
tsdat

Release 0.2.2

Carina Lansing, Maxwell Levin

Jan 17, 2022

CONTENTS

1	Prerequisites	3
2	Installation	5
3	Getting a Tsdats Pipeline Template	7
4	Running Your tsdat Pipeline	9
5	Configuring Tsdats	11
5.1	Configuration Files	12
5.1.1	Storage Config	12
5.1.2	Pipeline Config	14
5.2	Code Customizations	26
5.2.1	IngestPipeline Code Hooks	27
5.2.2	File Handlers	27
5.2.3	Converters	27
5.2.4	Quality Management	28
6	Examples and Tutorials	29
6.1	Examples	29
6.2	Tutorials	29
6.2.1	Local Data Ingest	29
7	API Reference	39
7.1	tsdat	39
7.1.1	Subpackages	39
7.1.2	Package Contents	132
8	Collaboration	161
8.1	Issues	161
8.1.1	Submit tsdat Issue	161
8.2	Contributing	161
8.2.1	Submit tsdat Pull Request	161
9	Acknowledgements	163
10	Tsdats	165
10.1	Quick Overview	165
10.2	Motivation	166
	Python Module Index	167

To get started developing a tsdat pipeline, we suggest following the following steps, which are explained in more detail in the linked sections:

1. *[Install tsdat](#)*
2. *[Get a template](#)*
3. *[Configure template](#)*
4. *[Run pipeline](#)*

PREREQUISITES

Tsdat requires [Python 3.8+](#)

INSTALLATION

You can install tsdat simply by running `pip install tsdat` in a console window.

GETTING A TSDAT PIPELINE TEMPLATE

The quickest way to set up a Tsdats pipeline is to use a GitHub repository template. You can find a list of template repositories for tsdat at <https://github.com/tsdat/template-repositories>.

Note: Currently, there are only two ingest templates available, but more will be added over time, including support for VAPs, multi-pipeline templates, and specific data models.

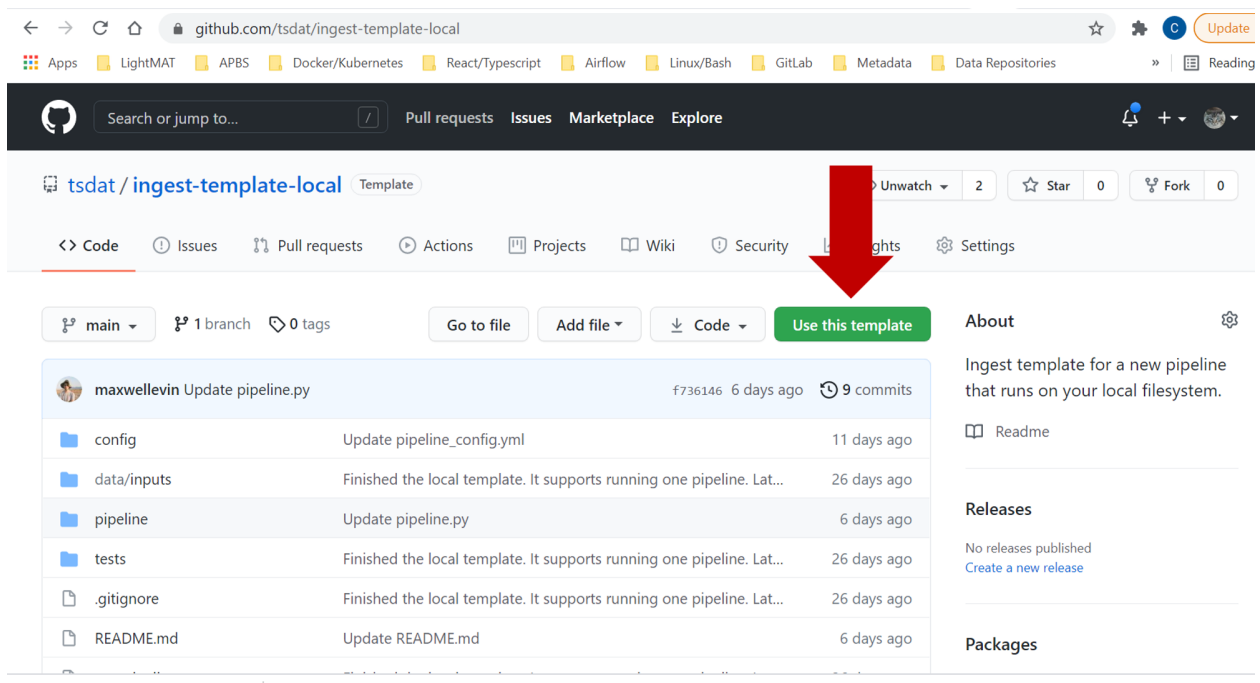
1. Local Ingest Template

Use this template to run ingest pipelines on your local computer.

2. AWS Ingest Template

Use this template to run ingest pipelines on AWS. (It requires an AWS account.)

Once you have selected the template to use, select the “Use this template” button to create a new repository at your specified location with the template contents.



Once you have created a new repository from the template, you can clone your repository to your local desktop and start developing. By default, the repository template will come pre-configured to run out-of-the-box on an example dataset.

See [configuring your pipeline](#) for more information on tsdat-specific configuration file and code customizations. In addition, make sure to read the **README.md** file associated with your template for any template-specific instructions.

RUNNING YOUR TSDAT PIPELINE

Once tsdat is installed and your pipeline template is configured, you can run it on your input data using the following code from a terminal window at the top level of your repository:

```
python3 run_pipeline.py
```

By default this will run the pipeline on all files in the **data/inputs** folder and it will run in **‘dev’ mode**, with all outputs going to the **storage/root** folder. To run the pipeline in production mode on a specific file, use the following syntax:

```
python3 run_pipeline.py $PATH_TO_YOUR_FILE --mode prod
```

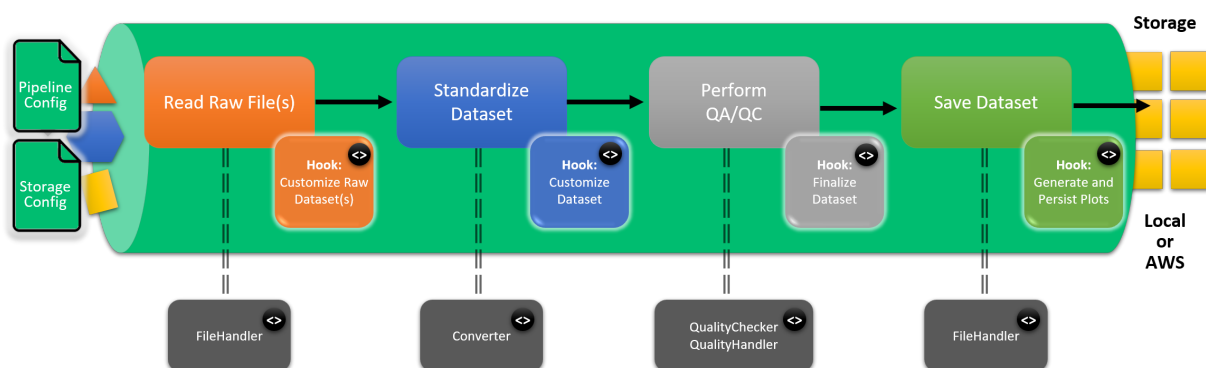
For command-line help:

```
python3 run_pipeline.py -h
```

For detailed examples of how to set up and use tsdat, consult the [Examples and Tutorials](#) section.

CONFIGURING TSDAT

Tsdat pipelines can be configured to tailor the specific data and metadata that will be contained in the standardized dataset. Tsdat pipelines provide multiple layers of configuration to allow the community to easily contribute common functionality (such as unit converters or file readers), to provide a low initial barrier of entry for basic ingests, and to allow full customization of the pipeline for very unique circumstances. The following figure illustrates the different phases of the pipeline along with multiple layers of configuration that Tsdat provides.



As shown in the figure, users can customize Tsdat in three ways:

1. **Configuration files** - shown as input to the pipeline on the left
2. **Code hooks** - indicated **inside** the pipeline with code (<>) bubbles. Code hooks are provided by extending the IngestPipeline base class to create custom pipeline behavior.
3. **Helper classes** - indicated **outside** the pipeline with code (<>) bubbles. Helper classes are described in more detail below and provide reusable, cross-pipeline functionality such as custom file readers or quality control checks. The specific helper classes that are used for a given pipeline are declared in the storage or pipeline config files.

More information on config file syntax and code hook base classes are provided below.

Note: Tsdat pipelines produce standardized datasets that follow the conventions and terminology provided in the [Data Standards Document](#). Please refer to this document for more detailed information about the format of standardized datasets.

5.1 Configuration Files

Configuration files provide an explicit, declarative way to define and customize the behavior of tsdat data pipelines. There are two types of configuration files:

1. **Storage config**
2. **Pipeline config**

This section breaks down the various properties of both types of configuration files and shows how these files can be modified to support a wide variety of data pipelines.

Note: Config files are written in yaml format. We recommend using an IDE with yaml support (such as VSCode) for editing your config files.

Note: In addition to your pre-configured pipeline template, see the [tsdat examples](#) folder for more configuration examples.

Note: In your pipeline template project, configuration files can be found in the `config/` folder.

5.1.1 Storage Config

The storage config file specifies which Storage class will be used to save processed data, declares configuration properties for that Storage (such as the root folder), and declares various FileHandler classes that will be used to read/write data with the specified file extensions.

Currently there are two provided storage classes:

1. **FilesystemStorage** - saves to local filesystem
2. **AwsStorage** - saves to an AWS bucket (requires an AWS account with admin privileges)

Each storage class has different configuration parameters, but they both share a common `file_handlers` section as explained below.

Note: Environment variables can be referenced in the storage config file using `${PARAMETER}` syntax in the yaml. Any referenced environment variables need to be set via the shell or via the `os.environ` dictionary from your `run_pipeline.py` file. The `CONFIG_DIR` environment parameter set automatically by tsdat and refers to the folder where the storage config file is located.

FilesystemStorage Parameters

```
storage:
  classname: tsdat.io.FilesystemStorage      # Choose from FilesystemStorage,
↳ or AwsStorage
  parameters:
    retain_input_files: True                  # Whether to keep input
↳ files after they are processed
    root_dir: ${CONFIG_DIR}/../storage/root  # The root dir where
↳ processed files will be stored
```

AwsStorage Parameters

```
storage:
  classname: tsdat.io.AwsStorage              # Choose from FilesystemStorage,
↳ or AwsStorage
  parameters:
    retain_input_files: True                  # Whether to keep input
↳ files after they are processed
    bucket_name: tsdat_test                  # The name of the AWS S3
↳ bucket where processed files will be stored
    root_dir: /storage/root                  # The root dir (key) prefix
↳ for all processed files created in the bucket
```

File Handlers

File Handlers declare the classes that should be used to read input and output files. Correspondingly, the `file_handlers` section in the yaml is split into two parts for input and output. For input files, you can specify a Python regular expression to match any specific file name pattern that should be read by that File Handler.

For output files, you can specify one or more formats. Tsdat will write processed data files using all the output formats specified. We recommend using the `NetCdfHandler` as this is the most powerful and flexible format that will support any data. However, other file formats may also be used such as `Parquet` or `CSV`. More output file handlers will be added over time.

```
file_handlers:
  input:
    sta:                                     # This is a label to identify your file
↳ handler
    file_pattern: '.*\.sta'                 # Use a Python regex to identify
↳ files this handler should process
    classname: pipeline.filehandlers.StaFileHandler # Declare
↳ the fully qualified name of the handler class

    output:
      netcdf:                                # This is a label to identify your file
↳ handler
      file_extension: '.nc'                  # Declare the file extension to use
↳ when writing output files
      classname: tsdat.io.filehandlers.NetCdfHandler # Declare the
↳ fully qualified name of the handler class
```

5.1.2 Pipeline Config

The pipeline config file is used to define how the pipeline will standardize input data. It defines all the pieces of your standardized dataset, as described in the [Data Standards Document](#). Specifically, it identifies the following components:

1. **Global attributes** - dataset metadata
2. **Dimensions** - shape of data
3. **Coordinate variables** - coordinate values for a specific dimension
4. **Data variables** - all other variables in the dataset
5. **Quality management** - quality tests to be performed for each variable and any associated corrections to be applied for failing tests.

Each pipeline template will include a starter pipeline config file in the config folder. It will work out of the box, but the configuration should be tweaked according to the specifics of your dataset.

A full annotated example of an ingest pipeline config file is provided below and can also be referenced in the [Tsdat Repository](#)

```

1 #####
2 # TSDAT (Time-Series Data) INGEST PIPELINE CONFIGURATION TEMPLATE
3 #
4 # This file contains an annotated example of how to configure an
5 # tsdat data ingest processing pipeline.
6 #####
7
8 # Specify the type of pipeline that will be run: Ingest or VAP
9 #
10 # Ingests are run against raw data and are used to convert
11 # proprietary instrument data files into standardized format, perform
12 # quality control checks against the data, and apply corrections as
13 # needed.
14 #
15 # VAPs are used to combine one or more lower-level standardized data
16 # files, optionally transform data to new coordinate grids, and/or
17 # to apply scientific algorithms to derive new variables that provide
18 # additional insights on the data.
19 pipeline:
20   type: "Ingest"
21
22   # Used to specify the level of data that this pipeline will use as
23   # input. For ingests, this will be used as the data level for raw data.
24   # If type: Ingest is specified, this defaults to "00"
25   # input_data_level: "00"
26
27   # Used to specify the level of data that this pipeline will produce.
28   # It is recommended that ingests use "a1" and VAPs should use "b1",
29   # but this is not enforced.
30   data_level: "a1"
31
32   # A label for the location where the data were obtained from
33   location_id: "humboldt_z05"
34
35   # A string consisting of any letters, digits, "-" or "_" that can
36   # be used to uniquely identify the instrument used to produce
37   # the data. To prevent confusion with the temporal resolution

```

(continues on next page)

(continued from previous page)

```

38 # of the instrument, the instrument identifier must not end
39 # with a number.
40 dataset_name: "buoy"
41
42 # An optional qualifier that distinguishes these data from other
43 # data sets produced by the same instrument. The qualifier
44 # must not end with a number.
45 #qualifier: "lidar"
46
47 # A optional description of the data temporal resolution
48 # (e.g., 30m, 1h, 200ms, 14d, 10Hz). All temporal resolution
49 # descriptors require a units identifier.
50 #temporal: "10m"
51
52 #####
53 # PART 1: DATASET DEFINITION
54 # Define dimensions, variables, and metadata that will be included
55 # in your processed, standardized data file.
56 #####
57 dataset_definition:
58 #-----
59 # Global Attributes (general metadata)
60 #
61 # All optional attributes are commented out. You may remove them
62 # if not applicable to your data.
63 #
64 # You may add any additional attributes as needed to describe your
65 # data collection and processing activities.
66 #-----
67 attributes:
68
69 # A succinct English language description of what is in the dataset.
70 # The value would be similar to a publication title.
71 # Example: "Atmospheric Radiation Measurement (ARM) program Best
72 # Estimate cloud and radiation measurements (ARMBECLDRAD) "
73 # This attribute is highly recommended but is not required.
74 title: "Buoy Dataset for Buoy #120"
75
76 # Longer English language description of the data.
77 # Example: "ARM best estimate hourly averaged QC controlled product,
78 # derived from ARM observational Value-Added Product data: ARSCL,
79 # MWRRET, QCRAD, TSI, and satellite; see input_files for the names of
80 # original files used in calculation of this product"
81 # This attribute is highly recommended but is not required.
82 description: "Example ingest dataset used for demonstration purposes."
83
84 # The version of the standards document this data conforms to.
85 # This attribute is highly recommended but is not required.
86 # conventions: "ME Data Pipeline Standards: Version 1.0"
87
88 # If an optional Digital Object Identifier (DOI) has been obtained
89 # for the data, it may be included here.
90 #doi: "10.21947/1671051"
91
92 # The institution who produced the data
93 # institution: "Pacific Northwest National Laboratory"
94

```

(continues on next page)

(continued from previous page)

```

95 # Include the url to the specific tagged release of the code
96 # used for this pipeline invocation.
97 # Example, https://github.com/clansing/twrnr/releases/tag/1.0.
98 # Note that MHKiT-Cloud will automatically create a new code
99 # release whenever the pipeline is deployed to production and
100 # record this attribute automatically.
101 code_url: "https://github.com/tsdat/tsdat/releases/tag/v0.2.2"
102
103 # Published or web-based references that describe the methods
104 # algorithms, or third party libraries used to process the data.
105 #references: "https://github.com/MHKit-Software/MHKit-Python"
106
107 # A more detailed description of the site location.
108 #location_meaning: "Buoy is located of the coast of Humboldt, CA"
109
110 # Name of instrument(s) used to collect data.
111 #instrument_name: "Wind Sentinel"
112
113 # Serial number of instrument(s) used to collect data.
114 #serial_number: "000011312"
115
116 # Description of instrument(s) used to collect data.
117 #instrument_meaning: "Self-powered floating buoy hosting a suite of_
↪ meteorological and marine instruments."
118
119 # Manufacturer of instrument(s) used to collect data.
120 #instrument_manufacturer: "AXYS Technologies Inc."
121
122 # The date(s) of the last time the instrument(s) was calibrated.
123 #last_calibration_date: "2020-10-01"
124
125 # The expected sampling interval of the instrument (e.g., "400 us")
126 #sampling_interval: "10 min"
127
128 #-----
129 # Dimensions (shape)
130 #-----
131 dimensions:
132 # All time series data must have a "time" dimension
133 # TODO: provide a link to the documentation online
134 time:
135     length: "unlimited"
136
137 #-----
138 # Variable Defaults
139 #
140 # Variable defaults can be used to specify a default dimension(s),
141 # data type, or variable attributes. This can be used to reduce the
142 # number of properties that a variable needs to define in this
143 # config file, which can be useful for vaps or ingests with many
144 # variables.
145 #
146 # Once a default property has been defined, (e.g. 'type: float64')
147 # that property becomes optional for all variables (e.g. No variables
148 # need to have a 'type' property).
149 #
150 # This section is entirely optional, so it is commented out.

```

(continues on next page)

(continued from previous page)

```

151 #-----
152 # variable_defaults:
153
154 # Optionally specify defaults for variable inputs. These defaults will
155 # only be applied to variables that have an 'input' property. This
156 # is to allow for variables that are created on the fly, but defined in
157 # the config file.
158 # input:
159
160 # If this is specified, the pipeline will attempt to match the file pattern
161 # to an input filename. This is useful for cases where a variable has the
162 # same name in multiple input files, but it should only be retrieved from
163 # one file.
164 # file_pattern: "buoy"
165
166 # Specify this to indicate that the variable must be retrieved. If this is
167 # set to True and the variable is not found in the input file the pipeline
168 # will crash. If this is set to False, the pipeline will continue.
169 # required: True
170
171 # Defaults for the converter used to translate input numpy arrays to
172 # numpy arrays used for calculations
173 # converter:
174
175 #-----
176 # Specify the classname of the converter to use as a default.
177 # A converter is used to convert the raw data into standardized
178 # values.
179 #
180 # Use the DefaultConverter for all non-time variables that
181 # use units supported by udunits2.
182 # https://www.unidata.ucar.edu/software/udunits/udunits-2.2.28/udunits2.html
183 ↪ #Database
184 #
185 # If your raw data has units that are not supported by udunits2,
186 # you can specify your own Converter class.
187 #-----
188 # classname: "tsdat.utils.converters.DefaultConverter"
189
190 # If the default converter always requires specific parameters, these
191 # can be defined here. Note that these parameters are not tied to the
192 # classname specified above and will be passed to all converters defined
193 # here.
194 # parameters:
195
196 # Example of parameter format:
197 # param_name: param_value
198
199 # The name(s) of the dimension(s) that dimension this data by
200 # default. For time-series tabular data, the following is a 'good'
201 # default to use:
202 # dims: [time]
203
204 # The data type to use by default. The data type must be one of:
205 # int8 (or byte), uint8 (or ubyte), int16 (or short), uint16 (or ushort),
206 # int32 (or int), uint32 (or uint), int64 (or long), uint64 (or ulong),
207 # float32 (or float), float64 (or double), char, str

```

(continues on next page)

(continued from previous page)

```

207     # type: float64
208
209     # Any attributes that should be defined by default
210     # attrs:
211
212     # Default _FillValue to use for missing data. Recommended to use
213     # -9999 because it is the default _FillValue according to CF
214     # conventions for netCDF data.
215     # _FillValue: -9999
216
217     #-----
218     # Variables
219     #-----
220     variables:
221
222     #-----
223     # All time series data must have a "time" coordinate variable which
224     # contains the data values for the time dimension
225     # TODO: provide a link to the documentation online
226     #-----
227     time: # Variable name as it will appear in the processed data
228
229     #-----
230     # The input section for each variable is used to specify the
231     # mapping between the raw data file and the processed output data
232     #-----
233     input:
234         # Name of the variable in the raw data
235         name: "DataTimeStamp"
236
237     #-----
238     # A converter is used to convert the raw data into standardized
239     # values.
240     #-----
241     # Use the StringTimeConverter if your raw data provides time
242     # as a formatted string.
243     converter:
244         classname: "tsdat.utils.converters.StringTimeConverter"
245         parameters:
246             # A list of timezones can be found here:
247             # https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
248             timezone: "US/Pacific"
249             time_format: "%Y-%m-%d %H:%M:%S"
250
251     # Use the TimestampTimeConverter if your raw data provides time
252     # as a numeric UTC timestamp
253     #converter:
254     #     classname: tsdat.utils.converters.TimestampTimeConverter
255     #     parameters:
256     #         # Unit of the numeric value as used by pandas.to_datetime (D,s,ms,us,ns)
257     #         unit: s
258
259     # The shape of this variable. All coordinate variables (e.g., time) must
260     # have a single dimension that exactly matches the variable name
261     dims: [time]
262
263     # The data type of the variable. Must be one of:

```

(continues on next page)

(continued from previous page)

```

264 # int8 (or byte), uint8 (or ubyte), int16 (or short), uint16 (or ushort),
265 # int32 (or int), uint32 (or uint), int64 (or long), uint64 (or ulong),
266 # float32 (or float), float64 (or double), char, str
267 type: int64
268
269 #-----
270 # The attrs section define the attributes (metadata) that will
271 # be set for this variable.
272 #
273 # All optional attributes are commented out. You may remove them
274 # if not applicable to your data.
275 #
276 # You may add any additional attributes as needed to describe your
277 # variables.
278 #
279 # Any metadata used for QC tests will be indicated.
280 #-----
281 attrs:
282
283 # A minimal description of what the variable represents.
284 long_name: "Time offset from epoch"
285
286 # A string exactly matching a value in from the CF or MRE
287 # Standard Name table, if a match exists
288 standard_name: time
289
290 # A CFUnits-compatible string indicating the units the data
291 # are measured in.
292 # https://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-
293 conventions.html#units
294 #
295 # Note: CF Standards require this exact format for time.
296 # UTC is strongly recommended.
297 # https://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-
298 conventions.html#time-coordinate
299 units: "seconds since 1970-01-01T00:00:00"
300
301 #-----
302 # Mean temperature variable (non-coordinate variable)
303 #-----
304 sea_surface_temperature: # Variable name as it will appear in the processed data
305
306 #-----
307 # The input section for each variable is used to specify the
308 # mapping between the raw data file and the processed output data
309 #-----
310 input:
311 # Name of the variable in the raw data
312 name: "Surface Temperature (C)"
313
314 # Units of the variable in the raw data
315 units: "degC"
316
317 # The shape of this variable
318 dims: [time]
319
320 # The data type of the variable. Can be one of:

```

(continues on next page)

(continued from previous page)

```

319 # [byte, ubyte, char, short, ushort, int32 (or int), uint32 (or uint),
320 # int64 (or long), uint64 (or ulong), float, double, string]
321 type: double
322
323 #-----
324 # The attrs section define the attributes (metadata) that will
325 # be set for this variable.
326 #
327 # All optional attributes are commented out. You may remove them
328 # if not applicable to your data.
329 #
330 # You may add any additional attributes as needed to describe your
331 # variables.
332 #
333 # Any metadata used for QC tests will be indicated.
334 #-----
335 attrs:
336 # A minimal description of what the variable represents.
337 long_name: "Mean sea surface temperature"
338
339 # An optional attribute to provide human-readable context for what this_
↪variable
340 # represents, how it was measured, or anything else that would be relevant to_
↪end-users.
341 #comment: Rolling 10-minute average sea surface temperature. Aligned such_
↪that the temperature reported at time 'n' represents the average across the_
↪interval (n-1, n].
342
343 # A CFUnits-compatible string indicating the units the data
344 # are measured in.
345 # https://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-
↪conventions.html#units
346 units: "degC"
347
348 # The value used to initialize the variable's data. Defaults to -9999.
349 # Coordinate variables must not use this attribute.
350 _FillValue: -9999
351
352 # An array of variable names that depend on the values from this variable._
↪This is primarily
353 # used to indicate if a variable has an ancillary qc variable.
354 # NOTE: QC ancillary variables will be automatically recorded via the MHKiT-
↪Cloud pipeline engine.
355 ancillary_variables: []
356
357 # A two-element array of [min, max] representing the smallest and largest_
↪valid values
358 # of a variable. Values outside valid_range will be filled with _FillValue.
359 valid_range: [-50, 50]
360
361 # The maximum allowed difference between any two consecutive values of a_
↪variable,
362 # values outside of which should be flagged as "Bad".
363 # This attribute is used for the valid_delta QC test. If not specified, this
364 # variable will be omitted from the test.
365 valid_delta: 0.25
366

```

(continues on next page)

(continued from previous page)

```

367     # A two-element array of [min, max] outside of which the data should be
↪ flagged as "Bad".
368     # This attribute is used for the fail_min and fail_max QC tests.
369     # If not specified, this variable will be omitted from these tests.
370     #fail_range: [0, 40]
371
372     # A two-element array of [min, max] outside of which the data should be
↪ flagged as "Indeterminate".
373     # This attribute is used for the warn_min and warn_max QC tests.
374     # If not specified, this variable will be omitted from these tests.
375     #warn_range: [0, 30]
376
377     # An array of strings indicating what corrections, if any, have been applied
↪ to the data.
378     #corrections_applied: []
379
380     # The height of the instrument above ground level (AGL), or in the case of
↪ above
381     # water, above the surface.
382     #sensor_height: "30m"
383
384     #-----
385     # Example of a variables that hold a single scalar value that
386     # is not present in the raw data.
387     #-----
388     latitude:
389         data: 71.323 #<-----The data field can be used to specify a pre-set value
390         type: float
391
392         #<-----This variable has no input, which means it will be set by
393         # the pipeline and not pulled from the raw data
394
395         #<-----This variable has no dimensions, which means it will be
396         # a scalar value
397
398         attrs:
399             long_name: "North latitude"
400             standard_name: "latitude"
401             comment: "Recorded latitude at the instrument location"
402             units: "degree_N"
403             valid_range: [-90.f, 90.f]
404
405     longitude:
406         data: -156.609
407         type: float
408         attrs:
409             long_name: "East longitude"
410             standard_name: "longitude"
411             comment: "Recorded longitude at the instrument location"
412             units: "degree_E"
413             valid_range: [-180.f, 180.f]
414
415     #-----
416     # Example of a variable that is derived by the processing pipeline
417     #-----
418     foo:
419         type: float

```

(continues on next page)

(continued from previous page)

```

420     #<-----This variable has no input, which means it will be set by
421     # the pipeline and not pulled from the raw data
422
423
424     dims: [time]
425
426     attrs:
427         long_name: "some other property"
428         units: "kg/m^3"
429         comment: "Computed from temp_mean point value using some formula..."
430         references: ["http://sccoos.org/data/autosss/", "http://sccoos.org/about/dmac/
431         ↪ "]
432
433     ---
434     #####
435     # PART 2: QC TESTS
436     # Define the QC tests that will be applied to variable data.
437     #####
438     coordinate_variable_qc_tests:
439         #-----
440         # The following section defines the default qc tests that will be
441         # performed on coordinate variables in a dataset. Note that by
442         # default, coordinate variable tests will NOT set a QC bit and
443         # will trigger a critical pipeline failure. This is because
444         # Problems with coordinate variables are considered to cause
445         # the dataset to be unusable and should be manually reviewed.
446         #
447         # However, the user may override the default coordinate variable
448         # tests and error handlers if they feel that data correction is
449         # warranted.
450         #
451         # For a complete list of tests provided by MHKiT-Cloud, please see
452         # the tsdat.qc.operators package.
453         #
454         # Users are also free to add custom tests defined by their own
455         # checker classes.
456         #-----
457     quality_management:
458         #-----
459         # The following section defines the default qc tests that will be
460         # performed on variables in a dataset.
461         #
462         # For a complete list of tests provided by MHKiT-Cloud, please see
463         # the tsdat.qc.operators package.
464         #
465         # Users are also free to add custom tests defined by their own
466         # checker classes.
467         #-----
468
469         #-----
470         # Checks on coordinate variables
471         #-----
472
473         # The name of the test.
474         manage_missing_coordinates:
475

```

(continues on next page)

(continued from previous page)

```

476 # Quality checker used to identify problematic variable values.
477 # Users can define their own quality checkers and link them here
478 checker:
479     # This quality checker will identify values that are missing,
480     # NaN, or equal to each variable's _FillValue
481     classname: "tsdat.qc.operators.CheckMissing"
482
483 # Quality handler used to manage problematic variable values.
484 # Users can define their own quality handlers and link them here.
485 handlers:
486     # This quality handler will cause the pipeline to fail
487     - classname: "tsdat.qc.error_handlers.FailPipeline"
488
489 # Which variables to apply the test to
490 variables:
491     # keyword to apply test to all coordinate variables
492     - COORDS
493
494 manage_coordinate_monotonicity:
495
496     checker:
497         # This quality checker will identify variables that are not
498         # strictly monotonic (That is, it identifies variables whose
499         # values are not strictly increasing or strictly decreasing)
500         classname: "tsdat.qc.operators.CheckMonotonic"
501
502     handlers:
503         - classname: "tsdat.qc.error_handlers.FailPipeline"
504
505     variables:
506         - COORDS
507
508 #-----
509 # Checks on data variables
510 #-----
511 manage_missing_values:
512
513     # The class that performs the quality check. Users are free
514     # to override with their own class if they want to change
515     # behavior.
516     checker:
517         classname: "tsdat.qc.operators.CheckMissing"
518
519     # Error handlers are optional and run after the test is
520     # performed if any of the values fail the test. Users
521     # may specify one or more error handlers which will be
522     # executed in sequence. Users are free to add their
523     # own QCErrorHandler subclass if they want to add custom
524     # behavior.
525     handlers:
526
527         # This error handler will replace any NaNs with _FillValue
528         - classname: "tsdat.qc.error_handlers.RemoveFailedValues"
529         # Quality handlers and all other objects that have a 'classname'
530         # property can take a dictionary of parameters. These
531         # parameters are made available to the object or class in the
532         # code and can be used to implement custom behavior with little

```

(continues on next page)

(continued from previous page)

```

533     # overhead.
534     parameters:
535
536         # The correction parameter is used by the RemoveFailedValues
537         # quality handler to append to a list of corrections for each
538         # variable that this handler is applied to. As a best practice,
539         # quality handlers that modify data values should use the
540         # correction parameter to update the 'corrections_applied'
541         # variable attribute on the variable this test is applied to.
542         correction: "Set NaN and missing values to _FillValue"
543
544
545     # This quality handler will record the results of the
546     # quality check in the ancillary qc variable for each
547     # variable this quality manager is applied to.
548     - classname: "tsdat.qc.error_handlers.RecordQualityResults"
549     parameters:
550
551         # The bit (1-32) used to record the results of this test.
552         # This is used to update the variable's ancillary qc
553         # variable.
554         bit: 1
555
556         # The assessment of the test. Must be either 'Bad' or 'Indeterminate'
557         assessment: "Bad"
558
559         # The description of the data quality from this check
560         meaning: "Value is equal to _FillValue or NaN"
561
562     variables:
563         # keyword to apply test to all non-coordinate variables
564         - DATA_VARS
565
566 manage_fail_min:
567     checker:
568         classname: "tsdat.qc.operators.CheckFailMin"
569     handlers:
570         - classname: "tsdat.qc.error_handlers.RecordQualityResults"
571         parameters:
572             bit: 2
573             assessment: "Bad"
574             meaning: "Value is less than the fail_range."
575     variables:
576         - DATA_VARS
577
578 manage_fail_max:
579     checker:
580         classname: "tsdat.qc.operators.CheckFailMax"
581     handlers:
582         - classname: "tsdat.qc.error_handlers.RecordQualityResults"
583         parameters:
584             bit: 3
585             assessment: "Bad"
586             meaning: "Value is greater than the fail_range."
587     variables:
588         - DATA_VARS
589

```

(continues on next page)

(continued from previous page)

```

590 manage_warn_min:
591     checker:
592         classname: "tsdat.qc.operators.CheckWarnMin"
593     handlers:
594         - classname: "tsdat.qc.error_handlers.RecordQualityResults"
595           parameters:
596             bit: 4
597             assessment: "Indeterminate"
598             meaning: "Value is less than the warn_range."
599     variables:
600         - DATA_VARS
601
602 manage_warn_max:
603     checker:
604         classname: "tsdat.qc.operators.CheckWarnMax"
605     handlers:
606         - classname: "tsdat.qc.error_handlers.RecordQualityResults"
607           parameters:
608             bit: 5
609             assessment: "Indeterminate"
610             meaning: "Value is greater than the warn_range."
611     variables:
612         - DATA_VARS
613
614 manage_valid_delta:
615     checker:
616         classname: "tsdat.qc.operators.CheckValidDelta"
617     parameters:
618         dim: time # specifies the dimension over which to compute the delta
619     handlers:
620         - classname: "tsdat.qc.error_handlers.RecordQualityResults"
621           parameters:
622             bit: 6
623             assessment: "Indeterminate"
624             meaning: "Difference between current and previous values exceeds valid_
↪delta."
625     variables:
626         - DATA_VARS
627
628 #-----
629 # Example of a user-created test that shows how to specify
630 # an error handler. Error handlers may be optionally added to
631 # any of the tests described above. (Note that this example will
632 # not work, it is just provided as an example of adding a
633 # custom QC test.)
634 #-----
635 # temp_test:
636
637 #     checker:
638 #         classname: "myproject.qc.operators.TestTemp"
639
640 # #-----
641 # # See the tsdat.qc.error_handlers package for a list of
642 # # available error handlers.
643 # #-----
644 #     handlers:

```

(continues on next page)

(continued from previous page)

```

646 #         # This handler will set bit number 7 on the ancillary qc
647 #         # variable for the variable(s) this test applies to.
648 #         - classname: "tsdat.qc.error_handlers.RecordQualityResults"
649 #         parameters:
650 #             bit: 7
651 #             assessment: "Indeterminate"
652 #             meaning: "Test for some special condition in temperature."
653
654 #         # This error handler will notify users via email. The
655 #         # datastream name, variable, and failing values will be
656 #         # included.
657 #         - classname: "tsdat.qc.error_handlers.SendEmailAWS"
658 #         parameters:
659 #             message: "Test failed..."
660 #             recipients: ["carina.lansing@pnnl.gov", "maxwell.levin@pnnl.gov"]
661
662 #         # Specifies the variable(s) this quality manager applies to
663 #         variables:
664 #             - temp_mean

```

5.2 Code Customizations

This section describes all the types of classes that can be extended in Tsdat to provide custom pipeline behavior. To start with, each pipeline will define a main Pipeline class which is used to run the pipeline itself. Each pipeline template will come with a Pipeline class pre-defined in the pipeline/pipeline.py file. The Pipeline class extends a specific base class depending upon the template that was selected. Currently, we only support one pipeline base class, `tsdat.pipeline.ingest_pipeline.IngestPipeline`. Later, support for VAP pipelines will be added. Each pipeline base class provides certain abstract methods which the developer can override if desired to customize pipeline functionality. In your template repository, your Pipeline class will come with all the hook methods stubbed out automatically (i.e., they will be included with an empty definition). Later as more templates are added - in particular to support specific data models- hook methods may be pre-filled out to implement prescribed calculations.

In addition to your Pipeline class, additional classes can be defined to provide specific behavior such as unit conversions, quality control tests, or reading/writing files. This section lists all of the custom classes that can be defined in Tsdat and what their purpose is.

Note: For more information on classes in Python, see <https://docs.python.org/3/tutorial/classes.html>

Note: We warmly encourage the community to contribute additional support classes such as FileHandlers and QC-Checkers.

5.2.1 IngestPipeline Code Hooks

The following hook methods (which can be easily identified because they all start with the ‘hook_’ prefix) are provided in the IngestPipeline template. They are listed in the order that they are executed in the pipeline.

hook_customize_raw_datasets Hook to allow for user customizations to one or more raw xarray Datasets before they merged and used to create the standardized dataset. This method would typically only be used if the user is combining multiple files into a single dataset. In this case, this method may be used to correct coordinates if they don’t match for all the files, or to change variable (column) names if two files have the same name for a variable, but they are two distinct variables.

This method can also be used to check for unique conditions in the raw data that should cause a pipeline failure if they are not met.

This method is called before the inputs are merged and converted to standard format as specified by the config file.

hook_customize_dataset Hook to allow for user customizations to the standardized dataset such as inserting a derived variable based on other variables in the dataset. This method is called immediately after the `apply_corrections` hook and before any QC tests are applied.

hook_finalize_dataset Hook to apply any final customizations to the dataset before it is saved. This hook is called after quality tests have been applied.

hook_generate_and_persist_plots Hook to allow users to create plots from the xarray dataset after processing and QC have been applied and just before the dataset is saved to disk.

5.2.2 File Handlers

File Handlers are classes that are used to read and write files. Each File Handler should extend the `tsdat.io.filehandlers.file_handlers.AbstractFileHandler` base class. The `AbstractFileHandler` base class defines two methods:

read Read a file into an XArray Dataset object.

write Write an XArray Dataset to file. This method only needs to be implemented for handlers that will be used to save processed data to persistent storage.

Each pipeline template comes with a default custom FileHandler implementation to use as an example if needed. In addition, see the [ImuFileHandler](#) for another example of writing a custom FileHandler to read raw instrument data.

The File Handlers that are to be used in your pipeline are configured in your *storage config file*

5.2.3 Converters

Converters are classes that are used to convert units from the raw data to standardized format. Each Converter should extend the `tsdat.utils.converters.Converter` base class. The Converter base class defines one method, **run**, which converts a numpy ndarray of variable data from the input units to the output units. Currently tsdat provides two converters for working with time data. `tsdat.utils.converters.StringTimeConverter` converts time values in a variety of string formats, and `tsdat.utils.converters.TimestampTimeConverter` converts time values in long integer format. In addition, tsdat provides a `tsdat.utils.converters.DefaultConverter` which converts any units from one udunits2 supported units type to another.

5.2.4 Quality Management

Two types of classes can be defined in your pipeline to ensure standardized data meets quality requirements:

QualityChecker Each QualityChecker performs a specific QC test on one or more variables in your dataset.

QualityHandler Each QualityHandler can be specified to run if a particular QC test fails. It can be used to correct invalid values, such as interpolating to fill gaps in the data.

The specific QCCheckers and QCHandlers used for a pipeline and the variables they run on are specified in the *pipeline config file*.

Quality Checkers

Quality Checkers are classes that are used to perform a QC test on a specific variable. Each Quality Checker should extend the `tsdat.qc.checkers.QualityChecker` base class, which defines a `run()` method that performs the check. Each QualityChecker defined in the pipeline config file will be automatically initialized by the pipeline and invoked on the specified variables. See the API Reference for a detailed description of the `QualityChecker.run()` method as well as a list of all QualityCheckers defined by Tsdats.

Quality Handlers

Quality Handlers are classes that are used to correct variable data when a specific quality test fails. An example is interpolating missing values to fill gaps. Each Quality Handler should extend the `tsdat.qc.handlers.QualityHandler` base class, which defines a `run()` method that performs the correction. Each QualityHandler defined in the pipeline config file will be automatically initialized by the pipeline and invoked on the specified variables. See the API Reference for a detailed description of the `QualityHandler.run()` method as well as a list of all QualityHandlers defined by Tsdats.

EXAMPLES AND TUTORIALS

We understand that many people learn better from examples than large walls of text and API references. That is why we have collected a set of examples and tutorials that we think are helpful for explaining how tsdat can be used to simplify the development of develop data pipelines and to show off some of the more advanced features of the library.

6.1 Examples

Tsdat hosts several examples on its [GitHub repository](#).

More examples coming soon.

6.2 Tutorials

We are starting to develop and collect written and video tutorials that provide walkthroughs of common tsdat workflows. See below for a list of tutorials:

6.2.1 Local Data Ingest

In this tutorial we will build a data ingestion pipeline to ingest some global marine data hosted by the National Oceanic and Atmospheric Administration's (NOAA) National Centers for Environmental Information (NCEI). The data can be found at <https://www.ncdc.noaa.gov/cdo-web/datasets> under the "Global Marine Data" section. This is a pretty simple and high-quality dataset, so this data ingest will be pretty straight-forward. We will walk through the following steps in this tutorial:

1. Examine and download the data
2. Set up a GitHub repository in which to build our ingestion pipeline
3. Modify configuration files and ingestion pipeline for our NCEI dataset
4. Run the ingest data pipeline on NCEI data

Now that we've outlined the goals of this tutorial and the steps that we will need to take to ingest this data we can get started with step #1.

Examining and downloading the data

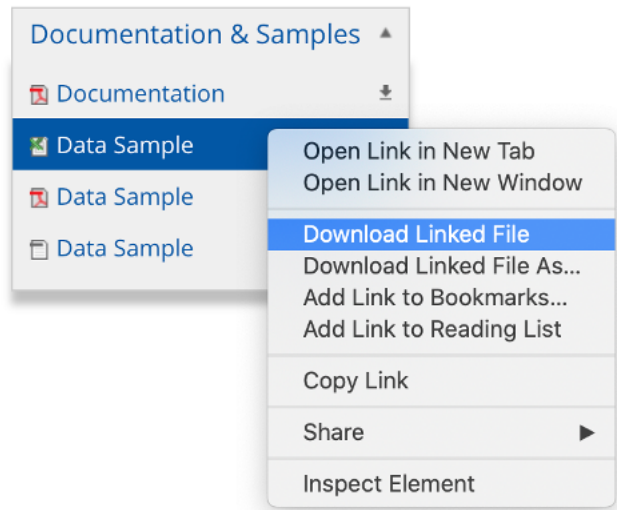
Navigate to <https://www.ncdc.noaa.gov/cdo-web/datasets> and download the documentation and a data sample from their global marine data section.

