mt001.bin", file_type='nims')
```

This will currently read in the following file types:

File Structure	MTH5 Key	File Types	Returns
NIMS	nims	[.bin, .bnn]	RunTS
Zonge Z3D	zen	[.z3d]	ChannelTS
USGS ASCII	usgs_ascii	[.asc, .gzip]	RunTS

NIMS and USGS ASCII will return a *mth5.timeseries.RunTS* object and Zonge Z3D returns a *mth5.timeseries.ChannelTS* object. The return type depends on the structure of the file. NIMS records each channel in a single block of data, so all channels are in a single file. Whereas, Z3D files are for a single channel. It might make sense in to return the same data type, but for now this is the way it is. Also returned are any extra metadata that might not belong to a channel or run. Specifically, most files have information about location and some other metadata about the station that could be helpful in filling out metadata for the station.

11.1.1 Adding Plugins

Everyone has their own file structure and therefore there will need to be various readers for the different data formats. If you have a data format that isn't supported adding a reader would be a welcomed contribution. To keep things somewhat uniform here are some guidelines to add a reader.

11.1.2 Reader Structure

The reader should be setup with having a class that contains the metadata which is inherited to a class that holds the data. This makes things a little easier to separate and read. It helps if the metadata has similar names as the standards but don't have to be it just means you have to do some translation.

It helps if you have properties, if attributes are not appropriate, for important information that is passed onto *mth5.timeseries.ChannelTS* or *mth5.timeseries.RunTS*.

```
from mth5.timeseries import ChannelTS, RunTS

class MyFileMetadata:
        """ Read in metadata into appropriate objects """
        def __init__(self, fn):
                self.fn = fn
                self.start = None
                self.end = None
                self.sample_rate = None

        def read_metadata():
                """ function to read in metadata and fill attribute values """
                pass

class MyFile(MyFileMetadata):
        """ inheret metadata and read data """
        def __init__(self, fn):
                self.fn = fn
                self.data = None

                super().__init__()

        @property
        def station_metadata(self):
                """ Any station metadata within the file """

                station_meta_dict = {}
                station_meta_dict['location.latitude'] = self.latitude

                return {'Station': station_meta_dict}

        @property
        def run_metadata(self):
                """ Any run metadata within the file """

                run_meta_dict = {}
                run_meta_dict['id'] = f"{self.station}a"
```

(continues on next page)

(continued from previous page)

```
                return {'Run': run_meta_dict}

        @property
        def channel_metadata(self):
                """ channel metadata filled from information in the file """
                channel_meta_dict = {}
                channel_meta_dict['time_period.start'] = self.start
                channel_meta_dict['time_period.end'] = self.end
                channel_meta_dict['sample_rate'] = self.sample_rate

                return {'Electric': channel_meta_dict}


        @property
        def ex(self):
                """ ex convenience property """
                # if a pandas dataframe or numpy structured array
                return timeseries.ChannelTS('electric',
                                                        data=self.data['ex'],
                                                        channel_metadata=self.channel_
↪metadata,
                                                        station_metadata=self.station_
↪metadata,
                                                        run_metadata=self.run_
↪metadata)

        def read_my_file(self):
                """ read in data """
                # suggest reading into a data type like numpy, pandas, xarray
                # xarray is the main object used for time series data in mth5
                return RunTS([self.ex, self.ey, self.hx, self.hy, self.hx])


def read_my_file(fn):
        """ the helper function to read the file """
        new_obj = MyFile(fn)
        return new_obj.read_my_file()
```

See also:

mth5.io.zen and *mth5.io.nims* for working examples.

Once you have come up a reader you can add it to the reader module. You just need to add a file name and associated file types.

In the dictionary in mth5.reader 'readers' add a line like:

```
"my_file": {"file_types": ["dat", "data"], "reader": my_file.read_my_file},
```

Then you can see if your reader works

```
>>> import mth5
>>> run = mth5.read_file(r"/home/mt_data/test.dat", file_type='my_file')
```

CHAPTER

TWELVE

CONVENTIONS

Some conventions that have been implemented:

- All channel names are lower case.

CHAPTER

THIRTEEN

MT METADATA

See mt_metadata documentation for more information on magnetotelluric time series and transfer function metadata standards.

CHAPTER

FOURTEEN

MTH5 PACKAGE

14.1 Subpackages

14.1.1 mth5.groups package

Subpackages

mth5.groups.filter_groups package

Submodules

mth5.groups.filter_groups.coefficient_filter_group module

Created on Wed Jun 9 08:58:15 2021

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.filter_groups.coefficient_filter_group.**CoefficientGroup**(*group*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

Container for Coefficient type filters

add_filter(*name*, *coefficient_metadata*)

Add a coefficient Filter

Parameters

- **name** (*TYPE*) – DESCRIPTION
- **coefficient_metadata** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

property filter_dict

Dictionary of available coefficient filters

Returns DESCRIPTION

Return type TYPE

from_object(*coefficient_object*)

make a filter from a mt_metadata.timeseries.filters.CoefficientFilter

Parameters **zpk_object** (mt_metadata.timeseries.filters.CoefficientFilter) – MT metadata Coefficient Filter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the ZPK filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.CoefficientFilter object

Returns DESCRIPTION

Return type TYPE

mth5.groups.filter_groups.fap_filter_group module

Created on Wed Jun 9 08:55:16 2021

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.filter_groups.fap_filter_group.**FAPGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for fap type filters

add_filter(*name*, *frequency*, *amplitude*, *phase*, *fap_metadata*)
create an HDF5 group/dataset from information given.

Parameters

- **name** (*string*) – Nane of the filter
- **poles** (*np.ndarray(dtype=complex)*) – poles of the filter as complex numbers
- **zeros** (*np.ndarray(dtype=comples)*) – zeros of the filter as complex numbers
- **fap_metadata** – metadata dictionary see

mt_metadata.timeseries.filters.PoleZeroFilter for details on entries :type fap_metadata: dictionary

property filter_dict
Dictionary of available fap filters

Returns DESCRIPTION

Return type TYPE

from_object(*fap_object*)
make a filter from a mt_metadata.timeseries.filters.PoleZeroFilter

Parameters **fap_object** (mt_metadata.timeseries.filters.PoleZeroFilter) – MT metadata PoleZeroFilter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the fap filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.pole_zeros_filter object :return: DESCRIPTION :rtype: TYPE

mth5.groups.filter_groups.fir_filter_group module

Created on Wed Jun 9 08:55:16 2021

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.filter_groups.fir_filter_group.**FIRGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for fir type filters

add_filter(*name*, *coefficients*, *fir_metadata*)
create an HDF5 group/dataset from information given.

Parameters

- **name** (*string*) – Nane of the filter
- **poles** (*np.ndarray(dtype=complex)*) – poles of the filter as complex numbers
- **zeros** (*np.ndarray(dtype=comples)*) – zeros of the filter as complex numbers
- **fir_metadata** – metadata dictionary see

mt_metadata.timeseries.filters.PoleZeroFilter for details on entries :type fir_metadata: dictionary

property filter_dict
Dictionary of available fir filters

Returns DESCRIPTION

Return type TYPE

from_object(*fir_object*)
make a filter from a mt_metadata.timeseries.filters.PoleZeroFilter

Parameters **fir_object** (mt_metadata.timeseries.filters.PoleZeroFilter) – MT metadata PoleZeroFilter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the fir filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.pole_zeros_filter object :return: DESCRIPTION :rtype: TYPE

mth5.groups.filter_groups.time_delay_filter_group module

Created on Wed Jun 9 09:01:55 2021

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.filter_groups.time_delay_filter_group.**TimeDelayGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for time_delay type filters

add_filter(*name*, *time_delay_metadata*)
create an HDF5 group/dataset from information given.

Parameters

- **name** (*string*) – Nane of the filter
- **poles** (*np.ndarray(dtype=complex)*) – poles of the filter as complex numbers
- **zeros** (*np.ndarray(dtype=comples)*) – zeros of the filter as complex numbers
- **time_delay_metadata** – metadata dictionary see

mt_metadata.timeseries.filters.PoleZeroFilter for details on entries :type time_delay_metadata: dictionary

property filter_dict
Dictionary of available time_delay filters

Returns DESCRIPTION

Return type TYPE

from_object(*time_delay_object*)
make a filter from a mt_metadata.timeseries.filters.PoleZeroFilter

Parameters **time_delay_object** (mt_metadata.timeseries.filters.PoleZeroFilter) – MT metadata PoleZeroFilter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the time_delay filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.pole_zeros_filter object :return: DESCRIPTION :rtype: TYPE

mth5.groups.filter_groups.zpk_filter_group module

Created on Wed Jun 9 08:55:16 2021

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.filter_groups.zpk_filter_group.**ZPKGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for ZPK type filters

add_filter(*name*, *poles*, *zeros*, *zpk_metadata*)
create an HDF5 group/dataset from information given.

Parameters

- **name** (*string*) – Nane of the filter
- **poles** (*np.ndarray(dtype=complex)*) – poles of the filter as complex numbers
- **zeros** (*np.ndarray(dtype=comples)*) – zeros of the filter as complex numbers
- **zpk_metadata** – metadata dictionary see

mt_metadata.timeseries.filters.PoleZeroFilter for details on entries :type zpk_metadata: dictionary

property filter_dict
Dictionary of available ZPK filters

Returns DESCRIPTION

Return type TYPE

from_object(*zpk_object*)
make a filter from a mt_metadata.timeseries.filters.PoleZeroFilter

Parameters **zpk_object** (mt_metadata.timeseries.filters.PoleZeroFilter) – MT metadata PoleZeroFilter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the ZPK filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.pole_zeros_filter object :return: DESCRIPTION :rtype: TYPE

Module contents

Import all Group objects

class mth5.groups.filter_groups.**CoefficientGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for Coefficient type filters

add_filter(*name*, *coefficient_metadata*)
Add a coefficient Filter

Parameters

- **name** (*TYPE*) – DESCRIPTION
- **coefficient_metadata** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

property filter_dict
Dictionary of available coefficient filters

Returns DESCRIPTION

Return type TYPE

from_object(*coefficient_object*)
make a filter from a `mt_metadata.timeseries.filters.CoefficientFilter`

Parameters **zpk_object** (`mt_metadata.timeseries.filters.CoefficientFilter`) – MT metadata Coefficient Filter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the ZPK filter

remove_filter()

to_object(*name*)
make a `mt_metadata.timeseries.filters.CoefficientFilter` object

Returns DESCRIPTION

Return type TYPE

class mth5.groups.filter_groups.**FAPGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for fap type filters

add_filter(*name*, *frequency*, *amplitude*, *phase*, *fap_metadata*)
create an HDF5 group/dataset from information given.

Parameters

- **name** (*string*) – Nane of the filter
- **poles** (*np.ndarray(dtype=complex)*) – poles of the filter as complex numbers
- **zeros** (*np.ndarray(dtype=comples)*) – zeros of the filter as complex numbers
- **fap_metadata** – metadata dictionary see

mt_metadata.timeseries.filters.PoleZeroFilter for details on entries :type fap_metadata: dictionary

property filter_dict
Dictionary of available fap filters

Returns DESCRIPTION

Return type TYPE

from_object(*fap_object*)
make a filter from a mt_metadata.timeseries.filters.PoleZeroFilter

Parameters **fap_object** (mt_metadata.timeseries.filters.PoleZeroFilter) – MT metadata PoleZeroFilter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the fap filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.pole_zeros_filter object :return: DESCRIPTION :rtype: TYPE

class mth5.groups.filter_groups.**FIRGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for fir type filters

add_filter(*name*, *coefficients*, *fir_metadata*)
create an HDF5 group/dataset from information given.

Parameters

- **name** (*string*) – Nane of the filter
- **poles** (*np.ndarray(dtype=complex)*) – poles of the filter as complex numbers
- **zeros** (*np.ndarray(dtype=comples)*) – zeros of the filter as complex numbers
- **fir_metadata** – metadata dictionary see

mt_metadata.timeseries.filters.PoleZeroFilter for details on entries :type fir_metadata: dictionary

property filter_dict
Dictionary of available fir filters

Returns DESCRIPTION

Return type TYPE

from_object(*fir_object*)
make a filter from a mt_metadata.timeseries.filters.PoleZeroFilter

Parameters **fir_object** (mt_metadata.timeseries.filters.PoleZeroFilter) – MT metadata PoleZeroFilter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the fir filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.pole_zeros_filter object :return: DESCRIPTION :rtype: TYPE

class mth5.groups.filter_groups.**TimeDelayGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for time_delay type filters

add_filter(*name*, *time_delay_metadata*)
create an HDF5 group/dataset from information given.

Parameters

- **name** (*string*) – Nane of the filter
- **poles** (*np.ndarray(dtype=complex)*) – poles of the filter as complex numbers
- **zeros** (*np.ndarray(dtype=comples)*) – zeros of the filter as complex numbers
- **time_delay_metadata** – metadata dictionary see

mt_metadata.timeseries.filters.PoleZeroFilter for details on entries :type time_delay_metadata: dictionary

property filter_dict
Dictionary of available time_delay filters

Returns DESCRIPTION

Return type TYPE

from_object(*time_delay_object*)
make a filter from a mt_metadata.timeseries.filters.PoleZeroFilter

Parameters **time_delay_object** (mt_metadata.timeseries.filters.PoleZeroFilter) – MT metadata PoleZeroFilter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the time_delay filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.pole_zeros_filter object :return: DESCRIPTION :rtype: TYPE

class mth5.groups.filter_groups.**ZPKGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Container for ZPK type filters

add_filter(*name*, *poles*, *zeros*, *zpk_metadata*)
create an HDF5 group/dataset from information given.

Parameters

- **name** (*string*) – Nane of the filter

- **poles** (*np.ndarray(dtype=complex)*) – poles of the filter as complex numbers
- **zeros** (*np.ndarray(dtype=comples)*) – zeros of the filter as complex numbers
- **zpk_metadata** – metadata dictionary see

mt_metadata.timeseries.filters.PoleZeroFilter for details on entries :type zpk_metadata: dictionary

property filter_dict
Dictionary of available ZPK filters

Returns DESCRIPTION

Return type TYPE

from_object(*zpk_object*)
make a filter from a mt_metadata.timeseries.filters.PoleZeroFilter

Parameters **zpk_object** (mt_metadata.timeseries.filters.PoleZeroFilter) – MT metadata PoleZeroFilter

get_filter(*name*)
Get a filter from the name

Parameters **name** (*string*) – name of the filter

Returns HDF5 group of the ZPK filter

remove_filter()

to_object(*name*)
make a mt_metadata.timeseries.filters.pole_zeros_filter object :return: DESCRIPTION :rtype: TYPE

Submodules

mth5.groups.base module

Base Group Class

Contains all the base functions that will be used by group classes.

Created on Fri May 29 15:09:48 2020

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.base.**BaseGroup**(*group*, *group_metadata=None*, ***kwargs*)
Bases: object

Generic object that will have functionality for reading/writing groups, including attributes. To access the hdf5 group directly use the *BaseGroup.hdf5_group* property.

```
>>> base = BaseGroup(hdf5_group)
>>> base.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *BaseGroup.write_metadata* method. This is a temporary solution working on an automatic updater if metadata is changed.

```
>>> base.metadata.existing_attribute = 'update_existing_attribute'
>>> base.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> base.metadata.add_base_attribute('new_attribute',
...                                  'new_attribute_value',
...                                  {'type':str,
...                                   'required':True,
...                                   'style':'free form',
...                                   'description': 'new attribute desc.',
...                                   'units':None,
...                                   'options':[],
...                                   'alias':[],
...                                   'example':'new attribute'})
```

Includes intializing functions that makes a summary table and writes metadata.

property dataset_options

property groups_list

initialize_group(***kwargs*)
Initialize group by making a summary table and writing metadata

property metadata
Metadata for the Group based on mt_metadata.timeseries

read_metadata()
read metadata from the HDF5 group into metadata object

write_metadata()
Write HDF5 metadata from metadata object.

mth5.groups.filters module

Created on Wed Dec 23 17:08:40 2020

Need to make a group for FAP and FIR filters.

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.filters.**FiltersGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Not implemented yet

add_filter(*filter_object*)
Add a filter dataset based on type

current types are:

- zpk –> zeros, poles, gain

- fap –> frequency look up table
- time_delay –> time delay filter
- coefficient –> coefficient filter

Parameters **filter_object** (mt_metadata.timeseries.filters) – An MT metadata filter object

property filter_dict

get_filter(*name*)
Get a filter by name

to_filter_object(*name*)
return the MT metadata representation of the filter

mth5.groups.master_station_run_channel module

Created on Wed Dec 23 17:18:29 2020

Note: Need to keep these groups together, if you split them into files you

get a circular import.

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.master_station_run_channel.**AuxiliaryDataset**(*group*, ***kwargs*)
Bases: *mth5.groups.master_station_run_channel.ChannelDataset*

Holds a channel dataset. This is a simple container for the data to make sure that the user has the flexibility to turn the channel into an object they want to deal with.

For now all the numpy type slicing can be used on *hdf5_dataset*

Parameters

- **dataset** (h5py.Dataset) – dataset object for the channel
- **dataset_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If the dataset is not of the correct type

Utilities will be written to create some common objects like:

- xarray.DataArray
- pandas.DataFrame
- zarr
- dask.Array

The benefit of these other objects is that they can be indexed by time, and they have much more buit-in funcionality.

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> channel = run.get_channel('Ex')
>>> channel
Channel Electric:
-------------------
            component:        Ey
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:00:01+00:00
    sample rate:      4096
```

class mth5.groups.master_station_run_channel.**ChannelDataset**(*dataset*, *dataset_metadata=None*, ***kwargs*)

Bases: `object`

Holds a channel dataset. This is a simple container for the data to make sure that the user has the flexibility to turn the channel into an object they want to deal with.

For now all the numpy type slicing can be used on *hdf5_dataset*

Parameters

- **dataset** (`h5py.Dataset`) – dataset object for the channel
- **dataset_metadata** ([`mth5.metadata.Electric` | `mth5.metadata.Magnetic` | `mth5.metadata.Auxiliary`], optional) – metadata container, defaults to None

Raises *MTH5Error* – If the dataset is not of the correct type

Utilities will be written to create some common objects like:

- xarray.DataArray
- pandas.DataFrame
- zarr
- dask.Array

The benefit of these other objects is that they can be indexed by time, and they have much more buit-in funcionality.

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> channel = run.get_channel('Ex')
>>> channel
Channel Electric:
-------------------
            component:        Ey
    data type:        electric
    data format:      float32
    data shape:       (4096,)
```

(continues on next page)

(continued from previous page)

```
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:00:01+00:00
    sample rate:      4096
```

property channel_entry
channel entry that will go into a full channel summary of the entire survey

property channel_response_filter

property end
return end time based on the data

extend_dataset(*new_data_array*, *start_time*, *sample_rate*, *fill=None*, *max_gap_seconds=1*, *fill_window=10*)
Append data according to how the start time aligns with existing data. If the start time is before existing start time the data is prepended, similarly if the start time is near the end data will be appended.

If the start time is within the existing time range, existing data will be replace with the new data.

If there is a gap between start or end time of the new data with the existing data you can either fill the data with a constant value or an error will be raise depending on the value of fill.

