tsdat

Release 0.2.2

Carina Lansing, Maxwell Levin

Oct 15, 2021

CONTENTS

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*

CHAPTER

ONE

PREREQUISITES

Tsdat requires Python 3.8+

CHAPTER

TWO

INSTALLATION

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

CHAPTER

THREE

GETTING A TSDAT PIPELINE TEMPLATE

The quickest way to set up a Tsdat 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.

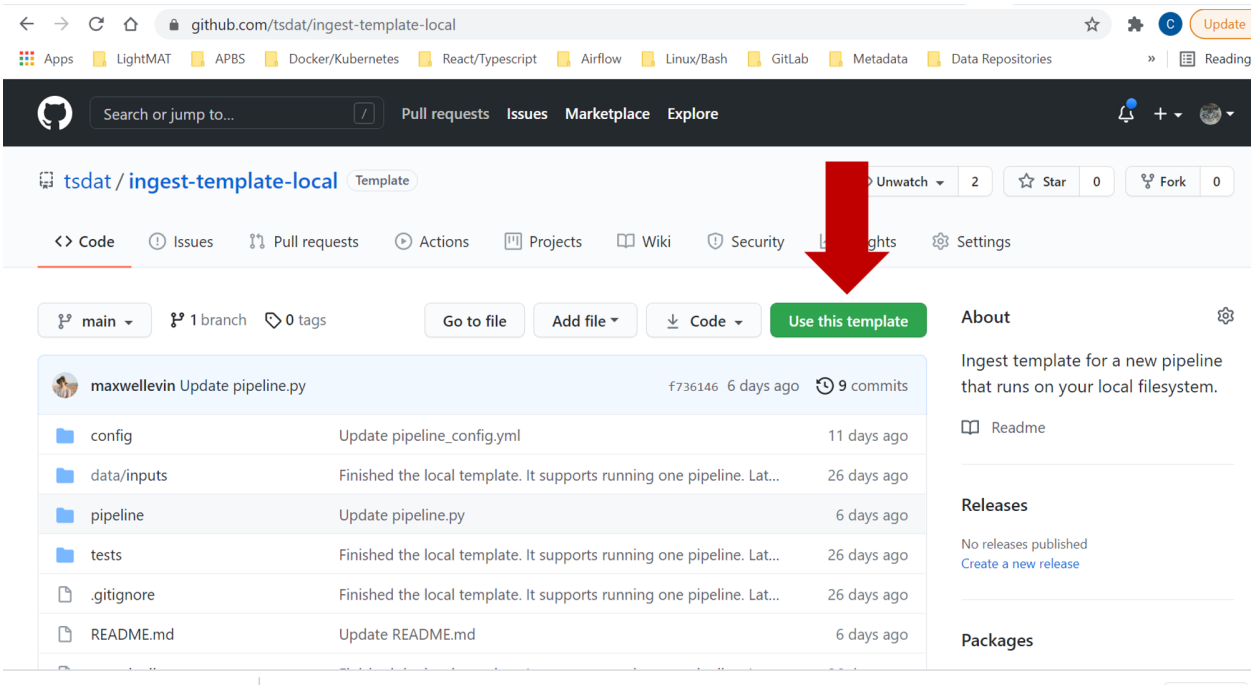
1. Local Ingest Template

   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 addtion, make sure to read the **README.md** file associated with your template for any template-specific instructions.

CHAPTER

FOUR

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
```

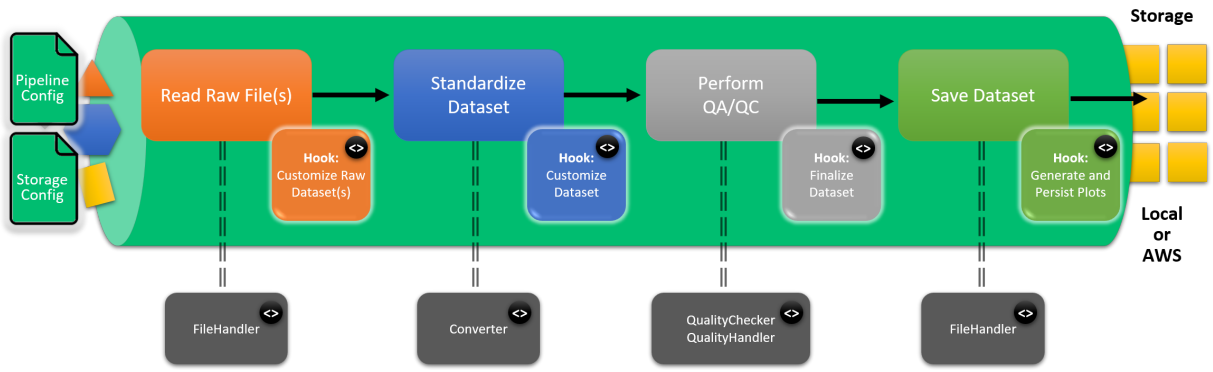
For detailed examples of how to set up and use tsdat, consult the *Examples and Tutorials* section.

CHAPTER

FIVE

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 intial 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 explict, 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 classses 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 priviledges)

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 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

```
########################################################################
# TSDAT (Time-Series Data) INGEST PIPELINE CONFIGURATION TEMPLATE
#
# This file contains an annotated example of how to configure an
# tsdat data ingest processing pipeline.
########################################################################

# Specify the type of pipeline that will be run:  Ingest or VAP
#
# Ingests are run against raw data and are used to convert
# proprietary instrument data files into standardized format, perform
# quality control checks against the data, and apply corrections as
# needed.
#
# VAPs are used to combine one or more lower-level standardized data
# files, optionally transform data to new coordinate grids, and/or
# to apply scientific algorithms to derive new variables that provide
# additional insights on the data.
pipeline:
  type: "Ingest"

  # Used to specify the level of data that this pipeline will use as
  # input. For ingests, this will be used as the data level for raw data.
  # If type: Ingest is specified, this defaults to "00"
  # input_data_level: "00"

  # Used to specify the level of data that this pipeline will produce.
  # It is recommended that ingests use "a1" and VAPs should use "b1",
  # but this is not enforced.
  data_level: "a1"

  # A label for the location where the data were obtained from
  location_id: "humboldt_z05"

  # A string consisting of any letters, digits, "-" or "_" that can
  # be used to uniquely identify the instrument used to produce
  # the data.  To prevent confusion with the temporal resolution
```

(continues on next page)

(continued from previous page)