Global Marine Data

Historical marine data are comprised of ship, buoy, and platform observations from 1662-present. Data values are delayed by one month. In addition to location, ship identification, ship speed, and ship direction, weather elements observed include: wind direction and speed, visibility, present and past weather, sea level pressure, dry bulb, wet bulb, and dew point temperatures, sea surface temperature, cloud data, wave data, and ice accretion. [More »](#)

■ [Search Tool](#) | ■ [Mapping Tool](#) | ■ [FTP](#)

- ⊕ Global Summary of the Month
- ⊕ Global Summary of the Year
- ⊕ Local Climatological Data
- ⊕ Normals Annual/Seasonal
- ⊕ Normals Daily
- ⊕ Normals Hourly
- ⊕ Normals Monthly
- ⊕ Precipitation 15 Minute



The documentation describes each variable in the sample dataset and will be extremely useful for updating our configuration file with the metadata for this dataset. The metadata we care most about are the units and user-friendly text descriptions of each variable, but we also need to be on the lookout for any inconsistencies or potential data problems that could complicate how we process this dataset. Take, for example, the following descriptions of the various temperature measurements that this dataset contains and note that the units are not necessarily the same between files in this dataset:

Air Temperature - Air temperature in tenths of degrees Celsius or Fahrenheit depending on user specification (standard or metric option).

Wet Bulb Temperature - Wet-bulb temperature in tenths of degrees Celsius or Fahrenheit depending on user specification (standard or metric option).

Dew Point Temperature - Dew point temperature in tenths of degrees Celsius or Fahrenheit depending on user specification (standard or metric option).

Sea Surface Temperature - Sea surface temperature in tenths of degrees Celsius or Fahrenheit depending on user specification (standard or metric option).

If we were collecting this data from multiple users, we would need to be aware of possible unit differences between files from different users and we would likely want to standardize the units so that they were all in Celsius or all in

Fahrenheit (Our preference is to use the metric system wherever possible). If we examine this data, it appears that the units are not metric – how unfortunate. Luckily, this is something that can easily be fixed by using tsdat.

Sea Level Pressure	Characteristics of Pressure	Tendency Pressure	Tendency Air	Temperature
29.83		8	0.0	70.5
29.93		1	0.0	66.2
30.02		1	0.0	61.2
30.14		0	0.0	64.6
30.12		8		66.7

Fig. 1: Selection from the sample dataset. It appears that units are recorded in the imperial system instead of the metric system – Sea Level Pressure is recorded in Hg instead of hPa (Hectopascal) and Air Temperature is recorded in degF (Fahrenheit) instead of degC (Celsius).

Creating a repository from a template

Now that we have the data and metadata that we will need for this example, let's move on to step #2 and set up a GitHub repository for our work. For this example, I will be using a [template repository](#) to speed things up, as this is one of the easiest ways to get started quickly. In this example I will be using [tsdat/ingest-template-local](#) as the basis for this example because what we are looking to do is read in the NCEI “raw” data and apply a set of corrections and changes to the dataset to bring it into the netCDF format – an ‘ingest’, in other words. To do this, navigate to <https://github.com/tsdat/ingest-template-local> and click “Use this template”.

tsdat / ingest-template-local Template Watch

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

[main](#) [1 branch](#) [0 tags](#) [Go to file](#) [Add file](#) [Code](#) [Use this template](#)

maxwellevin Update pipeline_config.yml 251cd95 5 days ago [7 commits](#)

File/Folder	Description	Time Ago
config	Update pipeline_config.yml	5 days ago
data/inputs	Finished the local template. It supports running one pipeline. Later ...	20 days ago
pipeline	Finished the local template. It supports running one pipeline. Later ...	20 days ago
tests	Finished the local template. It supports running one pipeline. Later ...	20 days ago
.gitignore	Finished the local template. It supports running one pipeline. Later ...	20 days ago
README.md	Update README.md	5 days ago
run_pipeline.py	Finished the local template. It supports running one pipeline. Later ...	20 days ago

ingest-template-local

This will open <https://github.com/tsdat/ingest-template-local/generate> (you can also just open this link directly) which will prompt you to name your repository. Go ahead and fill out the information however you would like and set the visibility to your preference. Once you are happy with it, click the green button at the bottom to create a repository from the template.

Create a new repository from ingest-template-local

The new repository will start with the same files and folders as [tsdat/ingest-template-local](#).

Owner *

Repository name *

 maxwellevin

/

ncei-global-marine-data-ingest ✓

Great repository names are short and memorable. Need inspiration? How about [silver-invention?](#)

Description (optional)

Pipeline for ingesting Global Marine Data hosted by NCEI at <https://www.ncdc.noaa.gov/cdo-web/datasets>.

☐

Public

Anyone on the internet can see this repository. You choose who can commit.

☒

Private

You choose who can see and commit to this repository.

☐ Include all branches

Copy all branches from tsdat/ingest-template-local and not just main.

Create repository from template

Click “Create repository from template” to create your own repository that you can work in for this example.

Go ahead and clone the repository to your local machine and open it up in whatever IDE you prefer.

Next install Python 3.8+ if you haven’t already done so and create an environment in which to manage your project’s dependencies. You can download and install Python here: <https://www.python.org>. When developing with intent to deploy to a production system, we recommend managing your environment using a [Docker Container](#) or an [Anaconda environment](#). For this tutorial, however, I will just be using Python’s built-in venv tool to manage python dependencies:

```
python3.8 -m venv ncei_env/
source ncei_env/bin/activate
pip install -r requirements.txt
```

This will install tsdat into our *ncei_env* virtual environment.

We now have everything we need to run the example ingest. Go ahead and do that:

```
python3.8 run_pipeline.py
```

Notice that a new *storage/* folder is created with the following contents:

These files contain the outputs of the ingest pipeline example that came with the ingest template we used. Note that there are two subdirectories here – one ends in “.00” and the other ends with “.a1”. This ending is called the “data level” and indicates the level of processing of the data, with “00” representing raw data that has been renamed according to the data standards that tsdat was developed under, and “a1” representing data that has been ingested,

maxwellevin / ncei-global-marine-data-ingest Private

generated from [tsdat/ingest-template-local](#)

Unwatch 1 Star 0 Fork 0

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main 1 branch 0 tags

Go to file Add file Code

maxwellevin Initial commit 2de908e now 1 commit

config	Initial commit	now
data/inputs	Initial commit	now
pipeline	Initial commit	now
tests	Initial commit	now
.gitignore	Initial commit	now
README.md	Initial commit	now
run_pipeline.py	Initial commit	now

README.md

ingest-template-local

Ingest template for a new pipeline that runs on your local filesystem.

About

Pipeline for ingesting Global Marine Data hosted by NCEI at <https://www.ncdc.noaa.gov/cdo-web/datasets>

Readme

Releases

No releases published
[Create a new release](#)

Packages

No packages published
[Publish your first package](#)

```

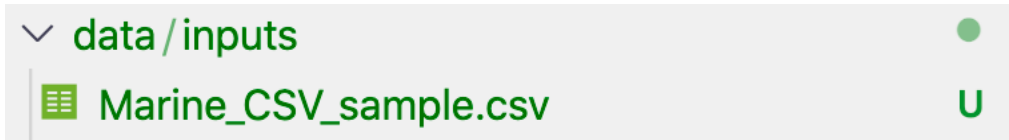
✓ storage / root / morro
  ✓ morro.buoy_z06-lidar-10min.00
    morro.buoy_z06-lidar-10min.00.20201201.001000.raw.lidar.z06.00.20201201.000000.sta
  ✓ morro.buoy_z06-lidar-10min.a1
    morro.buoy_z06-lidar-10min.a1.20201201.001000.nc
    morro.buoy_z06-lidar-10min.a1.20201201.001000.wind_speed_and_direction.png
    morro.buoy_z06-lidar-10min.a1.20201201.001000.wind_speeds.png

```

standardized, and optionally quality-controlled. For more information on the standards used to develop tsdat, please consult [our data standards](https://github.com/tsdat/data_standards).

Customizing the template repository

Now that all the setup work is done, let's start working on ingesting the NCEI data. First, we'll need to copy the sample data file into our data/inputs directory and pull up the documentation for us to reference:



We'll then want to start modifying the configuration files to work with our example. For one, the storage config files can change to use the `tsdat.io.filehandlers.CsvHandler` instead of the custom `FileHandler` used in the example by default. Additionally, if we examine the sample csv closely we can see that a mixture of tabs, commas, and spaces are used to separate the columns. While this somewhat works visually, many libraries have trouble parsing this. To solve this with tsdat, we can add some parameters to the storage configuration file to indicate how those gaps should be handled. Put together, the final storage config file looks like this:

```

1 storage:
2   classname: tsdat.io.FilesystemStorage
3   parameters:
4     retain_input_files: True
5     root_dir: ${CONFIG_DIR}/../storage/root
6
7   file_handlers:
8     input:
9       csv:
10        file_pattern: '.*\.csv'
11        classname: tsdat.io.filehandlers.CsvHandler
12        parameters:
13          read:
14            read_csv:
15              sep: ", *"
16              engine: "python"
17              index_col: False
18
19      output:
20        netcdf:
21          file_extension: ".nc"
22          classname: tsdat.io.filehandlers.NetCdfHandler

```

We'll then need to modify the pipeline configuration file to capture the variables and metadata we want to retain in this ingest. This part of the process can take some time, as it involves knowing or learning a lot of the context around the dataset and then writing it up succinctly and clearly so that your data users can quickly get a good understanding of what this dataset is and how to start using it. This part of the process is super specific to the particular dataset you are working on, so I will show only a snippet of the changes I have made here:

```

1 pipeline:
2   type: Ingest
3   location_id: arctic
4   dataset_name: ice_accretion
5   qualifier: ship_001
6   data_level: a1

```

(continues on next page)

(continued from previous page)

```

7
8 dataset_definition:
9   attributes:
10     title: "Marine Meteorological Measurements (Example Ingest)"
11     description: "Historical marine data are comprised of ship, buoy, and platform_
↪ observations."
12     conventions: "ME Data Pipeline Standards: Version 1.0"
13     institution: "National Oceanic and Atmospheric Administration"
14     code_url: "https://github.com/maxwellelevin/ncei-global-marine-data-ingest"
15
16   dimensions:
17     time:
18       length: unlimited
19
20   variables:
21     time:
22       input:
23         name: Time of Observation
24         converter:
25           classname: tsdat.utils.converters.StringTimeConverter
26           parameters:
27             time_format: "%Y-%m-%dT%H:%M:%S"
28         dims: [time]
29         type: long
30         attrs:
31           long_name: Time of Observation (UTC)
32           standard_name: time
33           units: seconds since 1970-01-01T00:00:00
34
35     pressure:
36       input:
37         name: Sea Level Pressure
38         dims: [time]
39         type: float
40         attrs:
41           long_name: Pressure at Sea Level
42           units: hPa
43
44     pressure_tendency_characteristics:
45       input:
46         name: Characteristics of Pressure Tendency
47         dims: [time]
48         type: int
49         attrs:
50           long_name: Characteristics of Pressure Tendency
51           comment: "-1=Data is missing, 0=Increasing, then decreasing, 1=Increasing_
↪ steadily or unsteadily, 2=Increasing steadily or unsteadily, 3=Decreasing or steady_
↪ then increasing OR increasing then increasing more rapidly, 4=Steady. Pressure same_
↪ as 3 hrs. ago, 5=Decreasing then increasing OR decreasing then decreasing more_
↪ slowly, 6=Decreasing, then steady OR decreasing, then decreasing more slowly,_
↪ 7=Decreasing steadily or unsteadily, 8=Steady or increasing then decreasing OR_
↪ decreasing then decreasing more rapidly"
52       _FillValue: -1
53
54   quality_management:
55
56   manage_missing_coordinates:

```

(continues on next page)

(continued from previous page)

```

57     checker:
58         classname: tsdat.qc.checkers.CheckMissing
59     handlers:
60         - classname: tsdat.qc.handlers.FailPipeline
61     variables:
62         - COORDS
63
64     manage_coordinate_monotonicity:
65         checker:
66             classname: tsdat.qc.checkers.CheckMonotonic
67         handlers:
68             - classname: tsdat.qc.handlers.FailPipeline
69         variables:
70             - COORDS

```

Finally, we will work on updating the customized pipeline that was written for the example ingest in the original template. I've removed several of the user hooks to keep this simple and also reworked the plotting hook so that it plots just the variables listed in the snippet above:

```

1  import os
2  import cmocean
3  import matplotlib.pyplot as plt
4  import pandas as pd
5  import xarray as xr
6  from tsdat.pipeline import IngestPipeline
7  from tsdat.utils import DSUtil
8
9  example_dir = os.path.abspath(os.path.dirname(__file__))
10 style_file = os.path.join(example_dir, "styling.mplstyle")
11 plt.style.use(style_file)
12
13
14 class Pipeline(IngestPipeline):
15
16     def hook_generate_and_persist_plots(self, dataset: xr.Dataset) -> None:
17         start_date = pd.to_datetime(dataset.time.data[0]).strftime('%Y-%m-%d')
18         final_date = pd.to_datetime(dataset.time.data[-1]).strftime('%Y-%m-%d')
19
20         filename = DSUtil.get_plot_filename(dataset, "pressure", "png")
21         with self.storage._tmp.get_temp_filepath(filename) as tmp_path:
22
23             fig, ax = plt.subplots(figsize=(10, 8), constrained_layout=True)
24             fig.suptitle(f"Pressure Observations from {start_date} to {final_date}")
25             dataset.pressure.plot(ax=ax, x="time", c=cmocean.cm.deep_r(0.5))
26
27             fig.savefig(tmp_path, dpi=100)
28             self.storage.save(tmp_path)
29             plt.close()
30
31     return

```

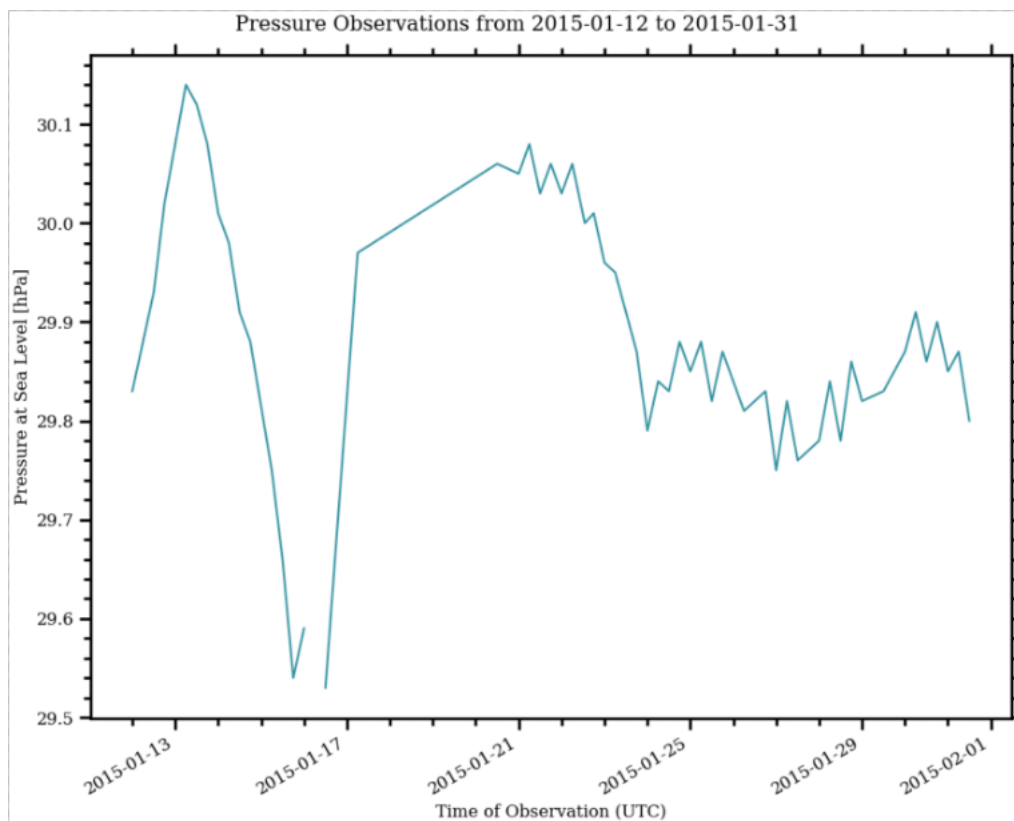
Running the pipeline

We can now re-run the pipeline using the same command as before

```
python3 run_pipeline.py
```

and it will produce the following results:

```
▼ storage/root
  ▼ arctic
    ▼ arctic.ice_accretion-ship_001.00
      📄 arctic.ice_accretion-ship_001.00.20150112.000000.raw.Marine_CSV_sample.csv
    ▼ arctic.ice_accretion-ship_001.a1
      📄 arctic.ice_accretion-ship_001.a1.20150112.000000.nc
      🖼️ arctic.ice_accretion-ship_001.a1.20150112.000000.pressure.png
```



API REFERENCE

This page contains auto-generated API reference documentation¹.

7.1 tsdat

7.1.1 Subpackages

`tsdat.config`

Module that wraps objects defined in pipeline and yaml configuration files.

Submodules

`tsdat.config.config`

Module Contents

Classes

<i>Config</i>	Wrapper for the pipeline configuration file.
---------------	--

class `tsdat.config.config.Config` (*dictionary: Dict*)

Wrapper for the pipeline configuration file.

Note: in most cases, `Config.load(filepath)` should be used to instantiate the `Config` class.

Parameters `dictionary` (*Dict*) – The pipeline configuration file as a dictionary.

`_parse_quality_managers` (*self, dictionary: Dict*) → `Dict[str, ts-dat.config.quality_manager_definition.QualityManagerDefinition]`
Extracts `QualityManagerDefinitions` from the config file.

Parameters `dictionary` (*Dict*) – The quality_management dictionary.

Returns Mapping of quality manager name to `QualityManagerDefinition`

Return type `Dict[str, QualityManagerDefinition]`

¹ Created with `sphinx-autoapi`

classmethod `load` (*self*, *filepaths*: *List[str]*)

Load one or more yaml pipeline configuration files. Multiple files should only be passed as input if the pipeline configuration file is split across multiple files.

Parameters `filepaths` (*List[str]*) – The path(s) to yaml configuration files to load.

Returns A Config object wrapping the yaml configuration file(s).

Return type *Config*

static `lint_yaml` (*filename*: *str*)

Lints a yaml file and raises an exception if an error is found.

Parameters `filename` (*str*) – The path to the file to lint.

Raises **Exception** – Raises an exception if an error is found.

`tsdat.config.dataset_definition`

Module Contents

Classes

DatasetDefinition

Wrapper for the `dataset_definition` portion of the pipeline config

class `tsdat.config.dataset_definition.DatasetDefinition` (*dictionary*: *Dict*, *datastream_name*: *str*)

Wrapper for the `dataset_definition` portion of the pipeline config file.

Parameters

- **dictionary** (*Dict*) – The portion of the config file corresponding with the dataset definition.
- **datastream_name** (*str*) – The name of the datastream that the config file is for.

_parse_dimensions (*self*, *dictionary*: *Dict*) → *Dict[str, ts-dat.config.dimension_definition.DimensionDefinition]*

Extracts the dimensions from the `dataset_definition` portion of the config file.

Parameters `dictionary` (*Dict*) – The `dataset_definition` dictionary from the config file.

Returns Returns a mapping of output dimension names to `DimensionDefinition` objects.

Return type *Dict[str, DimensionDefinition]*

_parse_variables (*self*, *dictionary*: *Dict*, *available_dimensions*: *Dict[str, ts-dat.config.dimension_definition.DimensionDefinition]*) → *Dict[str, ts-dat.config.variable_definition.VariableDefinition]*

Extracts the variables from the `dataset_definition` portion of the config file.

Parameters

- **dictionary** (*Dict*) – The `dataset_definition` dictionary from the config file.
- **available_dimensions** (*Dict[str, DimensionDefinition]*) – The `DimensionDefinition` objects that have already been parsed.

Returns Returns a mapping of output variable names to `VariableDefinition` objects.

Return type Dict[str, VariableDefinition]

_parse_coordinates (*self*, *vars*: Dict[str, [tsdat.config.variable_definition.VariableDefinition](#)]) → Tuple[Dict[str, [tsdat.config.variable_definition.VariableDefinition](#)], Dict[str, [tsdat.config.variable_definition.VariableDefinition](#)]]

Separates coordinate variables and data variables.

Determines which variables are coordinate variables and moves those variables from `self.vars` to `self.coords`. Coordinate variables are defined as variables that are dimensioned by themselves, i.e., `var.name == var.dim.name` is a true statement for coordinate variables, but false for data variables.

Parameters **vars** (Dict[str, VariableDefinition]) – The dictionary of VariableDefinition objects to check.

Returns The dictionary of dimensions in the dataset.

Return type Tuple[Dict[str, VariableDefinition], Dict[str, VariableDefinition]]

_validate_dataset_definition (*self*)

Performs sanity checks on the DatasetDefinition object.

Raises **DefinitionError** – If any sanity checks fail.

get_attr (*self*, *attribute_name*) → Any

Retrieves the value of the attribute requested, or None if it does not exist.

Parameters **attribute_name** (str) – The name of the attribute to retrieve.

Returns The value of the attribute, or None.

Return type Any

get_variable_names (*self*) → List[str]

Retrieves the list of variable names. Note that this excludes coordinate variables.

Returns The list of variable names.

Return type List[str]

get_variable (*self*, *variable_name*: str) → [tsdat.config.variable_definition.VariableDefinition](#)

Attempts to retrieve the requested variable. First searches the data variables, then searches the coordinate variables. Returns None if no data or coordinate variables have been defined with the requested variable name.

Parameters **variable_name** (str) – The name of the variable to retrieve.

Returns Returns the VariableDefinition for the variable, or None if the variable could not be found.

Return type VariableDefinition

get_coordinates (*self*, *variable*: [tsdat.config.variable_definition.VariableDefinition](#)) → List[[tsdat.config.variable_definition.VariableDefinition](#)]

Returns the coordinate VariableDefinition object(s) that dimension the requested VariableDefinition.

Parameters **variable** (VariableDefinition) – The VariableDefinition whose coordinate variables should be retrieved.

Returns A list of VariableDefinition coordinate variables that dimension the provided VariableDefinition.

Return type List[VariableDefinition]

get_static_variables (*self*) → List[[tsdat.config.variable_definition.VariableDefinition](#)]

Retrieves a list of static VariableDefinition objects. A variable is defined as static if it has a “data” section

in the config file, which would mean that the variable’s data is defined statically. For example, in the config file snippet below, “depth” is a static variable:

```
depth:
  data: [4, 8, 12]
  dims: [depth]
  type: int
  attrs:
    long_name: Depth
    units: m
```

Returns The list of static VariableDefinition objects.

Return type List[VariableDefinition]

`tsdat.config.dimension_definition`

Module Contents

Classes

<i>DimKeys</i>	Class that provides a handle for keys in the Dimensions section fo the
<i>DimensionDefinition</i>	Class to represent dimensions defined in the pipeline config file.

class `tsdat.config.dimension_definition.DimKeys`

Class that provides a handle for keys in the Dimensions section fo the dataset_definition

LENGTH = length

class `tsdat.config.dimension_definition.DimensionDefinition` (*name: str, length: Union[str, int]*)

Class to represent dimensions defined in the pipeline config file.

Parameters

- **name** (*str*) – The name of the dimension
- **length** (*Union[str, int]*) – The length of the dimension. This should be one of: "unlimited", "variable", or a positive *int*. The ‘time’ dimension should always have length of "unlimited".

is_unlimited (*self*) → bool

Returns True is the dimension has unlimited length. Represented by setting the length attribute to "unlimited".

Returns True if the dimension has unlimited length.

Return type bool

is_variable_length (*self*) → bool

Returns True if the dimension has variable length, meaning that the dimension’s length is set at runtime. Represented by setting the length to "variable".

Returns True if the dimension has variable length, False otherwise.

Return type bool

`tsdat.config.keys`

Module Contents

Classes

<i>Keys</i>	Class that provides a handle for keys in the pipeline config file.
-------------	--

```
class tsdat.config.keys.Keys
    Class that provides a handle for keys in the pipeline config file.

    PIPELINE = pipeline
    DATASET_DEFINITION = dataset_definition
    DEFAULTS = variable_defaults
    QUALITY_MANAGEMENT = quality_management
    ATTRIBUTES = attributes
    DIMENSIONS = dimensions
    VARIABLES = variables
    ALL = ALL
```

`tsdat.config.pipeline_definition`

Module Contents

Classes

<i>PipelineKeys</i>	Class that provides a handle for keys in the pipeline section of the
<i>PipelineDefinition</i>	Wrapper for the pipeline portion of the pipeline config file.

```
class tsdat.config.pipeline_definition.PipelineKeys
    Class that provides a handle for keys in the pipeline section of the pipeline config file.

    TYPE = type
    INPUT_DATA_LEVEL = input_data_level
    OUTPUT_DATA_LEVEL = data_level
    LOCATION_ID = location_id
    DATASET_NAME = dataset_name
    QUALIFIER = qualifier
    TEMPORAL = temporal
```

```
class tsdat.config.pipeline_definition.PipelineDefinition (dictionary: Dict[str, Dict])
```

Wrapper for the pipeline portion of the pipeline config file.

Parameters **dictionary** (*Dict[str]*) – The pipeline component of the pipeline config file.

Raises **DefinitionError** – Raises DefinitionError if one of the file naming components contains an illegal character.

check_file_name_components (*self*)

Performs sanity checks on the config properties used in naming files output by tsdat pipelines.

Raises **DefinitionError** – Raises DefinitionError if a component has been set improperly.

tsdat.config.quality_manager_definition

Module Contents

Classes

<i>QualityManagerKeys</i>	Class that provides a handle for keys in the quality management section
<i>QualityManagerDefinition</i>	Wrapper for the quality_management portion of the pipeline config

```
class tsdat.config.quality_manager_definition.QualityManagerKeys
```

Class that provides a handle for keys in the quality management section of the pipeline config file.

VARIABLES = **variables**

EXCLUDE = **exclude**

CHECKER = **checker**

HANDLERS = **handlers**

```
class tsdat.config.quality_manager_definition.QualityManagerDefinition (name: str, dictionary: Dict)
```

Wrapper for the quality_management portion of the pipeline config file.

Parameters

- **name** (*str*) – The name of the quality manager in the config file.
- **dictionary** (*Dict*) – The dictionary contents of the quality manager from the config file.

tsdat.config.utils

Module Contents

Functions

<code>configure_yaml()</code>	Configure yaml to automatically substitute environment variables
<code>instantiate_handler(*args, handler_desc: Dict = None) → Union[object, List[object]]</code>	Class to instantiate one or more classes given a dictionary containing
<code>_instantiate_class(*args, **kwargs)</code>	Instantiates a python class given args and kwargs.
<code>_parse_fully_qualified_name(fully_qualified_name: str) → Tuple[str, str]</code>	Splits a fully qualified name into the module name and the class name.

tsdat.config.utils.**configure_yaml**()

Configure yaml to automatically substitute environment variables referenced by the following syntax:
`${VAR_NAME}`

tsdat.config.utils.**instantiate_handler**(*args, handler_desc: Dict = None) → Union[object, List[object]]

Class to instantiate one or more classes given a dictionary containing the path to the class to instantiate and its parameters (optional). This method returns the handle(s) to the instantiated class(es).

Parameters **handler_desc** (*Dict, optional*) – The dictionary containing at least a `classname` entry, which should be a str that links to a python module on the PYTHON-PATH. The `handler_desc` can also contain a `parameters` entry, which will be passed as a keyword argument to classes instantiated by this method. This parameter defaults to None.

Returns The class, or list of classes specified by the `handler_desc`

Return type Union[object, List[object]]

tsdat.config.utils.**_instantiate_class**(*args, **kwargs)

Instantiates a python class given args and kwargs.

Returns The python class.

Return type object

tsdat.config.utils.**_parse_fully_qualified_name**(fully_qualified_name: str) → Tuple[str, str]

Splits a fully qualified name into the module name and the class name.

Parameters **fully_qualified_name** (*str*) – The fully qualified classname.

Returns Returns the module name and class name.

Return type Tuple[str, str]

`tsdat.config.variable_definition`

Module Contents

Classes

<i>VarKeys</i>	Class that provides a handle for keys in the variables section of the
<i>VarInputKeys</i>	Class that provides a handle for keys in the variable input section of
<i>ConverterKeys</i>	Class that provides a handle for keys in the converter section of
<i>VarInput</i>	Class to explicitly encode fields set by the variable's input source
<i>VariableDefinition</i>	Class to encode variable definitions from the config file. Also provides

class `tsdat.config.variable_definition.VarKeys`

Class that provides a handle for keys in the variables section of the pipeline config file.

INPUT = `input`

DIMS = `dims`

TYPE = `type`

ATTRS = `attrs`

class `tsdat.config.variable_definition.VarInputKeys`

Class that provides a handle for keys in the variable input section of the pipeline config file.

NAME = `name`

CONVERTER = `converter`

UNITS = `units`

REQUIRED = `required`

class `tsdat.config.variable_definition.ConverterKeys`

Class that provides a handle for keys in the converter section of the pipeline config file.

CLASSNAME = `classname`

PARAMETERS = `parameters`

class `tsdat.config.variable_definition.VarInput` (*dictionary: Dict, defaults: Union[Dict, None] = None*)

Class to explicitly encode fields set by the variable's input source defined by the yaml file.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with a variable's input section from the config file.
- **defaults** (*Dict, optional*) – The default input parameters, defaults to { }

is_required (*self*) → bool

```
class tsdat.config.variable_definition.VariableDefinition (name: str, dictionary: Dict, available_dimensions: Dict[str, tsdat.config.dimension_definition.DimensionDefinition], defaults: Union[Dict, None] = None)
```

Class to encode variable definitions from the config file. Also provides a few utility methods.

Parameters

- **name** (*str*) – The name of the variable in the output file.
- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.

:param available_dimensions: A mapping of dimension name to DimensionDefinition objects.

Parameters defaults (*Dict, optional*) – The defaults to use when instantiating this VariableDefinition object, defaults to {}.

_parse_input (*self, dictionary: Dict, defaults: Union[Dict, None] = None*) → *VarInput*
Parses the variable's input property, if it has one, from the variable dictionary.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (*Dict, optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A VarInput object for this VariableDefinition, or None.

Return type *VarInput*

_parse_attributes (*self, dictionary: Dict, defaults: Union[Dict, None] = None*) → *Dict[str, Any]*
Parses the variable's attributes from the variable dictionary.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (*Dict, optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A mapping of attribute name to attribute value.

Return type *Dict[str, Any]*

_parse_dimensions (*self, dictionary: Dict, available_dimensions: Dict[str, tsdat.config.dimension_definition.DimensionDefinition], defaults: Union[Dict, None] = None*) → *Dict[str, tsdat.config.dimension_definition.DimensionDefinition]*
Parses the variable's dimensions from the variable dictionary.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **available_dimensions** – A mapping of dimension name to DimensionDefinition.

- **defaults** (*Dict*, *optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A mapping of dimension name to DimensionDefinition objects.

Return type Dict[str, DimensionDefinition]

`_parse_data_type` (*self*, *dictionary: Dict*, *defaults: Union[Dict, None] = None*) → object

Parses the data_type string and returns the appropriate numpy data type (i.e. “float” -> np.float).

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (*Dict*, *optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Raises **KeyError** – Raises KeyError if the data type in the dictionary does not match a valid data type.

Returns The numpy data type corresponding with the type provided in the yaml file, or data_type if the provided data_type is not in the ME Data Standards list of data types.

Return type object

`add_fillvalue_if_none` (*self*, *attributes: Dict[str, Any]*) → Dict[str, Any]

Adds the _FillValue attribute to the provided attributes dictionary if the _FillValue attribute has not already been defined and returns the modified attributes dictionary.

Parameters **attributes** (*Dict[str, Any]*) – The dictionary containing user-defined variable attributes.

Returns The dictionary containing user-defined variable attributes. Is guaranteed to have a _FillValue attribute.

Return type Dict[str, Any]

`is_constant` (*self*) → bool

Returns True if the variable is a constant. A variable is constant if it does not have any dimensions.

Returns True if the variable is constant, False otherwise.

Return type bool

`is_predefined` (*self*) → bool

Returns True if the variable’s data was predefined in the config yaml file.

Returns True if the variable is predefined, False otherwise.

Return type bool

`is_coordinate` (*self*) → bool

Returns True if the variable is a coordinate variable. A variable is defined as a coordinate variable if it is dimensioned by itself.

Returns True if the variable is a coordinate variable, False otherwise.

Return type bool

`is_derived` (*self*) → bool

Return True if the variable is derived. A variable is derived if it does not have an input and it is not predefined.

Returns True if the Variable is derived, False otherwise.

Return type bool

has_converter (*self*) → bool

Returns True if the variable has an input converter defined, False otherwise.

Returns True if the Variable has a converter defined, False otherwise.

Return type bool

is_required (*self*) → bool

Returns True if the variable has the 'required' property defined and the 'required' property evaluates to True. A required variable is a variable which must be retrieved in the input dataset. If a required variable is not in the input dataset, the process should crash.

Returns True if the variable is required, False otherwise.

Return type bool

has_input (*self*) → bool

Return True if the variable is copied from an input dataset, regardless of whether or not unit and/or naming conversions should be applied.

Returns True if the Variable has an input defined, False otherwise.

Return type bool

get_input_name (*self*) → str

Returns the name of the variable in the input if defined, otherwise returns None.

Returns The name of the variable in the input, or None.

Return type str

get_input_units (*self*) → str

If the variable has input, returns the units of the input variable or the output units if no input units are defined.

Returns The units of the input variable data.

Return type str

get_output_units (*self*) → str

Returns the units of the output data or None if no units attribute has been defined.

Returns The units of the output variable data.

Return type str

get_coordinate_names (*self*) → List[str]

Returns the names of the coordinate VariableDefinition(s) that this VariableDefinition is dimensioned by.

Returns A list of dimension/coordinate variable names.

Return type List[str]

get_shape (*self*) → Tuple[int]

Returns the shape of the data attribute on the VariableDefinition.

Raises **KeyError** – Raises a KeyError if the data attribute has not been set yet.

Returns The shape of the VariableDefinition's data, or None.

Return type Tuple[int]

get_data_type (*self*) → numpy.dtype

Retrieves the variable's data type.

Returns Returns the data type of the variable's data as a numpy dtype.

Return type np.dtype

get_FillValue (*self*) → int

Retrieves the variable's _FillValue attribute, using -9999 as a default if it has not been defined.

Returns Returns the variable's _FillValue.

Return type int

run_converter (*self*, *data*: numpy.ndarray) → numpy.ndarray

If the variable has an input converter, runs the input converter for the input/output units on the provided data.

Parameters **data** (*np.ndarray*) – The data to be converted.

Returns Returns the data after it has been run through the variable's converter.

Return type np.ndarray

to_dict (*self*) → Dict

Returns the Variable as a dictionary to be used to initialize an empty xarray Dataset or DataArray.

Returns a dictionary like (Example is for *temperature*):

```
{
  "dims": ["time"],
  "data": [],
  "attrs": {"units": "degC"}
}
```

Returns A dictionary representation of the variable.

Return type Dict

Package Contents

Classes

<i>Config</i>	Wrapper for the pipeline configuration file.
<i>Keys</i>	Class that provides a handle for keys in the pipeline config file.
<i>DimensionDefinition</i>	Class to represent dimensions defined in the pipeline config file.
<i>PipelineDefinition</i>	Wrapper for the pipeline portion of the pipeline config file.
<i>VariableDefinition</i>	Class to encode variable definitions from the config file. Also provides
<i>DatasetDefinition</i>	Wrapper for the dataset_definition portion of the pipeline config
<i>QualityManagerDefinition</i>	Wrapper for the quality_management portion of the pipeline config

class tsdat.config.**Config** (*dictionary*: Dict)

Wrapper for the pipeline configuration file.

Note: in most cases, `Config.load(filepath)` should be used to instantiate the `Config` class.

Parameters `dictionary` (*Dict*) – The pipeline configuration file as a dictionary.

`_parse_quality_managers` (*self*, *dictionary: Dict*) → *Dict[str, tsdat.config.quality_manager_definition.QualityManagerDefinition]*

Extracts `QualityManagerDefinitions` from the config file.

Parameters `dictionary` (*Dict*) – The quality_management dictionary.

Returns Mapping of quality manager name to `QualityManagerDefinition`

Return type *Dict[str, QualityManagerDefinition]*

classmethod `load` (*self*, *filepaths: List[str]*)

Load one or more yaml pipeline configuration files. Multiple files should only be passed as input if the pipeline configuration file is split across multiple files.

Parameters `filepaths` (*List[str]*) – The path(s) to yaml configuration files to load.

Returns A `Config` object wrapping the yaml configuration file(s).

Return type *Config*

static `lint_yaml` (*filename: str*)

Lints a yaml file and raises an exception if an error is found.

Parameters `filename` (*str*) – The path to the file to lint.

Raises **Exception** – Raises an exception if an error is found.

class `tsdat.config.Keys`

Class that provides a handle for keys in the pipeline config file.

PIPELINE = `pipeline`

DATASET_DEFINITION = `dataset_definition`

DEFAULTS = `variable_defaults`

QUALITY_MANAGEMENT = `quality_management`

ATTRIBUTES = `attributes`

DIMENSIONS = `dimensions`

VARIABLES = `variables`

ALL = `ALL`

class `tsdat.config.DimensionDefinition` (*name: str, length: Union[str, int]*)

Class to represent dimensions defined in the pipeline config file.

Parameters

- **name** (*str*) – The name of the dimension
- **length** (*Union[str, int]*) – The length of the dimension. This should be one of: "unlimited", "variable", or a positive *int*. The 'time' dimension should always have length of "unlimited".

is_unlimited (*self*) → *bool*

Returns `True` if the dimension has unlimited length. Represented by setting the length attribute to "unlimited".

Returns `True` if the dimension has unlimited length.

Return type *bool*

is_variable_length (*self*) → bool

Returns True if the dimension has variable length, meaning that the dimension's length is set at runtime. Represented by setting the length to "variable".

Returns True if the dimension has variable length, False otherwise.

Return type bool

class tsdat.config.PipelineDefinition (*dictionary: Dict[str, Dict]*)

Wrapper for the pipeline portion of the pipeline config file.

Parameters **dictionary** (*Dict[str]*) – The pipeline component of the pipeline config file.

Raises **DefinitionError** – Raises DefinitionError if one of the file naming components contains an illegal character.

check_file_name_components (*self*)

Performs sanity checks on the config properties used in naming files output by tsdat pipelines.

Raises **DefinitionError** – Raises DefinitionError if a component has been set improperly.

class tsdat.config.VariableDefinition (*name: str, dictionary: Dict, available_dimensions: Dict[str, tsdat.config.dimension_definition.DimensionDefinition], defaults: Union[Dict, None] = None*)

Class to encode variable definitions from the config file. Also provides a few utility methods.

Parameters

- **name** (*str*) – The name of the variable in the output file.
- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.

:param **available_dimensions**: A mapping of dimension name to DimensionDefinition objects.

Parameters **defaults** (*Dict, optional*) – The defaults to use when instantiating this VariableDefinition object, defaults to {}.

_parse_input (*self, dictionary: Dict, defaults: Union[Dict, None] = None*) → VarInput

Parses the variable's input property, if it has one, from the variable dictionary.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (*Dict, optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A VarInput object for this VariableDefinition, or None.

Return type VarInput

_parse_attributes (*self, dictionary: Dict, defaults: Union[Dict, None] = None*) → Dict[str, Any]

Parses the variable's attributes from the variable dictionary.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (*Dict, optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A mapping of attribute name to attribute value.

Return type Dict[str, Any]

_parse_dimensions (*self*, *dictionary*: Dict, *available_dimensions*: Dict[str, tsdat.config.dimension_definition.DimensionDefinition], *faults*: Union[Dict, None] = None) → Dict[str, tsdat.config.dimension_definition.DimensionDefinition]

Parses the variable's dimensions from the variable dictionary.

Parameters

- **dictionary** (Dict) – The dictionary entry corresponding with this variable in the config file.
- **available_dimensions** – A mapping of dimension name to DimensionDefinition.
- **defaults** (Dict, optional) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A mapping of dimension name to DimensionDefinition objects.

Return type Dict[str, DimensionDefinition]

_parse_data_type (*self*, *dictionary*: Dict, *defaults*: Union[Dict, None] = None) → object
Parses the data_type string and returns the appropriate numpy data type (i.e. “float” -> np.float).

Parameters

- **dictionary** (Dict) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (Dict, optional) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Raises **KeyError** – Raises KeyError if the data type in the dictionary does not match a valid data type.

Returns The numpy data type corresponding with the type provided in the yaml file, or data_type if the provided data_type is not in the ME Data Standards list of data types.

Return type object

add_fillvalue_if_none (*self*, *attributes*: Dict[str, Any]) → Dict[str, Any]

Adds the _FillValue attribute to the provided attributes dictionary if the _FillValue attribute has not already been defined and returns the modified attributes dictionary.

Parameters **attributes** (Dict[str, Any]) – The dictionary containing user-defined variable attributes.

Returns The dictionary containing user-defined variable attributes. Is guaranteed to have a _FillValue attribute.

Return type Dict[str, Any]

is_constant (*self*) → bool

Returns True if the variable is a constant. A variable is constant if it does not have any dimensions.

Returns True if the variable is constant, False otherwise.

Return type bool

is_predefined (*self*) → bool

Returns True if the variable's data was predefined in the config yaml file.

Returns True if the variable is predefined, False otherwise.

Return type bool

is_coordinate (*self*) → bool

Returns True if the variable is a coordinate variable. A variable is defined as a coordinate variable if it is dimensioned by itself.

Returns True if the variable is a coordinate variable, False otherwise.

Return type bool

is_derived (*self*) → bool

Return True if the variable is derived. A variable is derived if it does not have an input and it is not predefined.

Returns True if the Variable is derived, False otherwise.

Return type bool

has_converter (*self*) → bool

Returns True if the variable has an input converter defined, False otherwise.

Returns True if the Variable has a converter defined, False otherwise.

Return type bool

is_required (*self*) → bool

Returns True if the variable has the 'required' property defined and the 'required' property evaluates to True. A required variable is a variable which must be retrieved in the input dataset. If a required variable is not in the input dataset, the process should crash.

Returns True if the variable is required, False otherwise.

Return type bool

has_input (*self*) → bool

Return True if the variable is copied from an input dataset, regardless of whether or not unit and/or naming conversions should be applied.

Returns True if the Variable has an input defined, False otherwise.

Return type bool

get_input_name (*self*) → str

Returns the name of the variable in the input if defined, otherwise returns None.

Returns The name of the variable in the input, or None.

Return type str

get_input_units (*self*) → str

If the variable has input, returns the units of the input variable or the output units if no input units are defined.

Returns The units of the input variable data.

Return type str

get_output_units (*self*) → str

Returns the units of the output data or None if no units attribute has been defined.