Parameters

- **new_data_array** (numpy.ndarray) – new data array with shape (npts,)
- **start_time** (string or mth5.utils.mttime.MTime) – start time of the new data array in UTC
- **sample_rate** (*float*) – Sample rate of the new data array, must match existing sample rate
- **fill** (*string, None, float, integer*) – If there is a data gap how do you want to fill the gap * None: will raise an *mth5.utils.exceptions.MTH5Error* * 'mean': will fill with the mean of each data set within the fill window * 'median': will fill with the median of each data set within the fill window * value: can be an integer or float to fill the gap * 'nan': will fill the gap with NaN
- **max_gap_seconds** (*float or integer*) – sets a maximum number of seconds the gap can be. Anything over this number will raise a *mth5.utils.exceptions.MTH5Error*.
- **fill_window** (*integer*) – number of points from the end of each data set to estimate fill value from.

Raises mth5.utils.excptions.MTH5Error if sample rate is not the same, or fill value is not understood,

Rubric

```
>>> ex = mth5_obj.get_channel('MT001', 'MT001a', 'Ex')
>>> ex.n_samples
4096
>>> ex.end
2015-01-08T19:32:09.500000+00:00
>>> t = timeseries.ChannelTS('electric',
...                          data=2*np.cos(4 * np.pi * .05 *          ...           ␣
↪                          np.linspace(0,4096l num=4096) *
...                                      .01),
```

(continues on next page)

(continued from previous page)

```
...                     channel_metadata={'electric':{
...                        'component': 'ex',
...                        'sample_rate': 8,
...                        'time_period.start':(ex.end+(1)).iso_str}})
>>> ex.extend_dataset(t.ts, t.start, t.sample_rate, fill='median',
...                   max_gap_seconds=2)
2020-07-02T18:02:47 - mth5.groups.Electric.extend_dataset - INFO -
filling data gap with 1.0385180759767025
>>> ex.n_samples
8200
>>> ex.end
2015-01-08T19:40:42.500000+00:00
```

from_channel_ts(*channel_ts_obj*, *how='replace'*, *fill=None*, *max_gap_seconds=1*, *fill_window=10*)
fill data set from a *mth5.timeseries.ChannelTS* object.

Will check for time alignement, and metadata.

Parameters

- **channel_ts_obj** (*mth5.timeseries.ChannelTS*) – time series object
- **how** – how the new array will be input to the existing dataset:
  - 'replace' -> replace the entire dataset nothing is left over.
  - 'extend' -> add onto the existing dataset, any overlapping values will be rewritten, if there are gaps between data sets those will be handled depending on the value of fill.

    param fill If there is a data gap how do you want to fill the gap:
    - None -> will raise an *mth5.utils.exceptions.MTH5Error*
    - 'mean'-> will fill with the mean of each data set within the fill window
    - 'median' -> will fill with the median of each data set within the fill window
    - value -> can be an integer or float to fill the gap
    - 'nan' -> will fill the gap with NaN
- **max_gap_seconds** (*float or integer*) – sets a maximum number of seconds the gap can be. Anything over this number will raise a *mth5.utils.exceptions.MTH5Error*.
- **fill_window** (*integer*) – number of points from the end of each data set to estimate fill value from.

from_xarray(*data_array*, *how='replace'*, *fill=None*, *max_gap_seconds=1*, *fill_window=10*)
fill data set from a `xarray.DataArray` object.

Will check for time alignement, and metadata.

Parameters

- **data_array_obj** – Xarray data array
- **how** – how the new array will be input to the existing dataset:
  - 'replace' -> replace the entire dataset nothing is left over.

– 'extend' -> add onto the existing dataset, any overlapping

values will be rewritten, if there are gaps between data sets those will be handled depending on the value of fill.

param fill If there is a data gap how do you want to fill the gap:

– None -> will raise an *mth5.utils.exceptions.MTH5Error*

– **'mean'-> will fill with the mean of each data set within** the fill window

– **'median' -> will fill with the median of each data set** within the fill window

– value -> can be an integer or float to fill the gap

– 'nan' -> will fill the gap with NaN

- **max_gap_seconds** (*float or integer*) – sets a maximum number of seconds the gap can be. Anything over this number will raise a *mth5.utils.exceptions.MTH5Error*.
- **fill_window** (*integer*) – number of points from the end of each data set to estimate fill value from.

get_index_from_time(*given_time*)
get the appropriate index for a given time.
Parameters **given_time** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

property master_station_group
shortcut to master station group

property n_samples

read_metadata()
Read metadata from the HDF5 file into the metadata container, that way it can be validated.

replace_dataset(*new_data_array*)
replace the entire dataset with a new one, nothing left behind
Parameters **new_data_array** (numpy.ndarray) – new data array shape (npts,)

property run_group
shortcut to run group

property sample_rate

property start

property station_group
shortcut to station group

property table_entry
Creat a table entry to put into the run summary table.

property time_index
time index given parameters in metadata :rtype: pandas.DatetimeIndex
Type return

time_slice(*start_time*, *end_time=None*, *n_samples=None*, *return_type='channel_ts'*)

Get a time slice from the channel and return the appropriate type

- numpy array with metadata
- pandas.Dataframe with metadata
- xarray.DataFrame with metadata
- *mth5.timeseries.ChannelTS* 'default'
- dask.DataFrame with metadata 'not yet'

Parameters

- **start_time** (string or `mth5.utils.mttime.MTime`) – start time of the slice
- **end_time** (string or `mth5.utils.mttime.MTime`, optional) – end time of the slice
- **n_samples** (*integer, optional*) – number of samples to read in

Returns the correct container for the time series.

Return type [`xarray.DataArray` | `pandas.DataFrame` | *mth5.timeseries.ChannelTS* | `numpy.ndarray`]

Raises ValueError if both end_time and n_samples are None or given.

Example with number of samples

```
>>> ex = mth5_obj.get_channel('FL001', 'FL001a', 'Ex')
>>> ex_slice = ex.time_slice("2015-01-08T19:49:15", n_samples=4096)
>>> ex_slice
<xarray.DataArray (time: 4096)>
array([0.93115046, 0.14233688, 0.87917119, ..., 0.26073634, 0.7137319 ,
       0.88154395])
Coordinates:
  * time     (time) datetime64[ns] 2015-01-08T19:49:15 ... 2015-01-08T19:57:46.
↪875000
Attributes:
    ac.end:                      None
    ac.start:                    None
    ...

>>> type(ex_slice)
mth5.timeseries.ChannelTS

# plot the time series
>>> ex_slice.ts.plot()
```

Example with start and end time

```
>>> ex_slice = ex.time_slice("2015-01-08T19:49:15",
...                          end_time="2015-01-09T19:49:15")
```

Raises Example

```
>>> ex_slice = ex.time_slice("2015-01-08T19:49:15",
...                          end_time="2015-01-09T19:49:15",
...                          n_samples=4096)
ValueError: Must input either end_time or n_samples, not both.
```

to_channel_ts()

Returns a Timeseries with the appropriate time index and metadata

Return type *mth5.timeseries.ChannelTS*

loads from memory (nearly half the size of xarray alone, not sure why)

to_dataframe()

Returns a dataframe where data is stored in the 'data' column and attributes are stored in the experimental attrs attribute

Return type pandas.DataFrame

Note: that metadta will not be validated if changed in an xarray.

loads into RAM

to_numpy()

Returns a numpy structured array with 2 columns (time, channel_data)

Return type numpy.core.records

data is a builtin to numpy and cannot be used as a name

loads into RAM

to_xarray()

Returns an xarray DataArray with appropriate metadata and the appropriate time index.

Return type xarray.DataArray

Note: that metadta will not be validated if changed in an xarray.

loads from memory

write_metadata()

Write metadata from the metadata container to the HDF5 attrs dictionary.

class mth5.groups.master_station_run_channel.**ElectricDataset**(*group*, ***kwargs*)

Bases: *mth5.groups.master_station_run_channel.ChannelDataset*

Holds a channel dataset. This is a simple container for the data to make sure that the user has the flexibility to turn the channel into an object they want to deal with.

For now all the numpy type slicing can be used on *hdf5_dataset*

Parameters

- **dataset** (h5py.Dataset) – dataset object for the channel
- **dataset_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If the dataset is not of the correct type

Utilities will be written to create some common objects like:

- xarray.DataArray
- pandas.DataFrame
- zarr
- dask.Array

The benefit of these other objects is that they can be indexed by time, and they have much more buit-in funcionality.

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> channel = run.get_channel('Ex')
>>> channel
Channel Electric:
-------------------
            component:        Ey
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:00:01+00:00
    sample rate:      4096
```

class mth5.groups.master_station_run_channel.**MagneticDataset**(*group*, ***kwargs*)

Bases: *mth5.groups.master_station_run_channel.ChannelDataset*

Holds a channel dataset. This is a simple container for the data to make sure that the user has the flexibility to turn the channel into an object they want to deal with.

For now all the numpy type slicing can be used on *hdf5_dataset*

Parameters

- **dataset** (h5py.Dataset) – dataset object for the channel
- **dataset_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If the dataset is not of the correct type

Utilities will be written to create some common objects like:

- xarray.DataArray
- pandas.DataFrame
- zarr
- dask.Array

The benefit of these other objects is that they can be indexed by time, and they have much more buit-in funcionality.

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> channel = run.get_channel('Ex')
>>> channel
Channel Electric:
-------------------
            component:        Ey
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:00:01+00:00
    sample rate:      4096
```

class mth5.groups.master_station_run_channel.**MasterStationGroup**(*group*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

Utility class to holds information about the stations within a survey and accompanying metadata. This class is next level down from Survey for stations `/Survey/Stations`. This class provides methods to add and get stations. A summary table of all existing stations is also provided as a convenience look up table to make searching easier.

To access MasterStationGroup from an open MTH5 file:

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> stations = mth5_obj.stations_group
```

To check what stations exist

```
>>> stations.groups_list
['summary', 'MT001', 'MT002', 'MT003']
```

To access the hdf5 group directly use *SurveyGroup.hdf5_group*.

```
>>> stations.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *SurveyGroup.write_metadata()* method. This is a temporary solution, working on an automatic updater if metadata is changed.

```
>>> stations.metadata.existing_attribute = 'update_existing_attribute'
>>> stations.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> stations.metadata.add_base_attribute('new_attribute',
>>> ...                                  'new_attribute_value',
>>> ...                                  {'type':str,
>>> ...                                   'required':True,
>>> ...                                   'style':'free form',
>>> ...                                   'description': 'new attribute desc.',
>>> ...                                   'units':None,
>>> ...                                   'options':[],
>>> ...                                   'alias':[],
>>> ...                                   'example':'new attribute
```

To add a station:

```
>>> new_station = stations.add_station('new_station')
>>> stations
/Survey/Stations:
====================
    --> Dataset: summary
    ......................
    |- Group: new_station
```

(continues on next page)

(continued from previous page)

```
    ---------------------
        --> Dataset: summary
        ......................
```

Add a station with metadata:

```
>>> from mth5.metadata import Station
>>> station_metadata = Station()
>>> station_metadata.id = 'MT004'
>>> station_metadata.time_period.start = '2020-01-01T12:30:00'
>>> station_metadata.location.latitude = 40.000
>>> station_metadata.location.longitude = -120.000
>>> new_station = stations.add_station('Test_01', station_metadata)
>>> # to look at the metadata
>>> new_station.metadata
{
    "station": {
        "acquired_by.author": null,
        "acquired_by.comments": null,
        "id": "MT004",
        ...
        }
}
```

See also:

mth5.metadata for details on how to add metadata from various files and python objects.

To remove a station:

```
>>> stations.remove_station('new_station')
>>> stations
/Survey/Stations:
====================
    --> Dataset: summary
    ......................
```

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

To get a station:

```
>>> existing_station = stations.get_station('existing_station_name')
>>> existing_station
/Survey/Stations/existing_station_name:
========================================
    --> Dataset: summary
    ......................
    |- Group: run_01
    ----------------
        --> Dataset: summary
```

(continues on next page)

(continued from previous page)

```
        ......................
        --> Dataset: Ex
        ......................
        --> Dataset: Ey
        ......................
        --> Dataset: Hx
        ......................
        --> Dataset: Hy
        ......................
        --> Dataset: Hz
        ......................
```

A summary table is provided to make searching easier. The table summarized all stations within a survey. To see what names are in the summary table:

```
>>> stations.summary_table.dtype.descr
[('id', ('|S5', {'h5py_encoding': 'ascii'})),
 ('start', ('|S32', {'h5py_encoding': 'ascii'})),
 ('end', ('|S32', {'h5py_encoding': 'ascii'})),
 ('components', ('|S100', {'h5py_encoding': 'ascii'})),
 ('measurement_type', ('|S12', {'h5py_encoding': 'ascii'})),
 ('sample_rate', '<f8')]
```

Note: When a station is added an entry is added to the summary table, where the information is pulled from the metadata.

```
>>> stations.summary_table
index |   id    |            start             |             end
 | components | measurement_type | sample_rate
 ------------------------------------------------------------------------------
 ------------------------------------------------
 0   |  Test_01   |  1980-01-01T00:00:00+00:00 |  1980-01-01T00:00:00+00:00
 |  Ex,Ey,Hx,Hy,Hz   |  BBMT   | 100
```

add_station(*station_name*, *station_metadata=None*)

Add a station with metadata if given with the path: `/Survey/Stations/station_name`
If the station already exists, will return that station and nothing is added.

Parameters

- **station_name** (*string*) – Name of the station, should be the same as metadata.id
- **station_metadata** (`mth5.metadata.Station`, optional) – Station metadata container, defaults to None

Returns A convenience class for the added station

Return type `mth5_groups.StationGroup`

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> # one option
>>> stations = mth5_obj.stations_group
>>> new_station = stations.add_station('MT001')
>>> # another option
>>> new_staiton = mth5_obj.stations_group.add_station('MT001')
```

property channel_summary
Summary of all channels in the file.

get_station(*station_name*)
Get a station with the same name as station_name

Parameters **station_name** (*string*) – existing station name

Returns convenience station class

Return type mth5.mth5_groups.StationGroup

Raises ***MTH5Error*** – if the station name is not found.

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> # one option
>>> stations = mth5_obj.stations_group
>>> existing_station = stations.get_station('MT001')
>>> # another option
>>> existing_staiton = mth5_obj.stations_group.get_station('MT001')
MTH5Error: MT001 does not exist, check station_list for existing names
```

remove_station(*station_name*)
Remove a station from the file.

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Parameters **station_name** (*string*) – existing station name

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> # one option
>>> stations = mth5_obj.stations_group
>>> stations.remove_station('MT001')
>>> # another option
>>> mth5_obj.stations_group.remove_station('MT001')
```

property station_summary
Summary of stations in the file

Returns DESCRIPTION

Return type TYPE

class mth5.groups.master_station_run_channel.**RunGroup**(*group*, *run_metadata=None*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

RunGroup is a utility class to hold information about a single run and accompanying metadata. This class is the next level down from Stations –> `/Survey/Stations/station/station{a-z}`.

This class provides methods to add and get channels. A summary table of all existing channels in the run is also provided as a convenience look up table to make searching easier.

Parameters

- **group** (`h5py.Group`) – HDF5 group for a station, should have a path `/Survey/Stations/station_name/run_name`
- **station_metadata** (`mth5.metadata.Station`, optional) – metadata container, defaults to None

Access RunGroup from an open MTH5 file

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
```

Check what channels exist

```
>>> station.groups_list
['Ex', 'Ey', 'Hx', 'Hy']
```

To access the hdf5 group directly use *RunGroup.hdf5_group*

```
>>> station.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *SurveyGroup.write_metadata()* method. This is a temporary solution, working on an automatic updater if metadata is changed.

```
>>> run.metadata.existing_attribute = 'update_existing_attribute'
>>> run.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> station.metadata.add_base_attribute('new_attribute',
>>> ...                                 'new_attribute_value',
>>> ...                                 {'type':str,
>>> ...                                  'required':True,
>>> ...                                  'style':'free form',
>>> ...                                  'description': 'new attribute desc.',
>>> ...                                  'units':None,
>>> ...                                  'options':[],
>>> ...                                  'alias':[],
>>> ...                                  'example':'new attribute
```

Add a channel

```
>>> new_channel = run.add_channel('Ex', 'electric',
>>> ...                           data=numpy.random.rand(4096))
>>> new_run
/Survey/Stations/MT001/MT001a:
======================================
    --> Dataset: summary
    ......................
    --> Dataset: Ex
    ......................
    --> Dataset: Ey
    ......................
    --> Dataset: Hx
    ......................
    --> Dataset: Hy
    ......................
```

Add a channel with metadata

```
>>> from mth5.metadata import Electric
>>> ex_metadata = Electric()
>>> ex_metadata.time_period.start = '2020-01-01T12:30:00'
>>> ex_metadata.time_period.end = '2020-01-03T16:30:00'
>>> new_ex = run.add_channel('Ex', 'electric',
>>> ...                      channel_metadata=ex_metadata)
>>> # to look at the metadata
>>> new_ex.metadata
{
     "electric": {
        "ac.end": 1.2,
        "ac.start": 2.3,
        ...
        }
}
```

See also:

mth5.metadata for details on how to add metadata from various files and python objects.

Remove a channel

```
>>> run.remove_channel('Ex')
>>> station
/Survey/Stations/MT001/MT001a:
======================================
    --> Dataset: summary
    ......................
    --> Dataset: Ey
    ......................
    --> Dataset: Hx
    ......................
    --> Dataset: Hy
    ......................
```

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Get a channel

```
>>> existing_ex = stations.get_channel('Ex')
>>> existing_ex
Channel Electric:
-------------------
    data type:        Ex
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:32:+08:00
    sample rate:      8
```

Summary Table

A summary table is provided to make searching easier. The table summarized all stations within a survey. To see what names are in the summary table:

```
>>> run.summary_table.dtype.descr
[('component', ('|S5', {'h5py_encoding': 'ascii'})),
 ('start', ('|S32', {'h5py_encoding': 'ascii'})),
 ('end', ('|S32', {'h5py_encoding': 'ascii'})),
 ('n_samples', '<i4'),
 ('measurement_type', ('|S12', {'h5py_encoding': 'ascii'})),
 ('units', ('|S25', {'h5py_encoding': 'ascii'})),
 ('hdf5_reference', ('|O', {'ref': h5py.h5r.Reference}))]
```

Note: When a run is added an entry is added to the summary table, where the information is pulled from the metadata.