```
  # of the instrument, the instrument identifier must not end
  # with a number.
  dataset_name: "buoy"

  # An optional qualifier that distinguishes these data from other
  # data sets produced by the same instrument.  The qualifier
  # must not end with a number.
  #qualifier: "lidar"

  # A optional description of the data temporal resolution
  # (e.g., 30m, 1h, 200ms, 14d, 10Hz).  All temporal resolution
  # descriptors require a units identifier.
  #temporal: "10m"

########################################################################
# PART 1: DATASET DEFINITION
# Define dimensions, variables, and metadata that will be included
# in your processed, standardized data file.
########################################################################
dataset_definition:
  #-----------------------------------------------------------------
  # Global Attributes (general metadata)
  #
  # All optional attributes are commented out.  You may remove them
  # if not applicable to your data.
  #
  # You may add any additional attributes as needed to describe your
  # data collection and processing activities.
  #-----------------------------------------------------------------
  attributes:

    # A succinct English language description of what is in the dataset.
    # The value would be similar to a publication title.
    # Example: "Atmospheric Radiation Measurement (ARM) program Best
    # Estimate cloud and radiation measurements (ARMBECLDRAD)"
    # This attribute is highly recommended but is not required.
    title: "Buoy Dataset for Buoy #120"

    # Longer English language description of the data.
    # Example: "ARM best estimate hourly averaged QC controlled product,
    # derived from ARM observational Value-Added Product data: ARSCL,
    # MWRRET, QCRAD, TSI, and satellite; see input_files for the names of
    # original files used in calculation of this product"
    # This attribute is highly recommended but is not required.
    description: "Example ingest dataset used for demonstration purposes."

    # The version of the standards document this data conforms to.
    # This attribute is highly recommended but is not required.
    # conventions: "ME Data Pipeline Standards: Version 1.0"

    # If an optional Digital Object Identifier (DOI) has been obtained
    # for the data, it may be included here.
    #doi: "10.21947/1671051"

    # The institution who produced the data
    # institution: "Pacific Northwest National Laboratory"

```

(continues on next page)

(continued from previous page)

```
    # Include the url to the specific tagged release of the code
    # used for this pipeline invocation.
    # Example,  https://github.com/clansing/twrmr/releases/tag/1.0.
    # Note that MHKiT-Cloud will automatically create a new code
    # release whenever the pipeline is deployed to production and
    # record this attribute automatically.
    code_url: "https://github.com/tsdat/tsdat/releases/tag/v0.2.2"

    # Published or web-based references that describe the methods
    # algorithms, or third party libraries used to process the data.
    #references: "https://github.com/MHKiT-Software/MHKiT-Python"

    # A more detailed description of the site location.
    #location_meaning: "Buoy is located of the coast of Humboldt, CA"

    # Name of instrument(s) used to collect data.
    #instrument_name: "Wind Sentinel"

    # Serial number of instrument(s) used to collect data.
    #serial_number: "000011312"

    # Description of instrument(s) used to collect data.
    #instrument_meaning: "Self-powered floating buoy hosting a suite of␣
↪meteorological and marine instruments."

    # Manufacturer of instrument(s) used to collect data.
    #instrument_manufacturer: "AXYS Technologies Inc."

    # The date(s) of the last time the instrument(s) was calibrated.
    #last_calibration_date: "2020-10-01"

    # The expected sampling interval of the instrument (e.g., "400 us")
    #sampling_interval: "10 min"

  #------------------------------------------------------------------
  # Dimensions (shape)
  #------------------------------------------------------------------
  dimensions:
    # All time series data must have a "time" dimension
    # TODO: provide a link to the documentation online
    time:
        length: "unlimited"

  #------------------------------------------------------------------
  # Variable Defaults
  #
  # Variable defaults can be used to specify a default dimension(s),
  # data type, or variable attributes. This can be used to reduce the
  # number of properties that a variable needs to define in this
  # config file, which can be useful for vaps or ingests with many
  # variables.
  #
  # Once a default property has been defined, (e.g. 'type: float64')
  # that property becomes optional for all variables (e.g. No variables
  # need to have a 'type' property).
  #
  # This section is entirely optional, so it is commented out.
```

(continues on next page)

(continued from previous page)

```
  #------------------------------------------------------------------
  # variable_defaults:

    # Optionally specify defaults for variable inputs. These defaults will
    # only be applied to variables that have an 'input' property. This
    # is to allow for variables that are created on the fly, but defined in
    # the config file.
    # input:

      # If this is specified, the pipeline will attempt to match the file pattern
      # to an input filename. This is useful for cases where a variable has the
      # same name in multiple input files, but it should only be retrieved from
      # one file.
      # file_pattern: "buoy"

      # Specify this to indicate that the variable must be retrieved. If this is
      # set to True and the variable is not found in the input file the pipeline
      # will crash. If this is set to False, the pipeline will continue.
      # required: True

      # Defaults for the converter used to translate input numpy arrays to
      # numpy arrays used for calculations
      # converter:

        #--------------------------------------------------------------
        # Specify the classname of the converter to use as a default.
        # A converter is used to convert the raw data into standardized
        # values.
        #
        # Use the DefaultConverter for all non-time variables that
        # use units supported by udunits2.
        # https://www.unidata.ucar.edu/software/udunits/udunits-2.2.28/udunits2.html
↪#Database
        #
        # If your raw data has units that are not supported by udunits2,
        # you can specify your own Converter class.
        #--------------------------------------------------------------
        # classname: "tsdat.utils.converters.DefaultConverter"

        # If the default converter always requires specific parameters, these
        # can be defined here. Note that these parameters are not tied to the
        # classname specified above and will be passed to all converters defined
        # here.
        # parameters:

          # Example of parameter format:
          # param_name: param_value

    # The name(s) of the dimension(s) that dimension this data by
    # default. For time-series tabular data, the following is a 'good'
    # default to use:
    # dims: [time]

    # The data type to use by default. The data type must be one of:
    # int8 (or byte), uint8 (or ubyte), int16 (or short), uint16 (or ushort),
    # int32 (or int), uint32 (or uint), int64 (or long), uint64 (or ulong),
    # float32 (or float), float64 (or double), char, str
```

(continues on next page)

(continued from previous page)