Returns The units of the output variable data.

Return type str

get_coordinate_names (*self*) → List[str]

Returns the names of the coordinate VariableDefinition(s) that this VariableDefinition is dimensioned by.

Returns A list of dimension/coordinate variable names.

Return type List[str]

get_shape (*self*) → Tuple[int]

Returns the shape of the data attribute on the VariableDefinition.

Raises **KeyError** – Raises a KeyError if the data attribute has not been set yet.

Returns The shape of the VariableDefinition's data, or None.

Return type Tuple[int]

get_data_type (*self*) → numpy.dtype

Retrieves the variable's data type.

Returns Returns the data type of the variable's data as a numpy dtype.

Return type np.dtype

get_FillValue (*self*) → int

Retrieves the variable's _FillValue attribute, using -9999 as a default if it has not been defined.

Returns Returns the variable's _FillValue.

Return type int

run_converter (*self*, *data*: numpy.ndarray) → numpy.ndarray

If the variable has an input converter, runs the input converter for the input/output units on the provided data.

Parameters **data** (*np.ndarray*) – The data to be converted.

Returns Returns the data after it has been run through the variable's converter.

Return type np.ndarray

to_dict (*self*) → Dict

Returns the Variable as a dictionary to be used to initialize an empty xarray Dataset or DataArray.

Returns a dictionary like (Example is for *temperature*):

```
{
    "dims": ["time"],
    "data": [],
    "attrs": {"units": "degC"}
}
```

Returns A dictionary representation of the variable.

Return type Dict

class tsdat.config.DatasetDefinition (*dictionary*: Dict, *datastream_name*: str)

Wrapper for the dataset_definition portion of the pipeline config file.

Parameters

- **dictionary** (*Dict*) – The portion of the config file corresponding with the dataset definition.
- **datastream_name** (*str*) – The name of the datastream that the config file is for.

_parse_dimensions (*self*, *dictionary*: Dict) → Dict[str, *tsdat.config.dimension_definition.DimensionDefinition*]

Extracts the dimensions from the dataset_definition portion of the config file.

Parameters **dictionary** (*Dict*) – The dataset_definition dictionary from the config file.

Returns Returns a mapping of output dimension names to DimensionDefinition objects.

Return type Dict[str, *DimensionDefinition*]

_parse_variables (*self*, *dictionary*: *Dict*, *available_dimensions*: *Dict*[str, *tsdat.config.dimension_definition.DimensionDefinition*]) → Dict[str, *tsdat.config.variable_definition.VariableDefinition*]

Extracts the variables from the dataset_definition portion of the config file.

Parameters

- **dictionary** (*Dict*) – The dataset_definition dictionary from the config file.
- **available_dimensions** (*Dict*[str, *DimensionDefinition*]) – The DimensionDefinition objects that have already been parsed.

Returns Returns a mapping of output variable names to VariableDefinition objects.

Return type Dict[str, *VariableDefinition*]

_parse_coordinates (*self*, *vars*: *Dict*[str, *tsdat.config.variable_definition.VariableDefinition*]) → Tuple[Dict[str, *tsdat.config.variable_definition.VariableDefinition*], Dict[str, *tsdat.config.variable_definition.VariableDefinition*]]

Separates coordinate variables and data variables.

Determines which variables are coordinate variables and moves those variables from *self.vars* to *self.coords*. Coordinate variables are defined as variables that are dimensioned by themselves, i.e., *var.name == var.dim.name* is a true statement for coordinate variables, but false for data variables.

Parameters **vars** (*Dict*[str, *VariableDefinition*]) – The dictionary of VariableDefinition objects to check.

Returns The dictionary of dimensions in the dataset.

Return type Tuple[Dict[str, *VariableDefinition*], Dict[str, *VariableDefinition*]]

_validate_dataset_definition (*self*)

Performs sanity checks on the DatasetDefinition object.

Raises **DefinitionError** – If any sanity checks fail.

get_attr (*self*, *attribute_name*) → Any

Retrieves the value of the attribute requested, or None if it does not exist.

Parameters **attribute_name** (*str*) – The name of the attribute to retrieve.

Returns The value of the attribute, or None.

Return type Any

get_variable_names (*self*) → List[str]

Retrieves the list of variable names. Note that this excludes coordinate variables.

Returns The list of variable names.

Return type List[str]

get_variable (*self*, *variable_name*: *str*) → *tsdat.config.variable_definition.VariableDefinition*

Attempts to retrieve the requested variable. First searches the data variables, then searches the coordinate variables. Returns None if no data or coordinate variables have been defined with the requested variable name.

Parameters **variable_name** (*str*) – The name of the variable to retrieve.

Returns Returns the VariableDefinition for the variable, or None if the variable could not be found.

Return type *VariableDefinition*

get_coordinates (*self*, *variable*: *tsdat.config.variable_definition.VariableDefinition*) → *List[tsdat.config.variable_definition.VariableDefinition]*

Returns the coordinate VariableDefinition object(s) that dimension the requested VariableDefinition.

Parameters **variable** (*VariableDefinition*) – The VariableDefinition whose coordinate variables should be retrieved.

Returns A list of VariableDefinition coordinate variables that dimension the provided VariableDefinition.

Return type *List[VariableDefinition]*

get_static_variables (*self*) → *List[tsdat.config.variable_definition.VariableDefinition]*

Retrieves a list of static VariableDefinition objects. A variable is defined as static if it has a “data” section in the config file, which would mean that the variable’s data is defined statically. For example, in the config file snippet below, “depth” is a static variable:

```
depth:
  data: [4, 8, 12]
  dims: [depth]
  type: int
  attrs:
    long_name: Depth
    units: m
```

Returns The list of static VariableDefinition objects.

Return type *List[VariableDefinition]*

class *tsdat.config.QualityManagerDefinition* (*name*: *str*, *dictionary*: *Dict*)

Wrapper for the quality_management portion of the pipeline config file.

Parameters

- **name** (*str*) – The name of the quality manager in the config file.
- **dictionary** (*Dict*) – The dictionary contents of the quality manager from the config file.

tsdat.constants

Module that contains tsdat constants.

Submodules

tsdat.constants.constants

Module Contents

Classes

<i>VARs</i>	Class that adds keywords for referring to variables.
<i>ATTs</i>	Class that adds constants for interacting with tsdat data-model

```
class tsdat.constants.constants.VARs
    Class that adds keywords for referring to variables.

    ALL = ALL

    COORDS = COORDS

    DATA_VARS = DATA_VARS

class tsdat.constants.constants.ATTs
    Class that adds constants for interacting with tsdat data-model specific attributes.

    TITLE = title

    DESCRIPTION = description

    CONVENTIONS = conventions

    HISTORY = history

    DOI = doi

    INSTITUTION = institution

    CODE_URL = code_url

    REFERENCES = references

    INPUT_FILES = input_files

    LOCATION_ID = location_id

    DATASTREAM = datastream_name

    DATA_LEVEL = data_level

    LOCATION_DESCRIPTION = location_description

    INSTRUMENT_NAME = instrument_name

    SERIAL_NUMBER = serial_number

    INSTRUMENT_DESCRIPTION = instrument_description

    INSTRUMENT_MANUFACTURER = instrument_manufacturer

    AVERAGING_INTERVAL = averaging_interval

    SAMPLING_INTERVAL = sampling_interval

    UNITS = units

    VALID_DELTA = valid_delta

    VALID_RANGE = valid_range

    FAIL_RANGE = fail_range

    WARN_RANGE = warn_range

    FILL_VALUE = _FillValue

    CORRECTIONS_APPLIED = corrections_applied
```

Package Contents

Classes

<i>ATTS</i>	Class that adds constants for interacting with tsdat data-model
<i>VARs</i>	Class that adds keywords for referring to variables.

class tsdat.constants.ATTS

Class that adds constants for interacting with tsdat data-model specific attributes.

```

TITLE = title
DESCRIPTION = description
CONVENTIONS = conventions
HISTORY = history
DOI = doi
INSTITUTION = institution
CODE_URL = code_url
REFERENCES = references
INPUT_FILES = input_files
LOCATION_ID = location_id
DATASTREAM = datastream_name
DATA_LEVEL = data_level
LOCATION_DESCRIPTION = location_description
INSTRUMENT_NAME = instrument_name
SERIAL_NUMBER = serial_number
INSTRUMENT_DESCRIPTION = instrument_description
INSTRUMENT_MANUFACTURER = instrument_manufacturer
AVERAGING_INTERVAL = averaging_interval
SAMPLING_INTERVAL = sampling_interval
UNITS = units
VALID_DELTA = valid_delta
VALID_RANGE = valid_range
FAIL_RANGE = fail_range
WARN_RANGE = warn_range
FILL_VALUE = _FillValue
CORRECTIONS_APPLIED = corrections_applied

```

class tsdat.constants.VARS

Class that adds keywords for referring to variables.

```
ALL = ALL
COORDS = COORDS
DATA_VARS = DATA_VARS
```

`tsdat.exceptions`

Module that contains tsdat exception and warning classes

Submodules

`tsdat.exceptions.exceptions`

Module Contents

exception `tsdat.exceptions.exceptions.QCError`

Bases: `Exception`

Indicates that a given Quality Manager failed with a fatal error.

exception `tsdat.exceptions.exceptions.DefinitionError`

Bases: `Exception`

Indicates a fatal error within the YAML Dataset Definition.

Package Contents

exception `tsdat.exceptions.QCError`

Bases: `Exception`

Indicates that a given Quality Manager failed with a fatal error.

exception `tsdat.exceptions.DefinitionError`

Bases: `Exception`

Indicates a fatal error within the YAML Dataset Definition.

`tsdat.io`

The `tsdat.io` package provides the classes that the data pipeline uses to manage I/O for the pipeline. Specifically, it includes:

1. The `FileHandler` infrastructure used to read/write to/from specific file formats, and
2. The `Storage` infrastructure used to store/access processed data files

We warmly welcome community contributions to increase the list of supported `FileHandlers` and `Storage` destinations.

Subpackages

`tsdat.io.filehandlers`

This module contains the File Handlers that come packaged with tsdat in addition to methods for registering new File Handler objects.

Submodules

`tsdat.io.filehandlers.csv_handler`

Module Contents

Classes

<i>CsvHandler</i>	FileHandler to read from and write to CSV files. Takes a number of
-------------------	--

class `tsdat.io.filehandlers.csv_handler.CsvHandler` (*parameters: Union[Dict, None] = None*)

Bases: `tsdat.io.filehandlers.file_handlers.AbstractFileHandler`

FileHandler to read from and write to CSV files. Takes a number of parameters that are passed in from the storage config file. Parameters specified in the config file should follow the following example:

```
parameters:
  write:
    to_dataframe:
      # Parameters here will be passed to xr.Dataset.to_dataframe()
    to_csv:
      # Parameters here will be passed to pd.DataFrame.to_csv()
  read:
    read_csv:
      # Parameters here will be passed to pd.read_csv()
    to_xarray:
      # Parameters here will be passed to pd.DataFrame.to_xarray()
```

Parameters `parameters` (*Dict, optional*) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (*self, ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs*) → None
Saves the given dataset to a csv file.

Parameters

- **ds** (*xr.Dataset*) – The dataset to save.
- **filename** (*str*) – The path to where the file should be written to.
- **config** (*Config, optional*) – Optional Config object, defaults to None

read (*self, filename: str, **kwargs*) → *xarray.Dataset*
Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters `filename` (*str*) – The path to the file to read in.

Returns A `xr.Dataset` object.

Return type `xr.Dataset`

`tsdat.io.filehandlers.file_handlers`

Module Contents

Classes

<code>AbstractFileHandler</code>	Abstract class to define methods required by all File-Handlers. Classes
<code>FileHandler</code>	Class to provide methods to read and write files with a variety of

Functions

<code>register_filehandler(patterns: Union[str, List[str]]) → AbstractFileHandler</code>	Python decorator to register an <code>AbstractFileHandler</code> in the <code>FileHandler</code>
--	--

class `tsdat.io.filehandlers.file_handlers.AbstractFileHandler` (*parameters:*
Union[Dict,
None] = None)

Abstract class to define methods required by all `FileHandlers`. Classes derived from `AbstractFileHandler` should implement one or more of the following methods:

`write(ds: xr.Dataset, filename: str, config: Config, **kwargs)`

`read(filename: str, **kwargs) → xr.Dataset`

Parameters `parameters` (*Dict, optional*) – Parameters that were passed to the `FileHandler` when it was registered in the storage config file, defaults to `{}`.

write (*self, ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs*) → `None`
 Saves the given dataset to a file.

Parameters

- **ds** (*xr.Dataset*) – The dataset to save.
- **filename** (*str*) – The path to where the file should be written to.
- **config** (*Config, optional*) – Optional `Config` object, defaults to `None`

read (*self, filename: str, **kwargs*) → `xarray.Dataset`

Reads in the given file and converts it into an `Xarray` dataset for use in the pipeline.

Parameters `filename` (*str*) – The path to the file to read in.

Returns A `xr.Dataset` object.

Return type `xr.Dataset`

class `tsdat.io.filehandlers.file_handlers.FileHandler`
 Class to provide methods to read and write files with a variety of extensions.

FILEREADERS :`Dict[str, AbstractFileHandler]`

FILEWRITERS :Dict[str, AbstractFileHandler]

static `_get_handler` (*filename: str, method: Literal[read, write]*) → *AbstractFileHandler*

Given the filepath of the file to read or write and the FileHandler method to apply to the filepath, this method determines which previously-registered FileHandler should be used on the provided filepath.

Args: filename (str): The path to the file to read or write to. method (Literal[]): The method to apply to the file. Must be one of: “read”, “write”.

Returns: AbstractFileHandler: The FileHandler that should be applied.

static `write` (*ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs*) → *None*

Calls the appropriate FileHandler to write the dataset to the provided filename.

Args: ds (xr.Dataset): The dataset to save. filename (str): The path to the file where the dataset should be written. config (Config, optional): Optional Config object. Defaults to None.

static `read` (*filename: str, **kwargs*) → *xarray.Dataset*

Reads in the given file and converts it into an xarray dataset object using the registered FileHandler for the provided filepath.

Args: filename (str): The path to the file to read in.

Returns: xr.Dataset: The raw file as an Xarray.Dataset object.

static `register_file_handler` (*method: Literal[read, write], patterns: Union[str, List[str]], handler: AbstractFileHandler*)

Method to register a FileHandler for reading from or writing to files matching one or more provided file patterns.

Args: method (“Literal”): The method the FileHandler should call if the pattern is matched. Must be one of: “read”, “write”. patterns (Union[str, List[str]]): The file pattern(s) that determine if this FileHandler should be run on a given filepath. handler (AbstractFileHandler): The AbstractFileHandler to register.

`tsdat.io.filehandlers.file_handlers.register_filehandler` (*patterns: Union[str, List[str]]*) → *AbstractFileHandler*

Python decorator to register an AbstractFileHandler in the FileHandler object. The FileHandler object will be used by tsdat pipelines to read and write raw, intermediate, and processed data.

This decorator can be used to work with a specific AbstractFileHandler without having to specify a config file. This is useful when using an AbstractFileHandler for analysis or for tests outside of a pipeline. For tsdat pipelines, handlers should always be specified via the storage config file.

Example Usage:

```
import xarray as xr
from tsdat.io import register_filehandler, AbstractFileHandler

@register_filehandler(["*.nc", "*.cdf"])
class NetCdfHandler(AbstractFileHandler):
    def write(ds: xr.Dataset, filename: str, config: Config = None, **kwargs):
        ds.to_netcdf(filename)
    def read(filename: str, **kwargs) -> xr.Dataset:
        xr.load_dataset(filename)
```

Parameters `patterns` (*Union[str, List[str]]*) – The patterns (regex) that should be used to match a filepath to the AbstractFileHandler provided.

Returns The original AbstractFileHandler class, after it has been registered for use in tsdat pipelines.

Return type *AbstractFileHandler*

`tsdat.io.filehandlers.netcdf_handler`

Module Contents

Classes

<i>NetCdfHandler</i>	FileHandler to read from and write to netCDF files. Takes a number of
----------------------	--

class `tsdat.io.filehandlers.netcdf_handler.NetCdfHandler` (*parameters: Union[Dict, None] = None*)
Bases: `tsdat.io.filehandlers.file_handlers.AbstractFileHandler`

FileHandler to read from and write to netCDF files. Takes a number of parameters that are passed in from the storage config file. Parameters specified in the config file should follow the following example:

```
parameters:
  write:
    to_netcdf:
      # Parameters here will be passed to xr.Dataset.to_netcdf()
  read:
    load_dataset:
      # Parameters here will be passed to xr.load_dataset()
```

Parameters `parameters` (*Dict, optional*) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (*self, ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs*) → None
Saves the given dataset to a netCDF file.

Parameters

- **ds** (*xr.Dataset*) – The dataset to save.
- **filename** (*str*) – The path to where the file should be written to.
- **config** (*Config, optional*) – Optional Config object, defaults to None

read (*self, filename: str, **kwargs*) → *xarray.Dataset*
Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters **filename** (*str*) – The path to the file to read in.

Returns A *xr.Dataset* object.

Return type *xr.Dataset*

Package Contents

Classes

<i>AbstractFileHandler</i>	Abstract class to define methods required by all File-Handlers. Classes
<i>FileHandler</i>	Class to provide methods to read and write files with a variety of
<i>CsvHandler</i>	FileHandler to read from and write to CSV files. Takes a number of
<i>NetCdfHandler</i>	FileHandler to read from and write to netCDF files. Takes a number of

Functions

<i>register_filehandler</i> (patterns: Union[str, List[str]]) → AbstractFileHandler	Python decorator to register an AbstractFileHandler in the FileHandler
---	--

class tsdat.io.filehandlers.**AbstractFileHandler** (parameters: Union[Dict, None] = None)

Abstract class to define methods required by all FileHandlers. Classes derived from AbstractFileHandler should implement one or more of the following methods:

`write(ds: xr.Dataset, filename: str, config: Config, **kwargs)`

`read(filename: str, **kwargs) -> xr.Dataset`

Parameters **parameters** (Dict, optional) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (self, ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs) → None
Saves the given dataset to a file.

Parameters

- **ds** (xr.Dataset) – The dataset to save.
- **filename** (str) – The path to where the file should be written to.
- **config** (Config, optional) – Optional Config object, defaults to None

read (self, filename: str, **kwargs) → xarray.Dataset

Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters **filename** (str) – The path to the file to read in.

Returns A xr.Dataset object.

Return type xr.Dataset

class tsdat.io.filehandlers.**FileHandler**

Class to provide methods to read and write files with a variety of extensions.

FILEREADERS :Dict[str, AbstractFileHandler]

FILEWRITERS :Dict[str, AbstractFileHandler]

static `_get_handler` (*filename: str, method: Literal[read, write]*) → *AbstractFileHandler*

Given the filepath of the file to read or write and the FileHandler method to apply to the filepath, this method determines which previously-registered FileHandler should be used on the provided filepath.

Args: filename (str): The path to the file to read or write to. method (Literal[]): The method to apply to the file. Must be one of: “read”, “write”.

Returns: AbstractFileHandler: The FileHandler that should be applied.

static `write` (*ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs*) → *None*

Calls the appropriate FileHandler to write the dataset to the provided filename.

Args: ds (xr.Dataset): The dataset to save. filename (str): The path to the file where the dataset should be written. config (Config, optional): Optional Config object. Defaults to None.

static `read` (*filename: str, **kwargs*) → *xarray.Dataset*

Reads in the given file and converts it into an xarray dataset object using the registered FileHandler for the provided filepath.

Args: filename (str): The path to the file to read in.

Returns: xr.Dataset: The raw file as an Xarray.Dataset object.

static `register_file_handler` (*method: Literal[read, write], patterns: Union[str, List[str]], handler: AbstractFileHandler*)

Method to register a FileHandler for reading from or writing to files matching one or more provided file patterns.

Args: method (“Literal”): The method the FileHandler should call if the pattern is matched. Must be one of: “read”, “write”. patterns (Union[str, List[str]]): The file pattern(s) that determine if this FileHandler should be run on a given filepath. handler (AbstractFileHandler): The AbstractFileHandler to register.

`tsdat.io.filehandlers.register_filehandler` (*patterns: Union[str, List[str]]*) → *AbstractFileHandler*

Python decorator to register an AbstractFileHandler in the FileHandler object. The FileHandler object will be used by tsdat pipelines to read and write raw, intermediate, and processed data.

This decorator can be used to work with a specific AbstractFileHandler without having to specify a config file. This is useful when using an AbstractFileHandler for analysis or for tests outside of a pipeline. For tsdat pipelines, handlers should always be specified via the storage config file.

Example Usage:

```
import xarray as xr
from tsdat.io import register_filehandler, AbstractFileHandler

@register_filehandler(["*.nc", "*.cdf"])
class NetCdfHandler(AbstractFileHandler):
    def write(ds: xr.Dataset, filename: str, config: Config = None, **kwargs):
        ds.to_netcdf(filename)
    def read(filename: str, **kwargs) -> xr.Dataset:
        xr.load_dataset(filename)
```

Parameters `patterns` (*Union[str, List[str]]*) – The patterns (regex) that should be used to match a filepath to the AbstractFileHandler provided.

Returns The original AbstractFileHandler class, after it has been registered for use in tsdat pipelines.

Return type *AbstractFileHandler*

class tsdat.io.filehandlers.CsvHandler (parameters: Union[Dict, None] = None)
 Bases: tsdat.io.filehandlers.file_handlers.AbstractFileHandler

FileHandler to read from and write to CSV files. Takes a number of parameters that are passed in from the storage config file. Parameters specified in the config file should follow the following example:

```
parameters:
  write:
    to_dataframe:
      # Parameters here will be passed to xr.Dataset.to_dataframe()
    to_csv:
      # Parameters here will be passed to pd.DataFrame.to_csv()
  read:
    read_csv:
      # Parameters here will be passed to pd.read_csv()
    to_xarray:
      # Parameters here will be passed to pd.DataFrame.to_xarray()
```

Parameters **parameters** (Dict, optional) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (self, ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs) → None
 Saves the given dataset to a csv file.

Parameters

- **ds** (xr.Dataset) – The dataset to save.
- **filename** (str) – The path to where the file should be written to.
- **config** (Config, optional) – Optional Config object, defaults to None

read (self, filename: str, **kwargs) → xarray.Dataset
 Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters **filename** (str) – The path to the file to read in.

Returns A xr.Dataset object.

Return type xr.Dataset

class tsdat.io.filehandlers.NetCdfHandler (parameters: Union[Dict, None] = None)
 Bases: tsdat.io.filehandlers.file_handlers.AbstractFileHandler

FileHandler to read from and write to netCDF files. Takes a number of parameters that are passed in from the storage config file. Parameters specified in the config file should follow the following example:

```
parameters:
  write:
    to_netcdf:
      # Parameters here will be passed to xr.Dataset.to_netcdf()
  read:
    load_dataset:
      # Parameters here will be passed to xr.load_dataset()
```

Parameters **parameters** (Dict, optional) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (self, ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs) → None
 Saves the given dataset to a netCDF file.

Parameters

- **ds** (*xr.Dataset*) – The dataset to save.
- **filename** (*str*) – The path to where the file should be written to.
- **config** (*Config, optional*) – Optional Config object, defaults to None

read (*self, filename: str, **kwargs*) → *xarray.Dataset*

Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters **filename** (*str*) – The path to the file to read in.

Returns A *xr.Dataset* object.

Return type *xr.Dataset*

Submodules

`tsdat.io.aws_storage`

Module Contents**Classes**

<i>S3Path</i>	This class wraps a ‘special’ path string that lets us include the
<i>AwsTemporaryStorage</i>	Class used to store temporary files or perform
<i>AwsStorage</i>	DatastreamStorage subclass for an AWS S3-based filesystem.

Attributes

<i>SEPARATOR</i>

`tsdat.io.aws_storage.SEPARATOR = $$$`

class `tsdat.io.aws_storage.S3Path` (*bucket_name: str, bucket_path: str = "", region_name: str = None*)

Bases: `str`

This class wraps a ‘special’ path string that lets us include the bucket name and region in the path, so that we can use it seamlessly in boto3 APIs. We are creating our own string to hold the region, bucket & key (i.e., path), since boto3 needs all three in order to access a file.

Example: .. code-block:: python

```
s3_client = boto3.client('s3', region_name='eu-central-1') s3_client.download_file(bucket, key, download_path)
```

Parameters

- **bucket_name** (*str*) – The S3 bucket name where this file is located

- **bucket_path**(*str*, *optional*) – The key to access this file in the bucket
- **region_name**(*str*, *optional*) – The AWS region where this file is located, defaults to None, which inherits the default configured region.

__str__(*self*)

Return str(self).

property bucket_name(*self*)

property bucket_path(*self*)

property region_name(*self*)

join(*self*, **args*)

Joins segments in an S3 path. This method behaves exactly like os.path.join.

Returns A New S3Path with the additional segments added.

Return type *S3Path*

class tsdat.io.aws_storage.**AwsTemporaryStorage**(**args*, ***kwargs*)

Bases: *tsdat.io.TemporaryStorage*

Class used to store temporary files or perform filesystem actions on files other than datastream files that reside in the same AWS S3 bucket as the DatastreamStorage. This is a helper class intended to be used in the internals of pipeline implementations only. It is not meant as an external API for interacting with files in DatastreamStorage.

property base_path(*self*) → *S3Path*

clean(*self*)

Clean any extraneous files from the temp working dirs. Temp files could be in two places:

1. the local temp folder - used when fetching files from the store
2. the storage temp folder - used when extracting zip files in some stores (e.g., AWS)

This method removes the local temp folder. Child classes can extend this method to clean up their respective storage temp folders.

is_tarfile(*self*, *filepath*)

is_zipfile(*self*, *filepath*)

extract_tarfile(*self*, *filepath*: *S3Path*) → List[*S3Path*]

extract_zipfile(*self*, *filepath*) → List[*S3Path*]

extract_files(*self*, *list_or_filepath*: Union[*S3Path*, List[*S3Path*]]) → *tsdat.io.DisposableStorageTempFileList*

If provided a path to an archive file, this function will extract the archive into a temp directory IN THE SAME FILESYSTEM AS THE STORAGE. This means, for example that if storage was in an s3 bucket, then the files would be extracted to a temp dir in that s3 bucket. This is to prevent local disk limitations when running via Lambda.

If the file is not an archive, then the same file will be returned.

This method supports zip, tar, and tar.g file formats.

Parameters **file_path**(Union[*str*, List[*str*]]) – The path of a file or a list of files that should be processed together, located in the same filesystem as the storage.

Returns A list of paths to the files that were extracted. Files will be located in the temp area of the storage filesystem.

Return type DisposableStorageTempFileList

fetch (*self*, *file_path*: S3Path, *local_dir*=None, *disposable*=True) → *tsdat.io.DisposableLocalTempFile*

Fetch a file from temp storage to a local temp folder. If disposable is True, then a DisposableLocalTempFile will be returned so that it can be used with a context manager.

Parameters

- **file_path** (*str*) – The path of a file located in the same filesystem as the storage.
- **local_dir** (*[type]*, *optional*) – The destination folder for the file. If not specified, it will be created in the storage-approved local temp folder. defaults to None.
- **disposable** (*bool*, *optional*) – True if this file should be auto-deleted when it goes out of scope. Defaults to True.

Returns If disposable, return a DisposableLocalTempFile, otherwise return the path to the local file.

Return type Union[DisposableLocalTempFile, str]

fetch_previous_file (*self*, *datastream_name*: str, *start_time*: str) → *tsdat.io.DisposableLocalTempFile*

Look in DatastreamStorage for the first processed file before the given date.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.

Returns If a previous file was found, return the local path to the fetched file. Otherwise return None. (Return value wrapped in DisposableLocalTempFile so it can be auto-deleted if needed.)

Return type DisposableLocalTempFile

delete (*self*, *filepath*: S3Path) → None

Remove a file from storage temp area if the file exists. If the file does not exist, this method will NOT raise an exception.

Parameters **file_path** (*str*) – The path of a file located in the same filesystem as the storage.

listdir (*self*, *filepath*: S3Path) → List[S3Path]

upload (*self*, *local_path*: str, *s3_path*: S3Path)

class *tsdat.io.aws_storage.AwsStorage* (*parameters*: Union[Dict, None] = None)

Bases: *tsdat.io.DatastreamStorage*

DatastreamStorage subclass for an AWS S3-based filesystem.

Parameters **parameters** (*dict*, *optional*) – Dictionary of parameters that should be set automatically from the storage config file when this class is instantiated via the DatastreamStorage.from-config() method. Defaults to {}

Key parameters that should be set in the config file include

retain_input_files Whether the input files should be cleaned up after they are done processing

root_dir The bucket ‘key’ to use to prepend to all processed files created in the persistent store. Defaults to ‘root’

temp_dir The bucket ‘key’ to use to prepend to all temp files created in the S3 bucket.
Defaults to ‘temp’

bucket_name The name of the S3 bucket to store to

property s3_resource (*self*)

property s3_client (*self*)

property tmp (*self*)

Each subclass should define the tmp property, which provides access to a TemporaryStorage object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the DatastreamStorage. Is is not intended to be used outside of the pipeline.

Raises NotImplementedError – [description]

property root (*self*)

property temp_path (*self*)

find (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *filetype*: str = None) → List[S3Path]

Finds all files of the given type from the datastream store with the given datastream_name and timestamps from start_time (inclusive) up to end_time (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths in datastream storage in ascending order

Return type List[str]

fetch (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *local_path*: str = None, *filetype*: int = None) → *tsdat.io.DisposableLocalTempFileList*

Fetches files from the datastream store using the datastream_name, start_time, and end_time to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str*, *optional*) – The path to the directory where the data should be stored. Defaults to None.
- **filetype** (*int*, *optional*) – A file type from the DatastreamStorage.file_filters keys If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so it this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type DisposableLocalTempFileList:

save_local_path (*self*, *local_path*: *str*, *new_filename*: *str* = *None*)

Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str*, *optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for dataset_or_path. Must also follow ME Data Standards naming conventions. Defaults to None.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type Any

exists (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *int* = *None*) → bool

Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If none specified, all files will be checked. Defaults to None.

Returns True if data exists, False otherwise.

Return type bool

delete (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *int* = *None*) → None

Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be deleted. Defaults to None.

`tsdat.io.filesystem_storage`

Module Contents

Classes

<code>FilesystemTemporaryStorage</code>	Class used to store temporary files or perform
<code>FilesystemStorage</code>	Datastreamstorage subclass for a local Linux-based filesystem.

class `tsdat.io.filesystem_storage.FilesystemTemporaryStorage` (*storage: DatastreamStorage*)

Bases: `tsdat.io.TemporaryStorage`

Class used to store temporary files or perform filesystem actions on files other than datastream files that reside in the same local filesystem as the DatastreamStorage. This is a helper class intended to be used in the internals of pipeline implementations only. It is not meant as an external API for interacting with files in DatastreamStorage.

extract_files (*self*, *list_or_filepath: Union[str, List[str]]*) → *ts-dat.io.DisposableStorageTempFileList*

If provided a path to an archive file, this function will extract the archive into a temp directory IN THE SAME FILESYSTEM AS THE STORAGE. This means, for example that if storage was in an s3 bucket, then the files would be extracted to a temp dir in that s3 bucket. This is to prevent local disk limitations when running via Lambda.

If the file is not an archive, then the same file will be returned.

This method supports zip, tar, and tar.g file formats.

Parameters `file_path` (*Union[str, List[str]]*) – The path of a file or a list of files that should be processed together, located in the same filesystem as the storage.

Returns A list of paths to the files that were extracted. Files will be located in the temp area of the storage filesystem.

Return type `DisposableStorageTempFileList`

fetch (*self*, *file_path: str*, *local_dir=None*, *disposable=True*) → *Union[tsdat.io.DisposableLocalTempFile, str]*

Fetch a file from temp storage to a local temp folder. If disposable is True, then a DisposableLocalTempFile will be returned so that it can be used with a context manager.

Parameters

- **file_path** (*str*) – The path of a file located in the same filesystem as the storage.
- **local_dir** (*[type], optional*) – The destination folder for the file. If not specified, it will be created in the storage-approved local temp folder. defaults to None.
- **disposable** (*bool, optional*) – True if this file should be auto-deleted when it goes out of scope. Defaults to True.

Returns If disposable, return a DisposableLocalTempFile, otherwise return the path to the local file.

Return type `Union[DisposableLocalTempFile, str]`

fetch_previous_file (*self*, *datastream_name*: *str*, *start_time*: *str*) → *tsdat.io.DisposableLocalTempFile*

Look in DatastreamStorage for the first processed file before the given date.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.

Returns If a previous file was found, return the local path to the fetched file. Otherwise return None. (Return value wrapped in DisposableLocalTempFile so it can be auto-deleted if needed.)

Return type DisposableLocalTempFile

delete (*self*, *file_path*: *str*) → None

Remove a file from storage temp area if the file exists. If the file does not exist, this method will NOT raise an exception.

Parameters **file_path** (*str*) – The path of a file located in the same filesystem as the storage.

class `tsdat.io.filesystem_storage.FilesystemStorage` (*parameters*: *Union[Dict, None]* = None)

Bases: *tsdat.io.DatastreamStorage*

Datastreamstorage subclass for a local Linux-based filesystem.

TODO: rename to LocalStorage as this is more intuitive.

Parameters **parameters** (*dict*, *optional*) – Dictionary of parameters that should be set automatically from the storage config file when this class is instantiated via the DatastreamStorage.from-config() method. Defaults to {}

Key parameters that should be set in the config file include

retain_input_files Whether the input files should be cleaned up after they are done processing

root_dir The root path under which processed files will be stored.

property tmp (*self*)

Each subclass should define the tmp property, which provides access to a TemporaryStorage object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the DatastreamStorage. It is not intended to be used outside of the pipeline.

Raises **NotImplementedError** – [description]

find (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *str* = None) → List[str]

Finds all files of the given type from the datastream store with the given datastream_name and timestamps from start_time (inclusive) up to end_time (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.

- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths in datastream storage in ascending order

Return type List[str]

fetch (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *local_path*: *str* = *None*, *filetype*: *int* = *None*) → *tsdat.io.DisposableLocalTempFileList*

Fetches files from the datastream store using the datastream_name, start_time, and end_time to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str*, *optional*) – The path to the directory where the data should be stored. Defaults to None.
- **filetype** (*int*, *optional*) – A file type from the DatastreamStorage.file_filters keys If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so it this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type DisposableLocalTempFileList:

save_local_path (*self*, *local_path*: *str*, *new_filename*: *str* = *None*) → Any

Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str*, *optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for dataset_or_path. Must also follow ME Data Standards naming conventions. Defaults to None.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type Any

exists (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *int* = *None*) → bool

Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.

- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If none specified, all files will be checked. Defaults to None.

Returns True if data exists, False otherwise.

Return type bool

delete (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *int* = None) → None
Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be deleted. Defaults to None.

tsdat.io.storage

Module Contents

Classes

<i>DatastreamStorage</i>	DatastreamStorage is the base class for providing
<i>DisposableLocalTempFile</i>	DisposableLocalTempFile is a context manager wrapper class for a temp file on
<i>DisposableLocalTempFileList</i>	Provides a context manager wrapper class for a list of
<i>DisposableStorageTempFileList</i>	Provides is a context manager wrapper class for a list of
<i>TemporaryStorage</i>	Each DatastreamStorage should contain a corresponding

Functions

<i>_is_image(x)</i>
<i>_is_raw(x)</i>

tsdat.io.storage.**_is_image** (*x*)

tsdat.io.storage.**_is_raw** (*x*)

class tsdat.io.storage.**DatastreamStorage** (*parameters*: Union[Dict, None] = None)
Bases: abc.ABC

DatastreamStorage is the base class for providing access to processed data files in a persistent archive. DatastreamStorage provides shortcut methods to find files based upon date, datastream name, file type, etc. This is the class that should be used to save and retrieve processed data files. Use the DatastreamStorage.from_config()

method to construct the appropriate subclass instance based upon a storage config file.

default_file_type

file_filters

output_file_extensions

static from_config (*storage_config_file: str*)

Load a yaml config file which provides the storage constructor parameters.

Parameters *storage_config_file* (*str*) – The path to the config file to load

Returns A subclass instance created from the config file.

Return type *DatastreamStorage*

property tmp (*self*)

Each subclass should define the tmp property, which provides access to a TemporaryStorage object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the DatastreamStorage. Is is not intended to be used outside of the pipeline.

Raises **NotImplementedError** – [description]

abstract find (*self, datastream_name: str, start_time: str, end_time: str, filetype: str = None*) → *List[str]*

Finds all files of the given type from the datastream store with the given datastream_name and timestamps from start_time (inclusive) up to end_time (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.
- **filetype** (*str, optional*) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths in datastream storage in ascending order

Return type *List[str]*

abstract fetch (*self, datastream_name: str, start_time: str, end_time: str, local_path: str = None, filetype: int = None*)

Fetches files from the datastream store using the datastream_name, start_time, and end_time to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str, optional*) – The path to the directory where the data should be stored. Defaults to None.

- **filetype** (*int, optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to `None`.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type `DisposableLocalTempFileList`:

save (*self, dataset_or_path: Union[str, xarray.Dataset], new_filename: str = None*) → `List[Any]`

Saves a local file to the datastream store.

Parameters

- **dataset_or_path** (*Union[str, xr.Dataset]*) – The dataset or local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str, optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for `dataset_or_path`. Must also follow ME Data Standards naming conventions. Defaults to `None`.

Returns A list of paths where the saved files were stored in storage. Path type is dependent upon the specific storage subclass.

Return type `List[Any]`

abstract save_local_path (*self, local_path: str, new_filename: str = None*) → `Any`

Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str, optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for `dataset_or_path`. Must also follow ME Data Standards naming conventions. Defaults to `None`.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type `Any`

abstract exists (*self, datastream_name: str, start_time: str, end_time: str, filetype: str = None*)

→ `bool`

Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str, optional*) – A file type from the `DatastreamStorage.file_filters` keys. If none specified, all files will be checked. Defaults to `None`.

Returns True if data exists, False otherwise.

Return type bool

abstract delete (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *filetype*: str = None)
→ None

Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be deleted. Defaults to None.

class tsdat.io.storage.DisposableLocalTempFile (*filepath*: str, *disposable*=True)

DisposableLocalTempFile is a context manager wrapper class for a temp file on the LOCAL FILESYSTEM. It will ensure that the file is deleted when it goes out of scope.

Parameters

- **filepath** (*str*) – Path to a local temp file that could be deleted when it goes out of scope.
- **disposable** (*bool*, *optional*) – True if this file should be automatically deleted when it goes out of scope. Defaults to True.

__enter__ (*self*)

__exit__ (*self*, *type*, *value*, *traceback*)

class tsdat.io.storage.DisposableLocalTempFileList (*filepath_list*: List[str],
delete_on_exception=False,
disposable=True)

Bases: list

Provides a context manager wrapper class for a list of temp files on the LOCAL FILESYSTEM. It ensures that if specified, the files will be auto-deleted when the list goes out of scope.

Parameters

- **filepath_list** (*List[str]*) – A list of local temp files
- **delete_on_exception** (*bool*, *optional*) – Should the local temp files be deleted if an error was thrown when processing. Defaults to False.
- **disposable** (*bool*, *optional*) – Should the local temp files be auto-deleted when they go out of scope. Defaults to True.

__enter__ (*self*)

__exit__ (*self*, *type*, *value*, *traceback*)

class tsdat.io.storage.DisposableStorageTempFileList (*filepath_list*: List[str], *storage*,
disposable_files: Union[List,
None] = None)

Bases: list

Provides is a context manager wrapper class for a list of temp files on the STORAGE FILESYSTEM. It will ensure that the specified files are deleted when the list goes out of scope.

Parameters

- **filepath_list** (*List[str]*) – A list of files in temporary storage area

- **storage** (`TemporaryStorage`) – The temporary storage service used to clean up temporary files.
- **disposable_files** (`list`, `optional`) – Which of the files from the `filepath_list` should be auto-deleted when the list goes out of scope. Defaults to []

`__enter__` (`self`)

`__exit__` (`self`, `type`, `value`, `traceback`)

class `tsdat.io.storage.TemporaryStorage` (`storage`: `DatastreamStorage`)

Bases: `abc.ABC`

Each `DatastreamStorage` should contain a corresponding `TemporaryStorage` class which provides access to a `TemporaryStorage` object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the `DatastreamStorage`.

`TemporaryStorage` methods return a context manager so that the created temporary files can be automatically removed when they go out of scope.

`TemporaryStorage` is a helper class intended to be used in the internals of pipeline implementations only. It is not meant as an external API for interacting with files in `DatastreamStorage`.

TODO: rename to a more intuitive name...

Parameters `storage` (`DatastreamStorage`) – A reference to the corresponding `DatastreamStorage`

property `local_temp_folder` (`self`) → `str`

Default method to get a local temporary folder for use when retrieving files from temporary storage. This method should work for all filesystems, but can be overridden if needed by subclasses.

Returns Path to local temp folder

Return type `str`

clean (`self`)

Clean any extraneous files from the temp working dirs. Temp files could be in two places:

1. the local temp folder - used when fetching files from the store
2. the storage temp folder - used when extracting zip files in some stores (e.g., AWS)

This method removes the local temp folder. Child classes can extend this method to clean up their respective storage temp folders.

ignore_zip_check (`self`, `filepath`: `str`) → `bool`

Return true if this file should be excluded from the zip file check. We need this for Office documents, since they are actually zip files under the hood, so we don't want to try to unzip them.

Parameters `filepath` (`str`) – the file we are potentially extracting

Returns whether we should check if it is a zip or not

Return type `bool`

get_temp_filepath (`self`, `filename`: `str` = `None`, `disposable`: `bool` = `True`) → `DisposableLocalTempFile`

Construct a filepath for a temporary file that will be located in the storage-approved local temp folder and will be deleted when it goes out of scope.

Parameters

- **filename** (`str`, `optional`) – The filename to use for the temp file. If no filename is provided, one will be created. Defaults to `None`

- **disposable** (*bool, optional*) – If true, then wrap in DisposableLocalTempfile so that the file will be removed when it goes out of scope. Defaults to True.

Returns Path to the local file. The file will be automatically deleted when it goes out of scope.

Return type *DisposableLocalTempFile*

create_temp_dir (*self*) → *str*

Create a new, temporary directory under the local tmp area managed by TemporaryStorage.

Returns Path to the local dir.

Return type *str*

abstract extract_files (*self, file_path: Union[str, List[str]]*) → *DisposableStorageTempFileList*

If provided a path to an archive file, this function will extract the archive into a temp directory IN THE SAME FILESYSTEM AS THE STORAGE. This means, for example that if storage was in an s3 bucket, then the files would be extracted to a temp dir in that s3 bucket. This is to prevent local disk limitations when running via Lambda.

If the file is not an archive, then the same file will be returned.

This method supports zip, tar, and tar.g file formats.