```
>>> new_run.summary_table
index | component | start | end | n_samples | measurement_type | units |
hdf5_reference
--------------------------------------------------------------------------
-------------
```

add_channel(*channel_name*, *channel_type*, *data*, *channel_dtype='int32'*, *max_shape=(None)*, *chunks=True*, *channel_metadata=None*, ***kwargs*)

add a channel to the run

Parameters

- **channel_name** (*string*) – name of the channel
- **channel_type** (*string*) – [electric | magnetic | auxiliary]
- **channel_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If channel type is not correct

Returns Channel container

Return type [mth5.mth5_groups.ElectricDatset | mth5.mth5_groups.MagneticDatset | mth5.mth5_groups.AuxiliaryDatset]

```
>>> new_channel = run.add_channel('Ex', 'electric', None)
>>> new_channel
Channel Electric:
-------------------
                component:        None
        data type:        electric
        data format:      float32
        data shape:       (1,)
        start:            1980-01-01T00:00:00+00:00
        end:              1980-01-01T00:00:00+00:00
        sample rate:      None
```

property channel_summary

summary of channels in run :return: DESCRIPTION :rtype: TYPE

from_channel_ts(*channel_ts_obj*)

create a channel data set from a *mth5.timeseries.ChannelTS* object and update metadata.

Parameters **channel_ts_obj** (*mth5.timeseries.ChannelTS*) – a single time series object

Returns new channel dataset

Return type :class:`mth5.groups.ChannelDataset

from_runts(*run_ts_obj*, ***kwargs*)

create channel datasets from a *mth5.timeseries.RunTS* object and update metadata.

:parameter *mth5.timeseries.RunTS* run_ts_obj: Run object with all the appropriate channels and metadata.

Will create a run group and appropriate channel datasets.

get_channel(*channel_name*)

Get a channel from an existing name. Returns the appropriate container.

Parameters **channel_name** (*string*) – name of the channel

Returns Channel container

Return type [mth5.mth5_groups.ElectricDatset | mth5.mth5_groups.MagneticDatset | mth5.mth5_groups.AuxiliaryDatset]

Raises *MTH5Error* – If no channel is found

Example

```
>>> existing_channel = run.get_channel('Ex')
MTH5Error: Ex does not exist, check groups_list for existing names'
```

```
>>> run.groups_list
['Ey', 'Hx', 'Hz']
```

```
>>> existing_channel = run.get_channel('Ey')
>>> existing_channel
Channel Electric:
-------------------
                component:        Ey
        data type:        electric
        data format:      float32
        data shape:       (4096,)
        start:            1980-01-01T00:00:00+00:00
        end:              1980-01-01T00:00:01+00:00
        sample rate:      4096
```

property master_station_group
shortcut to master station group

property metadata
Overwrite get metadata to include channel information in the runs

remove_channel(*channel_name*)
Remove a run from the station.

Note: Deleting a channel is not as simple as del(channel). In HDF5 this does not free up memory, it simply removes the reference to that channel. The common way to get around this is to copy what you want into a new file, or overwrite the channel.

Parameters **station_name** (*string*) – existing station name

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> run.remove_channel('Ex')
```

property station_group
shortcut to station group

property table_entry
Get a run table entry

Returns a properly formatted run table entry

Return type numpy.ndarray with dtype:

```
>>> dtype([('id', 'S20'),
          ('start', 'S32'),
          ('end', 'S32'),
          ('components', 'S100'),
          ('measurement_type', 'S12'),
          ('sample_rate', float),
          ('hdf5_reference', h5py.ref_dtype)])
```

to_runts()
create a *mth5.timeseries.RunTS* object from channels of the run

Returns DESCRIPTION

Return type TYPE

validate_run_metadata()
Update metadata and table entries to ensure consistency
Returns DESCRIPTION

Return type TYPE

write_metadata()
Overwrite Base.write_metadata to include updating table entry Write HDF5 metadata from metadata object.

class mth5.groups.master_station_run_channel.**StationGroup**(*group*, *station_metadata=None*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

StationGroup is a utility class to hold information about a single station and accompanying metadata. This class is the next level down from Stations –> /Survey/Stations/station_name.

This class provides methods to add and get runs. A summary table of all existing runs in the station is also provided as a convenience look up table to make searching easier.

Parameters

- **group** (h5py.Group) – HDF5 group for a station, should have a path /Survey/Stations/station_name
- **station_metadata** (mth5.metadata.Station, optional) – metadata container, defaults to None

Usage

Access StationGroup from an open MTH5 file

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> station = mth5_obj.stations_group.get_station('MT001')
```

Check what runs exist

```
>>> station.groups_list
['MT001a', 'MT001b', 'MT001c', 'MT001d']
```

To access the hdf5 group directly use *StationGroup.hdf5_group*.

```
>>> station.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *SurveyGroup.write_metadata()* method. This is a temporary solution, working on an automatic updater if metadata is changed.

```
>>> station.metadata.existing_attribute = 'update_existing_attribute'
>>> station.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> station.metadata.add_base_attribute('new_attribute',
>>> ...                                 'new_attribute_value',
>>> ...                                 {'type':str,
>>> ...                                  'required':True,
>>> ...                                  'style':'free form',
>>> ...                                  'description': 'new attribute desc.',
>>> ...                                  'units':None,
>>> ...                                  'options':[],
>>> ...                                  'alias':[],
>>> ...                                  'example':'new attribute
```

To add a run

```
>>> new_run = stations.add_run('MT001e')
>>> new_run
/Survey/Stations/Test_01:
=========================
    |- Group: MT001e
    ----------------
        --> Dataset: summary
        ......................
    --> Dataset: summary
    ......................
```

Add a run with metadata

```
>>> from mth5.metadata import Run
>>> run_metadata = Run()
>>> run_metadata.time_period.start = '2020-01-01T12:30:00'
>>> run_metadata.time_period.end = '2020-01-03T16:30:00'
>>> run_metadata.location.latitude = 40.000
>>> run_metadata.location.longitude = -120.000
>>> new_run = runs.add_run('Test_01', run_metadata)
>>> # to look at the metadata
>>> new_run.metadata
{
    "run": {
        "acquired_by.author": "new_user",
        "acquired_by.comments": "First time",
        "channels_recorded_auxiliary": ['T'],
        ...
        }
}
```

See also:

mth5.metadata for details on how to add metadata from various files and python objects.

Remove a run

```
>>> station.remove_run('new_run')
>>> station
/Survey/Stations/Test_01:
=========================
```

(continues on next page)

(continued from previous page)

```
    --> Dataset: summary
    ......................
```

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Get a run

```
>>> existing_run = stations.get_station('existing_run')
>>> existing_run
/Survey/Stations/MT001/MT001a:
========================================
    --> Dataset: summary
    ......................
    --> Dataset: Ex
    ......................
    --> Dataset: Ey
    ......................
    --> Dataset: Hx
    ......................
    --> Dataset: Hy
    ......................
    --> Dataset: Hz
    ......................
```

Summary Table

A summary table is provided to make searching easier. The table summarized all stations within a survey. To see what names are in the summary table:

```
>>> new_run.summary_table.dtype.descr
[('id', ('|S20', {'h5py_encoding': 'ascii'})),
 ('start', ('|S32', {'h5py_encoding': 'ascii'})),
 ('end', ('|S32', {'h5py_encoding': 'ascii'})),
 ('components', ('|S100', {'h5py_encoding': 'ascii'})),
 ('measurement_type', ('|S12', {'h5py_encoding': 'ascii'})),
 ('sample_rate', '<f8'),
 ('hdf5_reference', ('|O', {'ref': h5py.h5r.Reference}))]
```

Note: When a run is added an entry is added to the summary table, where the information is pulled from the metadata.

```
>>> station.summary_table
index | id | start | end | components | measurement_type | sample_rate |
hdf5_reference
--------------------------------------------------------------------------------
-------------
```

add_run(*run_name*, *run_metadata=None*)
Add a run to a station.

Parameters

- **run_name** (*string*) – run name, should be id{a-z}
- **metadata** (mth5.metadata.Station, optional) – metadata container, defaults to None

need to be able to fill an entry in the summary table.

get_run(*run_name*)
get a run from run name

Parameters **run_name** (*string*) – existing run name

Returns Run object

Return type mth5.mth5_groups.RunGroup

```
>>> existing_run = station.get_run('MT001')
```

locate_run(*sample_rate*, *start*)
Locate a run based on sample rate and start time from the summary table

Parameters

- **sample_rate** (*float*) – sample rate in samples/seconds
- **start** (string or mth5.utils.mttime.MTime) – start time

Returns appropriate run name, None if not found

Return type string or None

make_run_name()
Make a run name that will be the next alphabet letter extracted from the run list. Expects that all runs are labled as id{a-z}.

Returns metadata.id + next letter

Return type string

```
>>> station.metadata.id = 'MT001'
>>> station.make_run_name()
'MT001a'
```

property master_station_group
shortcut to master station group

property metadata
Overwrite get metadata to include run information in the station

property name

remove_run(*run_name*)
Remove a run from the station.

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Parameters **station_name** (*string*) – existing station name

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> # one option
>>> stations = mth5_obj.stations_group
>>> stations.remove_station('MT001')
>>> # another option
>>> mth5_obj.stations_group.remove_station('MT001')
```

property run_summary
Summary of runs in the station

Returns DESCRIPTION

Return type TYPE

property table_entry
make table entry

validate_station_metadata()
Check metadata from the runs and make sure it matches the station metadata

Returns DESCRIPTION

Return type TYPE

mth5.groups.reports module

Created on Wed Dec 23 17:03:53 2020

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.reports.**ReportsGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Not sure how to handle this yet

add_report(*report_name*, *report_metadata=None*, *report_data=None*)

Parameters

- **report_name** (*TYPE*) – DESCRIPTION
- **report_metadata** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **report_data** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns DESCRIPTION

Return type TYPE

mth5.groups.standards module

Created on Wed Dec 23 17:05:33 2020

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.standards.**StandardsGroup**(*group*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

The StandardsGroup is a convenience group that stores the metadata standards that were used to make the current file. This is to help a user understand the metadata directly from the file and not have to look up documentation that might not be updated.

The metadata standards are stored in the summary table `/Survey/Standards/summary`

```
>>> standards = mth5_obj.standards_group
>>> standards.summary_table
index | attribute | type | required | style | units | description |
options  |  alias |  example
--------------------------------------------------------------------------------
```

get_attribute_information(*attribute_name*)

get information about an attribute

The attribute name should be in the summary table.

Parameters **attribute_name** (*string*) – attribute name

Returns prints a description of the attribute

Raises *MTH5TableError* – if attribute is not found

```
>>> standars = mth5_obj.standards_group
>>> standards.get_attribute_information('survey.release_license')
survey.release_license
--------------------------
        type          : string
        required      : True
        style         : controlled vocabulary
        units         :
        description   : How the data can be used. The options are based on
                Creative Commons licenses. For details visit
                https://creativecommons.org/licenses/
        options       : CC-0,CC-BY,CC-BY-SA,CC-BY-ND,CC-BY-NC-SA,CC-BY-NC-ND
        alias         :
        example       : CC-0
```

initialize_group()

Initialize the group by making a summary table that summarizes the metadata standards used to describe the data.

Also, write generic metadata information.

property summary_table

summary_table_from_dict(*summary_dict*)

Fill summary table from a dictionary that summarizes the metadata for the entire survey.

Parameters **summary_dict** (*dictionary*) – Flattened dictionary of all metadata standards within the survey.

mth5.groups.standards.**summarize_metadata_standards**()
Summarize metadata standards into a dictionary

mth5.groups.survey module

Created on Wed Dec 23 16:59:45 2020

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.groups.survey.**SurveyGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Utility class to holds general information about the survey and accompanying metadata for an MT survey.

To access the hdf5 group directly use *SurveyGroup.hdf5_group*.

```
>>> survey = SurveyGroup(hdf5_group)
>>> survey.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *SurveyGroup.write_metadata()* method. This is a temporary solution, working on an automatic updater if metadata is changed.

```
>>> survey.metadata.existing_attribute = 'update_existing_attribute'
>>> survey.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> survey.metadata.add_base_attribute('new_attribute',
>>> ...                                'new_attribute_value',
>>> ...                                {'type':str,
>>> ...                                 'required':True,
>>> ...                                 'style':'free form',
>>> ...                                 'description': 'new attribute desc.',
>>> ...                                 'units':None,
>>> ...                                 'options':[],
>>> ...                                 'alias':[],
>>> ...                                 'example':'new attribute
```

Tip: If you want ot add stations, reports, etc to the survey this should be done from the MTH5 object. This is to avoid duplication, at least for now.

To look at what the structure of /Survey looks like:

```
>>> survey
/Survey:
====================
    |- Group: Filters
    -----------------
```

(continues on next page)

(continued from previous page)

```
        --> Dataset: summary
    -----------------
    |- Group: Reports
    -----------------
        --> Dataset: summary
        -----------------
    |- Group: Standards
    -------------------
        --> Dataset: summary
        -----------------
    |- Group: Stations
    ------------------
        --> Dataset: summary
        -----------------
```

property metadata
Overwrite get metadata to include station information

property stations_group

update_survey_metadata(*survey_dict=None*)
update start end dates and location corners from stations_group.summary_table

Module contents

Import all Group objects

class mth5.groups.**AuxiliaryDataset**(*group*, ***kwargs*)
Bases: *mth5.groups.master_station_run_channel.ChannelDataset*

Holds a channel dataset. This is a simple container for the data to make sure that the user has the flexibility to turn the channel into an object they want to deal with.

For now all the numpy type slicing can be used on *hdf5_dataset*

Parameters

- **dataset** (h5py.Dataset) – dataset object for the channel
- **dataset_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If the dataset is not of the correct type

Utilities will be written to create some common objects like:

- xarray.DataArray
- pandas.DataFrame
- zarr
- dask.Array

The benefit of these other objects is that they can be indexed by time, and they have much more buit-in funcionality.

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> channel = run.get_channel('Ex')
>>> channel
```

(continues on next page)

(continued from previous page)

```
Channel Electric:
------------------
            component:        Ey
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:00:01+00:00
    sample rate:      4096
```

class mth5.groups.**BaseGroup**(*group*, *group_metadata=None*, ***kwargs*)

Bases: `object`

Generic object that will have functionality for reading/writing groups, including attributes. To access the hdf5 group directly use the *BaseGroup.hdf5_group* property.

```
>>> base = BaseGroup(hdf5_group)
>>> base.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *BaseGroup.write_metadata* method. This is a temporary solution working on an automatic updater if metadata is changed.

```
>>> base.metadata.existing_attribute = 'update_existing_attribute'
>>> base.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> base.metadata.add_base_attribute('new_attribute',
...                                  'new_attribute_value',
...                                  {'type':str,
...                                   'required':True,
...                                   'style':'free form',
...                                   'description': 'new attribute desc.',
...                                   'units':None,
...                                   'options':[],
...                                   'alias':[],
...                                   'example':'new attribute'})
```

Includes intializing functions that makes a summary table and writes metadata.

property dataset_options

property groups_list

initialize_group(***kwargs*)

Initialize group by making a summary table and writing metadata

property metadata

Metadata for the Group based on mt_metadata.timeseries

read_metadata()

read metadata from the HDF5 group into metadata object

write_metadata()
Write HDF5 metadata from metadata object.

class mth5.groups.**ChannelDataset**(*dataset*, *dataset_metadata=None*, ***kwargs*)
Bases: object

Holds a channel dataset. This is a simple container for the data to make sure that the user has the flexibility to turn the channel into an object they want to deal with.

For now all the numpy type slicing can be used on *hdf5_dataset*

Parameters

- **dataset** (h5py.Dataset) – dataset object for the channel
- **dataset_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If the dataset is not of the correct type

Utilities will be written to create some common objects like:

- xarray.DataArray
- pandas.DataFrame
- zarr
- dask.Array

The benefit of these other objects is that they can be indexed by time, and they have much more buit-in funcionality.

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> channel = run.get_channel('Ex')
>>> channel
Channel Electric:
--------------------
            component:        Ey
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:00:01+00:00
    sample rate:      4096
```

property channel_entry
channel entry that will go into a full channel summary of the entire survey

property channel_response_filter

property end
return end time based on the data

extend_dataset(*new_data_array*, *start_time*, *sample_rate*, *fill=None*, *max_gap_seconds=1*, *fill_window=10*)
Append data according to how the start time aligns with existing data. If the start time is before existing start time the data is prepended, similarly if the start time is near the end data will be appended.

If the start time is within the existing time range, existing data will be replace with the new data.

If there is a gap between start or end time of the new data with the existing data you can either fill the data with a constant value or an error will be raise depending on the value of fill.

Parameters

- **new_data_array** (numpy.ndarray) – new data array with shape (npts,)
- **start_time** (string or mth5.utils.mttime.MTime) – start time of the new data array in UTC
- **sample_rate** (*float*) – Sample rate of the new data array, must match existing sample rate
- **fill** (*string, None, float, integer*) – If there is a data gap how do you want to fill the gap * None: will raise an *mth5.utils.exceptions.MTH5Error* * 'mean': will fill with the mean of each data set within the fill window * 'median': will fill with the median of each data set within the fill window * value: can be an integer or float to fill the gap * 'nan': will fill the gap with NaN
- **max_gap_seconds** (*float or integer*) – sets a maximum number of seconds the gap can be. Anything over this number will raise a *mth5.utils.exceptions.MTH5Error*.
- **fill_window** (*integer*) – number of points from the end of each data set to estimate fill value from.

Raises mth5.utils.excptions.MTH5Error if sample rate is not the same, or fill value is not understood,

Rubric

```
>>> ex = mth5_obj.get_channel('MT001', 'MT001a', 'Ex')
>>> ex.n_samples
4096
>>> ex.end
2015-01-08T19:32:09.500000+00:00
>>> t = timeseries.ChannelTS('electric',
...                      data=2*np.cos(4 * np.pi * .05 *          ...
↪                       np.linspace(0,4096l num=4096) *
...                                 .01),
...                      channel_metadata={'electric':{
...                         'component': 'ex',
...                         'sample_rate': 8,
...                         'time_period.start':(ex.end+(1)).iso_str}})
>>> ex.extend_dataset(t.ts, t.start, t.sample_rate, fill='median',
...                   max_gap_seconds=2)
2020-07-02T18:02:47 - mth5.groups.Electric.extend_dataset - INFO -
filling data gap with 1.0385180759767025
>>> ex.n_samples
8200
>>> ex.end
2015-01-08T19:40:42.500000+00:00
```

from_channel_ts(*channel_ts_obj*, *how='replace'*, *fill=None*, *max_gap_seconds=1*, *fill_window=10*)

fill data set from a *mth5.timeseries.ChannelTS* object.

Will check for time alignement, and metadata.

Parameters

- **channel_ts_obj** (*mth5.timeseries.ChannelTS*) – time series object
- **how** – how the new array will be input to the existing dataset:
  - 'replace' -> replace the entire dataset nothing is left over.

- 'extend' -> add onto the existing dataset, any overlapping values will be rewritten, if there are gaps between data sets those will be handled depending on the value of fill.

  param fill If there is a data gap how do you want to fill the gap:

  - None -> will raise an *mth5.utils.exceptions.MTH5Error*
  - 'mean'-> will fill with the mean of each data set within the fill window
  - 'median' -> will fill with the median of each data set within the fill window
  - value -> can be an integer or float to fill the gap
  - 'nan' -> will fill the gap with NaN

- **max_gap_seconds** (*float or integer*) – sets a maximum number of seconds the gap can be. Anything over this number will raise a *mth5.utils.exceptions.MTH5Error*.
- **fill_window** (*integer*) – number of points from the end of each data set to estimate fill value from.

from_xarray(*data_array*, *how='replace'*, *fill=None*, *max_gap_seconds=1*, *fill_window=10*)

fill data set from a `xarray.DataArray` object.

Will check for time alignement, and metadata.

Parameters

- **data_array_obj** – Xarray data array
- **how** – how the new array will be input to the existing dataset:
  - 'replace' -> replace the entire dataset nothing is left over.
  - 'extend' -> add onto the existing dataset, any overlapping

  values will be rewritten, if there are gaps between data sets those will be handled depending on the value of fill.

  param fill If there is a data gap how do you want to fill the gap:

  - None -> will raise an *mth5.utils.exceptions.MTH5Error*
  - **'mean'-> will fill with the mean of each data set within** the fill window
  - **'median' -> will fill with the median of each data set** within the fill window
  - value -> can be an integer or float to fill the gap
  - 'nan' -> will fill the gap with NaN

- **max_gap_seconds** (*float or integer*) – sets a maximum number of seconds the gap can be. Anything over this number will raise a *mth5.utils.exceptions.MTH5Error*.
- **fill_window** (*integer*) – number of points from the end of each data set to estimate fill value from.

get_index_from_time(*given_time*)
get the appropriate index for a given time.

Parameters **given_time** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

property master_station_group
shortcut to master station group

property n_samples

read_metadata()
Read metadata from the HDF5 file into the metadata container, that way it can be validated.

replace_dataset(*new_data_array*)
replace the entire dataset with a new one, nothing left behind

Parameters **new_data_array** (`numpy.ndarray`) – new data array shape (npts,)

property run_group
shortcut to run group

property sample_rate

property start

property station_group
shortcut to station group

property table_entry
Creat a table entry to put into the run summary table.

property time_index
time index given parameters in metadata :rtype: `pandas.DatetimeIndex`

Type return

time_slice(*start_time*, *end_time=None*, *n_samples=None*, *return_type='channel_ts'*)
Get a time slice from the channel and return the appropriate type

- numpy array with metadata
- pandas.Dataframe with metadata
- xarray.DataFrame with metadata
- *mth5.timeseries.ChannelTS* 'default'
- dask.DataFrame with metadata 'not yet'

Parameters

- **start_time** (string or `mth5.utils.mttime.MTime`) – start time of the slice
- **end_time** (string or `mth5.utils.mttime.MTime`, optional) – end time of the slice
- **n_samples** (*integer, optional*) – number of samples to read in

Returns the correct container for the time series.

Return type [`xarray.DataArray` | `pandas.DataFrame` | *mth5.timeseries.ChannelTS* | `numpy.ndarray`]

Raises ValueError if both end_time and n_samples are None or given.

Example with number of samples

```
>>> ex = mth5_obj.get_channel('FL001', 'FL001a', 'Ex')
>>> ex_slice = ex.time_slice("2015-01-08T19:49:15", n_samples=4096)
>>> ex_slice
<xarray.DataArray (time: 4096)>
array([0.93115046, 0.14233688, 0.87917119, ..., 0.26073634, 0.7137319 ,
       0.88154395])
Coordinates:
  * time     (time) datetime64[ns] 2015-01-08T19:49:15 ... 2015-01-08T19:57:46.
↪875000
Attributes:
    ac.end:                     None
    ac.start:                   None
    ...