```
    # type: float64

    # Any attributes that should be defined by default
    # attrs:

      # Default _FillValue to use for missing data. Recommended to use
      # -9999 because it is the default _FillValue according to CF
      # conventions for netCDF data.
      # _FillValue: -9999

  #---------------------------------------------------------------------
  # Variables
  #---------------------------------------------------------------------
  variables:

    #-------------------------------------------------------------------
    # All time series data must have a "time" coordinate variable which
    # contains the data values for the time dimension
    # TODO: provide a link to the documentation online
    #-------------------------------------------------------------------
    time:  # Variable name as it will appear in the processed data

      #-----------------------------------------------------------------
      # The input section for each variable is used to specify the
      # mapping between the raw data file and the processed output data
      #-----------------------------------------------------------------
      input:
        # Name of the variable in the raw data
        name: "DataTimeStamp"

        #---------------------------------------------------------------
        # A converter is used to convert the raw data into standardized
        # values.
        #---------------------------------------------------------------
        # Use the StringTimeConverter if your raw data provides time
        # as a formatted string.
        converter:
          classname: "tsdat.utils.converters.StringTimeConverter"
          parameters:
            # A list of timezones can be found here:
            # https://en.wikipedia.org/wiki/List_of_tz_database_time_zones
            timezone: "US/Pacific"
            time_format: "%Y-%m-%d %H:%M:%S"

        # Use the TimestampTimeConverter if your raw data provides time
        # as a numeric UTC timestamp
        #converter:
        #  classname: tsdat.utils.converters.TimestampTimeConverter
        #  parameters:
        #    # Unit of the numeric value as used by pandas.to_datetime (D,s,ms,us,ns)
        #    unit: s

      # The shape of this variable.  All coordinate variables (e.g., time) must
      # have a single dimension that exactly matches the variable name
      dims: [time]

      # The data type of the variable.  Must be one of:
```

(continues on next page)

(continued from previous page)

```
      # int8 (or byte), uint8 (or ubyte), int16 (or short), uint16 (or ushort),
      # int32 (or int), uint32 (or uint), int64 (or long), uint64 (or ulong),
      # float32 (or float), float64 (or double), char, str
      type: int64

      #---------------------------------------------------------------
      # The attrs section define the attributes (metadata) that will
      # be set for this variable.
      #
      # All optional attributes are commented out.  You may remove them
      # if not applicable to your data.
      #
      # You may add any additional attributes as needed to describe your
      # variables.
      #
      # Any metadata used for QC tests will be indicated.
      #---------------------------------------------------------------
      attrs:

        # A minimal description of what the variable represents.
        long_name: "Time offset from epoch"

        # A string exactly matching a value in from the CF or MRE
        # Standard Name table, if a match exists
        #standard_name: time

        # A CFUnits-compatible string indicating the units the data
        # are measured in.
        # https://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-
↪conventions.html#units
        #
        # Note:  CF Standards require this exact format for time.
        # UTC is strongly recommended.
        # https://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-
↪conventions.html#time-coordinate
        units: "seconds since 1970-01-01T00:00:00"

    #---------------------------------------------------------------
    # Mean temperature variable (non-coordinate variable)
    #---------------------------------------------------------------
    sea_surface_temperature: # Variable name as it will appear in the processed data

      #---------------------------------------------------------------
      # The input section for each variable is used to specify the
      # mapping between the raw data file and the processed output data
      #---------------------------------------------------------------
      input:
        # Name of the variable in the raw data
        name: "Surface Temperature (C)"

        # Units of the variable in the raw data
        units: "degC"

      # The shape of this variable
      dims: [time]

      # The data type of the variable.  Can be one of:
```

(continues on next page)

(continued from previous page)

```
      # [byte, ubyte, char, short, ushort, int32 (or int), uint32 (or uint),
      # int64 (or long), uint64 (or ulong), float, double, string]
      type: double

      #---------------------------------------------------------------
      # The attrs section define the attributes (metadata) that will
      # be set for this variable.
      #
      # All optional attributes are commented out.  You may remove them
      # if not applicable to your data.
      #
      # You may add any additional attributes as needed to describe your
      # variables.
      #
      # Any metadata used for QC tests will be indicated.
      #---------------------------------------------------------------
      attrs:
        # A minimal description of what the variable represents.
        long_name: "Mean sea surface temperature"

        # An optional attribute to provide human-readable context for what this variable
        # represents, how it was measured, or anything else that would be relevant to end-users.
        #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].

        # A CFUnits-compatible string indicating the units the data
        # are measured in.
        # https://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#units
        units: "degC"

        # The value used to initialize the variable's data. Defaults to -9999.
        # Coordinate variables must not use this attribute.
        #_FillValue: -9999

        # An array of variable names that depend on the values from this variable. This is primarily
        # used to indicate if a variable has an ancillary qc variable.
        # NOTE: QC ancillary variables will be automatically recorded via the MHKiT-Cloud pipeline engine.
        #ancillary_variables: []

        # A two-element array of [min, max] representing the smallest and largest valid values
        # of a variable.  Values outside valid_range will be filled with _FillValue.
        #valid_range: [-50, 50]

        # The maximum allowed difference between any two consecutive values of a variable,
        # values outside of which should be flagged as "Bad".
        # This attribute is used for the valid_delta QC test.  If not specified, this
        # variable will be omitted from the test.
        #valid_delta: 0.25

```

(continues on next page)

(continued from previous page)

```
        # A two-element array of [min, max] outside of which the data should be flagged as "Bad".
        # This attribute is used for the fail_min and fail_max QC tests.
        # If not specified, this variable will be omitted from these tests.
        #fail_range: [0, 40]

        # A two-element array of [min, max] outside of which the data should be flagged as "Indeterminate".
        # This attribute is used for the warn_min and warn_max QC tests.
        # If not specified, this variable will be omitted from these tests.
        #warn_range: [0, 30]

        # An array of strings indicating what corrections, if any, have been applied to the data.
        #corrections_applied: []

        # The height of the instrument above ground level (AGL), or in the case of above
        # water, above the surface.
        #sensor_height: "30m"

    #---------------------------------------------------------------------
    # Example of a variables that hold a single scalar value that
    # is not present in the raw data.
    #---------------------------------------------------------------------
    latitude:
      data: 71.323 #<-----The data field can be used to specify a pre-set value
      type: float

      #<-----This variable has no input, which means it will be set by
      # the pipeline and not pulled from the raw data