Parameters **file_path** (*Union[str, List[str]]*) – The path of a file or a list of files that should be processed together, located in the same filesystem as the storage.

Returns A list of paths to the files that were extracted. Files will be located in the temp area of the storage filesystem.

Return type *DisposableStorageTempFileList*

abstract fetch (*self, file_path: str, local_dir=None, disposable=True*) → *Union[DisposableLocalTempFile, str]*

Fetch a file from temp storage to a local temp folder. If disposable is True, then a DisposableLocalTempFile will be returned so that it can be used with a context manager.

Parameters

- **file_path** (*str*) – The path of a file located in the same filesystem as the storage.
- **local_dir** (*[type], optional*) – The destination folder for the file. If not specified, it will be created in the storage-approved local temp folder. defaults to None.
- **disposable** (*bool, optional*) – True if this file should be auto-deleted when it goes out of scope. Defaults to True.

Returns If disposable, return a DisposableLocalTempFile, otherwise return the path to the local file.

Return type *Union[DisposableLocalTempFile, str]*

abstract fetch_previous_file (*self, datastream_name: str, start_time: str*) → *DisposableLocalTempFile*

Look in DatastreamStorage for the first processed file before the given date.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.

Returns If a previous file was found, return the local path to the fetched file. Otherwise return None. (Return value wrapped in DisposableLocalTempFile so it can be auto-deleted if needed.)

Return type *DisposableLocalTempFile*

abstract delete (*self*, *file_path*: *str*)

Remove a file from storage temp area if the file exists. If the file does not exist, this method will NOT raise an exception.

Parameters *file_path* (*str*) – The path of a file located in the same filesystem as the storage.

Package Contents

Classes

<i>AbstractFileHandler</i>	Abstract class to define methods required by all FileHandlers. Classes
<i>FileHandler</i>	Class to provide methods to read and write files with a variety of
<i>CsvHandler</i>	FileHandler to read from and write to CSV files. Takes a number of
<i>NetCdfHandler</i>	FileHandler to read from and write to netCDF files. Takes a number of
<i>DatastreamStorage</i>	DatastreamStorage is the base class for providing
<i>TemporaryStorage</i>	Each DatastreamStorage should contain a corresponding
<i>DisposableLocalTempFile</i>	DisposableLocalTempFile is a context manager wrapper class for a temp file on
<i>DisposableStorageTempFileList</i>	Provides is a context manager wrapper class for a list of
<i>DisposableLocalTempFileList</i>	Provides a context manager wrapper class for a list of
<i>FilesystemStorage</i>	Datastreamstorage subclass for a local Linux-based filesystem.
<i>AwsStorage</i>	DatastreamStorage subclass for an AWS S3-based filesystem.
<i>S3Path</i>	This class wraps a ‘special’ path string that lets us include the

Functions

<i>register_filehandler</i> (patterns: Union[str, List[str]]) → AbstractFileHandler	Python decorator to register an AbstractFileHandler in the FileHandler
---	--

class `tsdat.io.AbstractFileHandler` (*parameters*: Union[Dict, None] = None)

Abstract class to define methods required by all FileHandlers. Classes derived from AbstractFileHandler should implement one or more of the following methods:

`write(ds: xr.Dataset, filename: str, config: Config, **kwargs)`

`read(filename: str, **kwargs) -> xr.Dataset`

Parameters `parameters` (*Dict*, *optional*) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (*self*, *ds*: *xarray.Dataset*, *filename*: *str*, *config*: *tsdat.config.Config* = *None*, ***kwargs*) → *None*
Saves the given dataset to a file.

Parameters

- **ds** (*xr.Dataset*) – The dataset to save.
- **filename** (*str*) – The path to where the file should be written to.
- **config** (*Config*, *optional*) – Optional Config object, defaults to None

read (*self*, *filename*: *str*, ***kwargs*) → *xarray.Dataset*
Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters **filename** (*str*) – The path to the file to read in.

Returns A *xr.Dataset* object.

Return type *xr.Dataset*

class *tsdat.io.FileHandler*

Class to provide methods to read and write files with a variety of extensions.

FILEREADERS :*Dict[str, AbstractFileHandler]*

FILEWRITERS :*Dict[str, AbstractFileHandler]*

static **_get_handler** (*filename*: *str*, *method*: *Literal[read, write]*) → *AbstractFileHandler*

Given the filepath of the file to read or write and the FileHandler method to apply to the filepath, this method determines which previously-registered FileHandler should be used on the provided filepath.

Args: *filename* (*str*): The path to the file to read or write to. *method* (*Literal*): The method to apply to the file. Must be one of: “read”, “write”.

Returns: *AbstractFileHandler*: The FileHandler that should be applied.

static **write** (*ds*: *xarray.Dataset*, *filename*: *str*, *config*: *tsdat.config.Config* = *None*, ***kwargs*) → *None*

Calls the appropriate FileHandler to write the dataset to the provided filename.

Args: *ds* (*xr.Dataset*): The dataset to save. *filename* (*str*): The path to the file where the dataset should be written. *config* (*Config*, *optional*): Optional Config object. Defaults to None.

static **read** (*filename*: *str*, ***kwargs*) → *xarray.Dataset*

Reads in the given file and converts it into an xarray dataset object using the registered FileHandler for the provided filepath.

Args: *filename* (*str*): The path to the file to read in.

Returns: *xr.Dataset*: The raw file as an Xarray.Dataset object.

static **register_file_handler** (*method*: *Literal[read, write]*, *patterns*: *Union[str, List[str]]*, *handler*: *AbstractFileHandler*)

Method to register a FileHandler for reading from or writing to files matching one or more provided file patterns.

Args: *method* (“Literal”): The method the FileHandler should call if the pattern is matched. Must be one of: “read”, “write”. *patterns* (*Union[str, List[str]]*): The file pattern(s) that determine if this FileHandler should be run on a given filepath. *handler* (*AbstractFileHandler*): The AbstractFileHandler to register.

`tsdat.io.register_filehandler` (*patterns: Union[str, List[str]]*) → *AbstractFileHandler*

Python decorator to register an `AbstractFileHandler` in the `FileHandler` object. The `FileHandler` object will be used by tsdat pipelines to read and write raw, intermediate, and processed data.

This decorator can be used to work with a specific `AbstractFileHandler` without having to specify a config file. This is useful when using an `AbstractFileHandler` for analysis or for tests outside of a pipeline. For tsdat pipelines, handlers should always be specified via the storage config file.

Example Usage:

```
import xarray as xr
from tsdat.io import register_filehandler, AbstractFileHandler

@register_filehandler(["*.nc", "*.cdf"])
class NetCdfHandler(AbstractFileHandler):
    def write(ds: xr.Dataset, filename: str, config: Config = None, **kwargs):
        ds.to_netcdf(filename)
    def read(filename: str, **kwargs) -> xr.Dataset:
        xr.load_dataset(filename)
```

Parameters `patterns` (*Union[str, List[str]]*) – The patterns (regex) that should be used to match a filepath to the `AbstractFileHandler` provided.

Returns The original `AbstractFileHandler` class, after it has been registered for use in tsdat pipelines.

Return type *AbstractFileHandler*

`class tsdat.io.CsvHandler` (*parameters: Union[Dict, None] = None*)

Bases: `tsdat.io.filehandlers.file_handlers.AbstractFileHandler`

`FileHandler` to read from and write to CSV files. Takes a number of parameters that are passed in from the storage config file. Parameters specified in the config file should follow the following example:

```
parameters:
write:
    to_dataframe:
        # Parameters here will be passed to xr.Dataset.to_dataframe()
    to_csv:
        # Parameters here will be passed to pd.DataFrame.to_csv()
read:
    read_csv:
        # Parameters here will be passed to pd.read_csv()
    to_xarray:
        # Parameters here will be passed to pd.DataFrame.to_xarray()
```

Parameters `parameters` (*Dict, optional*) – Parameters that were passed to the `FileHandler` when it was registered in the storage config file, defaults to `{}`.

write (*self, ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs*) → *None*
Saves the given dataset to a csv file.

Parameters

- **ds** (*xr.Dataset*) – The dataset to save.
- **filename** (*str*) – The path to where the file should be written to.
- **config** (*Config, optional*) – Optional `Config` object, defaults to `None`

read (*self*, *filename*: *str*, ***kwargs*) → *xarray.Dataset*

Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters *filename* (*str*) – The path to the file to read in.

Returns A *xr.Dataset* object.

Return type *xr.Dataset*

class *tsdat.io.NetCdfHandler* (*parameters*: *Union[Dict, None]* = *None*)

Bases: *tsdat.io.filehandlers.file_handlers.AbstractFileHandler*

FileHandler to read from and write to netCDF files. Takes a number of parameters that are passed in from the storage config file. Parameters specified in the config file should follow the following example:

```
parameters:
  write:
    to_netcdf:
      # Parameters here will be passed to xr.Dataset.to_netcdf()
  read:
    load_dataset:
      # Parameters here will be passed to xr.load_dataset()
```

Parameters *parameters* (*Dict*, *optional*) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (*self*, *ds*: *xarray.Dataset*, *filename*: *str*, *config*: *tsdat.config.Config* = *None*, ***kwargs*) → *None*

Saves the given dataset to a netCDF file.

Parameters

- *ds* (*xr.Dataset*) – The dataset to save.
- *filename* (*str*) – The path to where the file should be written to.
- *config* (*Config*, *optional*) – Optional Config object, defaults to *None*

read (*self*, *filename*: *str*, ***kwargs*) → *xarray.Dataset*

Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters *filename* (*str*) – The path to the file to read in.

Returns A *xr.Dataset* object.

Return type *xr.Dataset*

class *tsdat.io.DatastreamStorage* (*parameters*: *Union[Dict, None]* = *None*)

Bases: *abc.ABC*

DatastreamStorage is the base class for providing access to processed data files in a persistent archive. DatastreamStorage provides shortcut methods to find files based upon date, datastream name, file type, etc. This is the class that should be used to save and retrieve processed data files. Use the DatastreamStorage.from_config() method to construct the appropriate subclass instance based upon a storage config file.

default_file_type

file_filters

output_file_extensions

static from_config (*storage_config_file*: *str*)

Load a yaml config file which provides the storage constructor parameters.

Parameters *storage_config_file* (*str*) – The path to the config file to load

Returns A subclass instance created from the config file.

Return type *DatastreamStorage*

property tmp (*self*)

Each subclass should define the tmp property, which provides access to a TemporaryStorage object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the DatastreamStorage. Is is not intended to be used outside of the pipeline.

Raises **NotImplementedError** – [description]

abstract find (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *filetype*: str = None) → List[str]

Finds all files of the given type from the datastream store with the given datastream_name and timestamps from start_time (inclusive) up to end_time (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths in datastream storage in ascending order

Return type List[str]

abstract fetch (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *local_path*: str = None, *filetype*: int = None)

Fetches files from the datastream store using the datastream_name, start_time, and end_time to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str*, *optional*) – The path to the directory where the data should be stored. Defaults to None.
- **filetype** (*int*, *optional*) – A file type from the DatastreamStorage.file_filters keys If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so it this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type DisposableLocalTempFileList:

save (*self*, *dataset_or_path*: Union[str, xarray.Dataset], *new_filename*: str = None) → List[Any]

Saves a local file to the datastream store.

Parameters

- **dataset_or_path** (*Union[str, xr.Dataset]*) – The dataset or local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str, optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for dataset_or_path. Must also follow ME Data Standards naming conventions. Defaults to None.

Returns A list of paths where the saved files were stored in storage. Path type is dependent upon the specific storage subclass.

Return type List[Any]

abstract save_local_path (*self, local_path: str, new_filename: str = None*) → Any

Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str, optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for dataset_or_path. Must also follow ME Data Standards naming conventions. Defaults to None.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type Any

abstract exists (*self, datastream_name: str, start_time: str, end_time: str, filetype: str = None*) → bool

Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str, optional*) – A file type from the DatastreamStorage.file_filters keys. If none specified, all files will be checked. Defaults to None.

Returns True if data exists, False otherwise.

Return type bool

abstract delete (*self, datastream_name: str, start_time: str, end_time: str, filetype: str = None*) → None

Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.

- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be deleted. Defaults to None.

class `tsdat.io.TemporaryStorage` (*storage*: `DatastreamStorage`)

Bases: `abc.ABC`

Each `DatastreamStorage` should contain a corresponding `TemporaryStorage` class which provides access to a `TemporaryStorage` object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the `DatastreamStorage`.

`TemporaryStorage` methods return a context manager so that the created temporary files can be automatically removed when they go out of scope.

`TemporaryStorage` is a helper class intended to be used in the internals of pipeline implementations only. It is not meant as an external API for interacting with files in `DatastreamStorage`.

TODO: rename to a more intuitive name...

Parameters **storage** (`DatastreamStorage`) – A reference to the corresponding `DatastreamStorage`

property **local_temp_folder** (*self*) → `str`

Default method to get a local temporary folder for use when retrieving files from temporary storage. This method should work for all filesystems, but can be overridden if needed by subclasses.

Returns Path to local temp folder

Return type `str`

clean (*self*)

Clean any extraneous files from the temp working dirs. Temp files could be in two places:

1. the local temp folder - used when fetching files from the store
2. the storage temp folder - used when extracting zip files in some stores (e.g., AWS)

This method removes the local temp folder. Child classes can extend this method to clean up their respective storage temp folders.

ignore_zip_check (*self*, *filepath*: `str`) → `bool`

Return true if this file should be excluded from the zip file check. We need this for Office documents, since they are actually zip files under the hood, so we don't want to try to unzip them.

Parameters **filepath** (*str*) – the file we are potentially extracting

Returns whether we should check if it is a zip or not

Return type `bool`

get_temp_filepath (*self*, *filename*: `str` = `None`, *disposable*: `bool` = `True`) → `DisposableLocalTempFile`

Construct a filepath for a temporary file that will be located in the storage-approved local temp folder and will be deleted when it goes out of scope.

Parameters

- **filename** (*str*, *optional*) – The filename to use for the temp file. If no filename is provided, one will be created. Defaults to None
- **disposable** (*bool*, *optional*) – If true, then wrap in `DisposableLocalTempfile` so that the file will be removed when it goes out of scope. Defaults to True.

Returns Path to the local file. The file will be automatically deleted when it goes out of scope.

Return type *DisposableLocalTempFile*

create_temp_dir (*self*) → str

Create a new, temporary directory under the local tmp area managed by TemporaryStorage.

Returns Path to the local dir.

Return type str

abstract extract_files (*self*, *file_path*: Union[str, List[str]]) → *DisposableStorageTempFileList*

If provided a path to an archive file, this function will extract the archive into a temp directory IN THE SAME FILESYSTEM AS THE STORAGE. This means, for example that if storage was in an s3 bucket, then the files would be extracted to a temp dir in that s3 bucket. This is to prevent local disk limitations when running via Lambda.

If the file is not an archive, then the same file will be returned.

This method supports zip, tar, and tar.g file formats.

Parameters *file_path* (Union[str, List[str]]) – The path of a file or a list of files that should be processed together, located in the same filesystem as the storage.

Returns A list of paths to the files that were extracted. Files will be located in the temp area of the storage filesystem.

Return type *DisposableStorageTempFileList*

abstract fetch (*self*, *file_path*: str, *local_dir*=None, *disposable*=True) → Union[*DisposableLocalTempFile*, str]

Fetch a file from temp storage to a local temp folder. If disposable is True, then a DisposableLocalTempFile will be returned so that it can be used with a context manager.

Parameters

- **file_path** (*str*) – The path of a file located in the same filesystem as the storage.
- **local_dir** (*[type]*, *optional*) – The destination folder for the file. If not specified, it will be created in the storage-approved local temp folder. defaults to None.
- **disposable** (*bool*, *optional*) – True if this file should be auto-deleted when it goes out of scope. Defaults to True.

Returns If disposable, return a DisposableLocalTempFile, otherwise return the path to the local file.

Return type Union[*DisposableLocalTempFile*, str]

abstract fetch_previous_file (*self*, *datastream_name*: str, *start_time*: str) → *DisposableLocalTempFile*

Look in DatastreamStorage for the first processed file before the given date.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.

Returns If a previous file was found, return the local path to the fetched file. Otherwise return None. (Return value wrapped in DisposableLocalTempFile so it can be auto-deleted if needed.)

Return type *DisposableLocalTempFile*

abstract delete (*self*, *file_path*: *str*)

Remove a file from storage temp area if the file exists. If the file does not exist, this method will NOT raise an exception.

Parameters *file_path* (*str*) – The path of a file located in the same filesystem as the storage.

class `tsdat.io.DisposableLocalTempFile` (*filepath*: *str*, *disposable*=*True*)

`DisposableLocalTempFile` is a context manager wrapper class for a temp file on the LOCAL FILESYSTEM. It will ensure that the file is deleted when it goes out of scope.

Parameters

- **filepath** (*str*) – Path to a local temp file that could be deleted when it goes out of scope.
- **disposable** (*bool*, *optional*) – True if this file should be automatically deleted when it goes out of scope. Defaults to True.

`__enter__` (*self*)

`__exit__` (*self*, *type*, *value*, *traceback*)

class `tsdat.io.DisposableStorageTempFileList` (*filepath_list*: *List*[*str*], *storage*, *disposable_files*: *Union*[*List*, *None*] = *None*)

Bases: `list`

Provides is a context manager wrapper class for a list of temp files on the STORAGE FILESYSTEM. It will ensure that the specified files are deleted when the list goes out of scope.

Parameters

- **filepath_list** (*List*[*str*]) – A list of files in temporary storage area
- **storage** (`TemporaryStorage`) – The temporary storage service used to clean up temporary files.
- **disposable_files** (*list*, *optional*) – Which of the files from the *filepath_list* should be auto-deleted when the list goes out of scope. Defaults to []

`__enter__` (*self*)

`__exit__` (*self*, *type*, *value*, *traceback*)

class `tsdat.io.DisposableLocalTempFileList` (*filepath_list*: *List*[*str*], *delete_on_exception*=*False*, *disposable*=*True*)

Bases: `list`

Provides a context manager wrapper class for a list of temp files on the LOCAL FILESYSTEM. It ensures that if specified, the files will be auto-deleted when the list goes out of scope.

Parameters

- **filepath_list** (*List*[*str*]) – A list of local temp files
- **delete_on_exception** (*bool*, *optional*) – Should the local temp files be deleted if an error was thrown when processing. Defaults to False.
- **disposable** (*bool*, *optional*) – Should the local temp files be auto-deleted when they go out of scope. Defaults to True.

`__enter__` (*self*)

`__exit__` (*self*, *type*, *value*, *traceback*)

class `tsdat.io.FilesystemStorage` (*parameters: Union[Dict, None] = None*)

Bases: `tsdat.io.DatastreamStorage`

Datastreamstorage subclass for a local Linux-based filesystem.

TODO: rename to LocalStorage as this is more intuitive.

Parameters `parameters` (*dict, optional*) – Dictionary of parameters that should be set automatically from the storage config file when this class is instantiated via the `DatastreamStorage.from-config()` method. Defaults to {}

Key parameters that should be set in the config file include

retain_input_files Whether the input files should be cleaned up after they are done processing

root_dir The root path under which processed files will be stored.

property `tmp` (*self*)

Each subclass should define the `tmp` property, which provides access to a `TemporaryStorage` object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the `DatastreamStorage`. Is not intended to be used outside of the pipeline.

Raises `NotImplementedError` – [description]

find (*self, datastream_name: str, start_time: str, end_time: str, filetype: str = None*) → `List[str]`

Finds all files of the given type from the datastream store with the given `datastream_name` and timestamps from `start_time` (inclusive) up to `end_time` (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.
- **filetype** (*str, optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to `None`.

Returns A list of paths in datastream storage in ascending order

Return type `List[str]`

fetch (*self, datastream_name: str, start_time: str, end_time: str, local_path: str = None, filetype: int = None*) → `tsdat.io.DisposableLocalTempFileList`

Fetches files from the datastream store using the `datastream_name`, `start_time`, and `end_time` to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str, optional*) – The path to the directory where the data should be stored. Defaults to `None`.

- **filetype** (*int, optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to `None`.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type `DisposableLocalTempFileList`:

save_local_path (*self, local_path: str, new_filename: str = None*) → `Any`

Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str, optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for `dataset_or_path`. Must also follow ME Data Standards naming conventions. Defaults to `None`.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type `Any`

exists (*self, datastream_name: str, start_time: str, end_time: str, filetype: int = None*) → `bool`

Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str, optional*) – A file type from the `DatastreamStorage.file_filters` keys. If none specified, all files will be checked. Defaults to `None`.

Returns True if data exists, False otherwise.

Return type `bool`

delete (*self, datastream_name: str, start_time: str, end_time: str, filetype: int = None*) → `None`

Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str, optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be deleted. Defaults to `None`.

class `tsdat.io.AwsStorage` (*parameters: Union[Dict, None] = None*)

Bases: `tsdat.io.DatastreamStorage`

DatastreamStorage subclass for an AWS S3-based filesystem.

Parameters `parameters` (*dict, optional*) – Dictionary of parameters that should be set automatically from the storage config file when this class is instantiated via the `DatastreamStorage.from-config()` method. Defaults to `{}`

Key parameters that should be set in the config file include

retain_input_files Whether the input files should be cleaned up after they are done processing

root_dir The bucket ‘key’ to use to prepend to all processed files created in the persistent store. Defaults to ‘root’

temp_dir The bucket ‘key’ to use to prepend to all temp files created in the S3 bucket. Defaults to ‘temp’

bucket_name The name of the S3 bucket to store to

property `s3_resource` (*self*)

property `s3_client` (*self*)

property `tmp` (*self*)

Each subclass should define the `tmp` property, which provides access to a `TemporaryStorage` object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the `DatastreamStorage`. Is is not intended to be used outside of the pipeline.

Raises `NotImplementedError` – [description]

property `root` (*self*)

property `temp_path` (*self*)

find (*self*, *datastream_name: str*, *start_time: str*, *end_time: str*, *filetype: str = None*) → `List[S3Path]`

Finds all files of the given type from the datastream store with the given `datastream_name` and timestamps from `start_time` (inclusive) up to `end_time` (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.
- **filetype** (*str, optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to `None`.

Returns A list of paths in datastream storage in ascending order

Return type `List[str]`

fetch (*self*, *datastream_name: str*, *start_time: str*, *end_time: str*, *local_path: str = None*, *filetype: int = None*) → `tsdat.io.DisposableLocalTempFileList`

Fetches files from the datastream store using the `datastream_name`, `start_time`, and `end_time` to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.

- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str*, *optional*) – The path to the directory where the data should be stored. Defaults to None.
- **filetype** (*int*, *optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so it this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type `DisposableLocalTempFileList`:

save_local_path (*self*, *local_path*: *str*, *new_filename*: *str* = *None*)

Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str*, *optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for `dataset_or_path`. Must also follow ME Data Standards naming conventions. Defaults to None.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type `Any`

exists (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *int* = *None*) → `bool`

Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the `DatastreamStorage.file_filters` keys. If none specified, all files will be checked. Defaults to None.

Returns True if data exists, False otherwise.

Return type `bool`

delete (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *int* = *None*) → `None`

Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.

- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be deleted. Defaults to None.

class tsdat.io.S3Path(*bucket_name: str, bucket_path: str = "", region_name: str = None*)

Bases: `str`

This class wraps a ‘special’ path string that lets us include the bucket name and region in the path, so that we can use it seamlessly in boto3 APIs. We are creating our own string to hold the region, bucket & key (i.e., path), since boto3 needs all three in order to access a file.

Example: .. code-block:: python

```
s3_client = boto3.client('s3', region_name='eu-central-1') s3_client.download_file(bucket, key,
download_path)
```

Parameters

- **bucket_name** (*str*) – The S3 bucket name where this file is located
- **bucket_path** (*str*, *optional*) – The key to access this file in the bucket
- **region_name** (*str*, *optional*) – The AWS region where this file is located, defaults to None, which inherits the default configured region.

__str__ (*self*)

Return str(self).

property bucket_name (*self*)

property bucket_path (*self*)

property region_name (*self*)

join (*self, *args*)

Joins segments in an S3 path. This method behaves exactly like os.path.join.

Returns A New S3Path with the additional segments added.

Return type [S3Path](#)

tsdat.pipeline

This module contains pipeline classes that are used to process time series data from start to finish.

Submodules

[tsdat.pipeline.ingest_pipeline](#)

Module Contents

Classes

IngestPipeline

The IngestPipeline class is designed to read in raw, non-standardized

```
class tsdat.pipeline.ingest_pipeline.IngestPipeline (pipeline_config: Union[str,
                                                    tsdat.config.Config], storage_config: Union[str, tsdat.io.DatastreamStorage])
```

Bases: *tsdat.pipeline.pipeline.Pipeline*

The IngestPipeline class is designed to read in raw, non-standardized data and convert it to a standardized format by embedding metadata, applying quality checks and quality controls, and by saving the now-processed data in a standard file format.

run (*self*, *filepath*: Union[str, List[str]]) → xarray.Dataset
Runs the IngestPipeline from start to finish.

Parameters *filepath* (Union[str, List[str]]) – The path or list of paths to the file(s) to run the pipeline on.

hook_customize_dataset (*self*, *dataset*: xarray.Dataset, *raw_mapping*: Dict[str, xarray.Dataset]) → xarray.Dataset

Hook to allow for user customizations to the standardized dataset such as inserting a derived variable based on other variables in the dataset. This method is called immediately after the `standardize_dataset` method and before `QualityManagement` has been run.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to customize.
- **raw_mapping** (Dict[str, *xr.Dataset*]) – The raw dataset mapping.

Returns The customized dataset.

Return type *xr.Dataset*

hook_customize_raw_datasets (*self*, *raw_dataset_mapping*: Dict[str, xarray.Dataset]) → Dict[str, xarray.Dataset]

Hook to allow for user customizations to one or more raw xarray Datasets before they merged and used to create the standardized dataset. The `raw_dataset_mapping` will contain one entry for each file being used as input to the pipeline. The keys are the standardized raw file name, and the values are the datasets.

This method would typically only be used if the user is combining multiple files into a single dataset. In this case, this method may be used to correct coordinates if they don't match for all the files, or to change variable (column) names if two files have the same name for a variable, but they are two distinct variables.

This method can also be used to check for unique conditions in the raw data that should cause a pipeline failure if they are not met.

This method is called before the inputs are merged and converted to standard format as specified by the config file.

Parameters *raw_dataset_mapping* (Dict[str, *xr.Dataset*]) – The raw datasets to customize.

Returns The customized raw datasets.

Return type Dict[str, *xr.Dataset*]

hook_finalize_dataset (*self*, *dataset*: xarray.Dataset) → xarray.Dataset

Hook to apply any final customizations to the dataset before it is saved. This hook is called after `QualityManagement` has been run and immediately before the dataset is saved to file.

Parameters **dataset** (*xr.Dataset*) – The dataset to finalize.

Returns The finalized dataset to save.

Return type *xr.Dataset*

hook_generate_and_persist_plots (*self, dataset: xarray.Dataset*) → None

Hook to allow users to create plots from the xarray dataset after the dataset has been finalized and just before the dataset is saved to disk.

To save on filesystem space (which is limited when running on the cloud via a lambda function), this method should only write one plot to local storage at a time. An example of how this could be done is below:

```
filename = DSUtil.get_plot_filename(dataset, "sea_level", "png")
with self.storage._tmp.get_temp_filepath(filename) as tmp_path:
    fig, ax = plt.subplots(figsize=(10,5))
    ax.plot(dataset["time"].data, dataset["sea_level"].data)
    fig.save(tmp_path)
    storage.save(tmp_path)

filename = DSUtil.get_plot_filename(dataset, "qc_sea_level", "png")
with self.storage._tmp.get_temp_filepath(filename) as tmp_path:
    fig, ax = plt.subplots(figsize=(10,5))
    DSUtil.plot_qc(dataset, "sea_level", tmp_path)
    storage.save(tmp_path)
```

Parameters **dataset** (*xr.Dataset*) – The xarray dataset with customizations and Quality-Management applied.

read_and_persist_raw_files (*self, file_paths: List[str]*) → List[str]

Renames the provided raw files according to ME Data Standards file naming conventions for raw data files, and returns a list of the paths to the renamed files.

Parameters **file_paths** (*List[str]*) – A list of paths to the original raw files.

Returns A list of paths to the renamed files.

Return type List[str]

tsdat.pipeline.pipeline

Module Contents

Classes

<i>Pipeline</i>	This class serves as the base class for all tsdat data pipelines.
-----------------	---

```
class tsdat.pipeline.pipeline.Pipeline (pipeline_config: Union[str, ts-
                                     dat.config.Config], storage_config: Union[str,
                                     tsdat.io.DatastreamStorage])
```

Bases: *abc.ABC*

This class serves as the base class for all tsdat data pipelines.

Parameters

- **pipeline_config** (*Union[str, Config]*) – The pipeline config file. Can be either a config object, or the path to the pipeline config file that should be used with this pipeline.
- **storage_config** (*Union[str, DatastreamStorage]*) – The storage config file. Can be either a config object, or the path to the storage config file that should be used with this pipeline.

abstract run (*self, filepath: Union[str, List[str]]*)

This method is the entry point for the pipeline. It will take one or more file paths and process them from start to finish. All classes extending the Pipeline class must implement this method.

Parameters **filepath** (*Union[str, List[str]]*) – The path or list of paths to the file(s) to run the pipeline on.

standardize_dataset (*self, raw_mapping: Dict[str, xarray.Dataset]*) → *xarray.Dataset*

Standardizes the dataset by applying variable name and units conversions as defined by the pipeline config file. This method returns the standardized dataset.

Parameters **raw_mapping** (*Dict[str, xr.Dataset]*) – The raw dataset mapping.

Returns The standardized dataset.

Return type *xr.Dataset*

check_required_variables (*self, dataset: xarray.Dataset, dod: tsdat.config.DatasetDefinition*)

Function to throw an error if a required variable could not be retrieved.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to check.
- **dod** (*DatasetDefinition*) – The DatasetDefinition used to specify required variables.

Raises **Exception** – Raises an exception to indicate the variable could not be retrieved.

add_static_variables (*self, dataset: xarray.Dataset, dod: tsdat.config.DatasetDefinition*) → *xarray.Dataset*

Uses the DatasetDefinition to add static variables (variables whose data are defined in the pipeline config file) to the output dataset.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add static variables to.
- **dod** (*DatasetDefinition*) – The DatasetDefinition to pull data from.

Returns The original dataset with added variables from the config

Return type *xr.Dataset*

add_missing_variables (*self, dataset: xarray.Dataset, dod: tsdat.config.DatasetDefinition*) → *xarray.Dataset*

Uses the dataset definition to initialize variables that are defined in the dataset definition but did not have input. Uses the appropriate shape and _FillValue to initialize each variable.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add the variables to.
- **dod** (*DatasetDefinition*) – The DatasetDefinition to use.

Returns The original dataset with variables that still need to be initialized, initialized.

Return type *xr.Dataset*

add_attrs (*self*, *dataset*: *xarray.Dataset*, *raw_mapping*: *Dict[str, xarray.Dataset]*, *dod*: *tsdat.config.DatasetDefinition*) → *xarray.Dataset*

Adds global and variable-level attributes to the dataset from the DatasetDefinition object.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add attributes to.
- **raw_mapping** (*Dict[str, xr.Dataset]*) – The raw dataset mapping. Used to set the `input_files` global attribute.
- **dod** (*DatasetDefinition*) – The DatasetDefinition containing the attributes to add.

Returns The original dataset with the attributes added.

Return type *xr.Dataset*

get_previous_dataset (*self*, *dataset*: *xarray.Dataset*) → *xarray.Dataset*

Utility method to retrieve the previous set of data for the same datastream as the provided dataset from the DatastreamStorage.

Parameters **dataset** (*xr.Dataset*) – The reference dataset that will be used to search the DatastreamStore for prior data.

Returns The previous dataset from the DatastreamStorage if it exists, otherwise None.

Return type *xr.Dataset*

reduce_raw_datasets (*self*, *raw_mapping*: *Dict[str, xarray.Dataset]*, *definition*: *tsdat.config.DatasetDefinition*) → *List[xarray.Dataset]*

Removes unused variables from each raw dataset in the raw mapping and performs input to output naming and unit conversions as defined in the dataset definition.

Parameters

- **raw_mapping** (*Dict[str, xr.Dataset]*) – The raw xarray dataset mapping.
- **definition** (*DatasetDefinition*) – The DatasetDefinition used to select the variables to keep.

Returns A list of reduced datasets.

Return type *List[xr.Dataset]*

reduce_raw_dataset (*self*, *raw_dataset*: *xarray.Dataset*, *variable_definitions*: *List[tsdat.config.VariableDefinition]*, *definition*: *tsdat.config.DatasetDefinition*) → *xarray.Dataset*

Removes unused variables from the raw dataset provided and keeps only the variables and coordinates pertaining to the provided variable definitions. Also performs input to output naming and unit conversions as defined in the DatasetDefinition.

Parameters

- **raw_dataset** (*xr.Dataset*) – The raw dataset mapping.
- **variable_definitions** (*List[VariableDefinition]*) – List of variables to keep.
- **definition** (*DatasetDefinition*) – The DatasetDefinition used to select the variables to keep.

Returns The reduced dataset.

Return type *xr.Dataset*

store_and_reopen_dataset (*self*, *dataset*: *xarray.Dataset*) → *xarray.Dataset*

Uses the DatastreamStorage object to persist the dataset in the format specified by the storage config file.

Parameters *dataset* (*xr.Dataset*) – The dataset to store.

Returns The dataset after it has been saved to disk and reopened.

Return type *xr.Dataset*

Package Contents

Classes

<i>Pipeline</i>	This class serves as the base class for all tsdat data pipelines.
<i>IngestPipeline</i>	The IngestPipeline class is designed to read in raw, non-standardized

class `tsdat.pipeline.Pipeline` (*pipeline_config*: *Union[str, tsdat.config.Config]*, *storage_config*: *Union[str, tsdat.io.DatastreamStorage]*)

Bases: `abc.ABC`

This class serves as the base class for all tsdat data pipelines.

Parameters

- **pipeline_config** (*Union[str, Config]*) – The pipeline config file. Can be either a config object, or the path to the pipeline config file that should be used with this pipeline.
- **storage_config** (*Union[str, DatastreamStorage]*) – The storage config file. Can be either a config object, or the path to the storage config file that should be used with this pipeline.

abstract run (*self*, *filepath*: *Union[str, List[str]]*)

This method is the entry point for the pipeline. It will take one or more file paths and process them from start to finish. All classes extending the Pipeline class must implement this method.

Parameters *filepath* (*Union[str, List[str]]*) – The path or list of paths to the file(s) to run the pipeline on.

standardize_dataset (*self*, *raw_mapping*: *Dict[str, xarray.Dataset]*) → *xarray.Dataset*

Standardizes the dataset by applying variable name and units conversions as defined by the pipeline config file. This method returns the standardized dataset.

Parameters *raw_mapping* (*Dict[str, xr.Dataset]*) – The raw dataset mapping.

Returns The standardized dataset.

Return type *xr.Dataset*

check_required_variables (*self*, *dataset*: *xarray.Dataset*, *dod*: *tsdat.config.DatasetDefinition*)

Function to throw an error if a required variable could not be retrieved.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to check.
- **dod** (*DatasetDefinition*) – The DatasetDefinition used to specify required variables.

Raises Exception – Raises an exception to indicate the variable could not be retrieved.

add_static_variables (*self*, *dataset*: *xarray.Dataset*, *dod*: [tsdat.config.DatasetDefinition](#)) → *xarray.Dataset*

Uses the DatasetDefinition to add static variables (variables whose data are defined in the pipeline config file) to the output dataset.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add static variables to.
- **dod** (*DatasetDefinition*) – The DatasetDefinition to pull data from.

Returns The original dataset with added variables from the config

Return type *xr.Dataset*

add_missing_variables (*self*, *dataset*: *xarray.Dataset*, *dod*: [tsdat.config.DatasetDefinition](#)) → *xarray.Dataset*

Uses the dataset definition to initialize variables that are defined in the dataset definition but did not have input. Uses the appropriate shape and _FillValue to initialize each variable.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add the variables to.
- **dod** (*DatasetDefinition*) – The DatasetDefinition to use.

Returns The original dataset with variables that still need to be initialized, initialized.

Return type *xr.Dataset*

add_attrs (*self*, *dataset*: *xarray.Dataset*, *raw_mapping*: *Dict[str, xarray.Dataset]*, *dod*: [tsdat.config.DatasetDefinition](#)) → *xarray.Dataset*

Adds global and variable-level attributes to the dataset from the DatasetDefinition object.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add attributes to.
- **raw_mapping** (*Dict[str, xr.Dataset]*) – The raw dataset mapping. Used to set the `input_files` global attribute.
- **dod** (*DatasetDefinition*) – The DatasetDefinition containing the attributes to add.

Returns The original dataset with the attributes added.

Return type *xr.Dataset*

get_previous_dataset (*self*, *dataset*: *xarray.Dataset*) → *xarray.Dataset*

Utility method to retrieve the previous set of data for the same datastream as the provided dataset from the DatastreamStorage.

Parameters **dataset** (*xr.Dataset*) – The reference dataset that will be used to search the DatastreamStore for prior data.

Returns The previous dataset from the DatastreamStorage if it exists, otherwise None.

Return type *xr.Dataset*

reduce_raw_datasets (*self*, *raw_mapping*: *Dict[str, xarray.Dataset]*, *definition*: [tsdat.config.DatasetDefinition](#)) → *List[xarray.Dataset]*

Removes unused variables from each raw dataset in the raw mapping and performs input to output naming and unit conversions as defined in the dataset definition.

Parameters

- **raw_mapping** (*Dict[str, xr.Dataset]*) – The raw xarray dataset mapping.

- **definition** (*DatasetDefinition*) – The DatasetDefinition used to select the variables to keep.

Returns A list of reduced datasets.

Return type List[xr.Dataset]

reduce_raw_dataset (*self*, *raw_dataset*: *xarray.Dataset*, *variable_definitions*: List[*tsdat.config.VariableDefinition*], *definition*: *tsdat.config.DatasetDefinition*) → *xarray.Dataset*

Removes unused variables from the raw dataset provided and keeps only the variables and coordinates pertaining to the provided variable definitions. Also performs input to output naming and unit conversions as defined in the DatasetDefinition.

Parameters

- **raw_dataset** (*xr.Dataset*) – The raw dataset mapping.
- **variable_definitions** (List[*VariableDefinition*]) – List of variables to keep.
- **definition** (*DatasetDefinition*) – The DatasetDefinition used to select the variables to keep.

Returns The reduced dataset.

Return type *xr.Dataset*

store_and_reopen_dataset (*self*, *dataset*: *xarray.Dataset*) → *xarray.Dataset*

Uses the DatastreamStorage object to persist the dataset in the format specified by the storage config file.

Parameters **dataset** (*xr.Dataset*) – The dataset to store.

Returns The dataset after it has been saved to disk and reopened.

Return type *xr.Dataset*

class *tsdat.pipeline.IngestPipeline* (*pipeline_config*: Union[*str*, *tsdat.config.Config*], *storage_config*: Union[*str*, *tsdat.io.DatastreamStorage*])

Bases: *tsdat.pipeline.pipeline.Pipeline*

The IngestPipeline class is designed to read in raw, non-standardized data and convert it to a standardized format by embedding metadata, applying quality checks and quality controls, and by saving the now-processed data in a standard file format.

run (*self*, *filepath*: Union[*str*, List[*str*]]) → *xarray.Dataset*

Runs the IngestPipeline from start to finish.

Parameters **filepath** (Union[*str*, List[*str*]]) – The path or list of paths to the file(s) to run the pipeline on.

hook_customize_dataset (*self*, *dataset*: *xarray.Dataset*, *raw_mapping*: Dict[*str*, *xarray.Dataset*]) → *xarray.Dataset*

Hook to allow for user customizations to the standardized dataset such as inserting a derived variable based on other variables in the dataset. This method is called immediately after the `standardize_dataset` method and before `QualityManagement` has been run.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to customize.
- **raw_mapping** (Dict[*str*, *xr.Dataset*]) – The raw dataset mapping.

Returns The customized dataset.

Return type *xr.Dataset*

hook_customize_raw_datasets (*self*, *raw_dataset_mapping*: Dict[str, xarray.Dataset]) → Dict[str, xarray.Dataset]

Hook to allow for user customizations to one or more raw xarray Datasets before they merged and used to create the standardized dataset. The *raw_dataset_mapping* will contain one entry for each file being used as input to the pipeline. The keys are the standardized raw file name, and the values are the datasets.

This method would typically only be used if the user is combining multiple files into a single dataset. In this case, this method may be used to correct coordinates if they don't match for all the files, or to change variable (column) names if two files have the same name for a variable, but they are two distinct variables.

This method can also be used to check for unique conditions in the raw data that should cause a pipeline failure if they are not met.

This method is called before the inputs are merged and converted to standard format as specified by the config file.

Parameters *raw_dataset_mapping* (Dict[str, xr.Dataset]) – The raw datasets to customize.

Returns The customized raw datasets.

Return type Dict[str, xr.Dataset]

hook_finalize_dataset (*self*, *dataset*: xarray.Dataset) → xarray.Dataset

Hook to apply any final customizations to the dataset before it is saved. This hook is called after Quality-Management has been run and immediately before the dataset is saved to file.

Parameters *dataset* (xr.Dataset) – The dataset to finalize.

Returns The finalized dataset to save.

Return type xr.Dataset

hook_generate_and_persist_plots (*self*, *dataset*: xarray.Dataset) → None

Hook to allow users to create plots from the xarray dataset after the dataset has been finalized and just before the dataset is saved to disk.

To save on filesystem space (which is limited when running on the cloud via a lambda function), this method should only write one plot to local storage at a time. An example of how this could be done is below:

```
filename = DSUtil.get_plot_filename(dataset, "sea_level", "png")
with self.storage.tmp.get_temp_filepath(filename) as tmp_path:
    fig, ax = plt.subplots(figsize=(10,5))
    ax.plot(dataset["time"].data, dataset["sea_level"].data)
    fig.save(tmp_path)
    storage.save(tmp_path)

filename = DSUtil.get_plot_filename(dataset, "qc_sea_level", "png")
with self.storage.tmp.get_temp_filepath(filename) as tmp_path:
    fig, ax = plt.subplots(figsize=(10,5))
    DSUtil.plot_qc(dataset, "sea_level", tmp_path)
    storage.save(tmp_path)
```

Parameters *dataset* (xr.Dataset) – The xarray dataset with customizations and Quality-Management applied.

read_and_persist_raw_files (*self*, *file_paths*: List[str]) → List[str]

Renames the provided raw files according to ME Data Standards file naming conventions for raw data files, and returns a list of the paths to the renamed files.

Parameters `file_paths` (`List[str]`) – A list of paths to the original raw files.

Returns A list of paths to the renamed files.