>>> type(ex_slice)
mth5.timeseries.ChannelTS

# plot the time series
>>> ex_slice.ts.plot()
```

Example with start and end time

```
>>> ex_slice = ex.time_slice("2015-01-08T19:49:15",
...                          end_time="2015-01-09T19:49:15")
```

Raises Example

```
>>> ex_slice = ex.time_slice("2015-01-08T19:49:15",
...                          end_time="2015-01-09T19:49:15",
...                          n_samples=4096)
ValueError: Must input either end_time or n_samples, not both.
```

`to_channel_ts`()

Returns a Timeseries with the appropriate time index and metadata

Return type *mth5.timeseries.ChannelTS*

loads from memory (nearly half the size of xarray alone, not sure why)

`to_dataframe`()

Returns a dataframe where data is stored in the 'data' column and attributes are stored in the experimental attrs attribute

Return type `pandas.DataFrame`

Note: that metadta will not be validated if changed in an xarray.

loads into RAM

`to_numpy`()

Returns a numpy structured array with 2 columns (time, channel_data)

Return type `numpy.core.records`

data is a builtin to numpy and cannot be used as a name

loads into RAM

to_xarray()

Returns an xarray DataArray with appropriate metadata and the appropriate time index.

Return type xarray.DataArray

Note: that metadta will not be validated if changed in an xarray.

loads from memory

write_metadata()
Write metadata from the metadata container to the HDF5 attrs dictionary.

class mth5.groups.**ElectricDataset**(*group*, ***kwargs*)
Bases: *mth5.groups.master_station_run_channel.ChannelDataset*

Holds a channel dataset. This is a simple container for the data to make sure that the user has the flexibility to turn the channel into an object they want to deal with.

For now all the numpy type slicing can be used on *hdf5_dataset*

Parameters

- **dataset** (h5py.Dataset) – dataset object for the channel
- **dataset_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If the dataset is not of the correct type

Utilities will be written to create some common objects like:

- xarray.DataArray
- pandas.DataFrame
- zarr
- dask.Array

The benefit of these other objects is that they can be indexed by time, and they have much more buit-in funcionality.

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> channel = run.get_channel('Ex')
>>> channel
Channel Electric:
-------------------
            component:        Ey
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:00:01+00:00
    sample rate:      4096
```

class mth5.groups.**FiltersGroup**(*group*, ***kwargs*)
Bases: *mth5.groups.base.BaseGroup*

Not implemented yet

add_filter(*filter_object*)
Add a filter dataset based on type
current types are:

- zpk –> zeros, poles, gain
- fap –> frequency look up table
- time_delay –> time delay filter
- coefficient –> coefficient filter

Parameters **filter_object** (mt_metadata.timeseries.filters) – An MT metadata filter object

property filter_dict

get_filter(*name*)
Get a filter by name

to_filter_object(*name*)
return the MT metadata representation of the filter

class mth5.groups.**MagneticDataset**(*group*, ***kwargs*)
Bases: *mth5.groups.master_station_run_channel.ChannelDataset*

Holds a channel dataset. This is a simple container for the data to make sure that the user has the flexibility to turn the channel into an object they want to deal with.

For now all the numpy type slicing can be used on *hdf5_dataset*
Parameters

- **dataset** (h5py.Dataset) – dataset object for the channel
- **dataset_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If the dataset is not of the correct type
Utilities will be written to create some common objects like:

- xarray.DataArray
- pandas.DataFrame
- zarr
- dask.Array

The benefit of these other objects is that they can be indexed by time, and they have much more buit-in funcionality.

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> channel = run.get_channel('Ex')
>>> channel
Channel Electric:
--------------------
            component:        Ey
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:00:01+00:00
    sample rate:      4096
```

class mth5.groups.**MasterStationGroup**(*group*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

Utility class to holds information about the stations within a survey and accompanying metadata. This class is next level down from Survey for stations `/Survey/Stations`. This class provides methods to add and get stations. A summary table of all existing stations is also provided as a convenience look up table to make searching easier.

To access MasterStationGroup from an open MTH5 file:

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> stations = mth5_obj.stations_group
```

To check what stations exist

```
>>> stations.groups_list
['summary', 'MT001', 'MT002', 'MT003']
```

To access the hdf5 group directly use *SurveyGroup.hdf5_group*.

```
>>> stations.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *SurveyGroup.write_metadata()* method. This is a temporary solution, working on an automatic updater if metadata is changed.

```
>>> stations.metadata.existing_attribute = 'update_existing_attribute'
>>> stations.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> stations.metadata.add_base_attribute('new_attribute',
>>> ...                                  'new_attribute_value',
>>> ...                                  {'type':str,
>>> ...                                   'required':True,
>>> ...                                   'style':'free form',
>>> ...                                   'description': 'new attribute desc.',
>>> ...                                   'units':None,
>>> ...                                   'options':[],
>>> ...                                   'alias':[],
>>> ...                                   'example':'new attribute
```

To add a station:

```
>>> new_station = stations.add_station('new_station')
>>> stations
/Survey/Stations:
====================
    --> Dataset: summary
    ......................
    |- Group: new_station
```

(continues on next page)

(continued from previous page)

```
    ---------------------
        --> Dataset: summary
        ......................
```

Add a station with metadata:

```
>>> from mth5.metadata import Station
>>> station_metadata = Station()
>>> station_metadata.id = 'MT004'
>>> station_metadata.time_period.start = '2020-01-01T12:30:00'
>>> station_metadata.location.latitude = 40.000
>>> station_metadata.location.longitude = -120.000
>>> new_station = stations.add_station('Test_01', station_metadata)
>>> # to look at the metadata
>>> new_station.metadata
{
    "station": {
        "acquired_by.author": null,
        "acquired_by.comments": null,
        "id": "MT004",
        ...
        }
}
```

See also:

mth5.metadata for details on how to add metadata from various files and python objects.

To remove a station:

```
>>> stations.remove_station('new_station')
>>> stations
/Survey/Stations:
====================
    --> Dataset: summary
    ......................
```

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

To get a station:

```
>>> existing_station = stations.get_station('existing_station_name')
>>> existing_station
/Survey/Stations/existing_station_name:
========================================
    --> Dataset: summary
    ......................
    |- Group: run_01
    ----------------
        --> Dataset: summary
```

(continues on next page)

(continued from previous page)

```
        ......................
        --> Dataset: Ex
        ......................
        --> Dataset: Ey
        ......................
        --> Dataset: Hx
        ......................
        --> Dataset: Hy
        ......................
        --> Dataset: Hz
        ......................
```

A summary table is provided to make searching easier. The table summarized all stations within a survey. To see what names are in the summary table:

```
>>> stations.summary_table.dtype.descr
[('id', ('|S5', {'h5py_encoding': 'ascii'})),
 ('start', ('|S32', {'h5py_encoding': 'ascii'})),
 ('end', ('|S32', {'h5py_encoding': 'ascii'})),
 ('components', ('|S100', {'h5py_encoding': 'ascii'})),
 ('measurement_type', ('|S12', {'h5py_encoding': 'ascii'})),
 ('sample_rate', '<f8')]
```

Note: When a station is added an entry is added to the summary table, where the information is pulled from the metadata.

```
>>> stations.summary_table
index |   id    |            start             |             end
 | components | measurement_type | sample_rate
 ------------------------------------------------------------------------------
 ------------------------------------------------
 0   |  Test_01   |  1980-01-01T00:00:00+00:00 |  1980-01-01T00:00:00+00:00
 |  Ex,Ey,Hx,Hy,Hz   |  BBMT   | 100
```

add_station(*station_name*, *station_metadata=None*)

Add a station with metadata if given with the path: `/Survey/Stations/station_name`
If the station already exists, will return that station and nothing is added.

Parameters

- **station_name** (*string*) – Name of the station, should be the same as metadata.id
- **station_metadata** (`mth5.metadata.Station`, optional) – Station metadata container, defaults to None

Returns A convenience class for the added station

Return type `mth5_groups.StationGroup`

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> # one option
>>> stations = mth5_obj.stations_group
>>> new_station = stations.add_station('MT001')
>>> # another option
>>> new_staiton = mth5_obj.stations_group.add_station('MT001')
```

property channel_summary
Summary of all channels in the file.

get_station(*station_name*)
Get a station with the same name as station_name

Parameters **station_name** (*string*) – existing station name

Returns convenience station class

Return type mth5.mth5_groups.StationGroup

Raises ***MTH5Error*** – if the station name is not found.

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> # one option
>>> stations = mth5_obj.stations_group
>>> existing_station = stations.get_station('MT001')
>>> # another option
>>> existing_staiton = mth5_obj.stations_group.get_station('MT001')
MTH5Error: MT001 does not exist, check station_list for existing names
```

remove_station(*station_name*)
Remove a station from the file.

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Parameters **station_name** (*string*) – existing station name

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> # one option
>>> stations = mth5_obj.stations_group
>>> stations.remove_station('MT001')
>>> # another option
>>> mth5_obj.stations_group.remove_station('MT001')
```

property station_summary
Summary of stations in the file

Returns DESCRIPTION

Return type TYPE

class mth5.groups.**ReportsGroup**(*group*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

Not sure how to handle this yet

add_report(*report_name*, *report_metadata=None*, *report_data=None*)

Parameters

- **report_name** (*TYPE*) – DESCRIPTION
- **report_metadata** (*TYPE, optional*) – DESCRIPTION, defaults to None
- **report_data** (*TYPE, optional*) – DESCRIPTION, defaults to None

Returns DESCRIPTION

Return type TYPE

class mth5.groups.**RunGroup**(*group*, *run_metadata=None*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

RunGroup is a utility class to hold information about a single run and accompanying metadata. This class is the next level down from Stations –> `/Survey/Stations/station/station{a-z}`.

This class provides methods to add and get channels. A summary table of all existing channels in the run is also provided as a convenience look up table to make searching easier.

Parameters

- **group** (`h5py.Group`) – HDF5 group for a station, should have a path `/Survey/Stations/station_name/run_name`
- **station_metadata** (`mth5.metadata.Station`, optional) – metadata container, defaults to None

Access RunGroup from an open MTH5 file

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
```

Check what channels exist

```
>>> station.groups_list
['Ex', 'Ey', 'Hx', 'Hy']
```

To access the hdf5 group directly use *RunGroup.hdf5_group*

```
>>> station.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *SurveyGroup.write_metadata()* method. This is a temporary solution, working on an automatic updater if metadata is changed.

```
>>> run.metadata.existing_attribute = 'update_existing_attribute'
>>> run.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> station.metadata.add_base_attribute('new_attribute',
>>> ...                                 'new_attribute_value',
>>> ...                                 {'type':str,
>>> ...                                  'required':True,
>>> ...                                  'style':'free form',
>>> ...                                  'description': 'new attribute desc.',
>>> ...                                  'units':None,
>>> ...                                  'options':[],
>>> ...                                  'alias':[],
>>> ...                                  'example':'new attribute
```

Add a channel

```
>>> new_channel = run.add_channel('Ex', 'electric',
>>> ...                           data=numpy.random.rand(4096))
>>> new_run
/Survey/Stations/MT001/MT001a:
========================================
    --> Dataset: summary
    ......................
    --> Dataset: Ex
    ......................
    --> Dataset: Ey
    ......................
    --> Dataset: Hx
    ......................
    --> Dataset: Hy
    ......................
```

Add a channel with metadata

```
>>> from mth5.metadata import Electric
>>> ex_metadata = Electric()
>>> ex_metadata.time_period.start = '2020-01-01T12:30:00'
>>> ex_metadata.time_period.end = '2020-01-03T16:30:00'
>>> new_ex = run.add_channel('Ex', 'electric',
>>> ...                      channel_metadata=ex_metadata)
>>> # to look at the metadata
>>> new_ex.metadata
{
     "electric": {
        "ac.end": 1.2,
        "ac.start": 2.3,
        ...
        }
}
```

See also:

mth5.metadata for details on how to add metadata from various files and python objects.

Remove a channel

```
>>> run.remove_channel('Ex')
>>> station
/Survey/Stations/MT001/MT001a:
======================================
    --> Dataset: summary
    ......................
    --> Dataset: Ey
    ......................
    --> Dataset: Hx
    ......................
    --> Dataset: Hy
    ......................
```

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Get a channel

```
>>> existing_ex = stations.get_channel('Ex')
>>> existing_ex
Channel Electric:
-------------------
    data type:        Ex
    data type:        electric
    data format:      float32
    data shape:       (4096,)
    start:            1980-01-01T00:00:00+00:00
    end:              1980-01-01T00:32:+08:00
    sample rate:      8
```

Summary Table

A summary table is provided to make searching easier. The table summarized all stations within a survey. To see what names are in the summary table:

```
>>> run.summary_table.dtype.descr
[('component', ('|S5', {'h5py_encoding': 'ascii'})),
 ('start', ('|S32', {'h5py_encoding': 'ascii'})),
 ('end', ('|S32', {'h5py_encoding': 'ascii'})),
 ('n_samples', '<i4'),
 ('measurement_type', ('|S12', {'h5py_encoding': 'ascii'})),
 ('units', ('|S25', {'h5py_encoding': 'ascii'})),
 ('hdf5_reference', ('|O', {'ref': h5py.h5r.Reference}))]
```

Note: When a run is added an entry is added to the summary table, where the information is pulled from the metadata.

```
>>> new_run.summary_table
index | component | start | end | n_samples | measurement_type | units |
hdf5_reference
-----------------------------------------------------------------------------
-------------
```

add_channel(*channel_name*, *channel_type*, *data*, *channel_dtype='int32'*, *max_shape=(None)*, *chunks=True*, *channel_metadata=None*, ***kwargs*)

add a channel to the run

Parameters

- **channel_name** (*string*) – name of the channel
- **channel_type** (*string*) – [electric | magnetic | auxiliary]
- **channel_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If channel type is not correct

Returns Channel container

Return type [mth5.mth5_groups.ElectricDatset | mth5.mth5_groups.MagneticDatset | mth5.mth5_groups.AuxiliaryDatset]

```
>>> new_channel = run.add_channel('Ex', 'electric', None)
>>> new_channel
Channel Electric:
-------------------
                component:        None
        data type:        electric
        data format:      float32
        data shape:       (1,)
        start:            1980-01-01T00:00:00+00:00
        end:              1980-01-01T00:00:00+00:00
        sample rate:      None
```

property channel_summary

summary of channels in run :return: DESCRIPTION :rtype: TYPE

from_channel_ts(*channel_ts_obj*)

create a channel data set from a *mth5.timeseries.ChannelTS* object and update metadata.

Parameters **channel_ts_obj** (*mth5.timeseries.ChannelTS*) – a single time series object

Returns new channel dataset

Return type :class:`mth5.groups.ChannelDataset

from_runts(*run_ts_obj*, ***kwargs*)

create channel datasets from a *mth5.timeseries.RunTS* object and update metadata.

:parameter *mth5.timeseries.RunTS* run_ts_obj: Run object with all the appropriate channels and metadata.

Will create a run group and appropriate channel datasets.

get_channel(*channel_name*)

Get a channel from an existing name. Returns the appropriate container.

Parameters **channel_name** (*string*) – name of the channel

Returns Channel container

Return type [mth5.mth5_groups.ElectricDatset | mth5.mth5_groups.MagneticDatset | mth5.mth5_groups.AuxiliaryDatset]

Raises *MTH5Error* – If no channel is found

Example

```
>>> existing_channel = run.get_channel('Ex')
MTH5Error: Ex does not exist, check groups_list for existing names'
```

```
>>> run.groups_list
['Ey', 'Hx', 'Hz']
```

```
>>> existing_channel = run.get_channel('Ey')
>>> existing_channel
Channel Electric:
-------------------
                component:        Ey
        data type:        electric
        data format:      float32
        data shape:       (4096,)
        start:            1980-01-01T00:00:00+00:00
        end:              1980-01-01T00:00:01+00:00
        sample rate:      4096
```

property master_station_group
shortcut to master station group

property metadata
Overwrite get metadata to include channel information in the runs

remove_channel(*channel_name*)
Remove a run from the station.

Note: Deleting a channel is not as simple as del(channel). In HDF5 this does not free up memory, it simply removes the reference to that channel. The common way to get around this is to copy what you want into a new file, or overwrite the channel.