      #<-----This variable has no dimensions, which means it will be
      # a scalar value

      attrs:
        long_name: "North latitude"
        standard_name: "latitude"
        comment: "Recorded lattitude at the instrument location"
        units: "degree_N"
        valid_range: [-90.f, 90.f]

    longitude:
      data: -156.609
      type: float
      attrs:
        long_name: "East longitude"
        standard_name: "longitude"
        comment: "Recorded longitude at the instrument location"
        units: "degree_E"
        valid_range: [-180.f, 180.f]

    #---------------------------------------------------------------------
    # Example of a variable that is derived by the processing pipeline
    #---------------------------------------------------------------------
    foo:
      type: float
```

(continues on next page)

(continued from previous page)

```
      #<-----This variable has no input, which means it will be set by
      # the pipeline and not pulled from the raw data

      dims: [time]

      attrs:
        long_name: "some other property"
        units: "kg/m^3"
        comment: "Computed from temp_mean point value using some formula..."
        references: ["http://sccoos.org/data/autoss/", "http://sccoos.org/about/dmac/
↪"]

---
########################################################################
# PART 2: QC TESTS
# Define the QC tests that will be applied to variable data.
########################################################################
coordinate_variable_qc_tests:
  #-----------------------------------------------------------------
  # The following section defines the default qc tests that will be
  # performed on coordinate variables in a dataset.  Note that by
  # default, coordinate variable tests will NOT set a QC bit and
  # will trigger a critical pipeline failure.  This is because
  # Problems with coordinate variables are considered to cause
  # the dataset to be unusable and should be manually reviewed.
  #
  # However, the user may override the default coordinate variable
  # tests and error handlers if they feel that data correction is
  # warranted.
  #
  # For a complete list of tests provided by MHKiT-Cloud, please see
  # the tsdat.qc.operators package.
  #
  # Users are also free to add custom tests defined by their own
  # checker classes.
  #-----------------------------------------------------------------

quality_management:
  #-----------------------------------------------------------------
  # The following section defines the default qc tests that will be
  # performed on variables in a dataset.
  #
  # For a complete list of tests provided by MHKiT-Cloud, please see
  # the tsdat.qc.operators package.
  #
  # Users are also free to add custom tests defined by their own
  # checker classes.
  #-----------------------------------------------------------------

  #-----------------------------------------------------------------
  # Checks on coordinate variables
  #-----------------------------------------------------------------

  # The name of the test.
  manage_missing_coordinates:

```

(continues on next page)

(continued from previous page)

```
    # Quality checker used to identify problematic variable values.
    # Users can define their own quality checkers and link them here
    checker:
      # This quality checker will identify values that are missing,
      # NaN, or equal to each variable's _FillValue
      classname: "tsdat.qc.operators.CheckMissing"

    # Quality handler used to manage problematic variable values.
    # Users can define their own quality handlers and link them here.
    handlers:
      # This quality handler will cause the pipeline to fail
      - classname: "tsdat.qc.error_handlers.FailPipeline"

    # Which variables to apply the test to
    variables:
      # keyword to apply test to all coordinate variables
      - COORDS

  manage_coordinate_monotonicity:

    checker:
      # This quality checker will identify variables that are not
      # strictly monotonic (That is, it identifies variables whose
      # values are not strictly increasing or strictly decreasing)
      classname: "tsdat.qc.operators.CheckMonotonic"

    handlers:
      - classname: "tsdat.qc.error_handlers.FailPipeline"

    variables:
      - COORDS

  #---------------------------------------------------------------------
  # Checks on data variables
  #---------------------------------------------------------------------
  manage_missing_values:

    # The class that performs the quality check. Users are free
    # to override with their own class if they want to change
    # behavior.
    checker:
      classname: "tsdat.qc.operators.CheckMissing"

    # Error handlers are optional and run after the test is
    # performed if any of the values fail the test.  Users
    # may specify one or more error handlers which will be
    # executed in sequence.  Users are free to add their
    # own QCErrorHandler subclass if they want to add custom
    # behavior.
    handlers:

      # This error handler will replace any NaNs with _FillValue
      - classname: "tsdat.qc.error_handlers.RemoveFailedValues"
        # Quality handlers and all other objects that have a 'classname'
        # property can take a dictionary of parameters. These
        # parameters are made available to the object or class in the
        # code and can be used to implement custom behavior with little
```

(continues on next page)

(continued from previous page)

```
        # overhead.
        parameters:

          # The correction parameter is used by the RemoveFailedValues
          # quality handler to append to a list of corrections for each
          # variable that this handler is applied to. As a best practice,
          # quality handlers that modify data values should use the
          # correction parameter to update the 'corrections_applied'
          # variable attribute on the variable this test is applied to.
          correction: "Set NaN and missing values to _FillValue"


      # This quality handler will record the results of the
      # quality check in the ancillary qc variable for each
      # variable this quality manager is applied to.
      - classname: "tsdat.qc.error_handlers.RecordQualityResults"
        parameters:

          # The bit (1-32) used to record the results of this test.
          # This is used to update the variable's ancillary qc
          # variable.
          bit: 1

          # The assessment of the test.  Must be either 'Bad' or 'Indeterminate'
          assessment: "Bad"

          # The description of the data quality from this check
          meaning: "Value is equal to _FillValue or NaN"

    variables:
      # keyword to apply test to all non-coordinate variables
      - DATA_VARS

  manage_fail_min:
    checker:
      classname: "tsdat.qc.operators.CheckFailMin"
    handlers:
      - classname: "tsdat.qc.error_handlers.RecordQualityResults"
        parameters:
          bit: 2
          assessment: "Bad"
          meaning: "Value is less than the fail_range."
    variables:
      - DATA_VARS

  manage_fail_max:
    checker:
      classname: "tsdat.qc.operators.CheckFailMax"
    handlers:
      - classname: "tsdat.qc.error_handlers.RecordQualityResults"
        parameters:
          bit: 3
          assessment: "Bad"
          meaning: "Value is greater than the fail_range."
    variables:
      - DATA_VARS

```

(continues on next page)

(continued from previous page)