Return type `List[str]`

`tsdat.qc`

The `tsdat.qc` package provides the classes that the data pipeline uses to manage quality control/quality assurance for the dataset. This includes the infrastructure to run quality tests and handle failures, as well specific checkers and handlers that can be specified in the pipeline config file.

We warmly welcome community contributions to increase this default list.

Submodules

`tsdat.qc.checkers`

Module Contents

Classes

<i>QualityChecker</i>	Class containing the code to perform a single Quality Check on a
<i>CheckMissing</i>	Checks if any values are assigned to <code>_FillValue</code> or 'NaN' (for non-time
<i>CheckMin</i>	Check that no values for the specified variable are less than
<i>CheckMax</i>	Check that no values for the specified variable are greater than
<i>CheckValidMin</i>	Check that no values for the specified variable are less than
<i>CheckValidMax</i>	Check that no values for the specified variable are greater than
<i>CheckFailMin</i>	Check that no values for the specified variable are less than
<i>CheckFailMax</i>	Check that no values for the specified variable greater less than
<i>CheckWarnMin</i>	Check that no values for the specified variable are less than
<i>CheckWarnMax</i>	Check that no values for the specified variable are greater than
<i>CheckValidDelta</i>	Check that the difference between any two consecutive
<i>CheckMonotonic</i>	Checks that all values for the specified variable are either

class `tsdat.qc.checkers.QualityChecker` (`ds: xarray.Dataset`, `previous_data: xarray.Dataset`,
`definition: tsdat.config.QualityManagerDefinition`,
`parameters: Union[Dict, None] = None`)

Bases: `abc.ABC`

Class containing the code to perform a single Quality Check on a Dataset variable.

Parameters

- **ds** (*xr.Dataset*) – The dataset the checker will be applied to
- **previous_data** (*xr.Dataset*) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.
- **definition** (*QualityManagerDefinition*) – The quality manager definition as specified in the pipeline config file
- **parameters** (*dict, optional*) – A dictionary of checker-specific parameters specified in the pipeline config file. Defaults to {}

abstract run (*self, variable_name: str*) → *Optional[numpy.ndarray]*

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (*str*) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type *Optional[np.ndarray]*

class `tsdat.qc.checkers.CheckMissing` (*ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None*)

Bases: *QualityChecker*

Checks if any values are assigned to _FillValue or 'NaN' (for non-time variables) or checks if values are assigned to 'NaT' (for time variables). Also, for non-time variables, checks if values are above or below valid_range, as this is considered missing as well.

run (*self, variable_name: str*) → *Optional[numpy.ndarray]*

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (*str*) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type *Optional[np.ndarray]*

```
class tsdat.qc.checkers.CheckMin (ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None)
```

Bases: *QualityChecker*

Check that no values for the specified variable are less than a specified minimum threshold. The threshold value is an attribute set on the variable in question. The attribute name is specified in the quality checker definition in the pipeline config file by setting a param called 'key: ATTRIBUTE_NAME'.

If the key parameter is not set or the variable does not possess the specified attribute, this check will be skipped.

run (self, variable_name: str) → Optional[numpy.ndarray]

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (str) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type Optional[np.ndarray]

```
class tsdat.qc.checkers.CheckMax (ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None)
```

Bases: *QualityChecker*

Check that no values for the specified variable are greater than a specified maximum threshold. The threshold value is an attribute set on the variable in question. The attribute name is specified in the quality checker definition in the pipeline config file by setting a param called 'key: ATTRIBUTE_NAME'.

If the key parameter is not set or the variable does not possess the specified attribute, this check will be skipped.

run (self, variable_name: str) → Optional[numpy.ndarray]

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (str) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type Optional[np.ndarray]

class tsdat.qc.checkers.**CheckValidMin** (*ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters*)

Bases: *CheckMin*

Check that no values for the specified variable are less than the minimum vaue set by the ‘valid_range’ attribute. If the variable in question does not possess the ‘valid_range’ attribute, this check will be skipped.

class tsdat.qc.checkers.**CheckValidMax** (*ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters*)

Bases: *CheckMax*

Check that no values for the specified variable are greater than the maximum vaue set by the ‘valid_range’ attribute. If the variable in question does not possess the ‘valid_range’ attribute, this check will be skipped.

class tsdat.qc.checkers.**CheckFailMin** (*ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters*)

Bases: *CheckMin*

Check that no values for the specified variable are less than the minimum vaue set by the ‘fail_range’ attribute. If the variable in question does not possess the ‘fail_range’ attribute, this check will be skipped.

class tsdat.qc.checkers.**CheckFailMax** (*ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters*)

Bases: *CheckMax*

Check that no values for the specified variable greater less than the maximum vaue set by the ‘fail_range’ attribute. If the variable in question does not possess the ‘fail_range’ attribute, this check will be skipped.

class tsdat.qc.checkers.**CheckWarnMin** (*ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters*)

Bases: *CheckMin*

Check that no values for the specified variable are less than the minimum vaue set by the ‘warn_range’ attribute. If the variable in question does not possess the ‘warn_range’ attribute, this check will be skipped.

class tsdat.qc.checkers.**CheckWarnMax** (*ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters*)

Bases: *CheckMax*

Check that no values for the specified variable are greater than the maximum vaue set by the ‘warn_range’ attribute. If the variable in question does not possess the ‘warn_range’ attribute, this check will be skipped.

class tsdat.qc.checkers.**CheckValidDelta** (*ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None*)

Bases: *QualityChecker*

Check that the difference between any two consecutive values is not greater than the threshold set by the ‘valid_delta’ attribute. If the variable in question does not possess the ‘valid_delta’ attribute, this check will be skipped.

run (*self, variable_name: str*) → Optional[numpy.ndarray]

Check a dataset’s variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (*str*) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type Optional[np.ndarray]

```
class tsdat.qc.checkers.CheckMonotonic (ds: xarray.Dataset, previous_data: xarray.Dataset,  
                                         definition: tsdat.config.QualityManagerDefinition,  
                                         parameters: Union[Dict, None] = None)
```

Bases: *QualityChecker*

Checks that all values for the specified variable are either strictly increasing or strictly decreasing.

run (self, variable_name: str) → Optional[numpy.ndarray]

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (str) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type Optional[np.ndarray]

tsdat.qc.handlers

Module Contents

Classes

<i>QCParamKeys</i>	Symbolic constants used for referencing QC-related
<i>QualityHandler</i>	Class containing code to be executed if a particular quality check fails.
<i>RecordQualityResults</i>	Record the results of the quality check in an ancillary qc variable.
<i>RemoveFailedValues</i>	Replace all the failed values with _FillValue
<i>SortDatasetByCoordinate</i>	Sort coordinate data using xr.Dataset.sortby(). Accepts the following

continues on next page

Table 29 – continued from previous page

<i>SendEmailAWS</i>	Send an email to the recipients using AWS services.
<i>FailPipeline</i>	Throw an exception, halting the pipeline & indicating a critical error

class tsdat.qc.handlers.QCParamKeys

Symbolic constants used for referencing QC-related fields in the pipeline config file

QC_BIT = bit

ASSESSMENT = assessment

TEST_MEANING = meaning

CORRECTION = correction

class tsdat.qc.handlers.QualityHandler (*ds*: *xarray.Dataset*, *previous_data*:
xarray.Dataset, *quality_manager*: *ts-*
dat.config.QualityManagerDefinition, *parameters*:
Union[Dict, None] = *None*)

Bases: *abc.ABC*

Class containing code to be executed if a particular quality check fails.

Parameters

- **ds** (*xr.Dataset*) – The dataset the handler will be applied to
- **previous_data** (*xr.Dataset*) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.
- **quality_manager** (*QualityManagerDefinition*) – The quality_manager definition as specified in the pipeline config file
- **parameters** (*dict*, *optional*) – A dictionary of handler-specific parameters specified in the pipeline config file. Defaults to {}

abstract run (*self*, *variable_name*: *str*, *results_array*: *numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

record_correction (*self*, *variable_name*: *str*)

If a correction was made to variable data to fix invalid values as detected by a quality check, this method will record the fix to the appropriate variable attribute. The correction description will come from the handler params which get set in the pipeline config file.

Parameters **variable_name** (*str*) – Name

class tsdat.qc.handlers.RecordQualityResults (*ds*: *xarray.Dataset*, *previous_data*:
xarray.Dataset, *quality_manager*: *ts-*
dat.config.QualityManagerDefinition,
parameters: *Union[Dict, None]* = *None*)

Bases: *QualityHandler*

Record the results of the quality check in an ancillary qc variable.

run (*self*, *variable_name*: *str*, *results_array*: *numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

```
class tsdat.qc.handlers.RemoveFailedValues (ds: xarray.Dataset, previous_data:
                                             xarray.Dataset, quality_manager: ts-
                                             dat.config.QualityManagerDefinition, pa-
                                             rameters: Union[Dict, None] = None)
```

Bases: *QualityHandler*

Replace all the failed values with `_FillValue`

run (*self*, *variable_name*: *str*, *results_array*: *numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

```
class tsdat.qc.handlers.SortDatasetByCoordinate (ds: xarray.Dataset, previous_data:
                                                    xarray.Dataset, quality_manager: ts-
                                                    dat.config.QualityManagerDefinition,
                                                    parameters: Union[Dict, None] =
                                                    None)
```

Bases: *QualityHandler*

Sort coordinate data using `xr.Dataset.sortby()`. Accepts the following parameters:

```
parameters:
    # Whether or not to sort in ascending order. Defaults to True.
ascending: True
```

run (*self*, *variable_name*: *str*, *results_array*: *numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

```
class tsdat.qc.handlers.SendEmailAWS (ds: xarray.Dataset, previous_data: xar-
                                         ray.Dataset, quality_manager: ts-
                                         dat.config.QualityManagerDefinition, parameters:
                                         Union[Dict, None] = None)
```

Bases: *QualityHandler*

Send an email to the recipients using AWS services.

run (*self*, *variable_name*: *str*, *results_array*: *numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

```
class tsdat.qc.handlers.FailPipeline (ds:          xarray.Dataset,  previous_data:      xar-
                                     ray.Dataset,    quality_manager:      ts-
                                     dat.config.QualityManagerDefinition,  parameters:
                                     Union[Dict, None] = None)
```

Bases: *QualityHandler*

Throw an exception, halting the pipeline & indicating a critical error

run (*self*, *variable_name*: *str*, *results_array*: *numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

`tsdat.qc.qc`

Module Contents

Classes

<i>QualityManagement</i>	Class that provides static helper functions for providing quality
<i>QualityManager</i>	Applies a single Quality Manager to the given Dataset, as defined by

class `tsdat.qc.qc.QualityManagement`

Class that provides static helper functions for providing quality control checks on a tsdat-standardized xarray dataset.

static run (*ds*: *xarray.Dataset*, *config*: *tsdat.config.Config*, *previous_data*: *xarray.Dataset*) → *xarray.Dataset*

Applies the Quality Managers defined in the given Config to this dataset. QC results will be embedded in the dataset. QC metadata will be stored as attributes, and QC flags will be stored as a bitwise integer in new companion **qc_** variables that are added to the dataset. This method will create QC companion variables if they don't exist.

Parameters

- **ds** (*xr.Dataset*) – The dataset to apply quality managers to
- **config** (*Config*) – A configuration definition (loaded from yaml)
- **previous_data** (*xr.Dataset*) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.

Returns The dataset after quality checkers and handlers have been applied.

Return type *xr.Dataset*

```
class tsdat.qc.qc.QualityManager (ds: xarray.Dataset, config: tsdat.config.Config, definition:
                                tsdat.config.QualityManagerDefinition, previous_data: xar-
                                ray.Dataset)
```

Applies a single Quality Manager to the given Dataset, as defined by the Config

Parameters

- **ds** (*xr.Dataset*) – The dataset for which we will perform quality management.
- **config** (*Config*) – The Config from the pipeline definition file.
- **definition** (*QualityManagerDefinition*) – Definition of the quality test this class manages.
- **previous_data** (*xr.Dataset*) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.

run (*self*) → *xarray.Dataset*

Runs the QualityChecker and QualityHandler(s) for each specified variable as defined in the config file.

Returns The dataset after the quality checker and the quality handlers have been run.

Raises **QCError** – A QCError indicates that a fatal error has occurred.

Return type *xr.Dataset*

Package Contents

Classes

<i>QualityManagement</i>	Class that provides static helper functions for providing quality
<i>QualityManager</i>	Applies a single Quality Manager to the given Dataset, as defined by
<i>QualityChecker</i>	Class containing the code to perform a single Quality Check on a
<i>CheckWarnMax</i>	Check that no values for the specified variable are greater than
<i>CheckFailMax</i>	Check that no values for the specified variable greater less than
<i>CheckFailMin</i>	Check that no values for the specified variable are less than
<i>CheckMax</i>	Check that no values for the specified variable are greater than

continues on next page

Table 31 – continued from previous page

<i>CheckMin</i>	Check that no values for the specified variable are less than
<i>CheckMissing</i>	Checks if any values are assigned to <code>_FillValue</code> or 'NaN' (for non-time
<i>CheckMonotonic</i>	Checks that all values for the specified variable are either
<i>CheckValidDelta</i>	Check that the difference between any two consecutive
<i>CheckValidMax</i>	Check that no values for the specified variable are greater than
<i>CheckValidMin</i>	Check that no values for the specified variable are less than
<i>CheckWarnMin</i>	Check that no values for the specified variable are less than
<i>QualityHandler</i>	Class containing code to be executed if a particular quality check fails.
<i>QCParamKeys</i>	Symbolic constants used for referencing QC-related
<i>FailPipeline</i>	Throw an exception, halting the pipeline & indicating a critical error
<i>RecordQualityResults</i>	Record the results of the quality check in an ancillary qc variable.
<i>RemoveFailedValues</i>	Replace all the failed values with <code>_FillValue</code>
<i>SendEmailAWS</i>	Send an email to the recipients using AWS services.

class tsdat.qc.QualityManagement

Class that provides static helper functions for providing quality control checks on a tsdat-standardized xarray dataset.

static run (*ds*: xarray.Dataset, *config*: tsdat.config.Config, *previous_data*: xarray.Dataset) → xarray.Dataset

Applies the Quality Managers defined in the given Config to this dataset. QC results will be embedded in the dataset. QC metadata will be stored as attributes, and QC flags will be stored as a bitwise integer in new companion **qc_** variables that are added to the dataset. This method will create QC companion variables if they don't exist.

Parameters

- **ds** (*xr.Dataset*) – The dataset to apply quality managers to
- **config** (*Config*) – A configuration definition (loaded from yaml)
- **previous_data** (*xr.Dataset*) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.

Returns The dataset after quality checkers and handlers have been applied.

Return type *xr.Dataset*

class tsdat.qc.QualityManager (*ds*: xarray.Dataset, *config*: tsdat.config.Config, *definition*: tsdat.config.QualityManagerDefinition, *previous_data*: xarray.Dataset)

Applies a single Quality Manager to the given Dataset, as defined by the Config

Parameters

- **ds** (*xr.Dataset*) – The dataset for which we will perform quality management.
- **config** (*Config*) – The Config from the pipeline definition file.

- **definition** (*QualityManagerDefinition*) – Definition of the quality test this class manages.
- **previous_data** (*xr.Dataset*) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.

run (*self*) → *xarray.Dataset*

Runs the QualityChecker and QualityHandler(s) for each specified variable as defined in the config file.

Returns The dataset after the quality checker and the quality handlers have been run.

Raises **QCError** – A QCError indicates that a fatal error has occurred.

Return type *xr.Dataset*

```
class tsdat.qc.QualityChecker (ds: xarray.Dataset, previous_data: xarray.Dataset, definition:  
tsdat.config.QualityManagerDefinition, parameters: Union[Dict,  
None] = None)
```

Bases: *abc.ABC*

Class containing the code to perform a single Quality Check on a Dataset variable.

Parameters

- **ds** (*xr.Dataset*) – The dataset the checker will be applied to
- **previous_data** (*xr.Dataset*) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.
- **definition** (*QualityManagerDefinition*) – The quality manager definition as specified in the pipeline config file
- **parameters** (*dict, optional*) – A dictionary of checker-specific parameters specified in the pipeline config file. Defaults to { }

abstract run (*self, variable_name: str*) → *Optional[numpy.ndarray]*

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (*str*) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type *Optional[np.ndarray]*

```
class tsdat.qc.CheckWarnMax (ds: xarray.Dataset, previous_data: xarray.Dataset, definition: ts-  
dat.config.QualityManagerDefinition, parameters)
```

Bases: *CheckMax*

Check that no values for the specified variable are greater than the maximum value set by the 'warn_range' attribute. If the variable in question does not possess the 'warn_range' attribute, this check will be skipped.

```
class tsdat.qc.CheckFailMax(ds: xarray.Dataset, previous_data: xarray.Dataset, definition: ts-
                             dat.config.QualityManagerDefinition, parameters)
```

Bases: *CheckMax*

Check that no values for the specified variable greater less than the maximum vaue set by the ‘fail_range’ attribute. If the variable in question does not possess the ‘fail_range’ attribute, this check will be skipped.

```
class tsdat.qc.CheckFailMin(ds: xarray.Dataset, previous_data: xarray.Dataset, definition: ts-
                             dat.config.QualityManagerDefinition, parameters)
```

Bases: *CheckMin*

Check that no values for the specified variable are less than the minimum vaue set by the ‘fail_range’ attribute. If the variable in question does not possess the ‘fail_range’ attribute, this check will be skipped.

```
class tsdat.qc.CheckMax(ds: xarray.Dataset, previous_data: xarray.Dataset, definition: ts-
                        dat.config.QualityManagerDefinition, parameters: Union[Dict, None] =
                        None)
```

Bases: *QualityChecker*

Check that no values for the specified variable are greater than a specified maximum threshold. The threshold value is an attribute set on the variable in question. The attribute name is specified in the quality checker definition in the pipeline config file by setting a param called ‘key: ATTRIBUTE_NAME’.

If the key parameter is not set or the variable does not possess the specified attribute, this check will be skipped.

```
run(self, variable_name: str) → Optional[numpy.ndarray]
```

Check a dataset’s variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (*str*) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it’s easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type Optional[np.ndarray]

```
class tsdat.qc.CheckMin(ds: xarray.Dataset, previous_data: xarray.Dataset, definition: ts-
                        dat.config.QualityManagerDefinition, parameters: Union[Dict, None] =
                        None)
```

Bases: *QualityChecker*

Check that no values for the specified variable are less than a specified minimum threshold. The threshold value is an attribute set on the variable in question. The attribute name is specified in the quality checker definition in the pipeline config file by setting a param called ‘key: ATTRIBUTE_NAME’.

If the key parameter is not set or the variable does not possess the specified attribute, this check will be skipped.

```
run(self, variable_name: str) → Optional[numpy.ndarray]
```

Check a dataset’s variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (*str*) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type Optional[np.ndarray]

```
class tsdat.qc.CheckMissing(ds: xarray.Dataset, previous_data: xarray.Dataset, definition:
                           tsdat.config.QualityManagerDefinition, parameters: Union[Dict,
                           None] = None)
```

Bases: *QualityChecker*

Checks if any values are assigned to _FillValue or 'NaN' (for non-time variables) or checks if values are assigned to 'NaT' (for time variables). Also, for non-time variables, checks if values are above or below valid_range, as this is considered missing as well.

run (self, variable_name: str) → Optional[numpy.ndarray]

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (str) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an np.ndarray instead of an xr.DataArray because the DataArray contains coordinate indexes which can sometimes get out of sync when performing np arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return None.

Return type Optional[np.ndarray]

```
class tsdat.qc.CheckMonotonic(ds: xarray.Dataset, previous_data: xarray.Dataset, definition:
                              tsdat.config.QualityManagerDefinition, parameters: Union[Dict,
                              None] = None)
```

Bases: *QualityChecker*

Checks that all values for the specified variable are either strictly increasing or strictly decreasing.

run (self, variable_name: str) → Optional[numpy.ndarray]

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters **variable_name** (str) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an `np.ndarray` instead of an `xr.DataArray` because the `DataArray` contains coordinate indexes which can sometimes get out of sync when performing `np` arithmetic vector operations. So it's easier to just use `numpy` arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the `run` method should return `None`.

Return type `Optional[np.ndarray]`

```
class tsdat.qc.CheckValidDelta (ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None)
```

Bases: `QualityChecker`

Check that the difference between any two consecutive values is not greater than the threshold set by the 'valid_delta' attribute. If the variable in question does not possess the 'valid_delta' attribute, this check will be skipped.

run (*self*, *variable_name*: *str*) → `Optional[numpy.ndarray]`

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using `xarray` vectorized numerical operators.

Parameters *variable_name* (*str*) – The name of the variable to check

Returns

If the check was performed, return a `ndarray` of the same shape as the variable. Each value in the data array will be either `True` or `False`, depending upon the results of the check. `True` means the check failed. `False` means it succeeded.

Note that we are using an `np.ndarray` instead of an `xr.DataArray` because the `DataArray` contains coordinate indexes which can sometimes get out of sync when performing `np` arithmetic vector operations. So it's easier to just use `numpy` arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the `run` method should return `None`.

Return type `Optional[np.ndarray]`

```
class tsdat.qc.CheckValidMax (ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters)
```

Bases: `CheckMax`

Check that no values for the specified variable are greater than the maximum value set by the 'valid_range' attribute. If the variable in question does not possess the 'valid_range' attribute, this check will be skipped.

```
class tsdat.qc.CheckValidMin (ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters)
```

Bases: `CheckMin`

Check that no values for the specified variable are less than the minimum value set by the 'valid_range' attribute. If the variable in question does not possess the 'valid_range' attribute, this check will be skipped.

```
class tsdat.qc.CheckWarnMin (ds: xarray.Dataset, previous_data: xarray.Dataset, definition: tsdat.config.QualityManagerDefinition, parameters)
```

Bases: `CheckMin`

Check that no values for the specified variable are less than the minimum value set by the 'warn_range' attribute. If the variable in question does not possess the 'warn_range' attribute, this check will be skipped.

```
class tsdat.qc.QualityHandler (ds: xarray.Dataset, previous_data: xarray.Dataset, quality_manager: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None)
```

Bases: `abc.ABC`

Class containing code to be executed if a particular quality check fails.

Parameters

- **ds** (*xr.Dataset*) – The dataset the handler will be applied to
- **previous_data** (*xr.Dataset*) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.
- **quality_manager** (*QualityManagerDefinition*) – The quality_manager definition as specified in the pipeline config file
- **parameters** (*dict, optional*) – A dictionary of handler-specific parameters specified in the pipeline config file. Defaults to {}

abstract run (*self, variable_name: str, results_array: numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

record_correction (*self, variable_name: str*)

If a correction was made to variable data to fix invalid values as detected by a quality check, this method will record the fix to the appropriate variable attribute. The correction description will come from the handler params which get set in the pipeline config file.

Parameters **variable_name** (*str*) – Name

class tsdat.qc.QCParamKeys

Symbolic constants used for referencing QC-related fields in the pipeline config file

QC_BIT = `bit`

ASSESSMENT = `assessment`

TEST_MEANING = `meaning`

CORRECTION = `correction`

class tsdat.qc.FailPipeline (*ds: xarray.Dataset, previous_data: xarray.Dataset, quality_manager: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None*)

Bases: [QualityHandler](#)

Throw an exception, halting the pipeline & indicating a critical error

run (*self, variable_name: str, results_array: numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

```
class tsdat.qc.RecordQualityResults (ds: xarray.Dataset, previous_data: xarray.Dataset, quality_manager: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None)
```

Bases: *QualityHandler*

Record the results of the quality check in an ancillary qc variable.

```
run (self, variable_name: str, results_array: numpy.ndarray)
```

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

```
class tsdat.qc.RemoveFailedValues (ds: xarray.Dataset, previous_data: xarray.Dataset, quality_manager: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None)
```

Bases: *QualityHandler*

Replace all the failed values with _FillValue

```
run (self, variable_name: str, results_array: numpy.ndarray)
```

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

```
class tsdat.qc.SendEmailAWS (ds: xarray.Dataset, previous_data: xarray.Dataset, quality_manager: tsdat.config.QualityManagerDefinition, parameters: Union[Dict, None] = None)
```

Bases: *QualityHandler*

Send an email to the recipients using AWS services.

```
run (self, variable_name: str, results_array: numpy.ndarray)
```

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

tsdat.utils

The tsdat.utils package provides helper classes for working with XArray datasets.

Submodules

tsdat.utils.converters

Module Contents

Classes

<i>Converter</i>	Base class for converting data arrays from one units to another.
<i>DefaultConverter</i>	Default class for converting units on data arrays. This class utilizes
<i>StringTimeConverter</i>	Convert a time string to a np.datetime64, which is needed for xarray.
<i>TimestampTimeConverter</i>	Convert a numeric UTC timestamp to a np.datetime64, which is needed for

class tsdat.utils.converters.**Converter** (*parameters: Union[Dict, None] = None*)

Bases: abc.ABC

Base class for converting data arrays from one units to another. Users can extend this class if they have a special units conversion for their input data that cannot be resolved with the default converter classes.

Parameters **parameters** (*dict, optional*) – A dictionary of converter-specific parameters which get passed from the pipeline config file. Defaults to { }

abstract run (*self, data: numpy.ndarray, in_units: str, out_units: str*) → numpy.ndarray

Convert the input data from in_units to out_units.

Parameters

- **data** (*np.ndarray*) – Data array to be modified.
- **in_units** (*str*) – Current units of the data array.
- **out_units** (*str*) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

class tsdat.utils.converters.**DefaultConverter** (*parameters: Union[Dict, None] = None*)

Bases: *Converter*

Default class for converting units on data arrays. This class utilizes ACT.utils.data_utils.convert_units, and should work for most variables except time (see StringTimeConverter and TimestampTimeConverter)

run (*self, data: numpy.ndarray, in_units: str, out_units: str*) → numpy.ndarray

Convert the input data from in_units to out_units.

Parameters

- **data** (*np.ndarray*) – Data array to be modified.

- **in_units** (*str*) – Current units of the data array.
- **out_units** (*str*) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

```
class tsdat.utils.converters.StringTimeConverter (parameters: Union[Dict, None] =
None)
```

Bases: *Converter*

Convert a time string to a np.datetime64, which is needed for xarray. This class utilizes pd.to_datetime to perform the conversion.

One of the parameters should be 'time_format', which is the the strftime to parse time, eg "%d/%m/%Y". Note that "%f" will parse all the way up to nanoseconds. See strftime documentation for more information on choices.

Parameters **parameters** (*dict*, *optional*) – dictionary of converter-specific parameters.
Defaults to {}.

run (*self*, *data*: numpy.ndarray, *in_units*: *str*, *out_units*: *str*) → numpy.ndarray
Convert the input data from in_units to out_units.

Parameters

- **data** (*np.ndarray*) – Data array to be modified.
- **in_units** (*str*) – Current units of the data array.
- **out_units** (*str*) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

```
class tsdat.utils.converters.TimestampTimeConverter (parameters: Union[Dict, None]
= None)
```

Bases: *Converter*

Convert a numeric UTC timestamp to a np.datetime64, which is needed for xarray. This class utilizes pd.to_datetime to perform the conversion.

One of the parameters should be 'unit'. This parameter denotes the time unit (e.g., D,s,ms,us,ns), which is an integer or float number. The timestamp will be based off the unix epoch start.

Parameters **parameters** (*dict*, *optional*) – A dictionary of converter-specific parameters
which get passed from the pipeline config file. Defaults to {}

run (*self*, *data*: numpy.ndarray, *in_units*: *str*, *out_units*: *str*) → numpy.ndarray
Convert the input data from in_units to out_units.

Parameters

- **data** (*np.ndarray*) – Data array to be modified.
- **in_units** (*str*) – Current units of the data array.
- **out_units** (*str*) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

tsdat.utils.dsutils

Module Contents

Classes

<i>DSUtil</i>	Provides helper functions for xarray.Dataset
class tsdat.utils.dsutils.DSUtil	
Provides helper functions for xarray.Dataset	
static record_corrections_applied (<i>ds: xarray.Dataset, variable: str, correction: str</i>)	
Records a description of a correction made to a variable to the corrections_applied corresponding attribute.	
Parameters	
• ds (<i>xr.Dataset</i>) – Dataset containing the corrected variable • variable (<i>str</i>) – The name of the variable that was corrected • correction (<i>str</i>) – A description of the correction	
static datetime64_to_string (<i>datetime64: numpy.datetime64</i>) → Tuple[str, str]	
Convert a datetime64 object to formatted string.	
Parameters datetime64 (<i>Union[np.ndarray, np.datetime64]</i>) – The datetime64 object	
Returns A tuple of strings representing the formatted date. The first string is the day in ‘yyyymmdd’ format. The second string is the time in ‘hhmmss’ format.	
Return type Tuple[str, str]	
static datetime64_to_timestamp (<i>variable_data: numpy.ndarray</i>) → numpy.ndarray	
Converts each datetime64 value to a timestamp in same units as the variable (eg., seconds, nanoseconds).	
Parameters variable_data (<i>np.ndarray</i>) – ndarray of variable data	
Returns An ndarray of the same shape, with time values converted to long timestamps (e.g., int64)	
Return type np.ndarray	
static get_datastream_name (<i>ds: xarray.Dataset = None, config=None</i>) → str	
Returns the datastream name defined in the dataset or in the provided pipeline configuration.	
Parameters	
• ds (<i>xr.Dataset, optional.</i>) – The data as an xarray dataset; defaults to None • config (<i>Config, optional</i>) – The Config object used to assist reading time data from the raw_dataset; defaults to None.	
Returns The datastream name	
Return type str	
static get_end_time (<i>ds: xarray.Dataset</i>) → Tuple[str, str]	
Convenience method to get the end date and time from a xarray dataset.	
Parameters ds (<i>xr.Dataset</i>) – The dataset	
Returns A tuple of [day, time] as formatted strings representing the last time point in the dataset.	

Return type Tuple[str, str]

static get_fill_value (*ds*: xarray.Dataset, *variable_name*: str)

Get the value of the _FillValue attribute for the given variable.

Parameters

- **ds** (*xr.Dataset*) – The dataset
- **variable_name** (*str*) – A variable in the dataset

Returns The value of the _FillValue attr or None if it is not defined

Return type same data type of the variable (int, float, etc.) or None

static get_non_qc_variable_names (*ds*: xarray.Dataset) → List[str]

Get a list of all data variables in the dataset that are NOT qc variables.

Parameters **ds** (*xr.Dataset*) – A dataset

Returns List of non-qc data variable names

Return type List[str]

static get_raw_end_time (*raw_ds*: xarray.Dataset, *time_var_definition*: ts-
dat.VariableDefinition) → Tuple[str, str]

Convenience method to get the end date and time from a raw xarray dataset. This uses *time_var_definition.get_input_name()* as the dataset key for the time variable and additionally uses the input's *Converter* object if applicable.

Parameters

- **raw_ds** (*xr.Dataset*) – A raw dataset (not standardized)
- **time_var_definition** (*VariableDefinition*) – The 'time' variable definition from the pipeline config

Returns A tuple of strings representing the last time data point in the dataset. The first string is the day in 'yyyymmdd' format. The second string is the time in 'hhmmss' format.

Return type Tuple[str, str]

static get_raw_start_time (*raw_ds*: xarray.Dataset, *time_var_definition*: ts-
dat.config.VariableDefinition) → Tuple[str, str]

Convenience method to get the start date and time from a raw xarray dataset. This uses *time_var_definition.get_input_name()* as the dataset key for the time variable and additionally uses the input's *Converter* object if applicable.

Parameters

- **raw_ds** (*xr.Dataset*) – A raw dataset (not standardized)
- **time_var_definition** (*VariableDefinition*) – The 'time' variable definition from the pipeline config

Returns A tuple of strings representing the first time data point in the dataset. The first string is the day in 'yyyymmdd' format. The second string is the time in 'hhmmss' format.

Return type Tuple[str, str]

static get_coordinate_variable_names (*ds*: xarray.Dataset) → List[str]

Get a list of all coordinate variables in this dataset.

Parameters **ds** (*xr.Dataset*) – The dataset

Returns List of coordinate variable names

Return type List[str]

static get_start_time (*ds*: xarray.Dataset) → Tuple[str, str]

Convenience method to get the start date and time from a xarray dataset.

Parameters *ds* (*xr.Dataset*) – A standardized dataset

Returns A tuple of strings representing the first time data point in the dataset. The first string is the day in 'yyyymmdd' format. The second string is the time in 'hmmss' format.

Return type Tuple[str, str]

static get_metadata (*ds*: xarray.Dataset) → Dict

Get a dictionary of all global and variable attributes in a dataset. Global atts are found under the 'attributes' key and variable atts are found under the 'variables' key.

Parameters *ds* (*xr.Dataset*) – A dataset

Returns A dictionary of global & variable attributes

Return type Dict

static plot_qc (*ds*: xarray.Dataset, *variable_name*: str, *filename*: str = None, ***kwargs*) → act.plotting.TimeSeriesDisplay

Create a QC plot for the given variable. This is based on the ACT library: https://arm-doe.github.io/ACT/source/auto_examples/plot_qc.html#sphx-glr-source-auto-examples-plot-qc-py

We provide a convenience wrapper method for basic QC plots of a variable, but we recommend to use ACT directly and look at their examples for more complex plots like plotting variables in two different datasets.

TODO: Depending on use cases, we will likely add more arguments to be able to quickly produce the most common types of QC plots.

Parameters

- **ds** (*xr.Dataset*) – A dataset
- **variable_name** (*str*) – The variable to plot
- **filename** (*str*, *optional*) – The filename for the image. Saves the plot as this filename if provided.

static get_plot_filename (*dataset*: xarray.Dataset, *plot_description*: str, *extension*: str) → str

Returns the filename for a plot according to MHKIT-Cloud Data standards. The dataset is used to determine the *datastream_name* and start date/time. The standards dictate that a plot filename should follow the format: *datastream_name.date.time.description.extension*.

Parameters

- **dataset** (*xr.Dataset*) – The dataset from which the plot data is drawn from. This is used to collect the *datastream_name* and start date/time.
- **plot_description** (*str*) – The description of the plot. Should be as brief as possible and contain no spaces. Underscores may be used.
- **extension** (*str*) – The file extension for the plot.

Returns The standardized plot filename.

Return type str

static get_dataset_filename (*dataset*: xarray.Dataset, *file_extension*='nc') → str

Given an xarray dataset this function will return the base filename of the dataset according to MHKIT-Cloud data standards. The base filename does not include the directory structure where the file should be saved, only the name of the file itself, e.g. *z05.ExampleBuoyDatastream.b1.20201230.000000.nc*

Parameters

- **dataset** (*xr.Dataset*) – The dataset whose filename should be generated.
- **file_extension** (*str*, *optional*) – The file extension to use. Defaults to “.nc”

Returns The base filename of the dataset.

Return type *str*

static get_raw_filename (*raw_dataset: xarray.Dataset*, *old_filename: str*, *config*) → *str*

Returns the appropriate raw filename of the raw dataset according to MHKIT-Cloud naming conventions. Uses the config object to parse the start date and time from the raw dataset for use in the new filename.

The new filename will follow the MHKIT-Cloud Data standards for raw filenames, ie: *datastream_name.date.time.raw.old_filename*, where the data level used in the *datastream_name* is *00*.

Parameters

- **raw_dataset** (*xr.Dataset*) – The raw data as an xarray dataset.
- **old_filename** (*str*) – The name of the original raw file.
- **config** (*Config*) – The Config object used to assist reading time data from the *raw_dataset*.

Returns The standardized filename of the raw file.

Return type *str*

static get_date_from_filename (*filename: str*) → *str*

Given a filename that conforms to MHKIT-Cloud Data Standards, return the date of the first point of data in the file.

Parameters filename (*str*) – The filename or path to the file.

Returns The date, in “yyyymmdd.hhmmss” format.

Return type *str*

static get_datastream_name_from_filename (*filename: str*) → *Optional[str]*

Given a filename that conforms to MHKIT-Cloud Data Standards, return the datastream name. Datastream name is everything to the left of the third ‘.’ in the filename.

e.g., *humboldt_ca.buoy_data.b1.20210120.000000.nc*

Parameters filename (*str*) – The filename or path to the file.

Returns The datastream name, or *None* if filename is not in proper format.

Return type *Optional[str]*

static get_datastream_directory (*datastream_name: str*, *root: str = ""*) → *str*

Given the *datastream_name* and an optional *root*, returns the path to where the datastream should be located. Does NOT create the directory where the datastream should be located.

Parameters

- **datastream_name** (*str*) – The name of the datastream whose directory path should be generated.
- **root** (*str*, *optional*) – The directory to use as the root of the directory structure. Defaults to *None*. Defaults to “”

Returns The path to the directory where the datastream should be located.

Return type *str*

static `is_image(filename: str) → bool`

Detect the mimetype from the file extension and use it to determine if the file is an image or not

Parameters `filename` (*str*) – The name of the file to check

Returns True if the file extension matches an image mimetype

Return type bool

Package Contents

Classes

<code>DSUtil</code>	Provides helper functions for <code>xarray.Dataset</code>
<code>Converter</code>	Base class for converting data arrays from one units to another.
<code>DefaultConverter</code>	Default class for converting units on data arrays. This class utilizes
<code>StringTimeConverter</code>	Convert a time string to a <code>np.datetime64</code> , which is needed for <code>xarray</code> .
<code>TimestampTimeConverter</code>	Convert a numeric UTC timestamp to a <code>np.datetime64</code> , which is needed for

class `tsdat.utils.DSUtil`

Provides helper functions for `xarray.Dataset`

static `record_corrections_applied(ds: xarray.Dataset, variable: str, correction: str)`

Records a description of a correction made to a variable to the `corrections_applied` corresponding attribute.

Parameters

- `ds` (`xr.Dataset`) – Dataset containing the corrected variable
- `variable` (*str*) – The name of the variable that was corrected
- `correction` (*str*) – A description of the correction

static `datetime64_to_string(datetime64: numpy.datetime64) → Tuple[str, str]`

Convert a `datetime64` object to formatted string.

Parameters `datetime64` (`Union[np.ndarray, np.datetime64]`) – The `datetime64` object

Returns A tuple of strings representing the formatted date. The first string is the day in ‘yyyymm-dd’ format. The second string is the time in ‘hhmmss’ format.

Return type `Tuple[str, str]`

static `datetime64_to_timestamp(variable_data: numpy.ndarray) → numpy.ndarray`

Converts each `datetime64` value to a timestamp in same units as the variable (eg., seconds, nanoseconds).

Parameters `variable_data` (`np.ndarray`) – `ndarray` of variable data

Returns An `ndarray` of the same shape, with time values converted to long timestamps (e.g., `int64`)

Return type `np.ndarray`

static `get_datastream_name(ds: xarray.Dataset = None, config=None) → str`

Returns the datastream name defined in the dataset or in the provided pipeline configuration.

Parameters

- **ds** (*xr.Dataset*, *optional*.) – The data as an xarray dataset; defaults to None
- **config** (*Config*, *optional*) – The Config object used to assist reading time data from the raw_dataset; defaults to None.

Returns The datastream name

Return type str

static get_end_time (*ds: xarray.Dataset*) → Tuple[str, str]
Convenience method to get the end date and time from a xarray dataset.

Parameters **ds** (*xr.Dataset*) – The dataset

Returns A tuple of [day, time] as formatted strings representing the last time point in the dataset.

Return type Tuple[str, str]

static get_fill_value (*ds: xarray.Dataset*, *variable_name: str*)
Get the value of the _FillValue attribute for the given variable.

Parameters

- **ds** (*xr.Dataset*) – The dataset
- **variable_name** (*str*) – A variable in the dataset

Returns The value of the _FillValue attr or None if it is not defined

Return type same data type of the variable (int, float, etc.) or None

static get_non_qc_variable_names (*ds: xarray.Dataset*) → List[str]
Get a list of all data variables in the dataset that are NOT qc variables.

Parameters **ds** (*xr.Dataset*) – A dataset

Returns List of non-qc data variable names

Return type List[str]

static get_raw_end_time (*raw_ds: xarray.Dataset*, *time_var_definition: ts-*
dat.VariableDefinition) → Tuple[str, str]

Convenience method to get the end date and time from a raw xarray dataset. This uses *time_var_definition.get_input_name()* as the dataset key for the time variable and additionally uses the input's *Converter* object if applicable.

Parameters

- **raw_ds** (*xr.Dataset*) – A raw dataset (not standardized)
- **time_var_definition** (*VariableDefinition*) – The 'time' variable definition from the pipeline config

Returns A tuple of strings representing the last time data point in the dataset. The first string is the day in 'yyyymmdd' format. The second string is the time in 'hhmmss' format.

Return type Tuple[str, str]

static get_raw_start_time (*raw_ds: xarray.Dataset*, *time_var_definition: ts-*
dat.config.VariableDefinition) → Tuple[str, str]

Convenience method to get the start date and time from a raw xarray dataset. This uses *time_var_definition.get_input_name()* as the dataset key for the time variable and additionally uses the input's *Converter* object if applicable.

Parameters

- **raw_ds** (*xr.Dataset*) – A raw dataset (not standardized)
- **time_var_definition** (*VariableDefinition*) – The ‘time’ variable definition from the pipeline config

Returns A tuple of strings representing the first time data point in the dataset. The first string is the day in ‘yyyymmdd’ format. The second string is the time in ‘hhmmss’ format.

Return type Tuple[str, str]

static get_coordinate_variable_names (*ds: xarray.Dataset*) → List[str]

Get a list of all coordinate variables in this dataset.

Parameters **ds** (*xr.Dataset*) – The dataset

Returns List of coordinate variable names

Return type List[str]

static get_start_time (*ds: xarray.Dataset*) → Tuple[str, str]

Convenience method to get the start date and time from a xarray dataset.

Parameters **ds** (*xr.Dataset*) – A standardized dataset

Returns A tuple of strings representing the first time data point in the dataset. The first string is the day in ‘yyyymmdd’ format. The second string is the time in ‘hhmmss’ format.

Return type Tuple[str, str]

static get_metadata (*ds: xarray.Dataset*) → Dict

Get a dictionary of all global and variable attributes in a dataset. Global atts are found under the ‘attributes’ key and variable atts are found under the ‘variables’ key.

Parameters **ds** (*xr.Dataset*) – A dataset

Returns A dictionary of global & variable attributes

Return type Dict

static plot_qc (*ds: xarray.Dataset, variable_name: str, filename: str = None, **kwargs*) → *act.plotting.TimeSeriesDisplay*

Create a QC plot for the given variable. This is based on the ACT library: https://arm-doe.github.io/ACT/source/auto_examples/plot_qc.html#sphx-glr-source-auto-examples-plot-qc-py

We provide a convenience wrapper method for basic QC plots of a variable, but we recommend to use ACT directly and look at their examples for more complex plots like plotting variables in two different datasets.

TODO: Depending on use cases, we will likely add more arguments to be able to quickly produce the most common types of QC plots.