Parameters **station_name** (*string*) – existing station name

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> run = mth5_obj.stations_group.get_station('MT001').get_run('MT001a')
>>> run.remove_channel('Ex')
```

property station_group
shortcut to station group

property table_entry
Get a run table entry

Returns a properly formatted run table entry

Return type numpy.ndarray with dtype:

```
>>> dtype([('id', 'S20'),
          ('start', 'S32'),
          ('end', 'S32'),
          ('components', 'S100'),
          ('measurement_type', 'S12'),
          ('sample_rate', float),
          ('hdf5_reference', h5py.ref_dtype)])
```

to_runts()

create a *mth5.timeseries.RunTS* object from channels of the run

Returns DESCRIPTION

Return type TYPE

validate_run_metadata()

Update metadata and table entries to ensure consistency

Returns DESCRIPTION

Return type TYPE

write_metadata()

Overwrite Base.write_metadata to include updating table entry Write HDF5 metadata from metadata object.

class mth5.groups.**StandardsGroup**(*group*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

The StandardsGroup is a convenience group that stores the metadata standards that were used to make the current file. This is to help a user understand the metadata directly from the file and not have to look up documentation that might not be updated.

The metadata standards are stored in the summary table /Survey/Standards/summary

```
>>> standards = mth5_obj.standards_group
>>> standards.summary_table
index | attribute | type | required | style | units | description |
options  |  alias |  example
----------------------------------------------------------------------------
```

get_attribute_information(*attribute_name*)

get information about an attribute

The attribute name should be in the summary table.

Parameters **attribute_name** (*string*) – attribute name

Returns prints a description of the attribute

Raises *MTH5TableError* – if attribute is not found

```
>>> standars = mth5_obj.standards_group
>>> standards.get_attribute_information('survey.release_license')
survey.release_license
--------------------------
        type          : string
        required      : True
```

(continues on next page)

(continued from previous page)

```
        style         : controlled vocabulary
        units         :
        description   : How the data can be used. The options are based on
                Creative Commons licenses. For details visit
                https://creativecommons.org/licenses/
        options       : CC-0,CC-BY,CC-BY-SA,CC-BY-ND,CC-BY-NC-SA,CC-BY-NC-ND
        alias         :
        example       : CC-0
```

initialize_group()

Initialize the group by making a summary table that summarizes the metadata standards used to describe the data.

Also, write generic metadata information.

property summary_table

summary_table_from_dict(*summary_dict*)

Fill summary table from a dictionary that summarizes the metadata for the entire survey.

Parameters **summary_dict** (*dictionary*) – Flattened dictionary of all metadata standards within the survey.

class mth5.groups.**StationGroup**(*group*, *station_metadata=None*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

StationGroup is a utility class to hold information about a single station and accompanying metadata. This class is the next level down from Stations –> `/Survey/Stations/station_name`.

This class provides methods to add and get runs. A summary table of all existing runs in the station is also provided as a convenience look up table to make searching easier.

Parameters

- **group** (h5py.Group) – HDF5 group for a station, should have a path `/Survey/Stations/station_name`
- **station_metadata** (mth5.metadata.Station, optional) – metadata container, defaults to None

Usage

Access StationGroup from an open MTH5 file

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> station = mth5_obj.stations_group.get_station('MT001')
```

Check what runs exist

```
>>> station.groups_list
['MT001a', 'MT001b', 'MT001c', 'MT001d']
```

To access the hdf5 group directly use *StationGroup.hdf5_group*.

```
>>> station.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the meta-

data standards. If you change attributes in metadata object, you should run the *SurveyGroup.write_metadata()* method. This is a temporary solution, working on an automatic updater if metadata is changed.

```
>>> station.metadata.existing_attribute = 'update_existing_attribute'
>>> station.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> station.metadata.add_base_attribute('new_attribute',
>>> ...                                 'new_attribute_value',
>>> ...                                 {'type':str,
>>> ...                                  'required':True,
>>> ...                                  'style':'free form',
>>> ...                                  'description': 'new attribute desc.',
>>> ...                                  'units':None,
>>> ...                                  'options':[],
>>> ...                                  'alias':[],
>>> ...                                  'example':'new attribute
```

To add a run

```
>>> new_run = stations.add_run('MT001e')
>>> new_run
/Survey/Stations/Test_01:
=========================
    |- Group: MT001e
    ----------------
        --> Dataset: summary
        ......................
    --> Dataset: summary
    ......................
```

Add a run with metadata

```
>>> from mth5.metadata import Run
>>> run_metadata = Run()
>>> run_metadata.time_period.start = '2020-01-01T12:30:00'
>>> run_metadata.time_period.end = '2020-01-03T16:30:00'
>>> run_metadata.location.latitude = 40.000
>>> run_metadata.location.longitude = -120.000
>>> new_run = runs.add_run('Test_01', run_metadata)
>>> # to look at the metadata
>>> new_run.metadata
{
    "run": {
        "acquired_by.author": "new_user",
        "acquired_by.comments": "First time",
        "channels_recorded_auxiliary": ['T'],
        ...
        }
}
```

See also:

mth5.metadata for details on how to add metadata from various files and python objects.

Remove a run

```
>>> station.remove_run('new_run')
>>> station
/Survey/Stations/Test_01:
=========================
    --> Dataset: summary
    ......................
```

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Get a run

```
>>> existing_run = stations.get_station('existing_run')
>>> existing_run
/Survey/Stations/MT001/MT001a:
======================================
    --> Dataset: summary
    ......................
    --> Dataset: Ex
    ......................
    --> Dataset: Ey
    ......................
    --> Dataset: Hx
    ......................
    --> Dataset: Hy
    ......................
    --> Dataset: Hz
    ......................
```

Summary Table

A summary table is provided to make searching easier. The table summarized all stations within a survey. To see what names are in the summary table:

```
>>> new_run.summary_table.dtype.descr
[('id', ('|S20', {'h5py_encoding': 'ascii'})),
 ('start', ('|S32', {'h5py_encoding': 'ascii'})),
 ('end', ('|S32', {'h5py_encoding': 'ascii'})),
 ('components', ('|S100', {'h5py_encoding': 'ascii'})),
 ('measurement_type', ('|S12', {'h5py_encoding': 'ascii'})),
 ('sample_rate', '<f8'),
 ('hdf5_reference', ('|O', {'ref': h5py.h5r.Reference}))]
```

Note: When a run is added an entry is added to the summary table, where the information is pulled from the metadata.

```
>>> station.summary_table
index | id | start | end | components | measurement_type | sample_rate |
hdf5_reference
-------------------------------------------------------------------------
-------------
```

add_run(*run_name*, *run_metadata=None*)

Add a run to a station.

Parameters

- **run_name** (*string*) – run name, should be id{a-z}
- **metadata** (mth5.metadata.Station, optional) – metadata container, defaults to None

need to be able to fill an entry in the summary table.

get_run(*run_name*)

get a run from run name

Parameters **run_name** (*string*) – existing run name

Returns Run object

Return type mth5.mth5_groups.RunGroup

```
>>> existing_run = station.get_run('MT001')
```

locate_run(*sample_rate*, *start*)

Locate a run based on sample rate and start time from the summary table

Parameters

- **sample_rate** (*float*) – sample rate in samples/seconds
- **start** (string or mth5.utils.mttime.MTime) – start time

Returns appropriate run name, None if not found

Return type string or None

make_run_name()

Make a run name that will be the next alphabet letter extracted from the run list. Expects that all runs are labled as id{a-z}.

Returns metadata.id + next letter

Return type string

```
>>> station.metadata.id = 'MT001'
>>> station.make_run_name()
'MT001a'
```

property master_station_group

shortcut to master station group

property metadata

Overwrite get metadata to include run information in the station

property name

remove_run(*run_name*)

Remove a run from the station.

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Parameters **station_name** (*string*) – existing station name

Example

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> mth5_obj.open_mth5(r"/test.mth5", mode='a')
>>> # one option
>>> stations = mth5_obj.stations_group
>>> stations.remove_station('MT001')
>>> # another option
>>> mth5_obj.stations_group.remove_station('MT001')
```

property run_summary

Summary of runs in the station

Returns DESCRIPTION

Return type TYPE

property table_entry

make table entry

validate_station_metadata()

Check metadata from the runs and make sure it matches the station metadata

Returns DESCRIPTION

Return type TYPE

class mth5.groups.**SurveyGroup**(*group*, ***kwargs*)

Bases: *mth5.groups.base.BaseGroup*

Utility class to holds general information about the survey and accompanying metadata for an MT survey.

To access the hdf5 group directly use *SurveyGroup.hdf5_group*.

```
>>> survey = SurveyGroup(hdf5_group)
>>> survey.hdf5_group.ref
<HDF5 Group Reference>
```

Note: All attributes should be input into the metadata object, that way all input will be validated against the metadata standards. If you change attributes in metadata object, you should run the *SurveyGroup.write_metadata()* method. This is a temporary solution, working on an automatic updater if metadata is changed.

```
>>> survey.metadata.existing_attribute = 'update_existing_attribute'
>>> survey.write_metadata()
```

If you want to add a new attribute this should be done using the *metadata.add_base_attribute* method.

```
>>> survey.metadata.add_base_attribute('new_attribute',
>>> ...                                'new_attribute_value',
```

(continues on next page)

(continued from previous page)

```
>>> ...                                 {'type':str,
>>> ...                                  'required':True,
>>> ...                                  'style':'free form',
>>> ...                                  'description': 'new attribute desc.',
>>> ...                                  'units':None,
>>> ...                                  'options':[],
>>> ...                                  'alias':[],
>>> ...                                  'example':'new attribute
```

Tip: If you want ot add stations, reports, etc to the survey this should be done from the MTH5 object. This is to avoid duplication, at least for now.

To look at what the structure of `/Survey` looks like:

```
>>> survey
/Survey:
====================
    |- Group: Filters
    -----------------
        --> Dataset: summary
    -----------------
    |- Group: Reports
    -----------------
        --> Dataset: summary
        -----------------
    |- Group: Standards
    -------------------
        --> Dataset: summary
        -----------------
    |- Group: Stations
    ------------------
        --> Dataset: summary
        -----------------
```

property metadata
Overwrite get metadata to include station information

property stations_group

update_survey_metadata(*survey_dict=None*)
update start end dates and location corners from stations_group.summary_table

14.1.2 mth5.io package

Submodules

mth5.io.miniseed module

Created on Wed Sep 30 10:20:12 2020

author Jared Peacock

license MIT

mth5.io.miniseed.**read_miniseed**(*fn*)

Read a miniseed file into a *mth5.timeseries.RunTS* object

Parameters **fn** (*string*) – full path to the miniseed file

Returns

Return type TYPE

mth5.io.nims module

NIMS

- deals with reading in NIMS DATA.BIN files

This is a translation from Matlab codes written and edited by:

- Anna Kelbert
- Paul Bedrosian
- Esteban Bowles-Martinez
- Possibly others.

I've tested it against a version, and it matches. The data/GPS gaps I still don't understand so for now the time series is just made continuous and the number of missing seconds is clipped from the end of the time series.

Note: this only works for 8Hz data for now

---

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.io.nims.**GPS**(*gps_string*, *index=0*)

Bases: object

class to parse GPS stamp from the NIMS

Depending on the type of Stamp different attributes will be filled.

GPRMC has full date and time information and declination GPGGA has elevation data

---

Note: GPGGA date is set to 1980-01-01 so that the time can be estimated. Should use GPRMC for accurate date/time information.

---

property declination

geomagnetic declination in degrees from north

property elevation
elevation in meters

property fix
GPS fixed

property gps_type
GPRMC or GPGGA

property latitude
Latitude in decimal degrees, WGS84

property longitude
Latitude in decimal degrees, WGS84

parse_gps_string(*gps_string*)
Parse a raw gps string from the NIMS and set appropriate attributes. GPS string will first be validated, then parsed.

Parameters **gps_string** (*string*) – raw GPS string to be parsed

property time_stamp
return a datetime object of the time stamp

validate_gps_list(*gps_list*)
check to make sure the gps stamp is the correct format, checks each element for the proper format

Parameters **gps_list** (*list*) – a parsed gps string from a NIMS

Raises *mth5.io.nims.GPSError* if anything is wrong.

validate_gps_string(*gps_string*)
make sure the string is valid, remove any binary numbers and find the end of the string as '*'

Parameters **gps_string** (*string*) – raw GPS string to be validated

Returns validated string or None if there is something wrong

exception mth5.io.nims.**GPSError**
Bases: Exception

class mth5.io.nims.**NIMS**(*fn=None*)
Bases: *mth5.io.nims.NIMSHeader*

NIMS Class will read in a NIMS DATA.BIN file.

A fast way to read the binary files are to first read in the GPS strings, the third byte in each block as a character and parse that into valid GPS stamps.

Then read in the entire data set as unsigned 8 bit integers and reshape the data to be n seconds x block size. Then parse that array into the status information and data.

I only have a limited amount of .BIN files to test so this will likely break if there are issues such as data gaps. This has been tested against the matlab program loadNIMS by Anna Kelbert and the match for all the .bin files I have. If something looks weird check it against that program.

Warning: Currently Only 8 Hz data is supported

align_data(*data_array*, *stamps*)
Need to match up the first good GPS stamp with the data

Do this by using the first GPS stamp and assuming that the time from the first time stamp to the start is the index value.

put the data into a pandas data frame that is indexed by time

Parameters

- **data_array** (*array*) – structure array with columns for each component [hx, hy, hz, ex, ey]
- **stamps** (*list*) – list of GPS stamps [[status_index, [GPRMC, GPGGA]]]

Returns pandas DataFrame with colums of components and indexed by time initialized by the start time.

Note: Data gaps are squeezed cause not sure what a gap actually means.

property box_temperature
data logger temperature, sampled at 1 second

calibrate_data(*ts*)
Apply calibrations to data

Note: this needs work, would not use this now.

check_timing(*stamps*)
make sure that there are the correct number of seconds in between the first and last GPS GPRMC stamps

Parameters **stamps** (*list*) – list of GPS stamps [[status_index, [GPRMC, GPGGA]]]

Returns [True | False] if data is valid or not.

Returns gap index locations

Note: currently it is assumed that if a data gap occurs the data can be squeezed to remove them. Probably a more elegant way of doing it.

property declination
median elevation value from all the GPS stamps in decimal degrees WGS84

Only get from the first stamp within the sets

property elevation
median elevation value from all the GPS stamps in decimal degrees WGS84

Only get from the first stamp within the sets

property end_time
start time is the first good GPS time stamp minus the seconds to the beginning of the time series.

property ex
EX

property ey
EY

find_sequence(*data_array*, *block_sequence=None*)
find a sequence in a given array

Parameters

- **data_array** (*array*) – array of the data with shape [n, m] where n is the number of seconds recorded m is the block length for a given sampling rate.
- **block_sequence** (*list*) – sequence pattern to locate *default* is [1, 131] the start of a data block.

Returns array of index locations where the sequence is found.

get_stamps(*nims_string*)
get a list of valid GPS strings and match synchronous GPRMC with GPGGA stamps if possible.
Parameters **nims_string** (*str*) – raw GPS string output by NIMS

property hx
HX

property hy
HY

property hz
HZ

property latitude
median latitude value from all the GPS stamps in decimal degrees WGS84

Only get from the GPRMC stamp as they should be duplicates

property longitude
median longitude value from all the GPS stamps in decimal degrees WGS84

Only get from the first stamp within the sets

make_dt_index(*start_time*, *sample_rate*, *stop_time=None*, *n_samples=None*)
make time index array

Note: date-time format should be YYYY-M-DDThh:mm:ss.ms UTC

Parameters

- **start_time** (*string*) – start time
- **end_time** (*string*) – end time
- **sample_rate** (*float*) – sample_rate in samples/second

match_status_with_gps_stamps(*status_array*, *gps_list*)
Match the index values from the status array with the index values of the GPS stamps. There appears to be a bit of wiggle room between when the lock is recorded and the stamp was actually recorded. This is typically 1 second and sometimes 2.
Parameters

- **status_array** (*array*) – array of status values from each data block
- **gps_list** (*list*) – list of valid GPS stamps [[GPRMC, GPGGA], . . .]

Note: I think there is a 2 second gap between the lock and the first stamp character.

---

read_nims(*fn=None*)
Read NIMS DATA.BIN file.

1. Read in the header information and stores those as attributes with the same names as in the header file.
2. Locate the beginning of the data blocks by looking for the first [1, 131, . . .] combo. Anything before that is cut out.
3. Make sure the data is a multiple of the block length, if the data is longer the extra bits are cut off.
4. Read in the GPS data (3rd byte of each block) as characters. Parses those into valid GPS stamps with appropriate index locations of where the '$' was found.

5. Read in the data as unsigned 8-bit integers and reshape the array into [N, data_block_length]. Parse this array into the status information and the data.
6. Remove duplicate blocks, by removing the first of the duplicates as suggested by Anna and Paul.
7. Match the GPS locks from the status with valid GPS stamps.
8. Check to make sure that there is the correct number of seconds between the first and last GPS stamp. The extra seconds are cut off from the end of the time series. Not sure if this is the best way to accommodate gaps in the data.

Note: The data and information array returned have the duplicates removed and the sequence reset to be monotonic.

Parameters **fn** (*str*) – full path to DATA.BIN file

Example

```
>>> from mth5.io import nims
>>> n = nims.NIMS(r"/home/mt_data/nims/mt001.bin")
```

remove_duplicates(*info_array*, *data_array*)

remove duplicate blocks, removing the first duplicate as suggested by Paul and Anna. Checks to make sure that the mag data are identical for the duplicate blocks. Removes the blocks from the information and data arrays and returns the reduced arrays. This should sync up the timing of GPS stamps and index values.

Parameters

- **info_array** (*np.array*) – structured array of block information
- **data_array** (*np.array*) – structured array of the data

Returns reduced information array

Returns reduced data array

Returns index of duplicates in raw data

property run_metadata

Run metadata

property start_time

start time is the first good GPS time stamp minus the seconds to the beginning of the time series.

property station_metadata

Station metadata from nims file

to_runts()

Get xarray for run

unwrap_sequence(*sequence*)

unwrap the sequence to be sequential numbers instead of modulated by 256. sets the first number to 0

Parameters **sequence** (*list*) – sequence of bytes numbers

Returns unwrapped number of counts

exception mth5.io.nims.**NIMSError**

Bases: Exception

class mth5.io.nims.**NIMSHeader**(*fn=None*)

Bases: object

class to hold the NIMS header information.