```yaml
  manage_warn_min:
    checker:
      classname: "tsdat.qc.operators.CheckWarnMin"
    handlers:
      - classname: "tsdat.qc.error_handlers.RecordQualityResults"
        parameters:
          bit: 4
          assessment: "Indeterminate"
          meaning: "Value is less than the warn_range."
    variables:
      - DATA_VARS

  manage_warn_max:
    checker:
      classname: "tsdat.qc.operators.CheckWarnMax"
    handlers:
      - classname: "tsdat.qc.error_handlers.RecordQualityResults"
        parameters:
          bit: 5
          assessment: "Indeterminate"
          meaning: "Value is greater than the warn_range."
    variables:
      - DATA_VARS

  manage_valid_delta:
    checker:
      classname: "tsdat.qc.operators.CheckValidDelta"
      parameters:
        dim: time  # specifies the dimension over which to compute the delta
    handlers:
      - classname: "tsdat.qc.error_handlers.RecordQualityResults"
        parameters:
          bit: 6
          assessment: "Indeterminate"
          meaning: "Difference between current and previous values exceeds valid_
↪delta."
    variables:
      - DATA_VARS

    #---------------------------------------------------------------
    # Example of a user-created test that shows how to specify
    # an error handler.  Error handlers may be optionally added to
    # any of the tests described above.  (Note that this example will
    # not work, it is just provided as an example of adding a
    # custom QC test.)
    #---------------------------------------------------------------
    # temp_test:

    #   checker:
    #     classname: "myproject.qc.operators.TestTemp"

    #   #-----------------------------------------------------------
    #   # See the tsdat.qc.error_handlers package for a list of
    #   # available error handlers.
    #   #-----------------------------------------------------------
    #   handlers:

```

(continues on next page)

(continued from previous page)

```
#       # This handler will set bit number 7 on the ancillary qc
#       # variable for the variable(s) this test applies to.
#     - classname: "tsdat.qc.error_handlers.RecordQualityResults"
#       parameters:
#         bit: 7
#         assessment: "Indeterminate"
#         meaning: "Test for some special condition in temperature."

#       # This error handler will notify users via email.  The
#       # datastream name, variable, and failing values will be
#       # included.
#     - classname: "tsdat.qc.error_handlers.SendEmailAWS"
#       parameters:
#         message: "Test failed..."
#         recipients: ["carina.lansing@pnnl.gov", "maxwell.levin@pnnl.gov"]

#   # Specifies the variable(s) this quality manager applies to
#   variables:
#     - 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 addtion, 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 Tsdat.

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 Tsdat.

CHAPTER

SIX

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

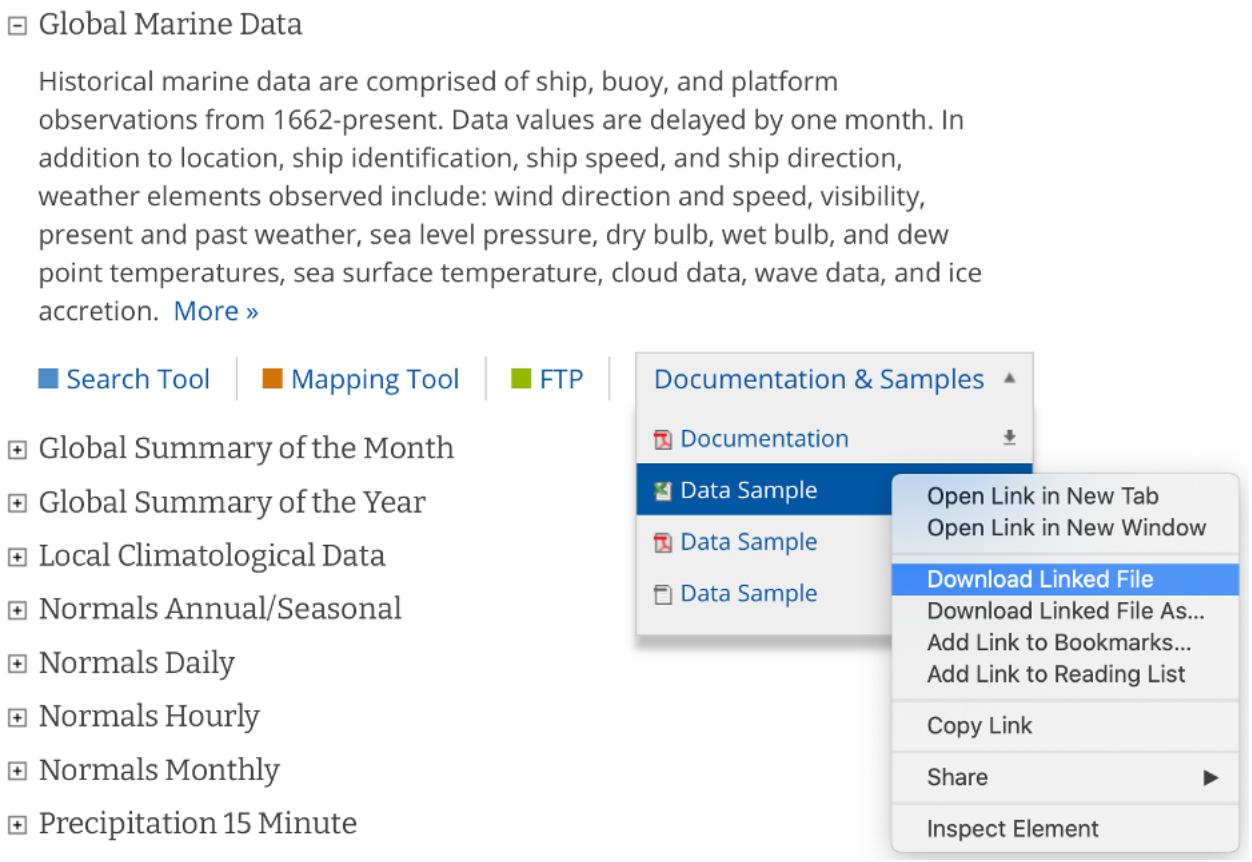
In this tutorial we will build a data ingestion pipeline to ingest some global marine data hosted by the National Oceanic and Atmospheric Administration's (NOAA) National Centers for Environmental Information (NCEI). The data can be found at https://www.ncdc.noaa.gov/cdo-web/datasets under the "Global 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.