Parameters

- **ds** (*xr.Dataset*) – A dataset
- **variable_name** (*str*) – The variable to plot
- **filename** (*str, optional*) – The filename for the image. Saves the plot as this filename if provided.

static get_plot_filename (*dataset: xarray.Dataset, plot_description: str, extension: str*) → str

Returns the filename for a plot according to MHKIT-Cloud Data standards. The dataset is used to determine the *datastream_name* and start date/time. The standards dictate that a plot filename should follow the format: *datastream_name.date.time.description.extension*.

Parameters

- **dataset** (*xr.Dataset*) – The dataset from which the plot data is drawn from. This is used to collect the `datastream_name` and start date/time.
- **plot_description** (*str*) – The description of the plot. Should be as brief as possible and contain no spaces. Underscores may be used.
- **extension** (*str*) – The file extension for the plot.

Returns The standardized plot filename.

Return type `str`

static get_dataset_filename (*dataset: xarray.Dataset, file_extension='nc'*) → `str`

Given an xarray dataset this function will return the base filename of the dataset according to MHKiT-Cloud data standards. The base filename does not include the directory structure where the file should be saved, only the name of the file itself, e.g. `z05.ExampleBuoyDatastream.b1.20201230.000000.nc`

Parameters

- **dataset** (*xr.Dataset*) – The dataset whose filename should be generated.
- **file_extension** (*str, optional*) – The file extension to use. Defaults to “.nc”

Returns The base filename of the dataset.

Return type `str`

static get_raw_filename (*raw_dataset: xarray.Dataset, old_filename: str, config*) → `str`

Returns the appropriate raw filename of the raw dataset according to MHKIT-Cloud naming conventions. Uses the config object to parse the start date and time from the raw dataset for use in the new filename.

The new filename will follow the MHKIT-Cloud Data standards for raw filenames, ie: `datastream_name.date.time.raw.old_filename`, where the data level used in the `datastream_name` is `00`.

Parameters

- **raw_dataset** (*xr.Dataset*) – The raw data as an xarray dataset.
- **old_filename** (*str*) – The name of the original raw file.
- **config** (*Config*) – The Config object used to assist reading time data from the `raw_dataset`.

Returns The standardized filename of the raw file.

Return type `str`

static get_date_from_filename (*filename: str*) → `str`

Given a filename that conforms to MHKiT-Cloud Data Standards, return the date of the first point of data in the file.

Parameters **filename** (*str*) – The filename or path to the file.

Returns The date, in “yyyymmdd.hhmmss” format.

Return type `str`

static get_datastream_name_from_filename (*filename: str*) → `Optional[str]`

Given a filename that conforms to MHKiT-Cloud Data Standards, return the datastream name. Datastream name is everything to the left of the third ‘.’ in the filename.

e.g., `humboldt_ca.buoy_data.b1.20210120.000000.nc`

Parameters **filename** (*str*) – The filename or path to the file.

Returns The datastream name, or `None` if filename is not in proper format.

Return type Optional[str]

static `get_datastream_directory` (*datastream_name: str, root: str = ""*) → str

Given the `datastream_name` and an optional `root`, returns the path to where the datastream should be located. Does NOT create the directory where the datastream should be located.

Parameters

- **datastream_name** (*str*) – The name of the datastream whose directory path should be generated.
- **root** (*str, optional*) – The directory to use as the root of the directory structure. Defaults to None. Defaults to ""

Returns The path to the directory where the datastream should be located.

Return type str

static `is_image` (*filename: str*) → bool

Detect the mimetype from the file extension and use it to determine if the file is an image or not

Parameters **filename** (*str*) – The name of the file to check

Returns True if the file extension matches an image mimetype

Return type bool

class `tsdat.utils.Converter` (*parameters: Union[Dict, None] = None*)

Bases: `abc.ABC`

Base class for converting data arrays from one units to another. Users can extend this class if they have a special units conversion for their input data that cannot be resolved with the default converter classes.

Parameters **parameters** (*dict, optional*) – A dictionary of converter-specific parameters which get passed from the pipeline config file. Defaults to {}

abstract `run` (*self, data: numpy.ndarray, in_units: str, out_units: str*) → `numpy.ndarray`

Convert the input data from `in_units` to `out_units`.

Parameters

- **data** (*np.ndarray*) – Data array to be modified.
- **in_units** (*str*) – Current units of the data array.
- **out_units** (*str*) – Units to be converted to.

Returns Data array converted into the new units.

Return type `np.ndarray`

class `tsdat.utils.DefaultConverter` (*parameters: Union[Dict, None] = None*)

Bases: `Converter`

Default class for converting units on data arrays. This class utilizes `ACT.utils.data_utils.convert_units`, and should work for most variables except time (see `StringTimeConverter` and `TimestampTimeConverter`)

run (*self, data: numpy.ndarray, in_units: str, out_units: str*) → `numpy.ndarray`

Convert the input data from `in_units` to `out_units`.

Parameters

- **data** (*np.ndarray*) – Data array to be modified.
- **in_units** (*str*) – Current units of the data array.
- **out_units** (*str*) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

class tsdat.utils.StringTimeConverter (parameters: Union[Dict, None] = None)

Bases: Converter

Convert a time string to a np.datetime64, which is needed for xarray. This class utilizes pd.to_datetime to perform the conversion.

One of the parameters should be 'time_format', which is the the strftime to parse time, eg "%d/%m/%Y". Note that "%f" will parse all the way up to nanoseconds. See strftime documentation for more information on choices.

Parameters **parameters** (dict, optional) – dictionary of converter-specific parameters.

Defaults to {}.

run (self, data: numpy.ndarray, in_units: str, out_units: str) → numpy.ndarray

Convert the input data from in_units to out_units.

Parameters

- **data** (np.ndarray) – Data array to be modified.
- **in_units** (str) – Current units of the data array.
- **out_units** (str) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

class tsdat.utils.TimestampTimeConverter (parameters: Union[Dict, None] = None)

Bases: Converter

Convert a numeric UTC timestamp to a np.datetime64, which is needed for xarray. This class utilizes pd.to_datetime to perform the conversion.

One of the parameters should be 'unit'. This parameter denotes the time unit (e.g., D,s,ms,us,ns), which is an integer or float number. The timestamp will be based off the unix epoch start.

Parameters **parameters** (dict, optional) – A dictionary of converter-specific parameters which get passed from the pipeline config file. Defaults to {}

run (self, data: numpy.ndarray, in_units: str, out_units: str) → numpy.ndarray

Convert the input data from in_units to out_units.

Parameters

- **data** (np.ndarray) – Data array to be modified.
- **in_units** (str) – Current units of the data array.
- **out_units** (str) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

7.1.2 Package Contents

Classes

<i>Config</i>	Wrapper for the pipeline configuration file.
<i>PipelineDefinition</i>	Wrapper for the pipeline portion of the pipeline config file.
<i>DatasetDefinition</i>	Wrapper for the dataset_definition portion of the pipeline config
<i>DimensionDefinition</i>	Class to represent dimensions defined in the pipeline config file.
<i>VariableDefinition</i>	Class to encode variable definitions from the config file. Also provides
<i>ATTS</i>	Class that adds constants for interacting with tsdat data-model
<i>VARs</i>	Class that adds keywords for referring to variables.
<i>DatastreamStorage</i>	DatastreamStorage is the base class for providing
<i>AwsStorage</i>	DatastreamStorage subclass for an AWS S3-based filesystem.
<i>S3Path</i>	This class wraps a ‘special’ path string that lets us include the
<i>FilesystemStorage</i>	Datastreamstorage subclass for a local Linux-based filesystem.
<i>AbstractFileHandler</i>	Abstract class to define methods required by all File-Handlers. Classes
<i>FileHandler</i>	Class to provide methods to read and write files with a variety of
<i>CsvHandler</i>	FileHandler to read from and write to CSV files. Takes a number of
<i>NetCdfHandler</i>	FileHandler to read from and write to netCDF files. Takes a number of
<i>Pipeline</i>	This class serves as the base class for all tsdat data pipelines.
<i>IngestPipeline</i>	The IngestPipeline class is designed to read in raw, non-standardized
<i>QualityChecker</i>	Class containing the code to perform a single Quality Check on a
<i>QualityHandler</i>	Class containing code to be executed if a particular quality check fails.
<i>DSUtil</i>	Provides helper functions for xarray.Dataset
<i>Converter</i>	Base class for converting data arrays from one units to another.
<i>DefaultConverter</i>	Default class for converting units on data arrays. This class utilizes
<i>StringTimeConverter</i>	Convert a time string to a np.datetime64, which is needed for xarray.
<i>TimestampTimeConverter</i>	Convert a numeric UTC timestamp to a np.datetime64, which is needed for

Functions

<code>register_filehandler(patterns: Union[str, List[str]]) → AbstractFileHandler</code>	Python decorator to register an AbstractFileHandler in the FileHandler
--	--

class `tsdat.Config` (*dictionary: Dict*)

Wrapper for the pipeline configuration file.

Note: in most cases, `Config.load(filepath)` should be used to instantiate the Config class.

Parameters `dictionary` (*Dict*) – The pipeline configuration file as a dictionary.

`_parse_quality_managers` (*self, dictionary: Dict*) → *Dict[str, ts-dat.config.quality_manager_definition.QualityManagerDefinition]*
 Extracts QualityManagerDefinitions from the config file.

Parameters `dictionary` (*Dict*) – The quality_management dictionary.

Returns Mapping of quality manager name to QualityManagerDefinition

Return type *Dict[str, QualityManagerDefinition]*

classmethod `load` (*self, filepaths: List[str]*)

Load one or more yaml pipeline configuration files. Multiple files should only be passed as input if the pipeline configuration file is split across multiple files.

Parameters `filepaths` (*List[str]*) – The path(s) to yaml configuration files to load.

Returns A Config object wrapping the yaml configuration file(s).

Return type *Config*

static `lint_yaml` (*filename: str*)

Lints a yaml file and raises an exception if an error is found.

Parameters `filename` (*str*) – The path to the file to lint.

Raises **Exception** – Raises an exception if an error is found.

class `tsdat.PipelineDefinition` (*dictionary: Dict[str, Dict]*)

Wrapper for the pipeline portion of the pipeline config file.

Parameters `dictionary` (*Dict[str]*) – The pipeline component of the pipeline config file.

Raises **DefinitionError** – Raises DefinitionError if one of the file naming components contains an illegal character.

check_file_name_components (*self*)

Performs sanity checks on the config properties used in naming files output by tsdat pipelines.

Raises **DefinitionError** – Raises DefinitionError if a component has been set improperly.

class `tsdat.DatasetDefinition` (*dictionary: Dict, datastream_name: str*)

Wrapper for the dataset_definition portion of the pipeline config file.

Parameters

- **`dictionary`** (*Dict*) – The portion of the config file corresponding with the dataset definition.

- **`datastream_name`** (*str*) – The name of the datastream that the config file is for.

`_parse_dimensions` (*self, dictionary: Dict*) → *Dict[str, ts-dat.config.dimension_definition.DimensionDefinition]*

Extracts the dimensions from the dataset_definition portion of the config file.

Parameters **dictionary** (*Dict*) – The dataset_definition dictionary from the config file.

Returns Returns a mapping of output dimension names to DimensionDefinition objects.

Return type Dict[str, *DimensionDefinition*]

_parse_variables (*self*, *dictionary*: *Dict*, *available_dimensions*: *Dict*[str, *tsdat.config.dimension_definition.DimensionDefinition*]) → Dict[str, *tsdat.config.variable_definition.VariableDefinition*]

Extracts the variables from the dataset_definition portion of the config file.

Parameters

- **dictionary** (*Dict*) – The dataset_definition dictionary from the config file.
- **available_dimensions** (*Dict*[str, *DimensionDefinition*]) – The DimensionDefinition objects that have already been parsed.

Returns Returns a mapping of output variable names to VariableDefinition objects.

Return type Dict[str, *VariableDefinition*]

_parse_coordinates (*self*, *vars*: *Dict*[str, *tsdat.config.variable_definition.VariableDefinition*]) → Tuple[Dict[str, *tsdat.config.variable_definition.VariableDefinition*], Dict[str, *tsdat.config.variable_definition.VariableDefinition*]]

Separates coordinate variables and data variables.

Determines which variables are coordinate variables and moves those variables from *self.vars* to *self.coords*. Coordinate variables are defined as variables that are dimensioned by themselves, i.e., *var.name == var.dim.name* is a true statement for coordinate variables, but false for data variables.

Parameters **vars** (*Dict*[str, *VariableDefinition*]) – The dictionary of VariableDefinition objects to check.

Returns The dictionary of dimensions in the dataset.

Return type Tuple[Dict[str, *VariableDefinition*], Dict[str, *VariableDefinition*]]

_validate_dataset_definition (*self*)

Performs sanity checks on the DatasetDefinition object.

Raises *DefinitionError* – If any sanity checks fail.

get_attr (*self*, *attribute_name*) → Any

Retrieves the value of the attribute requested, or None if it does not exist.

Parameters **attribute_name** (*str*) – The name of the attribute to retrieve.

Returns The value of the attribute, or None.

Return type Any

get_variable_names (*self*) → List[str]

Retrieves the list of variable names. Note that this excludes coordinate variables.

Returns The list of variable names.

Return type List[str]

get_variable (*self*, *variable_name*: *str*) → *tsdat.config.variable_definition.VariableDefinition*

Attempts to retrieve the requested variable. First searches the data variables, then searches the coordinate variables. Returns None if no data or coordinate variables have been defined with the requested variable name.

Parameters **variable_name** (*str*) – The name of the variable to retrieve.

Returns Returns the VariableDefinition for the variable, or None if the variable could not be found.

Return type *VariableDefinition*

get_coordinates (*self*, *variable*: *tsdat.config.variable_definition.VariableDefinition*) → *List[tsdat.config.variable_definition.VariableDefinition]*

Returns the coordinate VariableDefinition object(s) that dimension the requested VariableDefinition.

Parameters **variable** (*VariableDefinition*) – The VariableDefinition whose coordinate variables should be retrieved.

Returns A list of VariableDefinition coordinate variables that dimension the provided VariableDefinition.

Return type *List[VariableDefinition]*

get_static_variables (*self*) → *List[tsdat.config.variable_definition.VariableDefinition]*

Retrieves a list of static VariableDefinition objects. A variable is defined as static if it has a “data” section in the config file, which would mean that the variable’s data is defined statically. For example, in the config file snippet below, “depth” is a static variable:

```
depth:
  data: [4, 8, 12]
  dims: [depth]
  type: int
  attrs:
    long_name: Depth
    units: m
```

Returns The list of static VariableDefinition objects.

Return type *List[VariableDefinition]*

class *tsdat.DimensionDefinition* (*name*: *str*, *length*: *Union[str, int]*)

Class to represent dimensions defined in the pipeline config file.

Parameters

- **name** (*str*) – The name of the dimension
- **length** (*Union[str, int]*) – The length of the dimension. This should be one of: "unlimited", "variable", or a positive *int*. The ‘time’ dimension should always have length of "unlimited".

is_unlimited (*self*) → *bool*

Returns True if the dimension has unlimited length. Represented by setting the length attribute to "unlimited".

Returns True if the dimension has unlimited length.

Return type *bool*

is_variable_length (*self*) → *bool*

Returns True if the dimension has variable length, meaning that the dimension’s length is set at runtime. Represented by setting the length to "variable".

Returns True if the dimension has variable length, False otherwise.

Return type *bool*

```
class tsdat.VariableDefinition (name: str, dictionary: Dict, available_dimensions: Dict[str, ts-  
                                dat.config.dimension_definition.DimensionDefinition], defaults:  
                                Union[Dict, None] = None)
```

Class to encode variable definitions from the config file. Also provides a few utility methods.

Parameters

- **name** (*str*) – The name of the variable in the output file.
- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.

:param available_dimensions: A mapping of dimension name to DimensionDefinition objects.

Parameters defaults (*Dict, optional*) – The defaults to use when instantiating this VariableDefinition object, defaults to {}.

_parse_input (*self, dictionary: Dict, defaults: Union[Dict, None] = None*) → VarInput
Parses the variable's input property, if it has one, from the variable dictionary.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (*Dict, optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A VarInput object for this VariableDefinition, or None.

Return type VarInput

_parse_attributes (*self, dictionary: Dict, defaults: Union[Dict, None] = None*) → Dict[str, Any]
Parses the variable's attributes from the variable dictionary.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (*Dict, optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A mapping of attribute name to attribute value.

Return type Dict[str, Any]

```
_parse_dimensions (self, dictionary: Dict, available_dimensions: Dict[str, ts-  
                                dat.config.dimension_definition.DimensionDefinition],  
                                defaults: Union[Dict, None] = None) → Dict[str, ts-  
                                dat.config.dimension_definition.DimensionDefinition]
```

Parses the variable's dimensions from the variable dictionary.

Parameters

- **dictionary** (*Dict*) – The dictionary entry corresponding with this variable in the config file.
- **available_dimensions** – A mapping of dimension name to DimensionDefinition.
- **defaults** (*Dict, optional*) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Returns A mapping of dimension name to DimensionDefinition objects.

Return type Dict[str, *DimensionDefinition*]

`_parse_data_type` (*self*, *dictionary*: Dict, *defaults*: Union[Dict, None] = None) → object
 Parses the data_type string and returns the appropriate numpy data type (i.e. “float” -> np.float).

Parameters

- **dictionary** (Dict) – The dictionary entry corresponding with this variable in the config file.
- **defaults** (Dict, optional) – The defaults to use when instantiating the VariableDefinition object, defaults to {}.

Raises **KeyError** – Raises KeyError if the data type in the dictionary does not match a valid data type.

Returns The numpy data type corresponding with the type provided in the yaml file, or data_type if the provided data_type is not in the ME Data Standards list of data types.

Return type object

`add_fillvalue_if_none` (*self*, *attributes*: Dict[str, Any]) → Dict[str, Any]

Adds the _FillValue attribute to the provided attributes dictionary if the _FillValue attribute has not already been defined and returns the modified attributes dictionary.

Parameters **attributes** (Dict[str, Any]) – The dictionary containing user-defined variable attributes.

Returns The dictionary containing user-defined variable attributes. Is guaranteed to have a _FillValue attribute.

Return type Dict[str, Any]

`is_constant` (*self*) → bool

Returns True if the variable is a constant. A variable is constant if it does not have any dimensions.

Returns True if the variable is constant, False otherwise.

Return type bool

`is_predefined` (*self*) → bool

Returns True if the variable’s data was predefined in the config yaml file.

Returns True if the variable is predefined, False otherwise.

Return type bool

`is_coordinate` (*self*) → bool

Returns True if the variable is a coordinate variable. A variable is defined as a coordinate variable if it is dimensioned by itself.

Returns True if the variable is a coordinate variable, False otherwise.

Return type bool

`is_derived` (*self*) → bool

Return True if the variable is derived. A variable is derived if it does not have an input and it is not predefined.

Returns True if the Variable is derived, False otherwise.

Return type bool

`has_converter` (*self*) → bool

Returns True if the variable has an input converter defined, False otherwise.

Returns True if the Variable has a converter defined, False otherwise.

Return type bool

is_required (*self*) → bool

Returns True if the variable has the 'required' property defined and the 'required' property evaluates to True. A required variable is a variable which must be retrieved in the input dataset. If a required variable is not in the input dataset, the process should crash.

Returns True if the variable is required, False otherwise.

Return type bool

has_input (*self*) → bool

Return True if the variable is copied from an input dataset, regardless of whether or not unit and/or naming conversions should be applied.

Returns True if the Variable has an input defined, False otherwise.

Return type bool

get_input_name (*self*) → str

Returns the name of the variable in the input if defined, otherwise returns None.

Returns The name of the variable in the input, or None.

Return type str

get_input_units (*self*) → str

If the variable has input, returns the units of the input variable or the output units if no input units are defined.

Returns The units of the input variable data.

Return type str

get_output_units (*self*) → str

Returns the units of the output data or None if no units attribute has been defined.

Returns The units of the output variable data.

Return type str

get_coordinate_names (*self*) → List[str]

Returns the names of the coordinate VariableDefinition(s) that this VariableDefinition is dimensioned by.

Returns A list of dimension/coordinate variable names.

Return type List[str]

get_shape (*self*) → Tuple[int]

Returns the shape of the data attribute on the VariableDefinition.

Raises **KeyError** – Raises a KeyError if the data attribute has not been set yet.

Returns The shape of the VariableDefinition's data, or None.

Return type Tuple[int]

get_data_type (*self*) → numpy.dtype

Retrieves the variable's data type.

Returns Returns the data type of the variable's data as a numpy dtype.

Return type np.dtype

get_FillValue (*self*) → int

Retrieves the variable's `_FillValue` attribute, using -9999 as a default if it has not been defined.

Returns Returns the variable's `_FillValue`.

Return type int

run_converter (*self*, *data*: *numpy.ndarray*) → *numpy.ndarray*

If the variable has an input converter, runs the input converter for the input/output units on the provided data.

Parameters **data** (*np.ndarray*) – The data to be converted.

Returns Returns the data after it has been run through the variable's converter.

Return type *np.ndarray*

to_dict (*self*) → Dict

Returns the Variable as a dictionary to be used to initialize an empty xarray Dataset or DataArray.

Returns a dictionary like (Example is for *temperature*):

```
{
    "dims": ["time"],
    "data": [],
    "attrs": {"units": "degC"}
}
```

Returns A dictionary representation of the variable.

Return type Dict

class `tsdat.ATTS`

Class that adds constants for interacting with tsdat data-model specific attributes.

TITLE = `title`

DESCRIPTION = `description`

CONVENTIONS = `conventions`

HISTORY = `history`

DOI = `doi`

INSTITUTION = `institution`

CODE_URL = `code_url`

REFERENCES = `references`

INPUT_FILES = `input_files`

LOCATION_ID = `location_id`

DATASTREAM = `datastream_name`

DATA_LEVEL = `data_level`

LOCATION_DESCRIPTION = `location_description`

INSTRUMENT_NAME = `instrument_name`

SERIAL_NUMBER = `serial_number`

INSTRUMENT_DESCRIPTION = `instrument_description`

```
INSTRUMENT_MANUFACTURER = instrument_manufacturer
AVERAGING_INTERVAL = averaging_interval
SAMPLING_INTERVAL = sampling_interval
UNITS = units
VALID_DELTA = valid_delta
VALID_RANGE = valid_range
FAIL_RANGE = fail_range
WARN_RANGE = warn_range
FILL_VALUE = _FillValue
CORRECTIONS_APPLIED = corrections_applied
```

```
class tsdat.VARS
```

Class that adds keywords for referring to variables.

```
ALL = ALL
```

```
COORDS = COORDS
```

```
DATA_VARS = DATA_VARS
```

```
exception tsdat.DefinitionError
```

Bases: Exception

Indicates a fatal error within the YAML Dataset Definition.

```
exception tsdat.QCError
```

Bases: Exception

Indicates that a given Quality Manager failed with a fatal error.

```
class tsdat.DatastreamStorage (parameters: Union[Dict, None] = None)
```

Bases: abc.ABC

DatastreamStorage is the base class for providing access to processed data files in a persistent archive. DatastreamStorage provides shortcut methods to find files based upon date, datastream name, file type, etc. This is the class that should be used to save and retrieve processed data files. Use the DatastreamStorage.from_config() method to construct the appropriate subclass instance based upon a storage config file.

```
default_file_type
```

```
file_filters
```

```
output_file_extensions
```

```
static from_config (storage_config_file: str)
```

Load a yaml config file which provides the storage constructor parameters.

Parameters `storage_config_file` (*str*) – The path to the config file to load

Returns A subclass instance created from the config file.

Return type *DatastreamStorage*

```
property tmp (self)
```

Each subclass should define the tmp property, which provides access to a TemporaryStorage object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the DatastreamStorage. Is is not intended to be used outside of the pipeline.

Raises `NotImplementedError` – [description]

abstract find (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *str* = *None*) → *List*[*str*]

Finds all files of the given type from the datastream store with the given *datastream_name* and timestamps from *start_time* (inclusive) up to *end_time* (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The *datastream_name* as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to *None*.

Returns A list of paths in datastream storage in ascending order

Return type *List*[*str*]

abstract fetch (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *local_path*: *str* = *None*, *filetype*: *int* = *None*)

Fetches files from the datastream store using the *datastream_name*, *start_time*, and *end_time* to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The *datastream_name* as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str*, *optional*) – The path to the directory where the data should be stored. Defaults to *None*.
- **filetype** (*int*, *optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to *None*.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so it this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type `DisposableLocalTempFileList`:

save (*self*, *dataset_or_path*: *Union*[*str*, *xarray.Dataset*], *new_filename*: *str* = *None*) → *List*[*Any*]

Saves a local file to the datastream store.

Parameters

- **dataset_or_path** (*Union*[*str*, *xr.Dataset*]) – The dataset or local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str*, *optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for *dataset_or_path*. Must also follow ME Data Standards naming conventions. Defaults to *None*.

Returns A list of paths where the saved files were stored in storage. Path type is dependent upon the specific storage subclass.

Return type List[Any]

abstract save_local_path (*self*, *local_path*: str, *new_filename*: str = None) → Any
Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str*, *optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for dataset_or_path. Must also follow ME Data Standards naming conventions. Defaults to None.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type Any

abstract exists (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *filetype*: str = None) → bool
Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If none specified, all files will be checked. Defaults to None.

Returns True if data exists, False otherwise.

Return type bool

abstract delete (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *filetype*: str = None) → None
Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be deleted. Defaults to None.

class tsdat.AwsStorage (*parameters*: Union[Dict, None] = None)
Bases: tsdat.io.DatastreamStorage

DatastreamStorage subclass for an AWS S3-based filesystem.

Parameters `parameters` (*dict, optional*) – Dictionary of parameters that should be set automatically from the storage config file when this class is instantiated via the `DatstreamStorage.from-config()` method. Defaults to `{}`

Key parameters that should be set in the config file include

retain_input_files Whether the input files should be cleaned up after they are done processing

root_dir The bucket ‘key’ to use to prepend to all processed files created in the persistent store. Defaults to ‘root’

temp_dir The bucket ‘key’ to use to prepend to all temp files created in the S3 bucket. Defaults to ‘temp’

bucket_name The name of the S3 bucket to store to

property `s3_resource` (*self*)

property `s3_client` (*self*)

property `tmp` (*self*)

Each subclass should define the `tmp` property, which provides access to a `TemporaryStorage` object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the `DatstreamStorage`. Is is not intended to be used outside of the pipeline.

Raises `NotImplementedError` – [description]

property `root` (*self*)

property `temp_path` (*self*)

find (*self*, *datastream_name: str*, *start_time: str*, *end_time: str*, *filetype: str = None*) → `List[S3Path]`

Finds all files of the given type from the datastream store with the given `datastream_name` and timestamps from `start_time` (inclusive) up to `end_time` (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.
- **filetype** (*str, optional*) – A file type from the `DatstreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to `None`.

Returns A list of paths in datastream storage in ascending order

Return type `List[str]`

fetch (*self*, *datastream_name: str*, *start_time: str*, *end_time: str*, *local_path: str = None*, *filetype: int = None*) → `tsdat.io.DisposableLocalTempFileList`

Fetches files from the datastream store using the `datastream_name`, `start_time`, and `end_time` to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The `datastream_name` as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.

- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str*, *optional*) – The path to the directory where the data should be stored. Defaults to None.
- **filetype** (*int*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so it this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type DisposableLocalTempFileList:

save_local_path (*self*, *local_path*: *str*, *new_filename*: *str* = None)

Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str*, *optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for dataset_or_path. Must also follow ME Data Standards naming conventions. Defaults to None.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type Any

exists (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *int* = None) → bool

Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If none specified, all files will be checked. Defaults to None.

Returns True if data exists, False otherwise.

Return type bool

delete (*self*, *datastream_name*: *str*, *start_time*: *str*, *end_time*: *str*, *filetype*: *int* = None) → None

Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (*str*) – The datastream_name as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.

- **filetype** (*str*, *optional*) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be deleted. Defaults to None.

class tsdat.S3Path (*bucket_name: str, bucket_path: str = "", region_name: str = None*)

Bases: *str*

This class wraps a ‘special’ path string that lets us include the bucket name and region in the path, so that we can use it seamlessly in boto3 APIs. We are creating our own string to hold the region, bucket & key (i.e., path), since boto3 needs all three in order to access a file.

Example: .. code-block:: python

```
s3_client = boto3.client('s3', region_name='eu-central-1') s3_client.download_file(bucket, key,
download_path)
```

Parameters

- **bucket_name** (*str*) – The S3 bucket name where this file is located
- **bucket_path** (*str*, *optional*) – The key to access this file in the bucket
- **region_name** (*str*, *optional*) – The AWS region where this file is located, defaults to None, which inherits the default configured region.

__str__ (*self*)

Return str(self).

property **bucket_name** (*self*)

property **bucket_path** (*self*)

property **region_name** (*self*)

join (*self*, **args*)

Joins segments in an S3 path. This method behaves exactly like os.path.join.

Returns A New S3Path with the additional segments added.

Return type *S3Path*

class tsdat.FileSystemStorage (*parameters: Union[Dict, None] = None*)

Bases: *tsdat.io.DatastreamStorage*

Datastreamstorage subclass for a local Linux-based filesystem.

TODO: rename to LocalStorage as this is more intuitive.

Parameters **parameters** (*dict*, *optional*) – Dictionary of parameters that should be set automatically from the storage config file when this class is instantiated via the DatastreamStorage.from-config() method. Defaults to {}

Key parameters that should be set in the config file include

retain_input_files Whether the input files should be cleaned up after they are done processing

root_dir The root path under which processed files will e stored.

property **tmp** (*self*)

Each subclass should define the tmp property, which provides access to a TemporaryStorage object that is used to efficiently handle reading/writing temporary files used during the processing pipeline, or to perform filesystem actions on files other than processed datastream files that reside in the same filesystem as the DatastreamStorage. Is is not intended to be used outside of the pipeline.

Raises **NotImplementedError** – [description]

find (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *filetype*: str = None) → List[str]

Finds all files of the given type from the datastream store with the given *datastream_name* and timestamps from *start_time* (inclusive) up to *end_time* (exclusive). Returns a list of paths to files that match the criteria.

Parameters

- **datastream_name** (*str*) – The *datastream_name* as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106.000000” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108.000000” to search for data ending before January 8th, 2021.
- **filetype** (*str*, *optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths in datastream storage in ascending order

Return type List[str]

fetch (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *local_path*: str = None, *filetype*: int = None) → *tsdat.io.DisposableLocalTempFileList*

Fetches files from the datastream store using the *datastream_name*, *start_time*, and *end_time* to specify the file(s) to retrieve. If the local path is not specified, it is up to the subclass to determine where to put the retrieved file(s).

Parameters

- **datastream_name** (*str*) – The *datastream_name* as defined by ME Data Standards.
- **start_time** (*str*) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (*str*) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **local_path** (*str*, *optional*) – The path to the directory where the data should be stored. Defaults to None.
- **filetype** (*int*, *optional*) – A file type from the `DatastreamStorage.file_filters` keys. If no type is specified, all files will be returned. Defaults to None.

Returns A list of paths where the retrieved files were stored in local storage. This is a context manager class, so it this method should be called via the ‘with’ statement and all files referenced by the list will be cleaned up when it goes out of scope.

Return type `DisposableLocalTempFileList`:

save_local_path (*self*, *local_path*: str, *new_filename*: str = None) → Any

Given a path to a local file, save that file to the storage.

Parameters

- **local_path** (*str*) – Local path to the file to save. The file should be named according to ME Data Standards naming conventions so that this method can automatically parse the datastream, date, and time from the file name.
- **new_filename** (*str*, *optional*) – If provided, the new filename to save as. This parameter should ONLY be provided if using a local path for *dataset_or_path*. Must also follow ME Data Standards naming conventions. Defaults to None.

Returns The path where this file was stored in storage. Path type is dependent upon the specific storage subclass.

Return type Any

exists (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *filetype*: int = None) → bool
Checks if any data exists in the datastream store for the provided datastream and time range.

Parameters

- **datastream_name** (str) – The datastream_name as defined by ME Data Standards.
- **start_time** (str) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (str) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (str, optional) – A file type from the DatastreamStorage.file_filters keys. If none specified, all files will be checked. Defaults to None.

Returns True if data exists, False otherwise.

Return type bool

delete (*self*, *datastream_name*: str, *start_time*: str, *end_time*: str, *filetype*: int = None) → None
Deletes datastream data in the datastream store in between the specified time range.

Parameters

- **datastream_name** (str) – The datastream_name as defined by ME Data Standards.
- **start_time** (str) – The start time or date to start searching for data (inclusive). Should be like “20210106” to search for data beginning on or after January 6th, 2021.
- **end_time** (str) – The end time or date to stop searching for data (exclusive). Should be like “20210108” to search for data ending before January 8th, 2021.
- **filetype** (str, optional) – A file type from the DatastreamStorage.file_filters keys. If no type is specified, all files will be deleted. Defaults to None.

class tsdat.**AbstractFileHandler** (*parameters*: Union[Dict, None] = None)

Abstract class to define methods required by all FileHandlers. Classes derived from AbstractFileHandler should implement one or more of the following methods:

```
write(ds: xr.Dataset, filename: str, config: Config, **kwargs)
read(filename: str, **kwargs) -> xr.Dataset
```

Parameters **parameters** (Dict, optional) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (*self*, *ds*: xarray.Dataset, *filename*: str, *config*: tsdat.config.Config = None, ***kwargs*) → None
Saves the given dataset to a file.

Parameters

- **ds** (xr.Dataset) – The dataset to save.
- **filename** (str) – The path to where the file should be written to.
- **config** (Config, optional) – Optional Config object, defaults to None

read (*self*, *filename*: str, ***kwargs*) → xarray.Dataset

Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters **filename** (str) – The path to the file to read in.

Returns A xr.Dataset object.

Return type `xr.Dataset`

class `tsdat.FileHandler`

Class to provide methods to read and write files with a variety of extensions.

FILEREADERS :Dict[str, AbstractFileHandler]

FILEWRITERS :Dict[str, AbstractFileHandler]

static `_get_handler(filename: str, method: Literal[read, write]) → AbstractFileHandler`

Given the filepath of the file to read or write and the FileHandler method to apply to the filepath, this method determines which previously-registered FileHandler should be used on the provided filepath.

Args: filename (str): The path to the file to read or write to. method (Literal[]): The method to apply to the file. Must be one of: “read”, “write”.

Returns: AbstractFileHandler: The FileHandler that should be applied.

static `write(ds: xarray.Dataset, filename: str, config: tsdat.config.Config = None, **kwargs) → None`

Calls the appropriate FileHandler to write the dataset to the provided filename.

Args: ds (xr.Dataset): The dataset to save. filename (str): The path to the file where the dataset should be written. config (Config, optional): Optional Config object. Defaults to None.

static `read(filename: str, **kwargs) → xarray.Dataset`

Reads in the given file and converts it into an xarray dataset object using the registered FileHandler for the provided filepath.

Args: filename (str): The path to the file to read in.

Returns: xr.Dataset: The raw file as an Xarray.Dataset object.

static `register_file_handler(method: Literal[read, write], patterns: Union[str, List[str]], handler: AbstractFileHandler)`

Method to register a FileHandler for reading from or writing to files matching one or more provided file patterns.

Args: method (“Literal”): The method the FileHandler should call if the pattern is matched. Must be one of: “read”, “write”. patterns (Union[str, List[str]]): The file pattern(s) that determine if this FileHandler should be run on a given filepath. handler (AbstractFileHandler): The AbstractFileHandler to register.

class `tsdat.CsvHandler(parameters: Union[Dict, None] = None)`

Bases: `tsdat.io.filehandlers.file_handlers.AbstractFileHandler`

FileHandler to read from and write to CSV files. Takes a number of parameters that are passed in from the storage config file. Parameters specified in the config file should follow the following example:

```
parameters:
  write:
    to_dataframe:
      # Parameters here will be passed to xr.Dataset.to_dataframe()
    to_csv:
      # Parameters here will be passed to pd.DataFrame.to_csv()
  read:
    read_csv:
      # Parameters here will be passed to pd.read_csv()
    to_xarray:
      # Parameters here will be passed to pd.DataFrame.to_xarray()
```

Parameters `parameters` (Dict, optional) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to {}.

write (*self*, *ds*: *xarray.Dataset*, *filename*: *str*, *config*: *tsdat.config.Config* = *None*, ***kwargs*) → *None*
 Saves the given dataset to a csv file.

Parameters

- **ds** (*xr.Dataset*) – The dataset to save.
- **filename** (*str*) – The path to where the file should be written to.
- **config** (*Config*, *optional*) – Optional Config object, defaults to *None*

read (*self*, *filename*: *str*, ***kwargs*) → *xarray.Dataset*
 Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters **filename** (*str*) – The path to the file to read in.

Returns A *xr.Dataset* object.

Return type *xr.Dataset*

class *tsdat.NetCdfHandler* (*parameters*: *Union[Dict, None]* = *None*)

Bases: *tsdat.io.filehandlers.file_handlers.AbstractFileHandler*

FileHandler to read from and write to netCDF files. Takes a number of parameters that are passed in from the storage config file. Parameters specified in the config file should follow the following example:

```
parameters:
  write:
    to_netcdf:
      # Parameters here will be passed to xr.Dataset.to_netcdf()
  read:
    load_dataset:
      # Parameters here will be passed to xr.load_dataset()
```

Parameters **parameters** (*Dict*, *optional*) – Parameters that were passed to the FileHandler when it was registered in the storage config file, defaults to *{}*.

write (*self*, *ds*: *xarray.Dataset*, *filename*: *str*, *config*: *tsdat.config.Config* = *None*, ***kwargs*) → *None*
 Saves the given dataset to a netCDF file.

Parameters

- **ds** (*xr.Dataset*) – The dataset to save.
- **filename** (*str*) – The path to where the file should be written to.
- **config** (*Config*, *optional*) – Optional Config object, defaults to *None*

read (*self*, *filename*: *str*, ***kwargs*) → *xarray.Dataset*
 Reads in the given file and converts it into an Xarray dataset for use in the pipeline.

Parameters **filename** (*str*) – The path to the file to read in.

Returns A *xr.Dataset* object.

Return type *xr.Dataset*

tsdat.register_filehandler (*patterns*: *Union[str, List[str]]*) → *AbstractFileHandler*

Python decorator to register an *AbstractFileHandler* in the FileHandler object. The FileHandler object will be used by *tsdat* pipelines to read and write raw, intermediate, and processed data.

This decorator can be used to work with a specific *AbstractFileHandler* without having to specify a config file. This is useful when using an *AbstractFileHandler* for analysis or for tests outside of a pipeline. For *tsdat* pipelines, handlers should always be specified via the storage config file.

Example Usage:

```
import xarray as xr
from tsdat.io import register_filehandler, AbstractFileHandler

@register_filehandler(["*.nc", "*.cdf"])
class NetCdfHandler(AbstractFileHandler):
    def write(ds: xr.Dataset, filename: str, config: Config = None, **kwargs):
        ds.to_netcdf(filename)
    def read(filename: str, **kwargs) -> xr.Dataset:
        xr.load_dataset(filename)
```

Parameters `patterns` (`Union[str, List[str]]`) – The patterns (regex) that should be used to match a filepath to the `AbstractFileHandler` provided.

Returns The original `AbstractFileHandler` class, after it has been registered for use in tsdat pipelines.

Return type `AbstractFileHandler`

class `tsdat.Pipeline` (`pipeline_config: Union[str, tsdat.config.Config]`, `storage_config: Union[str, tsdat.io.DatastreamStorage]`)

Bases: `abc.ABC`

This class serves as the base class for all tsdat data pipelines.

Parameters

- **pipeline_config** (`Union[str, Config]`) – The pipeline config file. Can be either a config object, or the path to the pipeline config file that should be used with this pipeline.
- **storage_config** (`Union[str, DatastreamStorage]`) – The storage config file. Can be either a config object, or the path to the storage config file that should be used with this pipeline.

abstract run (`self, filepath: Union[str, List[str]]`)

This method is the entry point for the pipeline. It will take one or more file paths and process them from start to finish. All classes extending the `Pipeline` class must implement this method.

Parameters `filepath` (`Union[str, List[str]]`) – The path or list of paths to the file(s) to run the pipeline on.

standardize_dataset (`self, raw_mapping: Dict[str, xarray.Dataset]`) → `xarray.Dataset`

Standardizes the dataset by applying variable name and units conversions as defined by the pipeline config file. This method returns the standardized dataset.

Parameters `raw_mapping` (`Dict[str, xr.Dataset]`) – The raw dataset mapping.

Returns The standardized dataset.

Return type `xr.Dataset`

check_required_variables (`self, dataset: xarray.Dataset, dod: tsdat.config.DatasetDefinition`)

Function to throw an error if a required variable could not be retrieved.

Parameters

- **dataset** (`xr.Dataset`) – The dataset to check.
- **dod** (`DatasetDefinition`) – The `DatasetDefinition` used to specify required variables.

Raises **Exception** – Raises an exception to indicate the variable could not be retrieved.

add_static_variables (*self*, *dataset*: *xarray.Dataset*, *dod*: [tsdat.config.DatasetDefinition](#)) → *xarray.Dataset*

Uses the DatasetDefinition to add static variables (variables whose data are defined in the pipeline config file) to the output dataset.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add static variables to.
- **dod** ([DatasetDefinition](#)) – The DatasetDefinition to pull data from.

Returns The original dataset with added variables from the config

Return type *xr.Dataset*

add_missing_variables (*self*, *dataset*: *xarray.Dataset*, *dod*: [tsdat.config.DatasetDefinition](#)) → *xarray.Dataset*

Uses the dataset definition to initialize variables that are defined in the dataset definition but did not have input. Uses the appropriate shape and _FillValue to initialize each variable.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add the variables to.
- **dod** ([DatasetDefinition](#)) – The DatasetDefinition to use.

Returns The original dataset with variables that still need to be initialized, initialized.

Return type *xr.Dataset*

add_attrs (*self*, *dataset*: *xarray.Dataset*, *raw_mapping*: *Dict[str, xarray.Dataset]*, *dod*: [tsdat.config.DatasetDefinition](#)) → *xarray.Dataset*

Adds global and variable-level attributes to the dataset from the DatasetDefinition object.

Parameters

- **dataset** (*xr.Dataset*) – The dataset to add attributes to.
- **raw_mapping** (*Dict[str, xr.Dataset]*) – The raw dataset mapping. Used to set the `input_files` global attribute.
- **dod** ([DatasetDefinition](#)) – The DatasetDefinition containing the attributes to add.

Returns The original dataset with the attributes added.

Return type *xr.Dataset*

get_previous_dataset (*self*, *dataset*: *xarray.Dataset*) → *xarray.Dataset*

Utility method to retrieve the previous set of data for the same datastream as the provided dataset from the DatastreamStorage.