A typical header looks like

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>user field>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
SITE NAME: Budwieser Spring
STATE/PROVINCE: CA
COUNTRY: USA
>>> The following code in double quotes is REQUIRED to start the NIMS <<
>>> The next 3 lines contain values required for processing <<<<<<<<<<<<
>>> The lines after that are optional <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
"300b"  <-- 2CHAR EXPERIMENT CODE + 3 CHAR SITE CODE + RUN LETTER
1105-3; 1305-3  <-- SYSTEM BOX I.D.; MAG HEAD ID (if different)
106  0 <-- N-S Ex WIRE LENGTH (m); HEADING (deg E mag N)
109  90 <-- E-W Ey WIRE LENGTH (m); HEADING (deg E mag N)
1          <-- N ELECTRODE ID
3           <-- E ELECTRODE ID
2           <-- S ELECTRODE ID
4           <-- W ELECTRODE ID
Cu           <-- GROUND ELECTRODE INFO
GPS INFO: 01/10/19 16:16:42 1616.7000 3443.6088 115.7350 W 946.6
OPERATOR: KP
COMMENT: N/S CRS: .95/.96 DCV: 3.5 ACV:1
E/W CRS: .85/.86 DCV: 1.5 ACV: 1
Redeployed site for run b b/c possible animal disturbance
```

parse_header_dict(*header_dict=None*)
parse the header dictionary into something useful

read_header(*fn=None*)
read header information

Parameters **fn** (string or `pathlib.Path`) – full path to file to read

Raises *mth5.io.nims.NIMSError* if something is not right.

class mth5.io.nims.**Response**(*system_id=None*, ***kwargs*)
Bases: `object`

class for instrument response functions.

property ex_filter

property ey_filter

get_electric_high_pass(*hardware='pc'*)
get the electric high pass filter based on the hardware

property hx_filter
HX filter

property hy_filter
HY Filter

property hz_filter

exception mth5.io.nims.**ResponseError**
Bases: `Exception`

mth5.io.nims.**read_nims**(*fn*)

Parameters **fn** (*TYPE*) – DESCRIPTION
Returns DESCRIPTION

Return type TYPE

mth5.io.reader module

This is a utility function to get the appropriate reader for a given file type and return the appropriate object of *mth5.timeseries*

This setup to be like plugins but a hack cause I did not find the time to set this up properly as a true plugin.

If you are writing your own reader you need the following structure:

- Class object that will read the given file
- a reader function that is read_{file_type}, for instance read_nims
- the return value is a `mth5.timeseries.MTTS` or

mth5.timeseries.RunTS object and any extra metadata in the form of a dictionary with keys as {level.attribute}.

rubric

class NewFile

def __init__(self, fn): self.fn = fn

def read_header(self): return header_information

def read_newfile(self): ex, ey, hx, hy, hz = read_in_channels_as_MTTS return RunTS([ex, ey, hx, hy, hz])

def read_newfile(fn): new_file_obj = NewFile(fn) run_obj = new_file_obj.read_newfile()

return run_obj, extra_metadata

Then add your reader to the reader dictionary so that those files can be read.

See also:

Existing readers for some guidance.

Created on Wed Aug 26 10:32:45 2020

author Jared Peacock

license MIT

`mth5.io.reader.`**get_reader**(*extension*)

get the proper reader for file extension

Parameters **extension** (*string*) – file extension
Returns the correct function to read the file
Return type function

`mth5.io.reader.`**read_file**(*fn*, *file_type=None*)

Parameters

- **fn** (string or `pathlib.Path`) – full path to file
- **file_type** (*string*) – a specific file time if the extension is ambiguous.

Returns MT time series object
Return type `mth5.timeseries.MTTS`

mth5.io.tools module

Created on Thu Aug 6 15:18:50 2020

author Jared Peacock

license MIT

mth5.io.usgs_ascii module

Created on Thu Aug 27 16:54:09 2020

author Jared Peacock

license MIT

class mth5.io.usgs_ascii.**AsciiMetadata**(*fn=None*, ***kwargs*)

Bases: object

Container for all the important metadata in a USGS ascii file.

Attributes	Description
SurveyID	Survey name
SiteID	Site name
RunID	Run number
SiteLatitude	Site latitude in decimal degrees WGS84
SiteLongitude	Site longitude in decimal degrees WGS84
SiteElevation	Site elevation according to national map meters
AcqStartTime	Start time of station YYYY-MM-DDThh:mm:ss UTC
AcqStopTime	Stop time of station YYYY-MM-DDThh:mm:ss UTC
AcqSmpFreq	Sampling rate samples/second
AcqNumSmp	Number of samples
Nchan	Number of channels
CoordinateSystem	[Geographic North | Geomagnetic North]
ChnSettings	Channel settings, see below
MissingDataFlag	Missing data value

ChnSettings

Keys	Description
ChnNum	SiteID+channel number
ChnID	Component [ex | ey | hx | hy | hz]
InstrumentID	Data logger + sensor number
Azimuth	Setup angle of componet in degrees relative to CoordinateSystem
Dipole_Length	Dipole length in meters

property AcqNumSmp

property AcqSmpFreq

property AcqStartTime

property AcqStopTime

property Nchan

property SiteElevation

get elevation from national map

property SiteID

property SiteLatitude

property SiteLongitude

get_component_info(*comp*)

Parameters **comp** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

read_metadata(*fn=None*, *meta_lines=None*)

Read in a meta from the raw string or file. Populate all metadata as attributes.

Parameters

- **fn** (*string*) – full path to USGS ascii file
- **meta_lines** (*list*) – lines of metadata to read

write_metadata(*chn_list=['Ex', 'Ey', 'Hx', 'Hy', 'Hz']*)

Write out metadata in the format of USGS ascii.

return list of metadate lines.

Note: meant to use ‘

‘.join(lines) to write out in a file.

class mth5.io.usgs_ascii.**USGSasc**(*fn=None*, ***kwargs*)

Bases: *mth5.io.usgs_ascii.AsciiMetadata*

Read and write USGS ascii formatted time series.

Attributes	Description
ts	Pandas dataframe holding the time series data
fn	Full path to .asc file
station_dir	Full path to station directory
meta_notes	Notes of how the station was collected

Methods	Description
get_z3d_db	Get Pandas dataframe for schedule block
locate_mtft24_cfg_fn	Look for a mtft24.cfg file in station_dir
get_metadata_from_mtft24	Get metadata from mtft24.cfg file
get_metadata_from_survey_csv	Get metadata from survey csv file
fill_metadata	Fill Metadata container from a meta_array
read_asc_file	Read in USGS ascii file
convert_electrics	Convert electric channels to mV/km
write_asc_file	Write an USGS ascii file
write_station_info_metadata	Write metadata to a .cfg file

Example

```
>>> zc = Z3DCollection()
>>> fn_list = zc.get_time_blocks(z3d_path)
>>> zm = USGSasc()
>>> zm.SurveyID = 'iMUSH'
>>> zm.get_z3d_db(fn_list[0])
>>> zm.read_mtft24_cfg()
>>> zm.CoordinateSystem = 'Geomagnetic North'
>>> zm.SurveyID = 'MT'
>>> zm.write_asc_file(str_fmt='%15.7e')
>>> zm.write_station_info_metadata()
```

property electric_channels

property ex

property ey

fill_metadata(*meta_arr*)

Fill in metadata from time array made by Z3DCollection.check_time_series.

Parameters **meta_arr** (*np.ndarray*) – structured array of metadata for the Z3D files to be combined.

property hx

HX

property hy

property hz

property magnetic_channels

read_asc_file(*fn=None*)

Read in a USGS ascii file and fill attributes accordingly.

Parameters **fn** (*string*) – full path to .asc file to be read in

property run_xarray

Get xarray for run

write_asc_file(*save_fn=None*, *chunk_size=1024*, *str_fmt='%15.7e'*, *full=True*, *compress=False*, *save_dir=None*, *compress_type='zip'*, *convert_electrics=True*)

Write an ascii file in the USGS ascii format.

Parameters

- **save_fn** (*string*) – full path to file name to save the merged ascii to
- **chunck_size** (*int*) – chunck size to write file in blocks, larger numbers are typically slower.
- **str_fmt** (*string*) – format of the data as written
- **full** (*boolean [True | False]*) – write out the complete file, mostly for testing.
- **compress** (*boolean [True | False]*) – compress file
- **compress_type** (*boolean [zip | gzip]*) – compress file using zip or gzip

mth5.io.usgs_ascii.**read_ascii**(*fn*)

read USGS ASCII formatted file

Parameters **fn** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

mth5.io.zen module

Zen

- Tools for reading and writing files for Zen and processing software
- Tools for copying data from SD cards
- Tools for copying schedules to SD cards

Created on Tue Jun 11 10:53:23 2013 Updated August 2020 (JP)

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.io.zen.**Z3D**(*fn=None*, ***kwargs*)

Bases: object

Deals with the raw Z3D files output by zen. :param **fn**: full path to .Z3D file to be read in :type **fn**: string

Attributes	Description	Default Value
_block_len	length of data block to read in as chunks faster reading	65536
_counts_to_mv_conversion	conversion factor to convert counts to mv	9.53674316406e-10
_gps_bytes	number of bytes for a gps stamp	16
_gps_dtype	data type for a gps stamp	see below
_gps_epoch	starting date of GPS time format is a tuple	**(1980, 1, 6, 0,** 0, 0, -1, -1, 0)
_gps_f0	first gps flag in raw binary	
_gps_f1	second gps flag in raw binary	
_gps_flag_0	first gps flag as an int32	2147483647
_gps_flag_1	second gps flag as an int32	-2147483648
_gps_stamp_length	bit length of gps stamp	64
_leap_seconds	leap seconds, difference between UTC time and GPS time. GPS time is ahead by this much	16
_week_len	week length in seconds	604800
df	sampling rate of the data	256
fn	Z3D file name	None
gps_flag	full gps flag	_gps_f0+_gps_f1
gps_stamps	np.ndarray of gps stamps	None
header	Z3DHeader object	Z3DHeader
metadata	Z3DMetadata	Z3DMetadata
schedule	Z3DSchedule	Z3DSchedule
time_series	np.ndarra(len_data)	None
units	units in which the data is in	counts
zen_schedule	time when zen was set to run	None

- **gps_dtype is formated as np.dtype([('flag0', np.int32),** ('flag1', np.int32), ('time', np.int32), ('lat', np.float64), ('lon', np.float64), ('num_sat', np.int32), ('gps_sens', np.int32), ('temperature', np.float32), ('voltage', np.float32), ('num_fpga', np.int32), ('num_adc', np.int32), ('pps_count', np.int32), ('dac_tune', np.int32), ('block_len', np.int32)])

Example

```
>>> import mtpy.usgs.zen as zen
>>> zt = zen.Zen3D(r"/home/mt/mt00/mt00_20150522_080000_256_EX.Z3D")
>>> zt.read_z3d()
>>> ------- Reading /home/mt/mt00/mt00_20150522_080000_256_EX.Z3D ----
↪-
    --> Reading data took: 0.322 seconds
    Scheduled time was 2015-05-22,08:00:16 (GPS time)
    1st good stamp was 2015-05-22,08:00:18 (GPS time)
    difference of 2.00 seconds
    found 6418 GPS time stamps
    found 1642752 data points
>>> zt.plot_time_series()
```

property azimuth
azimuth of instrument setup

property channel_metadata
Channel metadata

property channel_number

property channel_response

check_start_time()
check to make sure the scheduled start time is similar to the first good gps stamp

property coil_num
coil number

property coil_response
Make the coile response into a FAP filter

property component
channel

convert_counts_to_mv(*data*)
convert the time series from counts to millivolts

convert_gps_time()
convert gps time integer to relative seconds from gps_week

convert_mv_to_counts(*data*)
convert millivolts to counts assuming no other scaling has been applied

property counts2mv_filter
Create a counts2mv coefficient filter

property dipole_filter

property dipole_length
dipole length

property elevation
elevation in meters

property end

property filter_metadata
Filter metadata

get_UTC_date_time(*gps_week*, *gps_time*)
get the actual date and time of measurement as UTC.

Parameters

- **gps_week** (*int*) – integer value of gps_week that the data was collected
- **gps_time** (*int*) – number of seconds from beginning of gps_week

Returns mth5.utils.mttime.MTime

get_gps_stamp_index(*ts_data*, *old_version=False*)
locate the time stamps in a given time series.

Looks for gps_flag_0 first, if the file is newer, then makes sure the next value is gps_flag_1

Returns list of gps stamps indicies

get_gps_time(*gps_int*, *gps_week=0*)
from the gps integer get the time in seconds.

Parameters

- **gps_int** (*int*) – integer from the gps time stamp line
- **gps_week** (*int*) – relative gps week, if the number of seconds is larger than a week then a week is subtracted from the seconds and computed from gps_week += 1

Returns gps_time as number of seconds from the beginning of the relative gps week.

property latitude
latitude in decimal degrees

property longitude
longitude in decimal degrees

read_all_info()
Read header, schedule, and metadata

read_z3d(*Z3Dfn=None*)
read in z3d file and populate attributes accordingly

1. Read in the entire file as chunks as np.int32.
2. Extract the gps stamps and convert accordingly. Check to make sure gps time stamps are 1 second apart and incrementing as well as checking the number of data points between stamps is the same as the sampling rate.
3. **Converts gps_stamps['time'] to seconds relative to header.gps_week** Note we skip the first two gps stamps because there is something wrong with the data there due to some type of buffering. Therefore the first GPS time is when the time series starts, so you will notice that gps_stamps[0]['block_len'] = 0, this is because there is nothing previous to this time stamp and so the 'block_len' measures backwards from the corresponding time index.
4. Put the data chunks into Pandas data frame that is indexed by time

Example

```
>>> from mth5.io import zen
>>> z_obj = zen.Z3D(r"home/mt_data/zen/mt001.z3d")
>>> z_obj.read_z3d()
```

property run_metadata
Run metadata

property sample_rate
sampling rate

property start

property station
station name

property station_metadata
station metadta

to_channelts()
fill time series object

trim_data()
apparently need to skip the first 2 seconds of data because of something to do with the SD buffer

This method will be deprecated after field testing

validate_gps_time()
make sure each time stamp is 1 second apart

validate_time_blocks()
validate gps time stamps and make sure each block is the proper length

property zen_response

property zen_schedule
zen schedule data and time

class mth5.io.zen.**Z3DHeader**(*fn=None*, *fid=None*, ***kwargs*)
Bases: object

Read in the header information of a Z3D file and make each metadata entry an attirbute.

Parameters

- **fn** (string or pathlib.Path) – full path to Z3D file
- **fid** (*file*) – file object ex. open(Z3Dfile, 'rb')

Attributes	Definition
_header_len	lenght of header in bits (512)
ad_gain	gain of channel
ad_rate	sampling rate in Hz
alt	altitude of the station (not reliable)
attenchannelsmask	not sure
box_number	ZEN box number
box_serial	ZEN box serial number
channel	channel number of the file
channelserial	serial number of the channel board
duty	duty cycle of the transmitter
fpga_buildnum	build number of one of the boards
gpsweek	GPS week
header_str	full header string
lat	latitude of station
logterminal	not sure
long	longitude of the station
main_hex_buildnum	build number of the ZEN box in hexidecimal
numsats	number of gps satelites
period	period of the transmitter
tx_duty	transmitter duty cycle
tx_freq	transmitter frequency
version	version of the firmware

Example

```
>>> import mtpy.usgs.zen as zen
>>> Z3Dfn = r"/home/mt/mt01/mt01_20150522_080000_256_EX.Z3D"
>>> header_obj = zen.Z3DHeader()
>>> header_obj.read_header()
```

convert_value(*key_string*, *value_string*)
convert the value to the appropriate units given the key

property data_logger
Data logger name as ZEN{box_number}

read_header(*fn=None*, *fid=None*)
Read the header information into appropriate attributes

Parameters

- **fn** (string or `pathlib.Path`) – full path to Z3D file
- **fid** (*file*) – file object ex. open(Z3Dfile, 'rb')

Example

```
>>> import mtpy.usgs.zen as zen
>>> Z3Dfn = r"/home/mt/mt01/mt01_20150522_080000_256_EX.Z3D"
>>> header_obj = zen.Z3DHeader()
>>> header_obj.read_header()
```

class mth5.io.zen.**Z3DMetadata**(*fn=None*, *fid=None*, ***kwargs*)
Bases: `object`

Will read in the metadata information of a Z3D file and make each metadata entry an attirbute.The attributes are left in capitalization of the Z3D file.

Parameters

- **fn** (string or `pathlib.Path`) – full path to Z3D file
- **fid** (*file*) – file object ex. open(Z3Dfile, 'rb')

Attributes	Definition
_header_length	length of header in bits (512)
_metadata_length	length of metadata blocks (512)
_schedule_metadata_len	length of schedule meta data (512)
board_cal	board calibration np.ndarray()
cal_ant	antenna calibration
cal_board	board calibration
cal_ver	calibration version
ch_azimuth	channel azimuth
ch_cmp	channel component
ch_length	channel length (or # of coil)
ch_number	channel number on the ZEN board
ch_xyz1	channel xyz location (not sure)
ch_xyz2	channel xyz location (not sure)
coil_cal	coil calibration np.ndarray (freq, amp, phase)
fid	file object
find_metadata	boolean of finding metadata
fn	full path to Z3D file

continues on next page

Table 1 – continued from previous page

Attributes	Definition
gdp_operator	operater of the survey
gdp_progver	program version
job_by	job preformed by
job_for	job for
job_name	job name
job_number	job number
m_tell	location in the file where the last metadata block was found.
rx_aspace	electrode spacing
rx_sspace	not sure
rx_xazimuth	x azimuth of electrode
rx_xyz0	not sure
rx_yazimuth	y azimuth of electrode
survey_type	type of survey
unit_length	length units (m)

Example

```
>>> import mtpy.usgs.zen as zen
>>> Z3Dfn = r"/home/mt/mt01/mt01_20150522_080000_256_EX.Z3D"
>>> header_obj = zen.Z3DMetadata()
>>> header_obj.read_metadata()
```

read_metadata(*fn=None*, *fid=None*)

read meta data

Parameters

- **fn** (*string*) – full path to file, optional if already initialized.
- **fid** (*file*) – open file object, optional if already initialized.

class mth5.io.zen.**Z3DSchedule**(*fn=None*, *fid=None*, ***kwargs*)

Bases: object

Will read in the schedule information of a Z3D file and make each metadata entry an attirbute. The attributes are left in capitalization of the Z3D file.

Parameters

- **fn** (string or pathlib.Path) – full path to Z3D file
- **fid** (*file*) – file object ex. open(Z3Dfile, 'rb')

Attributes	Definition
AutoGain	Auto gain for the channel
Comment	Any comments for the schedule
Date	Date of when the schedule action was started YYYY-MM-DD
Duty	Duty cycle of the transmitter
FFTStacks	FFT stacks from the transmitter
Filename	Name of the file that the ZEN gives it
Gain	Gain of the channel
Log	Log the data [Y | N]
NewFile	Create a new file [Y | N]
Period	Period of the transmitter
RadioOn	Turn on the radio [Y | N]
SR	Sampling Rate in Hz
SamplesPerAcq	Samples per aquisition for transmitter
Sleep	Set the box to sleep [Y | N]
Sync	Sync with GPS [Y | N]
Time	Time the schedule action started HH:MM:SS (GPS time)
_header_len	length of header in bits (512)
_schedule_metadata_len	length of schedule metadata in bits (512)
fid	file object of the file
fn	file name to read in
meta_string	string of the schedule

Example