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

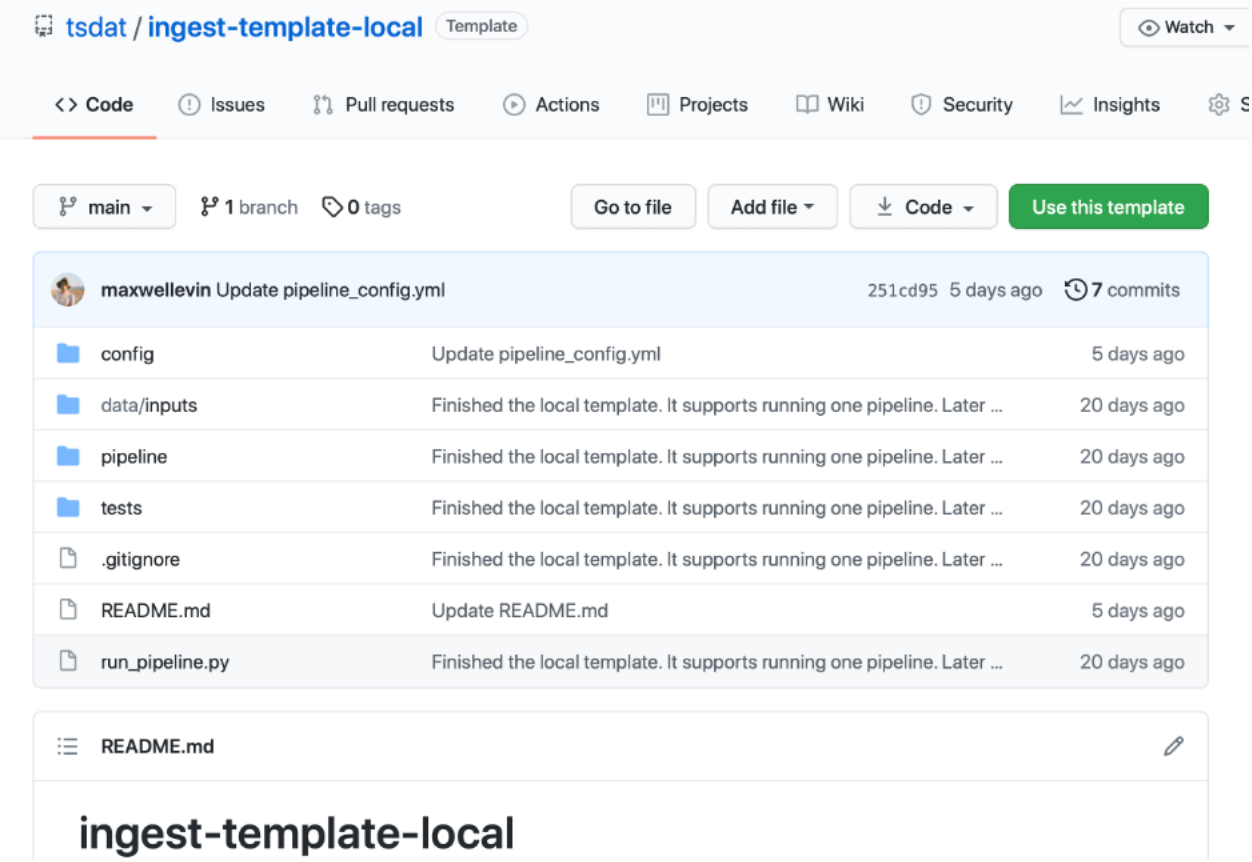
Fahrenheit (Our preference is to use the metric system wherever possible). If we examine this data, it appears that the units are not metric – how unfortunate. Luckily, this is something that can easily be fixed by using tsdat.

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

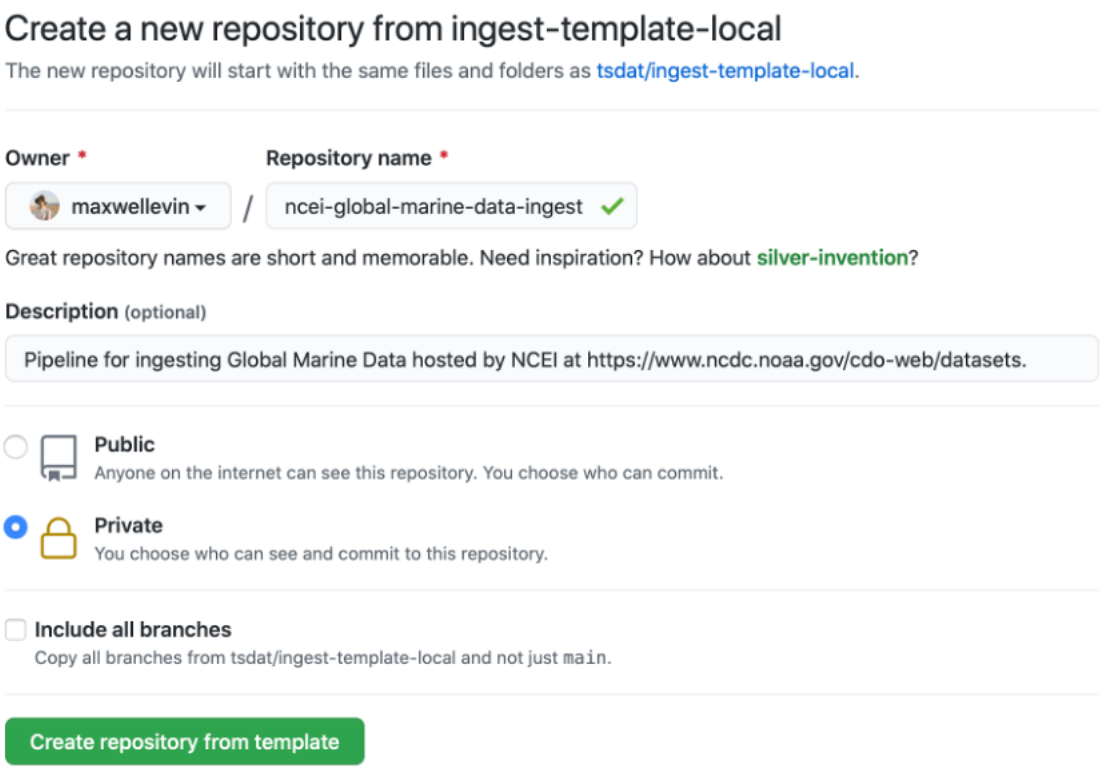
Fig. 1: Selection from the sample dataset. It appears that units are recorded in the imperial system instead of the metric system – Sea Level Pressure is recorded in Hg instead of hPa (Hectopascal) and Air Temperature is recorded in degF (Fahrenheit) instead of degC (Celsius).