Parameters **dataset** (*xr.Dataset*) – The reference dataset that will be used to search the DatastreamStore for prior data.

Returns The previous dataset from the DatastreamStorage if it exists, otherwise None.

Return type *xr.Dataset*

reduce_raw_datasets (*self*, *raw_mapping*: *Dict[str, xarray.Dataset]*, *definition*: [tsdat.config.DatasetDefinition](#)) → *List[xarray.Dataset]*

Removes unused variables from each raw dataset in the raw mapping and performs input to output naming and unit conversions as defined in the dataset definition.

Parameters

- **raw_mapping** (*Dict[str, xr.Dataset]*) – The raw xarray dataset mapping.

- **definition** (`DatasetDefinition`) – The `DatasetDefinition` used to select the variables to keep.

Returns A list of reduced datasets.

Return type `List[xr.Dataset]`

reduce_raw_dataset (*self*, *raw_dataset*: `xarray.Dataset`, *variable_definitions*: `List[tsdat.config.VariableDefinition]`, *definition*: `tsdat.config.DatasetDefinition`) → `xarray.Dataset`

Removes unused variables from the raw dataset provided and keeps only the variables and coordinates pertaining to the provided variable definitions. Also performs input to output naming and unit conversions as defined in the `DatasetDefinition`.

Parameters

- **raw_dataset** (`xr.Dataset`) – The raw dataset mapping.
- **variable_definitions** (`List[VariableDefinition]`) – List of variables to keep.
- **definition** (`DatasetDefinition`) – The `DatasetDefinition` used to select the variables to keep.

Returns The reduced dataset.

Return type `xr.Dataset`

store_and_reopen_dataset (*self*, *dataset*: `xarray.Dataset`) → `xarray.Dataset`

Uses the `DatastreamStorage` object to persist the dataset in the format specified by the storage config file.

Parameters **dataset** (`xr.Dataset`) – The dataset to store.

Returns The dataset after it has been saved to disk and reopened.

Return type `xr.Dataset`

class `tsdat.IngestPipeline` (*pipeline_config*: `Union[str, tsdat.config.Config]`, *storage_config*: `Union[str, tsdat.io.DatastreamStorage]`)

Bases: `tsdat.pipeline.pipeline.Pipeline`

The `IngestPipeline` class is designed to read in raw, non-standardized data and convert it to a standardized format by embedding metadata, applying quality checks and quality controls, and by saving the now-processed data in a standard file format.

run (*self*, *filepath*: `Union[str, List[str]]`) → `xarray.Dataset`

Runs the `IngestPipeline` from start to finish.

Parameters **filepath** (`Union[str, List[str]]`) – The path or list of paths to the file(s) to run the pipeline on.

hook_customize_dataset (*self*, *dataset*: `xarray.Dataset`, *raw_mapping*: `Dict[str, xarray.Dataset]`) → `xarray.Dataset`

Hook to allow for user customizations to the standardized dataset such as inserting a derived variable based on other variables in the dataset. This method is called immediately after the `standardize_dataset` method and before `QualityManagement` has been run.

Parameters

- **dataset** (`xr.Dataset`) – The dataset to customize.
- **raw_mapping** (`Dict[str, xr.Dataset]`) – The raw dataset mapping.

Returns The customized dataset.

Return type `xr.Dataset`

hook_customize_raw_datasets (*self*, *raw_dataset_mapping*: Dict[str, xarray.Dataset]) → Dict[str, xarray.Dataset]

Hook to allow for user customizations to one or more raw xarray Datasets before they merged and used to create the standardized dataset. The *raw_dataset_mapping* will contain one entry for each file being used as input to the pipeline. The keys are the standardized raw file name, and the values are the datasets.

This method would typically only be used if the user is combining multiple files into a single dataset. In this case, this method may be used to correct coordinates if they don't match for all the files, or to change variable (column) names if two files have the same name for a variable, but they are two distinct variables.

This method can also be used to check for unique conditions in the raw data that should cause a pipeline failure if they are not met.

This method is called before the inputs are merged and converted to standard format as specified by the config file.

Parameters *raw_dataset_mapping* (Dict[str, xr.Dataset]) – The raw datasets to customize.

Returns The customized raw datasets.

Return type Dict[str, xr.Dataset]

hook_finalize_dataset (*self*, *dataset*: xarray.Dataset) → xarray.Dataset

Hook to apply any final customizations to the dataset before it is saved. This hook is called after Quality-Management has been run and immediately before the dataset is saved to file.

Parameters *dataset* (xr.Dataset) – The dataset to finalize.

Returns The finalized dataset to save.

Return type xr.Dataset

hook_generate_and_persist_plots (*self*, *dataset*: xarray.Dataset) → None

Hook to allow users to create plots from the xarray dataset after the dataset has been finalized and just before the dataset is saved to disk.

To save on filesystem space (which is limited when running on the cloud via a lambda function), this method should only write one plot to local storage at a time. An example of how this could be done is below:

```
filename = DSUtil.get_plot_filename(dataset, "sea_level", "png")
with self.storage.tmp.get_temp_filepath(filename) as tmp_path:
    fig, ax = plt.subplots(figsize=(10,5))
    ax.plot(dataset["time"].data, dataset["sea_level"].data)
    fig.save(tmp_path)
    storage.save(tmp_path)

filename = DSUtil.get_plot_filename(dataset, "qc_sea_level", "png")
with self.storage.tmp.get_temp_filepath(filename) as tmp_path:
    fig, ax = plt.subplots(figsize=(10,5))
    DSUtil.plot_qc(dataset, "sea_level", tmp_path)
    storage.save(tmp_path)
```

Parameters *dataset* (xr.Dataset) – The xarray dataset with customizations and Quality-Management applied.

read_and_persist_raw_files (*self*, *file_paths*: List[str]) → List[str]

Renames the provided raw files according to ME Data Standards file naming conventions for raw data files, and returns a list of the paths to the renamed files.

Parameters `file_paths` (`List[str]`) – A list of paths to the original raw files.

Returns A list of paths to the renamed files.

Return type `List[str]`

```
class tsdat.QualityChecker(ds: xarray.Dataset, previous_data: xarray.Dataset, definition: ts-
                             dat.config.QualityManagerDefinition, parameters: Union[Dict, None]
                             = None)
```

Bases: `abc.ABC`

Class containing the code to perform a single Quality Check on a Dataset variable.

Parameters

- **ds** (`xr.Dataset`) – The dataset the checker will be applied to
- **previous_data** (`xr.Dataset`) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.
- **definition** (`QualityManagerDefinition`) – The quality manager definition as specified in the pipeline config file
- **parameters** (`dict, optional`) – A dictionary of checker-specific parameters specified in the pipeline config file. Defaults to {}

```
abstract run (self, variable_name: str) → Optional[numpy.ndarray]
```

Check a dataset's variable to see if it passes a quality check. These checks can be performed on the entire variable at one time by using xarray vectorized numerical operators.

Parameters `variable_name` (`str`) – The name of the variable to check

Returns

If the check was performed, return a ndarray of the same shape as the variable. Each value in the data array will be either True or False, depending upon the results of the check. True means the check failed. False means it succeeded.

Note that we are using an `np.ndarray` instead of an `xr.DataArray` because the `DataArray` contains coordinate indexes which can sometimes get out of sync when performing `np` arithmetic vector operations. So it's easier to just use numpy arrays.

If the check was skipped for some reason (i.e., it was not relevant given the current attributes defined for this dataset), then the run method should return `None`.

Return type `Optional[np.ndarray]`

```
class tsdat.QualityHandler(ds: xarray.Dataset, previous_data: xarray.Dataset, quality_manager:
                             tsdat.config.QualityManagerDefinition, parameters: Union[Dict,
                             None] = None)
```

Bases: `abc.ABC`

Class containing code to be executed if a particular quality check fails.

Parameters

- **ds** (`xr.Dataset`) – The dataset the handler will be applied to
- **previous_data** (`xr.Dataset`) – A dataset from the previous processing interval (i.e., file). This is used to check for consistency between files, such as for monotonic or delta checks when we need to check the previous value.
- **quality_manager** (`QualityManagerDefinition`) – The quality_manager definition as specified in the pipeline config file

- **parameters** (*dict*, *optional*) – A dictionary of handler-specific parameters specified in the pipeline config file. Defaults to {}

abstract run (*self*, *variable_name*: *str*, *results_array*: *numpy.ndarray*)

Perform a follow-on action if a quality check fails. This can be used to correct data if needed (such as replacing a bad value with missing value, emailing a contact person, or raising an exception if the failure constitutes a critical error).

Parameters

- **variable_name** (*str*) – Name of the variable that failed
- **results_array** (*np.ndarray*) – An array of True/False values for each data value of the variable. True means the check failed.

record_correction (*self*, *variable_name*: *str*)

If a correction was made to variable data to fix invalid values as detected by a quality check, this method will record the fix to the appropriate variable attribute. The correction description will come from the handler params which get set in the pipeline config file.

Parameters **variable_name** (*str*) – Name

class `tsdat.DSUtil`

Provides helper functions for `xarray.Dataset`

static record_corrections_applied (*ds*: *xarray.Dataset*, *variable*: *str*, *correction*: *str*)

Records a description of a correction made to a variable to the `corrections_applied` corresponding attribute.

Parameters

- **ds** (*xr.Dataset*) – Dataset containing the corrected variable
- **variable** (*str*) – The name of the variable that was corrected
- **correction** (*str*) – A description of the correction

static datetime64_to_string (*datetime64*: *numpy.datetime64*) → *Tuple*[*str*, *str*]

Convert a `datetime64` object to formatted string.

Parameters **datetime64** (*Union*[*np.ndarray*, *np.datetime64*]) – The `datetime64` object

Returns A tuple of strings representing the formatted date. The first string is the day in ‘yyyymm-dd’ format. The second string is the time in ‘hhmmss’ format.

Return type *Tuple*[*str*, *str*]

static datetime64_to_timestamp (*variable_data*: *numpy.ndarray*) → *numpy.ndarray*

Converts each `datetime64` value to a timestamp in same units as the variable (eg., seconds, nanoseconds).

Parameters **variable_data** (*np.ndarray*) – ndarray of variable data

Returns An ndarray of the same shape, with time values converted to long timestamps (e.g., `int64`)

Return type *np.ndarray*

static get_datastream_name (*ds*: *xarray.Dataset* = *None*, *config*=*None*) → *str*

Returns the datastream name defined in the dataset or in the provided pipeline configuration.

Parameters

- **ds** (*xr.Dataset*, *optional*.) – The data as an `xarray` dataset; defaults to `None`
- **config** (*Config*, *optional*) – The `Config` object used to assist reading time data from the `raw_dataset`; defaults to `None`.

Returns The datastream name

Return type str

static get_end_time (*ds*: xarray.Dataset) → Tuple[str, str]
Convenience method to get the end date and time from a xarray dataset.

Parameters **ds** (*xr.Dataset*) – The dataset

Returns A tuple of [day, time] as formatted strings representing the last time point in the dataset.

Return type Tuple[str, str]

static get_fill_value (*ds*: xarray.Dataset, *variable_name*: str)
Get the value of the _FillValue attribute for the given variable.

Parameters

- **ds** (*xr.Dataset*) – The dataset
- **variable_name** (*str*) – A variable in the dataset

Returns The value of the _FillValue attr or None if it is not defined

Return type same data type of the variable (int, float, etc.) or None

static get_non_qc_variable_names (*ds*: xarray.Dataset) → List[str]
Get a list of all data variables in the dataset that are NOT qc variables.

Parameters **ds** (*xr.Dataset*) – A dataset

Returns List of non-qc data variable names

Return type List[str]

static get_raw_end_time (*raw_ds*: xarray.Dataset, *time_var_definition*: ts-
dat.VariableDefinition) → Tuple[str, str]

Convenience method to get the end date and time from a raw xarray dataset. This uses *time_var_definition.get_input_name()* as the dataset key for the time variable and additionally uses the input's *Converter* object if applicable.

Parameters

- **raw_ds** (*xr.Dataset*) – A raw dataset (not standardized)
- **time_var_definition** (*VariableDefinition*) – The 'time' variable definition from the pipeline config

Returns A tuple of strings representing the last time data point in the dataset. The first string is the day in 'yyyymmdd' format. The second string is the time in 'hhmmss' format.

Return type Tuple[str, str]

static get_raw_start_time (*raw_ds*: xarray.Dataset, *time_var_definition*: ts-
dat.config.VariableDefinition) → Tuple[str, str]

Convenience method to get the start date and time from a raw xarray dataset. This uses *time_var_definition.get_input_name()* as the dataset key for the time variable and additionally uses the input's *Converter* object if applicable.

Parameters

- **raw_ds** (*xr.Dataset*) – A raw dataset (not standardized)
- **time_var_definition** (*VariableDefinition*) – The 'time' variable definition from the pipeline config

Returns A tuple of strings representing the first time data point in the dataset. The first string is the day in 'yyyymmdd' format. The second string is the time in 'hhmmss' format.

Return type Tuple[str, str]

static get_coordinate_variable_names (*ds: xarray.Dataset*) → List[str]

Get a list of all coordinate variables in this dataset.

Parameters *ds* (*xr.Dataset*) – The dataset

Returns List of coordinate variable names

Return type List[str]

static get_start_time (*ds: xarray.Dataset*) → Tuple[str, str]

Convenience method to get the start date and time from a xarray dataset.

Parameters *ds* (*xr.Dataset*) – A standardized dataset

Returns A tuple of strings representing the first time data point in the dataset. The first string is the day in 'yyyymmdd' format. The second string is the time in 'hhmmss' format.

Return type Tuple[str, str]

static get_metadata (*ds: xarray.Dataset*) → Dict

Get a dictionary of all global and variable attributes in a dataset. Global atts are found under the 'attributes' key and variable atts are found under the 'variables' key.

Parameters *ds* (*xr.Dataset*) – A dataset

Returns A dictionary of global & variable attributes

Return type Dict

static plot_qc (*ds: xarray.Dataset, variable_name: str, filename: str = None, **kwargs*) → act.plotting.TimeSeriesDisplay

Create a QC plot for the given variable. This is based on the ACT library: https://arm-doe.github.io/ACT/source/auto_examples/plot_qc.html#sphx-glr-source-auto-examples-plot-qc-py

We provide a convenience wrapper method for basic QC plots of a variable, but we recommend to use ACT directly and look at their examples for more complex plots like plotting variables in two different datasets.

TODO: Depending on use cases, we will likely add more arguments to be able to quickly produce the most common types of QC plots.

Parameters

- **ds** (*xr.Dataset*) – A dataset
- **variable_name** (*str*) – The variable to plot
- **filename** (*str, optional*) – The filename for the image. Saves the plot as this filename if provided.

static get_plot_filename (*dataset: xarray.Dataset, plot_description: str, extension: str*) → str

Returns the filename for a plot according to MHKIT-Cloud Data standards. The dataset is used to determine the *datastream_name* and start date/time. The standards dictate that a plot filename should follow the format: *datastream_name.date.time.description.extension*.

Parameters

- **dataset** (*xr.Dataset*) – The dataset from which the plot data is drawn from. This is used to collect the *datastream_name* and start date/time.

- **plot_description** (*str*) – The description of the plot. Should be as brief as possible and contain no spaces. Underscores may be used.
- **extension** (*str*) – The file extension for the plot.

Returns The standardized plot filename.

Return type *str*

static get_dataset_filename (*dataset: xarray.Dataset, file_extension='nc'*) → *str*

Given an xarray dataset this function will return the base filename of the dataset according to MHKIT-Cloud data standards. The base filename does not include the directory structure where the file should be saved, only the name of the file itself, e.g. z05.ExampleBuoyDatastream.b1.20201230.000000.nc

Parameters

- **dataset** (*xr.Dataset*) – The dataset whose filename should be generated.
- **file_extension** (*str, optional*) – The file extension to use. Defaults to “.nc”

Returns The base filename of the dataset.

Return type *str*

static get_raw_filename (*raw_dataset: xarray.Dataset, old_filename: str, config*) → *str*

Returns the appropriate raw filename of the raw dataset according to MHKIT-Cloud naming conventions. Uses the config object to parse the start date and time from the raw dataset for use in the new filename.

The new filename will follow the MHKIT-Cloud Data standards for raw filenames, ie: *datastream_name.date.time.raw.old_filename*, where the data level used in the *datastream_name* is *00*.

Parameters

- **raw_dataset** (*xr.Dataset*) – The raw data as an xarray dataset.
- **old_filename** (*str*) – The name of the original raw file.
- **config** (*Config*) – The Config object used to assist reading time data from the *raw_dataset*.

Returns The standardized filename of the raw file.

Return type *str*

static get_date_from_filename (*filename: str*) → *str*

Given a filename that conforms to MHKIT-Cloud Data Standards, return the date of the first point of data in the file.

Parameters **filename** (*str*) – The filename or path to the file.

Returns The date, in “yyyymmdd.hhmmss” format.

Return type *str*

static get_datastream_name_from_filename (*filename: str*) → *Optional[str]*

Given a filename that conforms to MHKIT-Cloud Data Standards, return the datastream name. Datastream name is everything to the left of the third ‘.’ in the filename.

e.g., humboldt_ca.buoy_data.b1.20210120.000000.nc

Parameters **filename** (*str*) – The filename or path to the file.

Returns The datastream name, or None if filename is not in proper format.

Return type *Optional[str]*

static `get_datastream_directory` (*datastream_name: str, root: str = ""*) → *str*

Given the `datastream_name` and an optional `root`, returns the path to where the datastream should be located. Does NOT create the directory where the datastream should be located.

Parameters

- **datastream_name** (*str*) – The name of the datastream whose directory path should be generated.
- **root** (*str, optional*) – The directory to use as the root of the directory structure. Defaults to `None`. Defaults to ""

Returns The path to the directory where the datastream should be located.

Return type *str*

static `is_image` (*filename: str*) → *bool*

Detect the mimetype from the file extension and use it to determine if the file is an image or not

Parameters **filename** (*str*) – The name of the file to check

Returns True if the file extension matches an image mimetype

Return type *bool*

class `tsdat.Converter` (*parameters: Union[Dict, None] = None*)

Bases: `abc.ABC`

Base class for converting data arrays from one units to another. Users can extend this class if they have a special units conversion for their input data that cannot be resolved with the default converter classes.

Parameters **parameters** (*dict, optional*) – A dictionary of converter-specific parameters which get passed from the pipeline config file. Defaults to {}

abstract `run` (*self, data: numpy.ndarray, in_units: str, out_units: str*) → *numpy.ndarray*

Convert the input data from `in_units` to `out_units`.

Parameters

- **data** (*np.ndarray*) – Data array to be modified.
- **in_units** (*str*) – Current units of the data array.
- **out_units** (*str*) – Units to be converted to.

Returns Data array converted into the new units.

Return type *np.ndarray*

class `tsdat.DefaultConverter` (*parameters: Union[Dict, None] = None*)

Bases: `Converter`

Default class for converting units on data arrays. This class utilizes `ACT.utils.data_utils.convert_units`, and should work for most variables except time (see `StringTimeConverter` and `TimestampTimeConverter`)

run (*self, data: numpy.ndarray, in_units: str, out_units: str*) → *numpy.ndarray*

Convert the input data from `in_units` to `out_units`.

Parameters

- **data** (*np.ndarray*) – Data array to be modified.
- **in_units** (*str*) – Current units of the data array.
- **out_units** (*str*) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

class tsdat.StringTimeConverter (parameters: Union[Dict, None] = None)

Bases: Converter

Convert a time string to a np.datetime64, which is needed for xarray. This class utilizes pd.to_datetime to perform the conversion.

One of the parameters should be 'time_format', which is the the strftime to parse time, eg "%d/%m/%Y". Note that "%f" will parse all the way up to nanoseconds. See strftime documentation for more information on choices.

Parameters **parameters** (dict, optional) – dictionary of converter-specific parameters.
Defaults to {}.

run (self, data: numpy.ndarray, in_units: str, out_units: str) → numpy.ndarray
Convert the input data from in_units to out_units.

Parameters

- **data** (np.ndarray) – Data array to be modified.
- **in_units** (str) – Current units of the data array.
- **out_units** (str) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

class tsdat.TimestampTimeConverter (parameters: Union[Dict, None] = None)

Bases: Converter

Convert a numeric UTC timestamp to a np.datetime64, which is needed for xarray. This class utilizes pd.to_datetime to perform the conversion.

One of the parameters should be 'unit'. This parameter denotes the time unit (e.g., D,s,ms,us,ns), which is an integer or float number. The timestamp will be based off the unix epoch start.

Parameters **parameters** (dict, optional) – A dictionary of converter-specific parameters
which get passed from the pipeline config file. Defaults to {}

run (self, data: numpy.ndarray, in_units: str, out_units: str) → numpy.ndarray
Convert the input data from in_units to out_units.

Parameters

- **data** (np.ndarray) – Data array to be modified.
- **in_units** (str) – Current units of the data array.
- **out_units** (str) – Units to be converted to.

Returns Data array converted into the new units.

Return type np.ndarray

COLLABORATION

tsdat is an open-source project that is still in its infancy. We enthusiastically welcome any feedback that helps us track down bugs or identify improvements. We also welcome community contributions in the form of new File Handlers, Quality Checkers, Quality Handlers, Converters, and Pipeline definitions.

8.1 Issues

Questions, feature requests, and bug reports for tsdat should be submitted to the GitHub Issues Page. The GitHub online forums are managed by the tsdat development team and users.

8.1.1 Submit tsdat Issue

8.2 Contributing

Software developers interested in contributing to the tsdat open-source software are encouraged to use GitHub to create a [Fork](#) of the repository into their GitHub user account. To include your additions to the tsdat code, please submit a [pull request](#) of the modified repository. Once reviewed by the tsdat development team, pull requests will be merged into the tsdat master branch, and included in future releases. Software developers - both within the tsdat development team and external collaborators - are expected to follow standard practices to document and test new code.

8.2.1 Submit tsdat Pull Request

ACKNOWLEDGEMENTS

tsdat was developed by Carina Lansing¹ and Maxwell Levin¹ with support and management from Chitra Sivaraman¹ and funding from the United States Water Power Technologies Office within the Department of Energy's Office of Energy Efficiency and Renewable Energy. We would like to thank Rebecca Fao², Calum Kenny², Raghavendra Krishnamurthy¹, Yangchao (Nino) Lin¹, and Eddie Schuman¹ for their feedback, testing, and support early on during the development of tsdat.

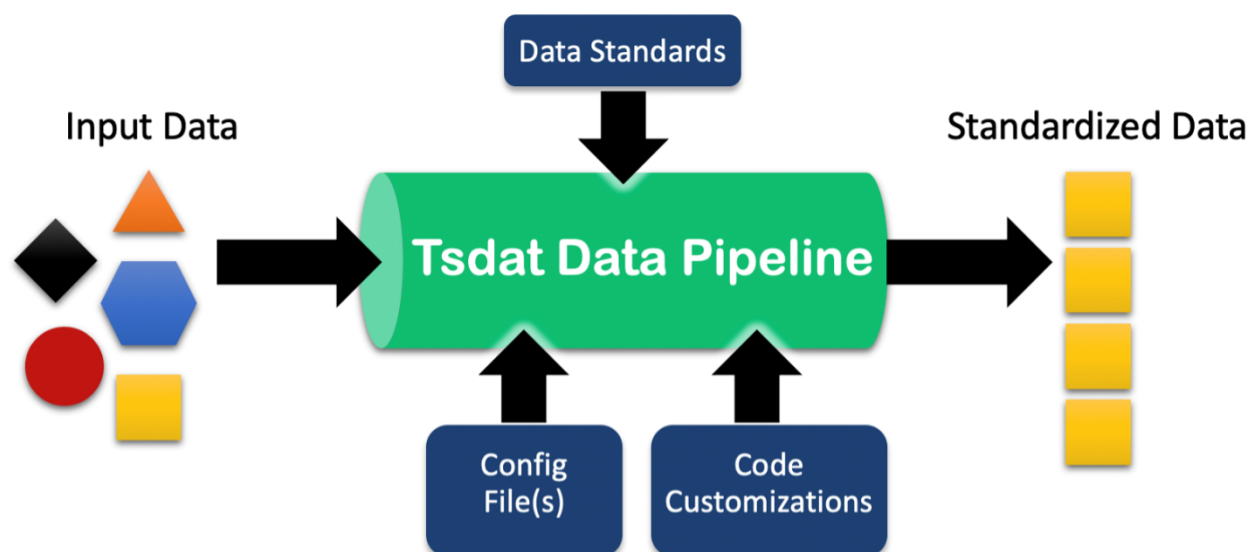
¹ Pacific Northwest National Laboratory

² National Renewable Energy Laboratory

tsdat is an open-source Python framework that makes creating pipelines to process and standardize time-series data more easy, clear, and quick to stand up so that you can spend less time data-wrangling and more time data- investigating.

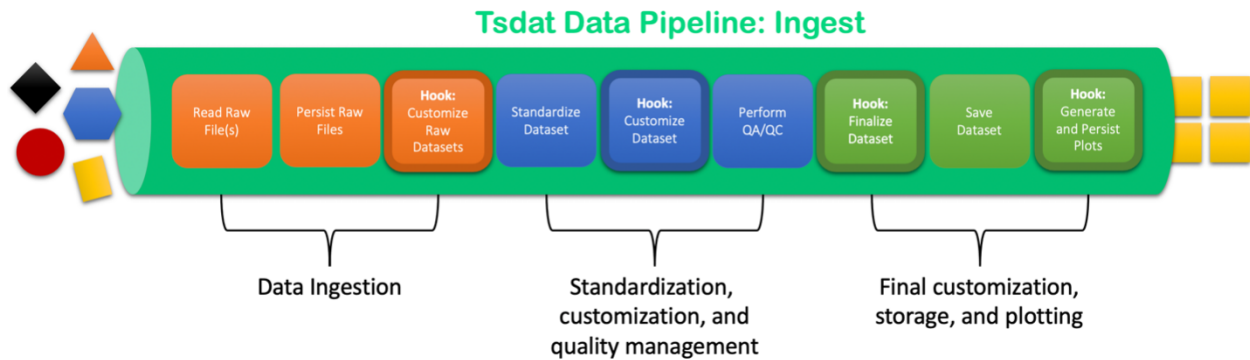
10.1 Quick Overview

Tsdat is a python library for standardizing time-series datasets. It uses yaml configuration files to specify the variable names and metadata that will be produced by tsdat data pipelines.



Tsdat data pipelines are primarily customizable through the aforementioned configuration files and also through user-defined code “hooks” that are triggered at various points in the pipeline.

Tsdat is built on top of [Xarray](#) and the [netCDF](#) file format frequently used in the Climate Science community. Tsdat was originally written for use in the Marine Energy community and was developed with data standards and best practices borrowed from the [ARM program](#), but the library and framework itself is applicable to any domain in which large datasets are collected.



10.2 Motivation

Too many datasets are difficult to use because the information needed to understand the data are buried away in technical reports and loose documentation that are often difficult to access and are not well-maintained. Even when you are able to get your hands on both the dataset and the metadata you need to understand the data, it can still be tricky to write code that reads each data file and handles edge cases. Additionally, as you process more and more datasets it can become cumbersome to keep track of and maintain all of the code you have written to process each of these datasets.

Wouldn't it just be much easier if all the data you worked with was in the same file format and had the same file structure? Wouldn't it take less time to learn about the dataset if each data file also contained the metadata you needed in order to conduct your analysis? Wouldn't it be nice if the data you worked with had been checked for quality and values that were suspect or bad had been flagged? That would all be great, right? This is the goal of tsdat, an open-source python library that aims to make it easier to produce high-quality datasets that are much more accessible to data users. Tsdat encourages following data standards and best practices when building data pipelines so that your data is clean, easy to understand, more accessible, and ultimately more valuable to your data users.

PYTHON MODULE INDEX

t

- [tsdat](#), 39
 - [tsdat.config](#), 39
 - [tsdat.config.config](#), 39
 - [tsdat.config.dataset_definition](#), 40
 - [tsdat.config.dimension_definition](#), 42
 - [tsdat.config.keys](#), 43
 - [tsdat.config.pipeline_definition](#), 43
 - [tsdat.config.quality_manager_definition](#), 44
 - [tsdat.config.utils](#), 45
 - [tsdat.config.variable_definition](#), 46
 - [tsdat.constants](#), 57
 - [tsdat.constants.constants](#), 57
 - [tsdat.exceptions](#), 60
 - [tsdat.exceptions.exceptions](#), 60
 - [tsdat.io](#), 60
 - [tsdat.io.aws_storage](#), 68
 - [tsdat.io.filehandlers](#), 61
 - [tsdat.io.filehandlers.csv_handler](#), 61
 - [tsdat.io.filehandlers.file_handlers](#), 62
 - [tsdat.io.filehandlers.netcdf_handler](#), 64
 - [tsdat.io.filesystem_storage](#), 73
 - [tsdat.io.storage](#), 76
 - [tsdat.pipeline](#), 95
 - [tsdat.pipeline.ingest_pipeline](#), 95
 - [tsdat.pipeline.pipeline](#), 97
 - [tsdat.qc](#), 104
 - [tsdat.qc.checkers](#), 104
 - [tsdat.qc.handlers](#), 108
 - [tsdat.qc.qc](#), 111
 - [tsdat.utils](#), 120
 - [tsdat.utils.converters](#), 120
 - [tsdat.utils.dsutils](#), 122

Symbols

<code>__enter__()</code>	(<i>tsdat.io.DisposableLocalTempFile</i> method), 90	<code>__parse_attributes()</code>	(<i>tsdat.config.VariableDefinition</i> method), 52
<code>__enter__()</code>	(<i>tsdat.io.DisposableLocalTempFileList</i> method), 90	<code>__parse_attributes()</code>	(<i>tsdat.config.variable_definition.VariableDefinition</i> method), 47
<code>__enter__()</code>	(<i>tsdat.io.DisposableStorageTempFileList</i> method), 90	<code>__parse_coordinates()</code>	(<i>tsdat.DatasetDefinition</i> method), 134
<code>__enter__()</code>	(<i>tsdat.io.storage.DisposableLocalTempFile</i> method), 79	<code>__parse_coordinates()</code>	(<i>tsdat.config.DatasetDefinition</i> method), 56
<code>__enter__()</code>	(<i>tsdat.io.storage.DisposableLocalTempFileList</i> method), 79	<code>__parse_coordinates()</code>	(<i>tsdat.config.dataset_definition.DatasetDefinition</i> method), 41
<code>__enter__()</code>	(<i>tsdat.io.storage.DisposableStorageTempFileList</i> method), 80	<code>__parse_data_type()</code>	(<i>tsdat.VariableDefinition</i> method), 137
<code>__exit__()</code>	(<i>tsdat.io.DisposableLocalTempFile</i> method), 90	<code>__parse_data_type()</code>	(<i>tsdat.config.VariableDefinition</i> method), 53
<code>__exit__()</code>	(<i>tsdat.io.DisposableLocalTempFileList</i> method), 90	<code>__parse_data_type()</code>	(<i>tsdat.config.variable_definition.VariableDefinition</i> method), 48
<code>__exit__()</code>	(<i>tsdat.io.DisposableStorageTempFileList</i> method), 90	<code>__parse_dimensions()</code>	(<i>tsdat.DatasetDefinition</i> method), 133
<code>__exit__()</code>	(<i>tsdat.io.storage.DisposableLocalTempFile</i> method), 79	<code>__parse_dimensions()</code>	(<i>tsdat.VariableDefinition</i> method), 136
<code>__exit__()</code>	(<i>tsdat.io.storage.DisposableLocalTempFileList</i> method), 79	<code>__parse_dimensions()</code>	(<i>tsdat.config.DatasetDefinition</i> method), 55
<code>__exit__()</code>	(<i>tsdat.io.storage.DisposableStorageTempFileList</i> method), 80	<code>__parse_dimensions()</code>	(<i>tsdat.config.VariableDefinition</i> method), 53
<code>__str__()</code>	(<i>tsdat.S3Path</i> method), 145	<code>__parse_dimensions()</code>	(<i>tsdat.config.dataset_definition.DatasetDefinition</i> method), 40
<code>__str__()</code>	(<i>tsdat.io.S3Path</i> method), 95	<code>__parse_dimensions()</code>	(<i>tsdat.config.variable_definition.VariableDefinition</i> method), 47
<code>__str__()</code>	(<i>tsdat.io.aws_storage.S3Path</i> method), 69	<code>__parse_fully_qualified_name()</code>	(in module <i>tsdat.config.utils</i>), 45
<code>_get_handler()</code>	(<i>tsdat.FileHandler</i> static method), 148	<code>__parse_input()</code>	(<i>tsdat.VariableDefinition</i> method), 136
<code>_get_handler()</code>	(<i>tsdat.io.FileHandler</i> static method), 83	<code>__parse_input()</code>	(<i>tsdat.config.VariableDefinition</i> method), 52
<code>_get_handler()</code>	(<i>tsdat.io.filehandlers.FileHandler</i> static method), 65	<code>__parse_input()</code>	(<i>tsdat.config.variable_definition.VariableDefinition</i> method), 47
<code>_get_handler()</code>	(<i>tsdat.io.filehandlers.file_handlers.FileHandler</i> static method), 63		
<code>_instantiate_class()</code>	(in module <i>tsdat.config.utils</i>), 45		
<code>_is_image()</code>	(in module <i>tsdat.io.storage</i>), 76		
<code>_is_raw()</code>	(in module <i>tsdat.io.storage</i>), 76		
<code>_parse_attributes()</code>	(<i>tsdat.VariableDefinition</i> method), 136		

method), 47
 _parse_quality_managers() (tsdat.Config method), 133
 _parse_quality_managers() (tsdat.config.Config method), 51
 _parse_quality_managers() (tsdat.config.config.Config method), 39
 _parse_variables() (tsdat.DatasetDefinition method), 134
 _parse_variables() (tsdat.config.DatasetDefinition method), 56
 _parse_variables() (tsdat.config.dataset_definition.DatasetDefinition method), 40
 _validate_dataset_definition() (tsdat.DatasetDefinition method), 134
 _validate_dataset_definition() (tsdat.config.DatasetDefinition method), 56
 _validate_dataset_definition() (tsdat.config.dataset_definition.DatasetDefinition method), 41

A

AbstractFileHandler (class in tsdat), 147
 AbstractFileHandler (class in tsdat.io), 82
 AbstractFileHandler (class in tsdat.io.filehandlers), 65
 AbstractFileHandler (class in tsdat.io.filehandlers.file_handlers), 62
 add_attrs() (tsdat.Pipeline method), 151
 add_attrs() (tsdat.pipeline.Pipeline method), 101
 add_attrs() (tsdat.pipeline.pipeline.Pipeline method), 98
 add_fillvalue_if_none() (tsdat.config.variable_definition.VariableDefinition method), 48
 add_fillvalue_if_none() (tsdat.config.VariableDefinition method), 53
 add_fillvalue_if_none() (tsdat.config.VariableDefinition method), 137
 add_missing_variables() (tsdat.Pipeline method), 151
 add_missing_variables() (tsdat.pipeline.Pipeline method), 101
 add_missing_variables() (tsdat.pipeline.pipeline.Pipeline method), 98
 add_static_variables() (tsdat.Pipeline method), 150
 add_static_variables() (tsdat.pipeline.Pipeline method), 101
 add_static_variables() (tsdat.pipeline.pipeline.Pipeline method), 98
 ALL (tsdat.config.Keys attribute), 51
 ALL (tsdat.config.keys.Keys attribute), 43

ALL (tsdat.constants.constants.VARS attribute), 58
 ALL (tsdat.constants.VARS attribute), 59
 ALL (tsdat.VARS attribute), 140
 ASSESSMENT (tsdat.qc.handlers.QCParamKeys attribute), 109
 ASSESSMENT (tsdat.qc.QCParamKeys attribute), 118
 ATTRIBUTES (tsdat.config.Keys attribute), 51
 ATTRIBUTES (tsdat.config.keys.Keys attribute), 43
 ATTRS (tsdat.config.variable_definition.VarKeys attribute), 46
 ATTS (class in tsdat), 139
 ATTS (class in tsdat.constants), 59
 ATTS (class in tsdat.constants.constants), 58
 AVERAGING_INTERVAL (tsdat.ATTS attribute), 140
 AVERAGING_INTERVAL (tsdat.constants.ATTS attribute), 59
 AVERAGING_INTERVAL (tsdat.constants.constants.ATTS attribute), 58
 AwsStorage (class in tsdat), 142
 AwsStorage (class in tsdat.io), 92
 AwsStorage (class in tsdat.io.aws_storage), 70
 AwsTemporaryStorage (class in tsdat.io.aws_storage), 69

B

base_path() (tsdat.io.aws_storage.AwsTemporaryStorage property), 69
 bucket_name() (tsdat.io.aws_storage.S3Path property), 69
 bucket_name() (tsdat.io.S3Path property), 95
 bucket_name() (tsdat.S3Path property), 145
 bucket_path() (tsdat.io.aws_storage.S3Path property), 69
 bucket_path() (tsdat.io.S3Path property), 95
 bucket_path() (tsdat.S3Path property), 145

C

check_file_name_components() (tsdat.config.pipeline_definition.PipelineDefinition method), 44
 check_file_name_components() (tsdat.config.PipelineDefinition method), 52
 check_file_name_components() (tsdat.PipelineDefinition method), 133
 check_required_variables() (tsdat.Pipeline method), 150
 check_required_variables() (tsdat.pipeline.Pipeline method), 100
 check_required_variables() (tsdat.pipeline.pipeline.Pipeline method), 98
 CHECKER (tsdat.config.quality_manager_definition.QualityManagerKeys attribute), 44
 CheckFailMax (class in tsdat.qc), 114

- CheckFailMax (class in *tsdat.qc.checkers*), 107
 CheckFailMin (class in *tsdat.qc*), 115
 CheckFailMin (class in *tsdat.qc.checkers*), 107
 CheckMax (class in *tsdat.qc*), 115
 CheckMax (class in *tsdat.qc.checkers*), 106
 CheckMin (class in *tsdat.qc*), 115
 CheckMin (class in *tsdat.qc.checkers*), 105
 CheckMissing (class in *tsdat.qc*), 116
 CheckMissing (class in *tsdat.qc.checkers*), 105
 CheckMonotonic (class in *tsdat.qc*), 116
 CheckMonotonic (class in *tsdat.qc.checkers*), 108
 CheckValidDelta (class in *tsdat.qc*), 117
 CheckValidDelta (class in *tsdat.qc.checkers*), 107
 CheckValidMax (class in *tsdat.qc*), 117
 CheckValidMax (class in *tsdat.qc.checkers*), 107
 CheckValidMin (class in *tsdat.qc*), 117
 CheckValidMin (class in *tsdat.qc.checkers*), 107
 CheckWarnMax (class in *tsdat.qc*), 114
 CheckWarnMax (class in *tsdat.qc.checkers*), 107
 CheckWarnMin (class in *tsdat.qc*), 117
 CheckWarnMin (class in *tsdat.qc.checkers*), 107
 CLASSNAME (*tsdat.config.variable_definition.ConverterKeys* attribute), 46
 clean() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 69
 clean() (*tsdat.io.storage.TemporaryStorage* method), 80
 clean() (*tsdat.io.TemporaryStorage* method), 88
 CODE_URL (*tsdat.ATTS* attribute), 139
 CODE_URL (*tsdat.constants.ATTS* attribute), 59
 CODE_URL (*tsdat.constants.constants.ATTS* attribute), 58
 Config (class in *tsdat*), 133
 Config (class in *tsdat.config*), 50
 Config (class in *tsdat.config.config*), 39
 configure_yaml() (in module *tsdat.config.utils*), 45
 CONVENTIONS (*tsdat.ATTS* attribute), 139
 CONVENTIONS (*tsdat.constants.ATTS* attribute), 59
 CONVENTIONS (*tsdat.constants.constants.ATTS* attribute), 58
 Converter (class in *tsdat*), 159
 Converter (class in *tsdat.utils*), 130
 Converter (class in *tsdat.utils.converters*), 120
 CONVERTER (*tsdat.config.variable_definition.VarInputKeys* attribute), 46
 ConverterKeys (class in *tsdat.config.variable_definition*), 46
 COORDS (*tsdat.constants.constants.VARS* attribute), 58
 COORDS (*tsdat.constants.VARS* attribute), 60
 COORDS (*tsdat.VARS* attribute), 140
 CORRECTION (*tsdat.qc.handlers.QCParamKeys* attribute), 109
 CORRECTION (*tsdat.qc.QCParamKeys* attribute), 118
 CORRECTIONS_APPLIED (*tsdat.ATTS* attribute), 140
 CORRECTIONS_APPLIED (*tsdat.constants.ATTS* attribute), 59
 CORRECTIONS_APPLIED (*tsdat.constants.constants.ATTS* attribute), 58
 create_temp_dir() (*tsdat.io.storage.TemporaryStorage* method), 81
 create_temp_dir() (*tsdat.io.TemporaryStorage* method), 89
 CsvHandler (class in *tsdat*), 148
 CsvHandler (class in *tsdat.io*), 84
 CsvHandler (class in *tsdat.io.filehandlers*), 66
 CsvHandler (class in *tsdat.io.filehandlers.csv_handler*), 61
- ## D
- DATA_LEVEL (*tsdat.ATTS* attribute), 139
 DATA_LEVEL (*tsdat.constants.ATTS* attribute), 59
 DATA_LEVEL (*tsdat.constants.constants.ATTS* attribute), 58
 DATA_VARS (*tsdat.constants.constants.VARS* attribute), 58
 DATA_VARS (*tsdat.constants.VARS* attribute), 60
 DATA_VARS (*tsdat.VARS* attribute), 140
 DATASET_DEFINITION (*tsdat.config.Keys* attribute), 51
 DATASET_DEFINITION (*tsdat.config.keys.Keys* attribute), 43
 DATASET_NAME (*tsdat.config.pipeline_definition.PipelineKeys* attribute), 43
 DatasetDefinition (class in *tsdat*), 133
 DatasetDefinition (class in *tsdat.config*), 55
 DatasetDefinition (class in *tsdat.config.dataset_definition*), 40
 DATASTREAM (*tsdat.ATTS* attribute), 139
 DATASTREAM (*tsdat.constants.ATTS* attribute), 59
 DATASTREAM (*tsdat.constants.constants.ATTS* attribute), 58
 DatastreamStorage (class in *tsdat*), 140
 DatastreamStorage (class in *tsdat.io*), 85
 DatastreamStorage (class in *tsdat.io.storage*), 76
 datetime64_to_string() (*tsdat.DSUtil* static method), 155
 datetime64_to_string() (*tsdat.utils.DSUtil* static method), 126
 datetime64_to_string() (*tsdat.utils.dsutils.DSUtil* static method), 122
 datetime64_to_timestamp() (*tsdat.DSUtil* static method), 155
 datetime64_to_timestamp() (*tsdat.utils.DSUtil* static method), 126
 datetime64_to_timestamp() (*tsdat.utils.dsutils.DSUtil* static method), 122