```
>>> import mtpy.usgs.zen as zen
>>> Z3Dfn = r"/home/mt/mt01/mt01_20150522_080000_256_EX.Z3D"
>>> header_obj = zen.Z3DSchedule()
>>> header_obj.read_schedule()
```

read_schedule(*fn=None*, *fid=None*)
read meta data string

exception mth5.io.zen.**ZenGPSError**
Bases: Exception

error for gps timing

exception mth5.io.zen.**ZenInputFileError**
Bases: Exception

error for input files

exception mth5.io.zen.**ZenSamplingRateError**
Bases: Exception

error for different sampling rates

mth5.io.zen.**read_z3d**(*fn*, *logger_file_handler=None*)
generic tool to read z3d file

Module contents

14.1.3 mth5.tables package

Submodules

mth5.tables.mth5_table module

Created on Wed Dec 23 16:53:55 2020

author Jared Peacock

license MIT

class mth5.tables.mth5_table.**MTH5Table**(*hdf5_dataset*)
Bases: object

Use the underlying NumPy basics, there are simple actions in this table, if a user wants to use something more sophisticated for querying they should try using a pandas table. In this case entries in the table are more difficult to change and datatypes need to be kept track of.

add_row(*row*, *index=None*)
Add a row to the table.

row must be of the same data type as the table

Parameters

- **row** (*TYPE*) – row entry for the table
- **index** (*integer, if None is given then the row is added to the end of the array*) – index of row to add

Returns index of the row added

Return type integer

check_dtypes(*other_dtype*)
Check to make sure datatypes match

property dtype

locate(*column*, *value*, *test='eq'*)
locate index where column is equal to value :param column: DESCRIPTION :type column: TYPE :param value: DESCRIPTION :type value: TYPE :type test: type of test to try * 'eq': equals * 'lt': less than * 'le': less than or equal to * 'gt': greater than * 'ge': greater than or equal to. * 'be': between or equal to * 'bt': between

If be or bt input value as a list of 2 values

Returns DESCRIPTION

Return type TYPE

property nrows

remove_row(*index*)
Remove a row

Note: that there is not index value within the array, so the indexing is on the fly. A user should use the HDF5 reference instead of index number that is the safest and most robust method.

Parameters **index** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

This isn't as easy as just deleteing an element. Need to delete the element from the weakly referenced array and then set the summary table dataset to the new array.

So set to a null array for now until a more clever option is found.

property shape

to_dataframe()

Convert the table into a `pandas.DataFrame` object.

Returns convert table into a `pandas.DataFrame` with the appropriate data types.

Return type `pandas.DataFrame`

update_row(*entry*)

Update an entry by first locating the index and then rewriting the entry.

Parameters **entry** (*np.ndarray*) – numpy array with same datatype as the table

Returns row index.

This doesn't work because you cannot test for hdf5_reference, should use add row and locate by index.

Module contents

class mth5.tables.**MTH5Table**(*hdf5_dataset*)

Bases: `object`

Use the underlying NumPy basics, there are simple actions in this table, if a user wants to use something more sophisticated for querying they should try using a pandas table. In this case entries in the table are more difficult to change and datatypes need to be kept track of.

add_row(*row*, *index=None*)

Add a row to the table.

row must be of the same data type as the table

Parameters

- **row** (*TYPE*) – row entry for the table
- **index** (*integer, if None is given then the row is added to the end of the array*) – index of row to add

Returns index of the row added

Return type integer

check_dtypes(*other_dtype*)

Check to make sure datatypes match

property dtype

locate(*column*, *value*, *test='eq'*)

locate index where column is equal to value :param column: DESCRIPTION :type column: TYPE :param value: DESCRIPTION :type value: TYPE :type test: type of test to try * 'eq': equals * 'lt': less than * 'le': less than or equal to * 'gt': greater than * 'ge': greater than or equal to. * 'be': between or equal to * 'bt': between

If be or bt input value as a list of 2 values

Returns DESCRIPTION

Return type TYPE

property nrows

remove_row(*index*)
Remove a row

Note: that there is not index value within the array, so the indexing is on the fly. A user should use the HDF5 reference instead of index number that is the safest and most robust method.

Parameters **index** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

This isn't as easy as just deleteing an element. Need to delete the element from the weakly referenced array and then set the summary table dataset to the new array.

So set to a null array for now until a more clever option is found.

property shape

to_dataframe()
Convert the table into a `pandas.DataFrame` object.

Returns convert table into a `pandas.DataFrame` with the appropriate data types.

Return type `pandas.DataFrame`

update_row(*entry*)
Update an entry by first locating the index and then rewriting the entry.

Parameters **entry** (*np.ndarray*) – numpy array with same datatype as the table

Returns row index.

This doesn't work because you cannot test for hdf5_reference, should use add row and locate by index.

14.1.4 mth5.timeseries package

Submodules

mth5.timeseries.channel_ts module

lists and arrays that goes on, seems easiest to convert all lists to strings and then convert them back if read in.

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.timeseries.channel_ts.**ChannelTS**(*channel_type='auxiliary'*, *data=None*, *channel_metadata=None*, *station_metadata=None*, *run_metadata=None*, ***kwargs*)

Bases: `object`

Note: Assumes equally spaced samples from the start time.

The time series is stored in an `xarray.Dataset` that has coordinates of time and is a 1-D array labeled 'data'. The `xarray.Dataset` can be accessed and set from the **:attribute:`ts`**. The data is stored in :attribute:'ts.data' and the time index is a coordinate of **:attribute:`ts`**.

The time coordinate is made from the start time, sample rate and number of samples. Currently, End time is a derived property and cannot be set.

Channel time series object is based on xarray and `mth5.metadata` therefore any type of interpolation, resampling, groupby, etc can be done using xarray methods.

There are 3 metadata classes that hold important metadata

- `mth5.metadata.Station` holds information about the station
- `mth5.metadata.Run` holds information about the run the channel

belongs to. * :class`mth5.metadata.Channel` holds information specific to the channel.
This way a single channel will hold all information needed to represent the channel.

Rubric

Example

```
>>> from mth5.timeseries import ChannelTS
>>> ts_obj = ChannelTS('auxiliary')
>>> ts_obj.sample_rate = 8
>>> ts_obj.start = '2020-01-01T12:00:00+00:00'
>>> ts_obj.ts = range(4096)
>>> ts_obj.station_metadata.id = 'MT001'
>>> ts_obj.run_metadata.id = 'MT001a'
>>> ts_obj.component = 'temperature'
>>> print(ts_obj)
        Station      = MT001
        Run          = MT001a
        Channel Type = auxiliary
    Component    = temperature
        Sample Rate  = 8.0
        Start        = 2020-01-01T12:00:00+00:00
        End          = 2020-01-01T12:08:31.875000+00:00
        N Samples    = 4096
```

```
>>> p = ts_obj.ts.plot()
```

property channel_response_filter
Full channel response filter

Returns DESCRIPTION

Return type TYPE

property channel_type
Channel Type

property component

property end
MTime object

from_obspy_trace(*obspy_trace*)
Fill data from an `obspy.core.Trace`

Parameters **obspy_trace** (*obspy.core.trace*) – Obspy trace object

get_slice(*start*, *end*)
Get a slice from the time series given a start and end time.

Looks for >= start & <= end

Uses loc to be exact with milliseconds

Parameters

- **start** (*TYPE*) – DESCRIPTION
- **end** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

property has_data

check to see if there is an index in the time series

property n_samples

number of samples

resample(*dec_factor=1*, *inplace=False*)

decimate the data by using scipy.signal.decimate

Parameters **dec_factor** (*int*) – decimation factor

- refills ts.data with decimated data and replaces sample_rate

property sample_rate

sample rate in samples/second

property start

MTime object

to_obspy_trace()

Convert the time series to an `obspy.core.trace.Trace` object. This will be helpful for converting between data pulled from IRIS and data going into IRIS.

Returns DESCRIPTION

Return type TYPE

to_xarray()

Returns a `xarray.DataArray` object of the channel timeseries this way metadata from the metadata class is updated upon return.

Returns Returns a `xarray.DataArray` object of the channel timeseries
this way metadata from the metadata class is updated upon return. :rtype: `xarray.DataArray`

```
>>> import numpy as np
>>> from mth5.timeseries import ChannelTS
>>> ex = ChannelTS("electric")
>>> ex.start = "2020-01-01T12:00:00"
>>> ex.sample_rate = 16
>>> ex.ts = np.random.rand(4096)
```

property ts

mth5.timeseries.channel_ts.**make_dt_coordinates**(*start_time*, *sample_rate*, *n_samples*, *logger*)

get the date time index from the data

Parameters

- **start_time** (*string*) – start time in time format
- **sample_rate** (*float*) – sample rate in samples per seconds
- **n_samples** (*int*) – number of samples in time series

:param `logging.logger` logger: logger class object

Returns date-time index

mth5.timeseries.run_ts module

lists and arrays that goes on, seems easiest to convert all lists to strings and then convert them back if read in.

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.timeseries.run_ts.**RunTS**(*array_list=None*, *run_metadata=None*, *station_metadata=None*)
Bases: object

holds all run ts in one aligned array

components –> {'ex': ex_xarray, 'ey': ey_xarray}

add_channel(*channel*)
Add a channel to the dataset, can be an xarray.DataArray or *mth5.timeseries.ChannelTS* object.

Need to be sure that the coordinates and dimensions are the same as the existing dataset, namely coordinates are time, and dimensions are the same, if the dimesions are larger than the existing dataset then the added channel will be clipped to the dimensions of the existing dataset.

If the start time is not the same nan's will be placed at locations where the timing does not match the current start time. This is a feature of xarray.

Parameters **channel** (xarray.DataArray or *mth5.timeseries.ChannelTS*) – a channel xarray or ChannelTS to add to the run

property channels

property dataset

property end

from_obspy_stream(*obspy_stream*)
Get a run from an obspy.core.stream which is a list of obspy.core.Trace objects.

Parameters **obspy_stream** (obspy.core.Stream) – Obspy Stream object

property has_data
check to see if there is data

plot()
plot the time series probably slow for large data sets

Returns DESCRIPTION

Return type TYPE

property sample_rate

set_dataset(*array_list*, *align_type='outer'*)

Parameters

- **array_list** (list of *mth5.timeseries.ChannelTS* objects) – list of xarrays
- **align_type** (*string*) – how the different times will be aligned * 'outer': use the union of object indexes * 'inner': use the intersection of object indexes * 'left': use indexes from the first object with each dimension * 'right': use indexes from the last object with each dimension * 'exact': instead of aligning, raise ValueError when indexes to be aligned are not equal * 'override': if indexes are of same size, rewrite indexes to be those of the first object with that dimension. Indexes for the same dimension must have the same size in all objects.

property start

property summarize_metadata
Get a summary of all the metadata
Returns DESCRIPTION

Return type TYPE

to_obspy_stream()
convert time series to an `obspy.core.Stream` which is like a list of `obspy.core.Trace` objects.
Returns An Obspy Stream object from the time series data

Return type `obspy.core.Stream`

validate_metadata()
Check to make sure that the metadata matches what is in the data set.

updates metadata from the data.

Check the start and end times, channels recorded :return: DESCRIPTION :rtype: TYPE

Module contents

class mth5.timeseries.**ChannelTS**(*channel_type='auxiliary'*, *data=None*, *channel_metadata=None*, *station_metadata=None*, *run_metadata=None*, ***kwargs*)
Bases: `object`

Note: Assumes equally spaced samples from the start time.

The time series is stored in an `xarray.Dataset` that has coordinates of time and is a 1-D array labeled 'data'. The `xarray.Dataset` can be accessed and set from the **:attribute:`ts`**. The data is stored in :attribute:'ts.data' and the time index is a coordinate of **:attribute:`ts`**.

The time coordinate is made from the start time, sample rate and number of samples. Currently, End time is a derived property and cannot be set.

Channel time series object is based on xarray and `mth5.metadata` therefore any type of interpolation, resampling, groupby, etc can be done using xarray methods.

There are 3 metadata classes that hold important metadata
- `mth5.metadata.Station` holds information about the station
- `mth5.metadata.Run` holds information about the run the channel

belongs to. * :class`mth5.metadata.Channel` holds information specific to the channel.
This way a single channel will hold all information needed to represent the channel.
Rubric

Example

```
>>> from mth5.timeseries import ChannelTS
>>> ts_obj = ChannelTS('auxiliary')
>>> ts_obj.sample_rate = 8
>>> ts_obj.start = '2020-01-01T12:00:00+00:00'
>>> ts_obj.ts = range(4096)
>>> ts_obj.station_metadata.id = 'MT001'
>>> ts_obj.run_metadata.id = 'MT001a'
>>> ts_obj.component = 'temperature'
```

(continues on next page)

(continued from previous page)

```
>>> print(ts_obj)
        Station      = MT001
        Run          = MT001a
        Channel Type = auxiliary
    Component    = temperature
        Sample Rate  = 8.0
        Start        = 2020-01-01T12:00:00+00:00
        End          = 2020-01-01T12:08:31.875000+00:00
        N Samples    = 4096
```

```
>>> p = ts_obj.ts.plot()
```

property channel_response_filter
Full channel response filter

Returns DESCRIPTION

Return type TYPE

property channel_type
Channel Type

property component

property end
MTime object

from_obspy_trace(*obspy_trace*)
Fill data from an `obspy.core.Trace`

Parameters **obspy_trace** (*obspy.core.trace*) – Obspy trace object

get_slice(*start*, *end*)
Get a slice from the time series given a start and end time.

Looks for >= start & <= end

Uses loc to be exact with milliseconds

Parameters

- **start** (*TYPE*) – DESCRIPTION
- **end** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

property has_data
check to see if there is an index in the time series

property n_samples
number of samples

resample(*dec_factor=1*, *inplace=False*)
decimate the data by using scipy.signal.decimate

Parameters **dec_factor** (*int*) – decimation factor

- refills ts.data with decimated data and replaces sample_rate

property sample_rate
sample rate in samples/second

property start
MTime object

to_obspy_trace()
Convert the time series to an obspy.core.trace.Trace object. This will be helpful for converting between data pulled from IRIS and data going into IRIS.
Returns DESCRIPTION

Return type TYPE

to_xarray()
Returns a xarray.DataArray object of the channel timeseries this way metadata from the metadata class is updated upon return.
Returns Returns a xarray.DataArray object of the channel timeseries
this way metadata from the metadata class is updated upon return. :rtype: xarray.DataArray

```
>>> import numpy as np
>>> from mth5.timeseries import ChannelTS
>>> ex = ChannelTS("electric")
>>> ex.start = "2020-01-01T12:00:00"
>>> ex.sample_rate = 16
>>> ex.ts = np.random.rand(4096)
```

property ts

class mth5.timeseries.**RunTS**(*array_list=None*, *run_metadata=None*, *station_metadata=None*)
Bases: object

holds all run ts in one aligned array

components –> {'ex': ex_xarray, 'ey': ey_xarray}

add_channel(*channel*)
Add a channel to the dataset, can be an xarray.DataArray or *mth5.timeseries.ChannelTS* object.

Need to be sure that the coordinates and dimensions are the same as the existing dataset, namely coordinates are time, and dimensions are the same, if the dimesions are larger than the existing dataset then the added channel will be clipped to the dimensions of the existing dataset.

If the start time is not the same nan's will be placed at locations where the timing does not match the current start time. This is a feature of xarray.
Parameters **channel** (xarray.DataArray or *mth5.timeseries.ChannelTS*) – a channel xarray or ChannelTS to add to the run

property channels

property dataset

property end

from_obspy_stream(*obspy_stream*)
Get a run from an obspy.core.stream which is a list of obspy.core.Trace objects.
Parameters **obspy_stream** (obspy.core.Stream) – Obspy Stream object

property has_data
check to see if there is data

plot()
plot the time series probably slow for large data sets
Returns DESCRIPTION

Return type TYPE

property sample_rate

set_dataset(*array_list*, *align_type='outer'*)

Parameters

- **array_list** (list of *mth5.timeseries.ChannelTS* objects) – list of xarrays
- **align_type** (*string*) – how the different times will be aligned * 'outer': use the union of object indexes * 'inner': use the intersection of object indexes * 'left': use indexes from the first object with each dimension * 'right': use indexes from the last object with each dimension * 'exact': instead of aligning, raise ValueError when indexes to be aligned are not equal * 'override': if indexes are of same size, rewrite indexes to be those of the first object with that dimension. Indexes for the same dimension must have the same size in all objects.

property start

property summarize_metadata

Get a summary of all the metadata

Returns DESCRIPTION

Return type TYPE

to_obspy_stream()

convert time series to an obspy.core.Stream which is like a list of obspy.core.Trace objects.

Returns An Obspy Stream object from the time series data

Return type obspy.core.Stream

validate_metadata()

Check to make sure that the metadata matches what is in the data set.

updates metadata from the data.

Check the start and end times, channels recorded :return: DESCRIPTION :rtype: TYPE

14.1.5 mth5.utils package

Submodules

mth5.utils.exceptions module

Exceptions raised by MTH5

Created on Wed May 13 19:07:21 2020

@author: jpeacock

exception mth5.utils.exceptions.**MTH5Error**

Bases: Exception

exception mth5.utils.exceptions.**MTH5TableError**

Bases: Exception

exception mth5.utils.exceptions.**MTSchemaError**

Bases: Exception

exception mth5.utils.exceptions.**MTTSError**

Bases: Exception

exception mth5.utils.exceptions.**MTTimeError**
Bases: Exception

mth5.utils.fdsn_tools module

Tools for FDSN standards

Created on Wed Sep 30 11:47:01 2020

author Jared Peacock

license MIT

mth5.utils.fdsn_tools.**get_location_code**(*channel_obj*)
Get the location code given the components and channel number
Parameters **channel_obj** (Channel) – Channel object
Returns 2 character location code
Return type string

mth5.utils.fdsn_tools.**get_measurement_code**(*measurement*)
get SEED sensor code given the measurement type
Parameters **measurement** (*string*) – measurement type, e.g. * temperature * electric * magnetic
Returns single character SEED sensor code, if the measurement type has not been defined yet Y is returned.
Return type string

mth5.utils.fdsn_tools.**get_orientation_code**(*azimuth*, *orientation='horizontal'*)
Get orientation code given angle and orientation. This is a general code and the true azimuth is stored in channel
Parameters **azimuth** (*float*) – angel assuming 0 is north, 90 is east, 0 is vertical down
Returns single character SEED orientation code
Return type string

mth5.utils.fdsn_tools.**get_period_code**(*sample_rate*)
Get the SEED sampling rate code given a sample rate
Parameters **sample_rate** (*float*) – sample rate in samples per second
Returns single character SEED sampling code
Return type string

mth5.utils.fdsn_tools.**make_channel_code**(*channel_obj*)
Make the 3 character SEED channel code
Parameters **channel_obj** (Channel) – Channel metadata
Returns 3 character channel code
Type string

mth5.utils.fdsn_tools.**read_channel_code**(*channel_code*)
read FDSN channel code
Parameters **channel_code** (*TYPE*) – DESCRIPTION
Returns DESCRIPTION
Return type TYPE

mth5.utils.mth5_logger module

Logger

Setup a logger with two handlers to remove redundancy between logs entries One is a stream handler for any messages to the console. The other is either a file handler or a null handler.

mth5.utils.mth5_logger.**load_logging_config**(*config_fn=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/mth5/*
configure/setup the logging according to the input configfile

Parameters **configfile** – .yml, .ini, .conf, .json, .yaml.

Its default is the logging.yml located in the same dir as this module. It can be modofied to use env variables to search for a log config file.

mth5.utils.mth5_logger.**setup_logger**(*logger_name*, *fn=None*, *level='debug'*)
Create a logger, can write to a separate file. This will write to the logs folder in the mt_metadata directory.

Parameters

- **logger_name** (*string*) – name of the logger, typically __name__
- **fn** (*TYPE, optional*) – file name to write to, defaults to None
- **level** (*TYPE, optional*) – DESCRIPTION, defaults to "debug"

Returns DESCRIPTION
Return type TYPE

mth5.utils.stationxml_translator module

This module provides tools to convert MT metadata to a StationXML file.

Created on Tue Jun 9 19:53:32 2020

@author: jpeacock

class mth5.utils.stationxml_translator.**MTToStationXML**(*inventory_object=None*)
Bases: object

Translate MT metadata to StationXML using Obspy Inventory classes.

Any metadata that does not fit under the StationXML schema will be added as extra metadata in the namespace MT.

MT metadata is mapped into StationXML

```
>>> # inventory mapping
Inventory
===========
  |--> Network (MT Survey)
  --------------
    |--> Station (MT Station)
    -------------
      |--> Channel (MT Channel + MT Run)
      -------------
```

Example

```
>>> from mth5.utils import translator
>>> from mth import metadata
>>> # survey_dict = metadata for survey
```

(continues on next page)

(continued from previous page)

```
>>> mt2xml = translator.MTToStationXML()
>>> mt_survey = metadata.Survey()
>>> mt_survey.from_dict(survey_dict)
>>> mt2xml.add_network(mt_survey)
```

Add a station from an xml file with root <station>

```
>>> from xml.etree import ElementTree as et
>>> mt_station = metadata.Station()
>>> mt_station.from_xml(et.parse("mt_station_xml_fn.xml").getroot())
>>> mt2xml.add_station(mt_station)
```

Add a channel from an json files with {channel:{}} and {run:{}} format