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".



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.



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 3.7+ 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 -m venv ncei_env/
source ncei_env/bin/activate
pip install tsdat
```

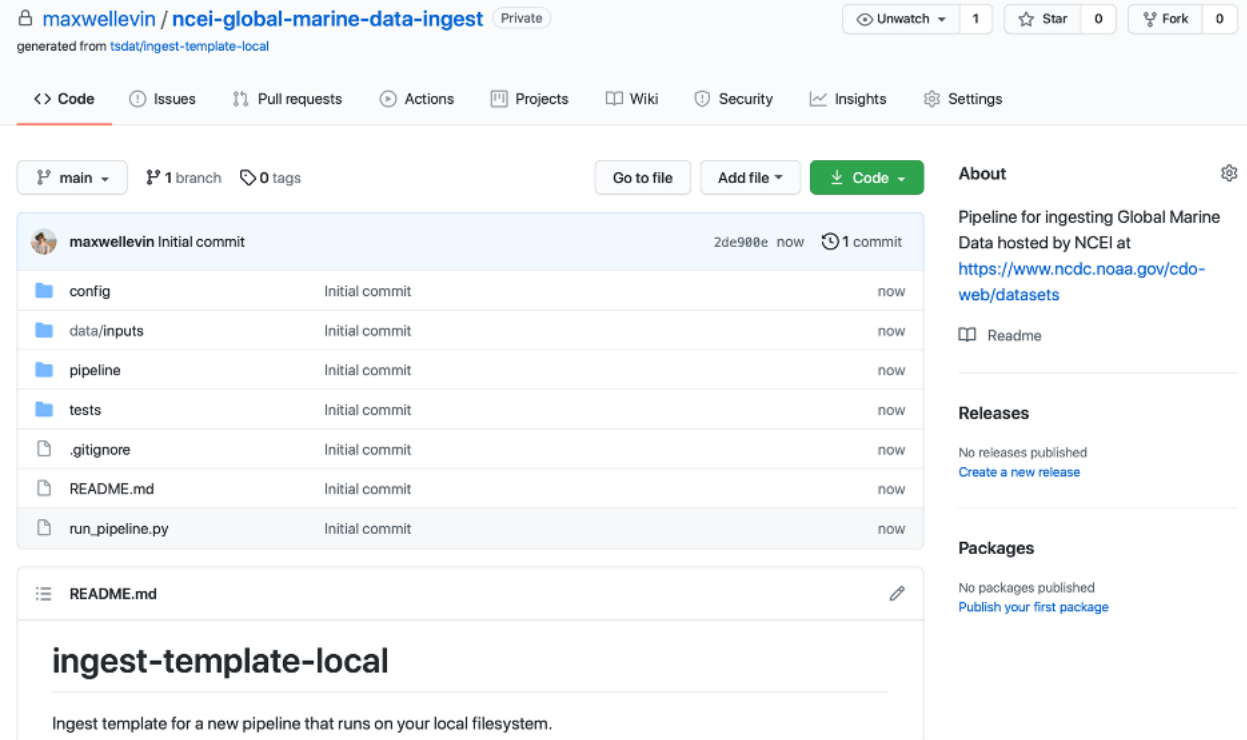
This will install tsdat into our *ncei_env* virtual environment.

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

```
python3 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
2de900e 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 https://github.com/ME- Data-Pipeline-Software/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:

```yaml
storage:
    classname:  tsdat.io.FilesystemStorage
    parameters:
        retain_input_files: True
        root_dir: ${CONFIG_DIR}/../storage/root

    file_handlers:
        input:
        csv:
            file_pattern: '.*\.csv'
            classname: pipeline.filehandlers.CsvHandler
            parameters:
                read:
                    sep: " *, *"
                    engine: "python"
                    index_col: False

        output:
        netcdf:
            file_extension: '.nc'
            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:

```yaml
pipeline:
    type: Ingest
    location_id: arctic
    dataset_name: ice_accretion
    qualifier: ship_001
    data_level: a1

```

(continues on next page)

(continued from previous page)

```yaml
dataset_definition:
    attributes:
        title: "Marine Meteorolical Measurements (Example Ingest)"
        description: "Historical marine data are comprised of ship, buoy, and platform observations."
        conventions: "ME Data Pipeline Standards: Version 1.0"
        institution: "National Oceanic and Atmospheric Administration"
        code_url: "https://github.com/maxwellevin/ncei-global-marine-data-ingest"

    dimensions:
        time:
            length: unlimited

    variables:
        time:
            input:
                name: Time of Observation
                converter:
                classname: tsdat.utils.converters.StringTimeConverter
                parameters:
                    time_format: "%Y-%m-%dT%H:%M:%S"
            dims: [time]
            type: long
            attrs:
                long_name: Time of Observation (UTC)
                standard_name: time
                units: seconds since 1970-01-01T00:00:00

        ice_accretion_source:
            input:
                name: Ice Accretion On Ship
            dims: [time]
            type: int
            attrs:
                long_name: Ice Accretion Source
                comment: "1=Icing from ocean spray, 2=Icing from fog, 3=Icing from spray and fog, 4=Icing
                from rain, 5=Icing from spray and rain"

        ice_accretion_thickness:
            input:
                name: Thickness of Ice Accretion On Ship
            dims: [time]
            type: int
            attrs:
                long_name: Ice Accretion Thickness
                units: cm

        pressure:
            input:
                name: Sea Level Pressure
            dims: [time]
            type: float
            attrs:
                long_name: Pressure at Sea Level
                units: hPa
```

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:

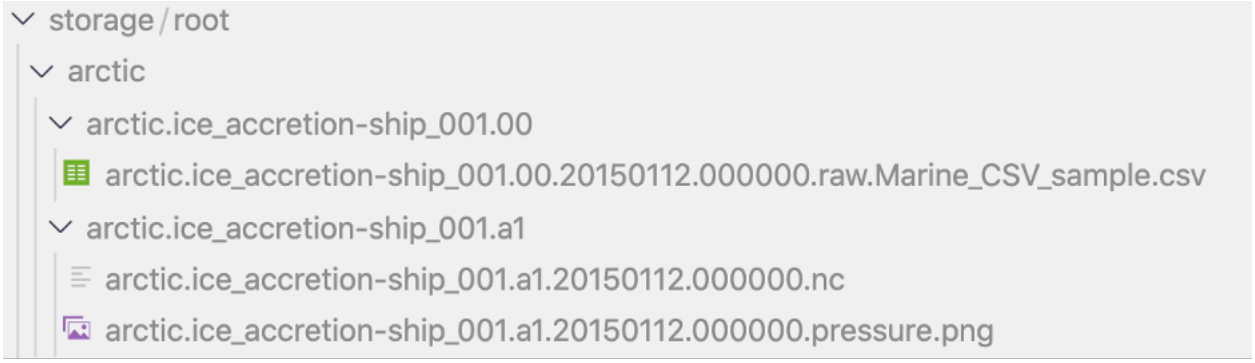
```
import os
import cmocean
import matplotlib.pyplot as plt
import pandas as pd
import xarray as xr
from tsdat.pipeline import IngestPipeline
from tsdat.utils import DSUtil