- default_file_type (*tsdat.DatastreamStorage* attribute), 140
- default_file_type (*tsdat.io.DatastreamStorage* attribute), 85
- default_file_type (*tsdat.io.storage.DatastreamStorage* attribute), 77
- DefaultConverter (*class in tsdat*), 159
- DefaultConverter (*class in tsdat.utils*), 130
- DefaultConverter (*class in tsdat.utils.converters*), 120
- DEFAULTS (*tsdat.config.Keys* attribute), 51
- DEFAULTS (*tsdat.config.keys.Keys* attribute), 43
- DefinitionError, 60, 140
- delete() (*tsdat.AwsStorage* method), 144
- delete() (*tsdat.DatastreamStorage* method), 142
- delete() (*tsdat.FilesystemStorage* method), 147
- delete() (*tsdat.io.aws_storage.AwsStorage* method), 72
- delete() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 70
- delete() (*tsdat.io.AwsStorage* method), 94
- delete() (*tsdat.io.DatastreamStorage* method), 87
- delete() (*tsdat.io.filesystem_storage.FilesystemStorage* method), 76
- delete() (*tsdat.io.filesystem_storage.FilesystemTemporaryStorage* method), 74
- delete() (*tsdat.io.FilesystemStorage* method), 92
- delete() (*tsdat.io.storage.DatastreamStorage* method), 79
- delete() (*tsdat.io.storage.TemporaryStorage* method), 82
- delete() (*tsdat.io.TemporaryStorage* method), 89
- DESCRIPTION (*tsdat.ATTS* attribute), 139
- DESCRIPTION (*tsdat.constants.ATTS* attribute), 59
- DESCRIPTION (*tsdat.constants.constants.ATTS* attribute), 58
- DimensionDefinition (*class in tsdat*), 135
- DimensionDefinition (*class in tsdat.config*), 51
- DimensionDefinition (*class in tsdat.config.dimension_definition*), 42
- DIMENSIONS (*tsdat.config.Keys* attribute), 51
- DIMENSIONS (*tsdat.config.keys.Keys* attribute), 43
- DimKeys (*class in tsdat.config.dimension_definition*), 42
- DIMS (*tsdat.config.variable_definition.VarKeys* attribute), 46
- DisposableLocalTempFile (*class in tsdat.io*), 90
- DisposableLocalTempFile (*class in tsdat.io.storage*), 79
- DisposableLocalTempFileList (*class in tsdat.io*), 90
- DisposableLocalTempFileList (*class in tsdat.io.storage*), 79
- DisposableStorageTempFileList (*class in tsdat.io*), 90
- DisposableStorageTempFileList (*class in tsdat.io.storage*), 79
- DOI (*tsdat.ATTS* attribute), 139
- DOI (*tsdat.constants.ATTS* attribute), 59
- DOI (*tsdat.constants.constants.ATTS* attribute), 58
- DSUtil (*class in tsdat*), 155
- DSUtil (*class in tsdat.utils*), 126
- DSUtil (*class in tsdat.utils.dsutils*), 122
- ## E
- EXCLUDE (*tsdat.config.quality_manager_definition.QualityManagerKeys* attribute), 44
- exists() (*tsdat.AwsStorage* method), 144
- exists() (*tsdat.DatastreamStorage* method), 142
- exists() (*tsdat.FilesystemStorage* method), 147
- exists() (*tsdat.io.aws_storage.AwsStorage* method), 72
- exists() (*tsdat.io.AwsStorage* method), 94
- exists() (*tsdat.io.DatastreamStorage* method), 87
- exists() (*tsdat.io.filesystem_storage.FilesystemStorage* method), 75
- exists() (*tsdat.io.FilesystemStorage* method), 92
- exists() (*tsdat.io.storage.DatastreamStorage* method), 78
- extract_files() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 69
- extract_files() (*tsdat.io.filesystem_storage.FilesystemTemporaryStorage* method), 73
- extract_files() (*tsdat.io.storage.TemporaryStorage* method), 81
- extract_files() (*tsdat.io.TemporaryStorage* method), 89
- extract_tarfile() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 69
- extract_zipfile() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 69
- ## F
- FAIL_RANGE (*tsdat.ATTS* attribute), 140
- FAIL_RANGE (*tsdat.constants.ATTS* attribute), 59
- FAIL_RANGE (*tsdat.constants.constants.ATTS* attribute), 58
- FailPipeline (*class in tsdat.qc*), 118
- FailPipeline (*class in tsdat.qc.handlers*), 111
- fetch() (*tsdat.AwsStorage* method), 143
- fetch() (*tsdat.DatastreamStorage* method), 141
- fetch() (*tsdat.FilesystemStorage* method), 146

fetch() (*tsdat.io.aws_storage.AwsStorage* method), 71
 fetch() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 70
 fetch() (*tsdat.io.AwsStorage* method), 93
 fetch() (*tsdat.io.DatastreamStorage* method), 86
 fetch() (*tsdat.io.filesystem_storage.FilesystemStorage* method), 75
 fetch() (*tsdat.io.filesystem_storage.FilesystemTemporaryStorage* method), 73
 fetch() (*tsdat.io.FilesystemStorage* method), 91
 fetch() (*tsdat.io.storage.DatastreamStorage* method), 77
 fetch() (*tsdat.io.storage.TemporaryStorage* method), 81
 fetch() (*tsdat.io.TemporaryStorage* method), 89
 fetch_previous_file() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 70
 fetch_previous_file() (*tsdat.io.filesystem_storage.FilesystemTemporaryStorage* method), 73
 fetch_previous_file() (*tsdat.io.storage.TemporaryStorage* method), 81
 fetch_previous_file() (*tsdat.io.TemporaryStorage* method), 89
 file_filters (*tsdat.DatastreamStorage* attribute), 140
 file_filters (*tsdat.io.DatastreamStorage* attribute), 85
 file_filters (*tsdat.io.storage.DatastreamStorage* attribute), 77
 FileHandler (class in *tsdat*), 148
 FileHandler (class in *tsdat.io*), 83
 FileHandler (class in *tsdat.io.filehandlers*), 65
 FileHandler (class in *tsdat.io.filehandlers.file_handlers*), 62
 FILEREADERS (*tsdat.FileHandler* attribute), 148
 FILEREADERS (*tsdat.io.FileHandler* attribute), 83
 FILEREADERS (*tsdat.io.filehandlers.file_handlers.FileHandler* attribute), 62
 FILEREADERS (*tsdat.io.filehandlers.FileHandler* attribute), 65
 FilesystemStorage (class in *tsdat*), 145
 FilesystemStorage (class in *tsdat.io*), 90
 FilesystemStorage (class in *tsdat.io.filesystem_storage*), 74
 FilesystemTemporaryStorage (class in *tsdat.io.filesystem_storage*), 73
 FILEWRITERS (*tsdat.FileHandler* attribute), 148
 FILEWRITERS (*tsdat.io.FileHandler* attribute), 83
 FILEWRITERS (*tsdat.io.filehandlers.file_handlers.FileHandler* attribute), 62
 FILEWRITERS (*tsdat.io.filehandlers.FileHandler* attribute), 65
 attribute), 65
 FILL_VALUE (*tsdat.ATTS* attribute), 140
 FILL_VALUE (*tsdat.constants.ATTS* attribute), 59
 FILL_VALUE (*tsdat.constants.constants.ATTS* attribute), 58
 find() (*tsdat.AwsStorage* method), 143
 find() (*tsdat.DatastreamStorage* method), 141
 find() (*tsdat.FilesystemStorage* method), 145
 find() (*tsdat.io.aws_storage.AwsStorage* method), 71
 find() (*tsdat.io.AwsStorage* method), 93
 find() (*tsdat.io.DatastreamStorage* method), 86
 find() (*tsdat.io.filesystem_storage.FilesystemStorage* method), 74
 find() (*tsdat.io.FilesystemStorage* method), 91
 find() (*tsdat.io.storage.DatastreamStorage* method), 77
 from_config() (*tsdat.DatastreamStorage* static method), 140
 from_config() (*tsdat.io.DatastreamStorage* static method), 85
 from_config() (*tsdat.io.storage.DatastreamStorage* static method), 77

G

get_attr() (*tsdat.config.dataset_definition.DatasetDefinition* method), 41
 get_attr() (*tsdat.config.DatasetDefinition* method), 56
 get_attr() (*tsdat.DatasetDefinition* method), 134
 get_coordinate_names() (*tsdat.config.variable_definition.VariableDefinition* method), 49
 get_coordinate_names() (*tsdat.config.VariableDefinition* method), 54
 get_coordinate_names() (*tsdat.VariableDefinition* method), 138
 get_coordinate_variable_names() (*tsdat.DSUtil* static method), 157
 get_coordinate_variable_names() (*tsdat.utils.DSUtil* static method), 128
 get_coordinate_variable_names() (*tsdat.utils.dsutils.DSUtil* static method), 123
 get_coordinates() (*tsdat.config.dataset_definition.DatasetDefinition* method), 41
 get_coordinates() (*tsdat.config.DatasetDefinition* method), 57
 get_coordinates() (*tsdat.DatasetDefinition* method), 135
 get_data_type() (*tsdat.config.variable_definition.VariableDefinition* method), 49
 get_data_type() (*tsdat.config.VariableDefinition* method), 55

`get_data_type()` (*tsdat.VariableDefinition* method), 138
`get_dataset_filename()` (*tsdat.DSUtil* static method), 158
`get_dataset_filename()` (*tsdat.utils.DSUtil* static method), 129
`get_dataset_filename()` (*tsdat.utils.dsutils.DSUtil* static method), 124
`get_datastream_directory()` (*tsdat.DSUtil* static method), 158
`get_datastream_directory()` (*tsdat.utils.DSUtil* static method), 130
`get_datastream_directory()` (*tsdat.utils.dsutils.DSUtil* static method), 125
`get_datastream_name()` (*tsdat.DSUtil* static method), 155
`get_datastream_name()` (*tsdat.utils.DSUtil* static method), 126
`get_datastream_name()` (*tsdat.utils.dsutils.DSUtil* static method), 122
`get_datastream_name_from_filename()` (*tsdat.DSUtil* static method), 158
`get_datastream_name_from_filename()` (*tsdat.utils.DSUtil* static method), 129
`get_datastream_name_from_filename()` (*tsdat.utils.dsutils.DSUtil* static method), 125
`get_date_from_filename()` (*tsdat.DSUtil* static method), 158
`get_date_from_filename()` (*tsdat.utils.DSUtil* static method), 129
`get_date_from_filename()` (*tsdat.utils.dsutils.DSUtil* static method), 125
`get_end_time()` (*tsdat.DSUtil* static method), 156
`get_end_time()` (*tsdat.utils.DSUtil* static method), 127
`get_end_time()` (*tsdat.utils.dsutils.DSUtil* static method), 122
`get_fill_value()` (*tsdat.DSUtil* static method), 156
`get_fill_value()` (*tsdat.utils.DSUtil* static method), 127
`get_fill_value()` (*tsdat.utils.dsutils.DSUtil* static method), 123
`get_FillValue()` (*tsdat.config.variable_definition.VariableDefinition* method), 50
`get_FillValue()` (*tsdat.config.VariableDefinition* method), 55
`get_FillValue()` (*tsdat.VariableDefinition* method), 138
`get_input_name()` (*tsdat.config.variable_definition.VariableDefinition* method), 49
`get_input_name()` (*tsdat.config.VariableDefinition* method), 54
`get_input_name()` (*tsdat.VariableDefinition* method), 138
`get_input_units()` (*tsdat.config.variable_definition.VariableDefinition* method), 49
`get_input_units()` (*tsdat.config.VariableDefinition* method), 54
`get_input_units()` (*tsdat.VariableDefinition* method), 138
`get_metadata()` (*tsdat.DSUtil* static method), 157
`get_metadata()` (*tsdat.utils.DSUtil* static method), 128
`get_metadata()` (*tsdat.utils.dsutils.DSUtil* static method), 124
`get_non_qc_variable_names()` (*tsdat.DSUtil* static method), 156
`get_non_qc_variable_names()` (*tsdat.utils.DSUtil* static method), 127
`get_non_qc_variable_names()` (*tsdat.utils.dsutils.DSUtil* static method), 123
`get_output_units()` (*tsdat.config.variable_definition.VariableDefinition* method), 49
`get_output_units()` (*tsdat.config.VariableDefinition* method), 54
`get_output_units()` (*tsdat.VariableDefinition* method), 138
`get_plot_filename()` (*tsdat.DSUtil* static method), 157
`get_plot_filename()` (*tsdat.utils.DSUtil* static method), 128
`get_plot_filename()` (*tsdat.utils.dsutils.DSUtil* static method), 124
`get_previous_dataset()` (*tsdat.Pipeline* method), 151
`get_previous_dataset()` (*tsdat.pipeline.Pipeline* method), 101
`get_previous_dataset()` (*tsdat.pipeline.pipeline.Pipeline* method), 99
`get_raw_end_time()` (*tsdat.DSUtil* static method), 156
`get_raw_end_time()` (*tsdat.utils.DSUtil* static method), 127
`get_raw_end_time()` (*tsdat.utils.dsutils.DSUtil* static method), 123
`get_raw_filename()` (*tsdat.DSUtil* static method), 158
`get_raw_filename()` (*tsdat.utils.DSUtil* static method), 129
`get_raw_filename()` (*tsdat.utils.dsutils.DSUtil* static method), 125
`get_raw_start_time()` (*tsdat.DSUtil* static method), 156
`get_raw_start_time()` (*tsdat.utils.DSUtil* static

- method), 127
- get_raw_start_time() (tsdat.utils.dsutils.DSUtil static method), 123
- get_shape() (tsdat.config.variable_definition.VariableDefinition method), 49
- get_shape() (tsdat.config.VariableDefinition method), 55
- get_shape() (tsdat.VariableDefinition method), 138
- get_start_time() (tsdat.DSUtil static method), 157
- get_start_time() (tsdat.utils.DSUtil static method), 128
- get_start_time() (tsdat.utils.dsutils.DSUtil static method), 124
- get_static_variables() (tsdat.config.dataset_definition.DatasetDefinition method), 41
- get_static_variables() (tsdat.config.DatasetDefinition method), 57
- get_static_variables() (tsdat.DatasetDefinition method), 135
- get_temp_filepath() (tsdat.io.storage.TemporaryStorage method), 80
- get_temp_filepath() (tsdat.io.TemporaryStorage method), 88
- get_variable() (tsdat.config.dataset_definition.DatasetDefinition method), 41
- get_variable() (tsdat.config.DatasetDefinition method), 56
- get_variable() (tsdat.DatasetDefinition method), 134
- get_variable_names() (tsdat.config.dataset_definition.DatasetDefinition method), 41
- get_variable_names() (tsdat.config.DatasetDefinition method), 56
- get_variable_names() (tsdat.DatasetDefinition method), 134
- ## H
- HANDLERS (tsdat.config.quality_manager_definition.QualityManager attribute), 44
- has_converter() (tsdat.config.variable_definition.VariableDefinition method), 49
- has_converter() (tsdat.config.VariableDefinition method), 54
- has_converter() (tsdat.VariableDefinition method), 137
- has_input() (tsdat.config.variable_definition.VariableDefinition method), 49
- has_input() (tsdat.config.VariableDefinition method), 54
- has_input() (tsdat.VariableDefinition method), 138
- HISTORY (tsdat.ATTS attribute), 139
- HISTORY (tsdat.constants.ATTS attribute), 59
- HISTORY (tsdat.constants.constants.ATTS attribute), 58
- hook_customize_dataset() (tsdat.IngestPipeline method), 152
- hook_customize_dataset() (tsdat.pipeline.ingest_pipeline.IngestPipeline method), 96
- hook_customize_dataset() (tsdat.pipeline.IngestPipeline method), 102
- hook_customize_raw_datasets() (tsdat.IngestPipeline method), 152
- hook_customize_raw_datasets() (tsdat.pipeline.ingest_pipeline.IngestPipeline method), 96
- hook_customize_raw_datasets() (tsdat.pipeline.IngestPipeline method), 102
- hook_finalize_dataset() (tsdat.IngestPipeline method), 153
- hook_finalize_dataset() (tsdat.pipeline.ingest_pipeline.IngestPipeline method), 96
- hook_finalize_dataset() (tsdat.pipeline.IngestPipeline method), 103
- hook_generate_and_persist_plots() (tsdat.IngestPipeline method), 153
- hook_generate_and_persist_plots() (tsdat.pipeline.ingest_pipeline.IngestPipeline method), 97
- hook_generate_and_persist_plots() (tsdat.pipeline.IngestPipeline method), 103
- ## I
- ignore_zip_check() (tsdat.io.storage.TemporaryStorage method), 80
- ignore_zip_check() (tsdat.io.TemporaryStorage method), 88
- IngestPipeline (class in tsdat), 152
- IngestPipeline (class in tsdat.pipeline), 102
- IngestPipelineKeys (class in tsdat.pipeline.ingest_pipeline), 96
- INPUT (tsdat.config.variable_definition.VarKeys attribute), 46
- INPUT_DATA_LEVEL (tsdat.config.pipeline_definition.PipelineKeys attribute), 43
- INPUT_FILES (tsdat.ATTS attribute), 139
- INPUT_FILES (tsdat.constants.ATTS attribute), 59
- INPUT_FILES (tsdat.constants.constants.ATTS attribute), 58
- instantiate_handler() (in module tsdat.config.utils), 45

INSTITUTION (*tsdat.ATTS* attribute), 139
 INSTITUTION (*tsdat.constants.ATTS* attribute), 59
 INSTITUTION (*tsdat.constants.constants.ATTS* attribute), 58
 INSTRUMENT_DESCRIPTION (*tsdat.ATTS* attribute), 139
 INSTRUMENT_DESCRIPTION (*tsdat.constants.ATTS* attribute), 59
 INSTRUMENT_DESCRIPTION (*tsdat.constants.constants.ATTS* attribute), 58
 INSTRUMENT_MANUFACTURER (*tsdat.ATTS* attribute), 139
 INSTRUMENT_MANUFACTURER (*tsdat.constants.ATTS* attribute), 59
 INSTRUMENT_MANUFACTURER (*tsdat.constants.constants.ATTS* attribute), 58
 INSTRUMENT_NAME (*tsdat.ATTS* attribute), 139
 INSTRUMENT_NAME (*tsdat.constants.ATTS* attribute), 59
 INSTRUMENT_NAME (*tsdat.constants.constants.ATTS* attribute), 58
 is_constant() (*tsdat.config.variable_definition.VariableDefinition* method), 48
 is_constant() (*tsdat.config.VariableDefinition* method), 53
 is_constant() (*tsdat.VariableDefinition* method), 137
 is_coordinate() (*tsdat.config.variable_definition.VariableDefinition* method), 48
 is_coordinate() (*tsdat.config.VariableDefinition* method), 54
 is_coordinate() (*tsdat.VariableDefinition* method), 137
 is_derived() (*tsdat.config.variable_definition.VariableDefinition* method), 48
 is_derived() (*tsdat.config.VariableDefinition* method), 54
 is_derived() (*tsdat.VariableDefinition* method), 137
 is_image() (*tsdat.DSUtil* static method), 159
 is_image() (*tsdat.utils.DSUtil* static method), 130
 is_image() (*tsdat.utils.dsutils.DSUtil* static method), 125
 is_predefined() (*tsdat.config.variable_definition.VariableDefinition* method), 48
 is_predefined() (*tsdat.config.VariableDefinition* method), 53
 is_predefined() (*tsdat.VariableDefinition* method), 137
 is_required() (*tsdat.config.variable_definition.VariableDefinition* method), 49
 is_required() (*tsdat.config.variable_definition.VarInput* method), 46
 is_required() (*tsdat.config.VariableDefinition* method), 54
 is_required() (*tsdat.VariableDefinition* method), 138
 is_tarfile() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 69
 is_unlimited() (*tsdat.config.dimension_definition.DimensionDefinition* method), 42
 is_unlimited() (*tsdat.config.DimensionDefinition* method), 51
 is_unlimited() (*tsdat.DimensionDefinition* method), 135
 is_variable_length() (*tsdat.config.dimension_definition.DimensionDefinition* method), 42
 is_variable_length() (*tsdat.config.DimensionDefinition* method), 51
 is_variable_length() (*tsdat.DimensionDefinition* method), 135
 is_zipfile() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 69

J

join() (*tsdat.io.aws_storage.S3Path* method), 69
 join() (*tsdat.io.S3Path* method), 95
 join() (*tsdat.S3Path* method), 145

K

Keys (class in *tsdat.config*), 51
 Keys (class in *tsdat.config.keys*), 43

L

LENGTH (*tsdat.config.dimension_definition.DimKeys* attribute), 42
 lint_yaml() (*tsdat.Config* static method), 133
 lint_yaml() (*tsdat.config.Config* static method), 51
 lint_yaml() (*tsdat.config.config.Config* static method), 40
 listdir() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 70
 load() (*tsdat.Config* class method), 133
 load() (*tsdat.config.Config* class method), 51
 load() (*tsdat.config.config.Config* class method), 39
 local_temp_folder() (*tsdat.io.storage.TemporaryStorage* property), 80

local_temp_folder() (*tsdat.io.TemporaryStorage* property), 88
 LOCATION_DESCRIPTION (*tsdat.ATTS* attribute), 139
 LOCATION_DESCRIPTION (*tsdat.constants.ATTS* attribute), 59
 LOCATION_DESCRIPTION (*tsdat.constants.constants.ATTS* attribute), 58
 LOCATION_ID (*tsdat.ATTS* attribute), 139
 LOCATION_ID (*tsdat.config.pipeline_definition.PipelineKeys* attribute), 43
 LOCATION_ID (*tsdat.constants.ATTS* attribute), 59
 LOCATION_ID (*tsdat.constants.constants.ATTS* attribute), 58

M

module

tsdat, 39
 tsdat.config, 39
 tsdat.config.config, 39
 tsdat.config.dataset_definition, 40
 tsdat.config.dimension_definition, 42
 tsdat.config.keys, 43
 tsdat.config.pipeline_definition, 43
 tsdat.config.quality_manager_definition, 44
 tsdat.config.utils, 45
 tsdat.config.variable_definition, 46
 tsdat.constants, 57
 tsdat.constants.constants, 57
 tsdat.exceptions, 60
 tsdat.exceptions.exceptions, 60
 tsdat.io, 60
 tsdat.io.aws_storage, 68
 tsdat.io.filehandlers, 61
 tsdat.io.filehandlers.csv_handler, 61
 tsdat.io.filehandlers.file_handlers, 62
 tsdat.io.filehandlers.netcdf_handler, 64
 tsdat.io.filesystem_storage, 73
 tsdat.io.storage, 76
 tsdat.pipeline, 95
 tsdat.pipeline.ingest_pipeline, 95
 tsdat.pipeline.pipeline, 97
 tsdat.qc, 104
 tsdat.qc.checkers, 104
 tsdat.qc.handlers, 108
 tsdat.qc.qc, 111
 tsdat.utils, 120
 tsdat.utils.converters, 120
 tsdat.utils.dsutils, 122

N

NAME (*tsdat.config.variable_definition.VarInputKeys* attribute), 46
 NetCdfHandler (class in *tsdat*), 149
 NetCdfHandler (class in *tsdat.io*), 85
 NetCdfHandler (class in *tsdat.io.filehandlers*), 67
 NetCdfHandler (class in *tsdat.io.filehandlers.netcdf_handler*), 64

OUTPUT_DATA_LEVEL (*tsdat.config.pipeline_definition.PipelineKeys* attribute), 43
 output_file_extensions (*tsdat.DatastreamStorage* attribute), 140
 output_file_extensions (*tsdat.io.DatastreamStorage* attribute), 85
 output_file_extensions (*tsdat.io.storage.DatastreamStorage* attribute), 77

P

PARAMETERS (*tsdat.config.variable_definition.ConverterKeys* attribute), 46
 Pipeline (class in *tsdat*), 150
 Pipeline (class in *tsdat.pipeline*), 100
 Pipeline (class in *tsdat.pipeline.pipeline*), 97
 PIPELINE (*tsdat.config.Keys* attribute), 51
 PIPELINE (*tsdat.config.keys.Keys* attribute), 43
 PipelineDefinition (class in *tsdat*), 133
 PipelineDefinition (class in *tsdat.config*), 52
 PipelineDefinition (class in *tsdat.config.pipeline_definition*), 43
 PipelineKeys (class in *tsdat.config.pipeline_definition*), 43
 plot_qc() (*tsdat.DSUtil* static method), 157
 plot_qc() (*tsdat.utils.DSUtil* static method), 128
 plot_qc() (*tsdat.utils.dsutils.DSUtil* static method), 124

Q

QC_BIT (*tsdat.qc.handlers.QCParamKeys* attribute), 109
 QC_BIT (*tsdat.qc.QCParamKeys* attribute), 118
 QCErrror, 60, 140
 QCParamKeys (class in *tsdat.qc*), 118
 QCParamKeys (class in *tsdat.qc.handlers*), 109
 QUALIFIER (*tsdat.config.pipeline_definition.PipelineKeys* attribute), 43
 QUALITY_MANAGEMENT (*tsdat.config.Keys* attribute), 51
 QUALITY_MANAGEMENT (*tsdat.config.keys.Keys* attribute), 43
 QualityChecker (class in *tsdat*), 154

QualityChecker (class in *tsdat.qc*), 114
 QualityChecker (class in *tsdat.qc.checkers*), 104
 QualityHandler (class in *tsdat*), 154
 QualityHandler (class in *tsdat.qc*), 117
 QualityHandler (class in *tsdat.qc.handlers*), 109
 QualityManagement (class in *tsdat.qc*), 113
 QualityManagement (class in *tsdat.qc.qc*), 111
 QualityManager (class in *tsdat.qc*), 113
 QualityManager (class in *tsdat.qc.qc*), 112
 QualityManagerDefinition (class in *tsdat.config*), 57
 QualityManagerDefinition (class in *tsdat.config.quality_manager_definition*), 44
 QualityManagerKeys (class in *tsdat.config.quality_manager_definition*), 44

R

read() (*tsdat.AbstractFileHandler* method), 147
 read() (*tsdat.CsvHandler* method), 149
 read() (*tsdat.FileHandler* static method), 148
 read() (*tsdat.io.AbstractFileHandler* method), 83
 read() (*tsdat.io.CsvHandler* method), 84
 read() (*tsdat.io.FileHandler* static method), 83
 read() (*tsdat.io.filehandlers.AbstractFileHandler* method), 65
 read() (*tsdat.io.filehandlers.csv_handler.CsvHandler* method), 61
 read() (*tsdat.io.filehandlers.CsvHandler* method), 67
 read() (*tsdat.io.filehandlers.file_handlers.AbstractFileHandler* method), 62
 read() (*tsdat.io.filehandlers.file_handlers.FileHandler* static method), 63
 read() (*tsdat.io.filehandlers.FileHandler* static method), 66
 read() (*tsdat.io.filehandlers.netcdf_handler.NetCdfHandler* method), 64
 read() (*tsdat.io.filehandlers.NetCdfHandler* method), 68
 read() (*tsdat.io.NetCdfHandler* method), 85
 read() (*tsdat.NetCdfHandler* method), 149
 read_and_persist_raw_files() (*tsdat.IngestPipeline* method), 153
 read_and_persist_raw_files() (*tsdat.pipeline.ingest_pipeline.IngestPipeline* method), 97
 read_and_persist_raw_files() (*tsdat.pipeline.IngestPipeline* method), 103
 record_correction() (*tsdat.qc.handlers.QualityHandler* method), 109
 record_correction() (*tsdat.qc.QualityHandler* method), 118
 record_correction() (*tsdat.QualityHandler* method), 155

record_corrections_applied() (*tsdat.DSUtil* static method), 155
 record_corrections_applied() (*tsdat.utils.DSUtil* static method), 126
 record_corrections_applied() (*tsdat.utils.dsutils.DSUtil* static method), 122
 RecordQualityResults (class in *tsdat.qc*), 118
 RecordQualityResults (class in *tsdat.qc.handlers*), 109
 reduce_raw_dataset() (*tsdat.Pipeline* method), 152
 reduce_raw_dataset() (*tsdat.pipeline.Pipeline* method), 102
 reduce_raw_dataset() (*tsdat.pipeline.pipeline.Pipeline* method), 99
 reduce_raw_datasets() (*tsdat.Pipeline* method), 151
 reduce_raw_datasets() (*tsdat.pipeline.Pipeline* method), 101
 reduce_raw_datasets() (*tsdat.pipeline.pipeline.Pipeline* method), 99
 REFERENCES (*tsdat.ATTS* attribute), 139
 REFERENCES (*tsdat.constants.ATTS* attribute), 59
 REFERENCES (*tsdat.constants.constants.ATTS* attribute), 58
 region_name() (*tsdat.io.aws_storage.S3Path* property), 69
 region_name() (*tsdat.io.S3Path* property), 95
 region_name() (*tsdat.S3Path* property), 145
 register_file_handler() (*tsdat.FileHandler* static method), 148
 register_file_handler() (*tsdat.io.FileHandler* static method), 83
 register_file_handler() (*tsdat.io.filehandlers.file_handlers.FileHandler* static method), 63
 register_file_handler() (*tsdat.io.filehandlers.FileHandler* static method), 66
 register_filehandler() (in module *tsdat*), 149
 register_filehandler() (in module *tsdat.io*), 83
 register_filehandler() (in module *tsdat.io.filehandlers*), 66
 register_filehandler() (in module *tsdat.io.filehandlers.file_handlers*), 63
 RemoveFailedValues (class in *tsdat.qc*), 119
 RemoveFailedValues (class in *tsdat.qc.handlers*), 110
 REQUIRED (*tsdat.config.variable_definition.VarInputKeys* attribute), 46
 root() (*tsdat.AwsStorage* property), 143
 root() (*tsdat.io.aws_storage.AwsStorage* property), 71
 root() (*tsdat.io.AwsStorage* property), 93
 run() (*tsdat.Converter* method), 159

- `run()` (*tsdat.DefaultConverter* method), 159
 - `run()` (*tsdat.IngestPipeline* method), 152
 - `run()` (*tsdat.Pipeline* method), 150
 - `run()` (*tsdat.pipeline.ingest_pipeline.IngestPipeline* method), 96
 - `run()` (*tsdat.pipeline.IngestPipeline* method), 102
 - `run()` (*tsdat.pipeline.Pipeline* method), 100
 - `run()` (*tsdat.pipeline.pipeline.Pipeline* method), 98
 - `run()` (*tsdat.qc.checkers.CheckMax* method), 106
 - `run()` (*tsdat.qc.checkers.CheckMin* method), 106
 - `run()` (*tsdat.qc.checkers.CheckMissing* method), 105
 - `run()` (*tsdat.qc.checkers.CheckMonotonic* method), 108
 - `run()` (*tsdat.qc.checkers.CheckValidDelta* method), 107
 - `run()` (*tsdat.qc.checkers.QualityChecker* method), 105
 - `run()` (*tsdat.qc.CheckMax* method), 115
 - `run()` (*tsdat.qc.CheckMin* method), 115
 - `run()` (*tsdat.qc.CheckMissing* method), 116
 - `run()` (*tsdat.qc.CheckMonotonic* method), 116
 - `run()` (*tsdat.qc.CheckValidDelta* method), 117
 - `run()` (*tsdat.qc.FailPipeline* method), 118
 - `run()` (*tsdat.qc.handlers.FailPipeline* method), 111
 - `run()` (*tsdat.qc.handlers.QualityHandler* method), 109
 - `run()` (*tsdat.qc.handlers.RecordQualityResults* method), 109
 - `run()` (*tsdat.qc.handlers.RemoveFailedValues* method), 110
 - `run()` (*tsdat.qc.handlers.SendEmailAWS* method), 111
 - `run()` (*tsdat.qc.handlers.SortDatasetByCoordinate* method), 110
 - `run()` (*tsdat.qc.qc.QualityManagement* static method), 111
 - `run()` (*tsdat.qc.qc.QualityManager* method), 112
 - `run()` (*tsdat.qc.QualityChecker* method), 114
 - `run()` (*tsdat.qc.QualityHandler* method), 118
 - `run()` (*tsdat.qc.QualityManagement* static method), 113
 - `run()` (*tsdat.qc.QualityManager* method), 114
 - `run()` (*tsdat.qc.RecordQualityResults* method), 119
 - `run()` (*tsdat.qc.RemoveFailedValues* method), 119
 - `run()` (*tsdat.qc.SendEmailAWS* method), 119
 - `run()` (*tsdat.QualityChecker* method), 154
 - `run()` (*tsdat.QualityHandler* method), 155
 - `run()` (*tsdat.StringTimeConverter* method), 160
 - `run()` (*tsdat.TimestampTimeConverter* method), 160
 - `run()` (*tsdat.utils.Converter* method), 130
 - `run()` (*tsdat.utils.converters.Converter* method), 120
 - `run()` (*tsdat.utils.converters.DefaultConverter* method), 120
 - `run()` (*tsdat.utils.converters.StringTimeConverter* method), 121
 - `run()` (*tsdat.utils.converters.TimestampTimeConverter* method), 121
 - `run()` (*tsdat.utils.DefaultConverter* method), 130
 - `run()` (*tsdat.utils.StringTimeConverter* method), 131
 - `run()` (*tsdat.utils.TimestampTimeConverter* method), 131
 - `run_converter()` (*tsdat.config.variable_definition.VariableDefinition* method), 50
 - `run_converter()` (*tsdat.config.VariableDefinition* method), 55
 - `run_converter()` (*tsdat.VariableDefinition* method), 139
- ## S
- `s3_client()` (*tsdat.AwsStorage* property), 143
 - `s3_client()` (*tsdat.io.aws_storage.AwsStorage* property), 71
 - `s3_client()` (*tsdat.io.AwsStorage* property), 93
 - `s3_resource()` (*tsdat.AwsStorage* property), 143
 - `s3_resource()` (*tsdat.io.aws_storage.AwsStorage* property), 71
 - `s3_resource()` (*tsdat.io.AwsStorage* property), 93
 - `S3Path` (class in *tsdat*), 145
 - `S3Path` (class in *tsdat.io*), 95
 - `S3Path` (class in *tsdat.io.aws_storage*), 68
 - `SAMPLING_INTERVAL` (*tsdat.ATTS* attribute), 140
 - `SAMPLING_INTERVAL` (*tsdat.constants.ATTS* attribute), 59
 - `SAMPLING_INTERVAL` (*tsdat.constants.constants.ATTS* attribute), 58
 - `save()` (*tsdat.DatastreamStorage* method), 141
 - `save()` (*tsdat.io.DatastreamStorage* method), 86
 - `save()` (*tsdat.io.storage.DatastreamStorage* method), 78
 - `save_local_path()` (*tsdat.AwsStorage* method), 144
 - `save_local_path()` (*tsdat.DatastreamStorage* method), 142
 - `save_local_path()` (*tsdat.FilesystemStorage* method), 146
 - `save_local_path()` (*tsdat.io.aws_storage.AwsStorage* method), 72
 - `save_local_path()` (*tsdat.io.AwsStorage* method), 94
 - `save_local_path()` (*tsdat.io.DatastreamStorage* method), 87
 - `save_local_path()` (*tsdat.io.filesystem_storage.FilesystemStorage* method), 75
 - `save_local_path()` (*tsdat.io.FilesystemStorage* method), 92
 - `save_local_path()` (*tsdat.io.storage.DatastreamStorage* method), 78
 - `SendEmailAWS` (class in *tsdat.qc*), 119

SendEmailAWS (class in *tsdat.qc.handlers*), 110
SEPARATOR (in module *tsdat.io.aws_storage*), 68
SERIAL_NUMBER (*tsdat.ATTS* attribute), 139
SERIAL_NUMBER (*tsdat.constants.ATTS* attribute), 59
SERIAL_NUMBER (*tsdat.constants.constants.ATTS* attribute), 58
SortDatasetByCoordinate (class in *tsdat.qc.handlers*), 110
standardize_dataset () (*tsdat.Pipeline* method), 150
standardize_dataset () (*tsdat.pipeline.Pipeline* method), 100
standardize_dataset () (*tsdat.pipeline.pipeline.Pipeline* method), 98
store_and_reopen_dataset () (*tsdat.Pipeline* method), 152
store_and_reopen_dataset () (*tsdat.pipeline.Pipeline* method), 102
store_and_reopen_dataset () (*tsdat.pipeline.pipeline.Pipeline* method), 99
StringTimeConverter (class in *tsdat*), 160
StringTimeConverter (class in *tsdat.utils*), 131
StringTimeConverter (class in *tsdat.utils.converters*), 121

T

temp_path () (*tsdat.AwsStorage* property), 143
temp_path () (*tsdat.io.aws_storage.AwsStorage* property), 71
temp_path () (*tsdat.io.AwsStorage* property), 93
TEMPORAL (*tsdat.config.pipeline_definition.PipelineKeys* attribute), 43
TemporaryStorage (class in *tsdat.io*), 88
TemporaryStorage (class in *tsdat.io.storage*), 80
TEST_MEANING (*tsdat.qc.handlers.QCParamKeys* attribute), 109
TEST_MEANING (*tsdat.qc.QCParamKeys* attribute), 118
TimestampTimeConverter (class in *tsdat*), 160
TimestampTimeConverter (class in *tsdat.utils*), 131
TimestampTimeConverter (class in *tsdat.utils.converters*), 121
TITLE (*tsdat.ATTS* attribute), 139
TITLE (*tsdat.constants.ATTS* attribute), 59
TITLE (*tsdat.constants.constants.ATTS* attribute), 58
tmp () (*tsdat.AwsStorage* property), 143
tmp () (*tsdat.DatastreamStorage* property), 140
tmp () (*tsdat.FilesystemStorage* property), 145
tmp () (*tsdat.io.aws_storage.AwsStorage* property), 71
tmp () (*tsdat.io.AwsStorage* property), 93
tmp () (*tsdat.io.DatastreamStorage* property), 86
tmp () (*tsdat.io.filesystem_storage.FilesystemStorage* property), 74
tmp () (*tsdat.io.FilesystemStorage* property), 91
tmp () (*tsdat.io.storage.DatastreamStorage* property), 77
to_dict () (*tsdat.config.variable_definition.VariableDefinition* method), 50
to_dict () (*tsdat.config.VariableDefinition* method), 55
to_dict () (*tsdat.VariableDefinition* method), 139
tsdat
 module, 39
tsdat.config
 module, 39
tsdat.config.config
 module, 39
tsdat.config.dataset_definition
 module, 40
tsdat.config.dimension_definition
 module, 42
tsdat.config.keys
 module, 43
tsdat.config.pipeline_definition
 module, 43
tsdat.config.quality_manager_definition
 module, 44
tsdat.config.utils
 module, 45
tsdat.config.variable_definition
 module, 46
tsdat.constants
 module, 57
tsdat.constants.constants
 module, 57
tsdat.exceptions
 module, 60
tsdat.exceptions.exceptions
 module, 60
tsdat.io
 module, 60
tsdat.io.aws_storage
 module, 68
tsdat.io.filehandlers
 module, 61
tsdat.io.filehandlers.csv_handler
 module, 61
tsdat.io.filehandlers.file_handlers
 module, 62
tsdat.io.filehandlers.netcdf_handler
 module, 64
tsdat.io.filesystem_storage
 module, 73
tsdat.io.storage
 module, 76
tsdat.pipeline
 module, 95
tsdat.pipeline.ingest_pipeline

module, 95
 tsdat.pipeline.pipeline
 module, 97
 tsdat.qc
 module, 104
 tsdat.qc.checkers
 module, 104
 tsdat.qc.handlers
 module, 108
 tsdat.qc.qc
 module, 111
 tsdat.utils
 module, 120
 tsdat.utils.converters
 module, 120
 tsdat.utils.dsutils
 module, 122
 TYPE (*tsdat.config.pipeline_definition.PipelineKeys* attribute), 43
 TYPE (*tsdat.config.variable_definition.VarKeys* attribute), 46

U

UNITS (*tsdat.ATTS* attribute), 140
 UNITS (*tsdat.config.variable_definition.VarInputKeys* attribute), 46
 UNITS (*tsdat.constants.ATTS* attribute), 59
 UNITS (*tsdat.constants.constants.ATTS* attribute), 58
 upload() (*tsdat.io.aws_storage.AwsTemporaryStorage* method), 70

V

VALID_DELTA (*tsdat.ATTS* attribute), 140
 VALID_DELTA (*tsdat.constants.ATTS* attribute), 59
 VALID_DELTA (*tsdat.constants.constants.ATTS* attribute), 58
 VALID_RANGE (*tsdat.ATTS* attribute), 140
 VALID_RANGE (*tsdat.constants.ATTS* attribute), 59
 VALID_RANGE (*tsdat.constants.constants.ATTS* attribute), 58
 VariableDefinition (*class in tsdat*), 135
 VariableDefinition (*class in tsdat.config*), 52
 VariableDefinition (*class in tsdat.config.variable_definition*), 46
 VARIABLES (*tsdat.config.Keys* attribute), 51
 VARIABLES (*tsdat.config.keys.Keys* attribute), 43
 VARIABLES (*tsdat.config.quality_manager_definition.QualityManagerKeys* attribute), 44
 VarInput (*class in tsdat.config.variable_definition*), 46
 VarInputKeys (*class in tsdat.config.variable_definition*), 46
 VarKeys (*class in tsdat.config.variable_definition*), 46
 VARS (*class in tsdat*), 140
 VARS (*class in tsdat.constants*), 59

VARS (*class in tsdat.constants.constants*), 58

W

WARN_RANGE (*tsdat.ATTS* attribute), 140
 WARN_RANGE (*tsdat.constants.ATTS* attribute), 59
 WARN_RANGE (*tsdat.constants.constants.ATTS* attribute), 58
 write() (*tsdat.AbstractFileHandler* method), 147
 write() (*tsdat.CsvHandler* method), 149
 write() (*tsdat.FileHandler* static method), 148
 write() (*tsdat.io.AbstractFileHandler* method), 83
 write() (*tsdat.io.CsvHandler* method), 84
 write() (*tsdat.io.FileHandler* static method), 83
 write() (*tsdat.io.filehandlers.AbstractFileHandler* method), 65
 write() (*tsdat.io.filehandlers.csv_handler.CsvHandler* method), 61
 write() (*tsdat.io.filehandlers.CsvHandler* method), 67
 write() (*tsdat.io.filehandlers.file_handlers.AbstractFileHandler* method), 62
 write() (*tsdat.io.filehandlers.file_handlers.FileHandler* static method), 63
 write() (*tsdat.io.filehandlers.FileHandler* static method), 66
 write() (*tsdat.io.filehandlers.netcdf_handler.NetCdfHandler* method), 64
 write() (*tsdat.io.filehandlers.NetCdfHandler* method), 67
 write() (*tsdat.io.NetCdfHandler* method), 85
 write() (*tsdat.NetCdfHandler* method), 149