```
>>> import json
>>> mt_electric = metadata.Electric()
>>> with open("electric_json_fn.json", 'r') as fid:
>>> ... mt_electric.from_json(json.load(fid))
>>> mt_run = metadata.Run()
>>> with open("run_json_fn.json", 'r') as fid:
>>> ... mt_run.from_json(json.load(fid))
>>> mt2xml.add_channel(mt_electric, mt_run, mt_station.archive_id)
```

add_channel(*mt_channel*, *mt_run*, *station*, *network_code=None*)

Add a station from a MT channel and run objects. The run object is needed to fill the datalogger information.

Will fill the appropriate metadata in Inventory.Network[network].station[station], any metadata that does not fit within the StationXML schema will be added as extra.

Parameters

- **mt_channel** (Channel, Electric, Magnetic, Auxiliary) – MT channel metadata
- **mt_run** – MT run metadata
- **station** (*5 character string*) – Station name to add the channel to
- **network_code** (*2 character code. optional*) – Network code that the station belongs to. Defaults to None which will use Inventory.networks[0]

Dtype mt_run Run

add_network(*mt_survey_obj*)

Add a network from an MT survey object. Will fill the appropriate metadata in Inventory.Network, any metadata that does not fit within the StationXML schema will be added as extra.

Parameters **mt_survey_obj** (Survey) – MT Survey metadata

add_station(*mt_station_obj*, *network_code=None*)

Add a station from an MT station object. Will fill the appropriate metadata in Inventory.Network[network].station, any metadata that does not fit within the StationXML schema will be added as extra.

Parameters

- **mt_station_obj** (Station) – MT station metadata
- **network_code** (*2 character code. optional*) – Network code that the station belongs to. Defaults to None which will use Inventory.networks[0]

find_network_index(*network*)
locate the index of given network
Parameters **network** (*2 character string*) – name of the network to look for
Returns index of network in inventory.networks
Return type int

find_station_index(*station*, *network=None*)
locate the index of given station in Inventory.networks[network].stations
Parameters
- **station** (*5 character string*) – 5 character SEED station name
- **network** (*2 character string, optional*) – Network name, defaults to None which will use Inventory.networks[0]

Returns Index of station in Inventory.networks[network].stations
Return type int

to_stationxml(*station_xml_fn*)
Write a StationXML file using Inventory.write
Parameters **station_xml_fn** (*string or Path*) – Full path to StationXML file
Returns path to StationXML
Return type Path

class mth5.utils.stationxml_translator.**StationXMLToMTH5**
Bases: object

Translate a station XML to MT metadata standards

class mth5.utils.stationxml_translator.**XMLNetworkMTSurvey**
Bases: object

translate back and forth between StationXML Network and MT Survey

network_to_survey(*network*)
Translate a StationXML Network object to MT Survey object
Parameters **network** (obspy.core.inventory.Network) – StationXML network element

mth5.utils.stationxml_translator.**add_custom_element**(*obj*, *custom_name*, *custom_value*, *units=None*, *attribs=None*, *namespace='MT'*)
Add a custom MT element to Obspy Inventory object
Parameters
- **obj** (Inventory) – Inventory object that will have the element added
- **custom_key** (*str*) – name of custom element, if the key has a '.' it will be recursively split to assure proper nesting.
- **custom_value** (*[int | float | string]*) – value of custom element

Example

```
>>> from obspy.core import inventory
>>> from obspy.util import AttribDict()
>>> channel_01 = inventory.Channel('SQE', "", 39.0, -112.0, 150, 0,
```

... azimuth=90, ... sample_rate=256, dip=0, ... types=['ELECTRIC POTENTIAL'])
>>> # add custom element >>> channel_01.extra = AttribDict({'namespace':'MT'})

>>> channel_01.extra.FieldNotes = AttribDict({'namespace':'MT'}) >>> channel_01.extra.FieldNotes.value = AttribDict({'namespace':'MT'}) >>> channel_01.extra.FieldNotes = add_custom_element(>>>... channel_01.extra.FieldNotes, >>>... 'ContactResistanceA', >>>... 1.2, >>>... units='kOhm')

mth5.utils.stationxml_translator.**flip_dict**(*original_dict*)

Flip keys and values of the dictionary

Parameters **original_dict** (*TYPE*) – DESCRIPTION
Returns DESCRIPTION
Return type TYPE

mth5.utils.stationxml_translator.**inventory_network_to_mt_survey**(*network_obj*)

Convert an inventory.Network oject to an `mth5.metadata.Survey` object

Parameters **network_obj** (*TYPE*) – DESCRIPTION
Returns DESCRIPTION
Return type TYPE

mth5.utils.stationxml_translator.**inventory_station_to_mt_station**(*inv_station_obj*)

Parameters **inv_station_obj** (*TYPE*) – DESCRIPTION
Returns DESCRIPTION
Return type TYPE

mth5.utils.stationxml_translator.**mt_channel_to_inventory_channel**(*channel_obj*, *run_obj*, *namespace*)

Translate MT channel metadata to inventory channel

Metadata that does not fit under StationXML schema is added as extra.

Parameters

- **channel_obj** (`Channel`, `Electric`, `Magnetic`, `Auxiliary`,) – MT channel metadata
- **run_obj** (`Run`) – MT run metadata to get data logger information

Returns StationXML channel
Return type `Channel`

mth5.utils.stationxml_translator.**mt_station_to_inventory_station**(*station_obj*, *namespace='MT'*)

Translate MT station metadata to inventory station

Metadata that does not fit under StationXML schema is added as extra.

Parameters **station_obj** (`Station`) – MT station metadata
Returns StationXML Station element
Return type `Station`

mth5.utils.stationxml_translator.**mt_survey_to_inventory_network**(*survey_obj*, *namespace='MT'*)

Translate MT survey metadata to inventory Network in StationXML

Metadata that does not fit under StationXML schema is added as extra.

Parameters **survey_obj** (`Survey`) – MT survey metadata
Returns DESCRIPTION
Return type TYPE

mth5.utils.stationxml_translator.**read_comment**(*inv_comment*)

Parameters **inv_comment** (*TYPE*) – DESCRIPTION
Returns DESCRIPTION
Return type TYPE

Module contents

14.2 Submodules

14.3 mth5.helpers module

Helper functions for HDF5

Created on Tue Jun 2 12:37:50 2020

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

mth5.helpers.**close_open_files**()

mth5.helpers.**get_tree**(*parent*)
Simple function to recursively print the contents of an hdf5 group :param parent: HDF5 (sub-)tree to print :type parent: h5py.Group

mth5.helpers.**inherit_doc_string**(*cls*)

mth5.helpers.**recursive_hdf5_tree**(*group*, *lines=[]*)

mth5.helpers.**to_numpy_type**(*value*)
Need to make the attributes friendly with Numpy and HDF5.

For numbers and bool this is straight forward they are automatically mapped in h5py to a numpy type.

But for strings this can be a challenge, especially a list of strings.

HDF5 should only deal with ASCII characters or Unicode. No binary data is allowed.

mth5.helpers.**validate_compression**(*compression*, *level*)
validate that the input compression is supported.

Parameters

- **compression** (*string, ['lzf' | 'gzip' | 'szip' | None]*) – type of lossless compression
- **level** (*string for 'szip' or int for 'gzip'*) – compression level if supported

Returns compression type
Return type string
Returns compressiong level
Return type string for 'szip' or int for 'gzip'
Raises ValueError if comporession or level are not supported
Raises TypeError if compression level is not a string

14.4 mth5.mth5 module

14.4.1 MTH5

MTH5 deals with reading and writing an MTH5 file, which are HDF5 files developed for magnetotelluric (MT) data. The code is based on h5py and therefor numpy. This is the simplest and we are not really dealing with large tables of data to warrant using pytables.

Created on Sun Dec 9 20:50:41 2018

copyright Jared Peacock (jpeacock@usgs.gov)

license MIT

class mth5.mth5.**MTH5**(*filename=None*, *compression='gzip'*, *compression_opts=9*, *shuffle=True*, *fletcher32=True*, *data_level=1*)

Bases: object

MTH5 is the main container for the HDF5 file format developed for MT data

It uses the metadata standards developled by the IRIS PASSCAL software group and defined in the metadata documentation.

MTH5 is built with h5py and therefore numpy. The structure follows the different levels of MT data collection: Survey

|_Reports |_Standards |_Filters |_Station

|_**Run** |_Channel

All timeseries data are stored as individual channels with the appropriate metadata defined for the given channel, i.e. electric, magnetic, auxiliary.

Each level is represented as a mth5 group class object which has methods to add, remove, and get a group from the level below. Each group has a metadata attribute that is the approprate metadata class object. For instance the SurveyGroup has an attribute metadata that is a `mth5.metadata.Survey` object. Metadata is stored in the HDF5 group attributes as (key, value) pairs.

All groups are represented by their structure tree and can be shown at any time from the command line.

Each level has a summary array of the contents of the levels below to hopefully make searching easier.

Parameters

- **filename** (string or `pathlib.Path`) – name of the to be or existing file
- **compression** – compression type. Supported lossless compressions are * 'lzf' - Available with every installation of h5py

  (C source code also available). Low to moderate compression, very fast. No options.

  - **'gzip' - Available with every installation of HDF5,** so it's best where portability is required. Good compression, moderate speed. compression_opts sets the compression level and may be an integer from 0 to 9, default is 3.
  - **'szip' - Patent-encumbered filter used in the NASA** community. Not available with all installations of HDF5 due to legal reasons. Consult the HDF5 docs for filter options.

- **compression_opts** (*string or int depending on compression type*) – compression options, see above
- **shuffle** (*boolean*) – Block-oriented compressors like GZIP or LZF work better when presented with runs of similar values. Enabling the shuffle filter rearranges the bytes in the chunk and may improve compression ratio. No significant speed penalty, lossless.
- **fletcher32** (*boolean*) – Adds a checksum to each chunk to detect data corruption. Attempts to read corrupted chunks will fail with an error. No significant speed penalty. Obviously shouldn't be used with lossy compression filters.
- **data_level** (*integer, defaults to 1*) – level the data are stored following levels defined by NASA ESDS
  - 0 - Raw data

  - 1 - Raw data with response information and full metadata
  - 2 - Derived product, raw data has been manipulated

Usage

- Open a new file and show initialized file

```
>>> from mth5 import mth5
>>> mth5_obj = mth5.MTH5()
>>> # Have a look at the dataset options
>>> mth5.dataset_options
{'compression': 'gzip',
 'compression_opts': 3,
 'shuffle': True,
 'fletcher32': True}
>>> mth5_obj.open_mth5(r"/home/mtdata/mt01.mth5", 'w')
>>> mth5_obj
/:
====================
    |- Group: Survey
    ----------------
        |- Group: Filters
        -----------------
            --> Dataset: summary
            ......................
        |- Group: Reports
        -----------------
            --> Dataset: summary
            ......................
        |- Group: Standards
        -------------------
            --> Dataset: summary
            ......................
        |- Group: Stations
        ------------------
            --> Dataset: summary
            ......................
```

- Add metadata for survey from a dictionary

```
>>> survey_dict = {'survey':{'acquired_by': 'me', 'archive_id': 'MTCND'}}
>>> survey = mth5_obj.survey_group
>>> survey.metadata.from_dict(survey_dict)
>>> survey.metadata
{
"survey": {
    "acquired_by.author": "me",
    "acquired_by.comments": null,
    "archive_id": "MTCND"
    ...}
}
```

- Add a station from the convenience function

```
>>> station = mth5_obj.add_station('MT001')
>>> mth5_obj
/:
====================
    |- Group: Survey
    ----------------
        |- Group: Filters
        -----------------
            --> Dataset: summary
            ......................
        |- Group: Reports
        -----------------
            --> Dataset: summary
            ......................
        |- Group: Standards
        -------------------
            --> Dataset: summary
            ......................
        |- Group: Stations
        ------------------
            |- Group: MT001
            ---------------
                --> Dataset: summary
                ......................
            --> Dataset: summary
            ......................
>>> station
/Survey/Stations/MT001:
====================
    --> Dataset: summary
    ......................
```

```
>>> data.schedule_01.ex[0:10] = np.nan
>>> data.calibration_hx[...] = np.logspace(-4, 4, 20)
```

Note: if replacing an entire array with a new one you need to use [...] otherwise the data will not be updated.

Warning: You can only replace entire arrays with arrays of the same size. Otherwise you need to delete the existing data and make a new dataset.

See also:

https://www.hdfgroup.org/ and https://www.h5py.org/

add_channel(*station_name*, *run_name*, *channel_name*, *channel_type*, *data*, *channel_metadata=None*)

Convenience function to add a channel using `mth5.stations_group.get_station().get_run().add_channel()`

add a channel to a given run for a given station

Parameters

- **station_name** (*string*) – existing station name

- **run_name** (*string*) – existing run name
- **channel_name** (*string*) – name of the channel
- **channel_type** (*string*) – [electric | magnetic | auxiliary]
- **channel_metadata** ([mth5.metadata.Electric | mth5.metadata.Magnetic | mth5.metadata.Auxiliary], optional) – metadata container, defaults to None

Raises *MTH5Error* – If channel type is not correct

Returns Channel container

Return type [mth5.mth5_groups.ElectricDatset | mth5.mth5_groups.MagneticDatset | mth5.mth5_groups.AuxiliaryDatset]

Example

```
>>> new_channel = mth5_obj.add_channel('MT001', 'MT001a''Ex',
>>> ...                                'electric', None)
>>> new_channel
Channel Electric:
-------------------
                component:        None
        data type:        electric
        data format:      float32
        data shape:       (1,)
        start:            1980-01-01T00:00:00+00:00
        end:              1980-01-01T00:00:00+00:00
        sample rate:      None
```

add_run(*station_name*, *run_name*, *run_metadata=None*)

Convenience function to add a run using mth5.stations_group.get_station(station_name).add_run()

Add a run to a given station.

Parameters

- **run_name** (*string*) – run name, should be archive_id{a-z}
- **metadata** (mth5.metadata.Station, optional) – metadata container, defaults to None

Example

```
>>> new_run = mth5_obj.add_run('MT001', 'MT001a')
```

add_station(*name*, *station_metadata=None*)

Convenience function to add a station using mth5.stations_group.add_station
Add a station with metadata if given with the path: /Survey/Stations/station_name
If the station already exists, will return that station and nothing is added.

Parameters

- **station_name** (*string*) – Name of the station, should be the same as metadata.archive_id
- **station_metadata** (mth5.metadata.Station, optional) – Station metadata container, defaults to None

Returns A convenience class for the added station

Return type mth5_groups.StationGroup

Example

```
>>> new_staiton = mth5_obj.add_station('MT001')
```

close_mth5()
close mth5 file to make sure everything is flushed to the file

property data_level
data level

property dataset_options
summary of dataset options

property file_type
File Type should be MTH5

property file_version
mth5 file version

property filename
file name of the hdf5 file

property filters_group
Convenience property for /Survey/Filters group

from_experiment(*experiment*, *survey_index=0*)
Fill out an MTH5 from a mt_metadata.timeseries.Experiment object given a survey_id

Parameters

- **experiment** (mt_metadata.timeseries.Experiment) – Experiment metadata
- **survey_index** (*int, defaults to 0*) – Index of the survey to write

from_reference(*h5_reference*)
Get an HDF5 group, dataset, etc from a reference

Parameters **h5_reference** (*TYPE*) – DESCRIPTION

Returns DESCRIPTION

Return type TYPE

get_channel(*station_name*, *run_name*, *channel_name*)
Convenience function to get a channel using mth5.stations_group.get_station().get_run().get_channel()

Get a channel from an existing name. Returns the appropriate container.

Parameters

- **station_name** (*string*) – existing station name
- **run_name** (*string*) – existing run name
- **channel_name** (*string*) – name of the channel

Returns Channel container

Return type [mth5.mth5_groups.ElectricDatset | mth5.mth5_groups.MagneticDatset | mth5.mth5_groups.AuxiliaryDatset]

Raises *MTH5Error* – If no channel is found

Example

```
>>> existing_channel = mth5_obj.get_channel(station_name,
>>> ...                                     run_name,
>>> ...                                     channel_name)
>>> existing_channel
Channel Electric:
-------------------
                component:        Ex
        data type:        electric
        data format:      float32
        data shape:       (4096,)
        start:            1980-01-01T00:00:00+00:00
        end:              1980-01-01T00:00:01+00:00
        sample rate:      4096
```

get_run(*station_name*, *run_name*)

Convenience function to get a run using `mth5.stations_group.get_station(station_name).get_run()`

get a run from run name for a given station

Parameters

- **station_name** (*string*) – existing station name
- **run_name** (*string*) – existing run name

Returns Run object

Return type `mth5.mth5_groups.RunGroup`

Example

```
>>> existing_run = mth5_obj.get_run('MT001', 'MT001a')
```

get_station(*station_name*)

Convenience function to get a station using `mth5.stations_group.get_station`

Get a station with the same name as station_name

Parameters **station_name** (*string*) – existing station name

Returns convenience station class

Return type `mth5.mth5_groups.StationGroup`

Raises ***MTH5Error*** – if the station name is not found.

Example

```
>>> existing_staiton = mth5_obj.get_station('MT001')
MTH5Error: MT001 does not exist, check station_list for existing names
```

h5_is_write()

check to see if the hdf5 file is open and writeable

has_group(*group_name*)

Check to see if the group name exists

open_mth5(*filename=None*, *mode='a'*)

open an mth5 file

Returns Survey Group

Type *groups.SurveyGroup*

Example

```
>>> from mth5 import mth5
>>> mth5_object = mth5.MTH5()
>>> survey_object = mth5_object.open_mth5('Test.mth5', 'w')
```

remove_channel(*station_name*, *run_name*, *channel_name*)

Convenience function to remove a channel using `mth5.stations_group.get_station().get_run().remove_channel()`

Remove a channel from a given run and station.

Note: Deleting a channel is not as simple as del(channel). In HDF5 this does not free up memory, it simply removes the reference to that channel. The common way to get around this is to copy what you want into a new file, or overwrite the channel.

Parameters

- **station_name** (*string*) – existing station name
- **run_name** (*string*) – existing run name
- **channel_name** (*string*) – existing station name

Example

```
>>> mth5_obj.remove_channel('MT001', 'MT001a', 'Ex')
```

remove_run(*station_name*, *run_name*)

Convenience function to add a run using `mth5.stations_group.get_station(station_name).remove_run()`

Remove a run from the station.

Note: Deleting a run is not as simple as del(run). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the run.

Parameters

- **station_name** (*string*) – existing station name
- **run_name** (*string*) – existing run name

Example

```
>>> mth5_obj.remove_station('MT001', 'MT001a')
```

remove_station(*station_name*)

Convenience function to remove a station using `mth5.stations_group.remove_station`

Remove a station from the file.

Note: Deleting a station is not as simple as del(station). In HDF5 this does not free up memory, it simply removes the reference to that station. The common way to get around this is to copy what you want into a new file, or overwrite the station.

Parameters **station_name** (*string*) – existing station name

Example

```
>>> mth5_obj.remove_station('MT001')
```

property reports_group
Convenience property for /Survey/Reports group

property software_name
software name that wrote the file

property standards_group
Convenience property for /Survey/Standards group

property station_list
list of existing stations names

property stations_group
Convenience property for /Survey/Stations group

property survey_group
Convenience property for /Survey group

to_experiment()
Create an `mt_metadata.timeseries.Experiment` object from the metadata contained in the MTH5 file.

Returns `mt_metadata.timeseries.Experiment`

validate_file()
Validate an open mth5 file

will test the attribute values and group names

Returns Boolean [True = valid, False = not valid]

Return type Boolean

14.5 Module contents

Top-level package for MTH5.

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

t

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z