example_dir = os.path.abspath(os.path.dirname(__file__))
style_file = os.path.join(example_dir, "styling.mplstyle")
plt.style.use(style_file)


class Pipeline(IngestPipeline):

    def hook_generate_and_persist_plots(self, dataset: xr.Dataset) -> None:
        start_date = pd.to_datetime(dataset.time.data[0]).strftime('%Y-%m-%d')
        final_date = pd.to_datetime(dataset.time.data[-1]).strftime('%Y-%m-%d')

        filename = DSUtil.get_plot_filename(dataset, "pressure", "png")
        with self.storage._tmp.get_temp_filepath(filename) as tmp_path:

            fig, ax = plt.subplots(figsize=(10, 8), constrained_layout=True)
            fig.suptitle(f"Pressure Observations from {start_date} to {final_date}")
            dataset.pressure.plot(ax=ax, x="time", c=cmocean.cm.deep_r(0.5))

            fig.savefig(tmp_path, dpi=100)
            self.storage.save(tmp_path)
            plt.close()

        return
```

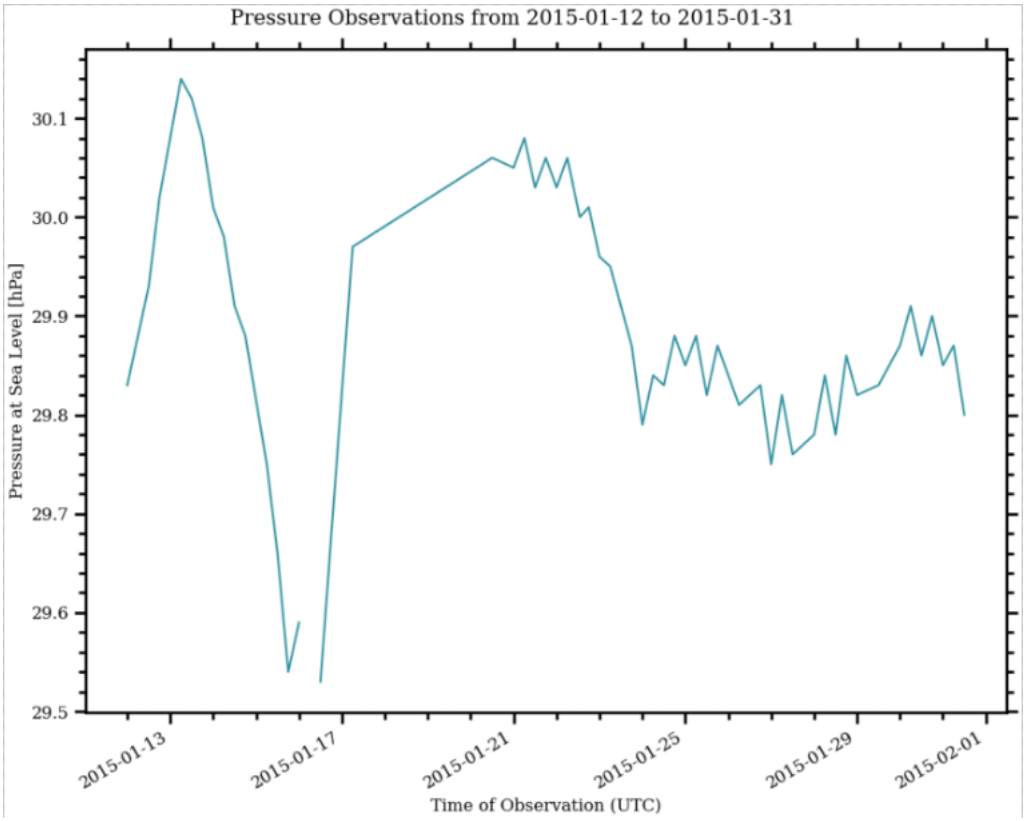
Running the pipeline

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

```
python3 run_pipeline.py
```

and it will produce the following results:




Pressure Observations from 2015-01-12 to 2015-01-31
Pressure at Sea Level [hPa]
30.1
30.0
29.9
29.8
29.7
29.6
29.5
2015-01-13
2015-01-17
2015-01-21
2015-01-25
2015-01-29
2015-02-01
Time of Observation (UTC)

CHAPTER

SEVEN

API REFERENCE

This page contains auto-generated API reference documentation[1].

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

Config	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, *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]

[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, *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*

Attemps 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

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

Keys	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

PipelineKeys	Class that provides a handle for keys in the pipeline section of the
PipelineDefinition	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

QualityManagerKeys	Class that provides a handle for keys in the quality management section
QualityManagerDefinition	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

configure_yaml()	Configure yaml to automatically substitute environment variables
instantiate_handler(*args, handler_desc: Dict = None) → Union[object, List[object]]	Class to instantiate one or more classes given a dictionary containing
_instantiate_class(*args, **kwargs)	Instantiates a python class given args and kwargs.
_parse_fully_qualified_name(fully_qualified_name: str) → Tuple[str, str]	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 PYTHONPATH. The `handler_desc` can also contain a `parameters` entry, which will is 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

VarKeys	Class that provides a handle for keys in the variables section of the
VarInputKeys	Class that provides a handle for keys in the variable input section of
ConverterKeys	Class that provides a handle for keys in the converter section of
VarInput	Class to explicitly encode fields set by the variable's input source
VariableDefinition	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.DimensionDefin*, *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 much 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 intialize 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

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

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]*, *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 much 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 intialize 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*

Attemps 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

VARS	Class that adds keywords for referring to variables.
ATTS	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_DESCRPTION = location_description

INSTRUMENT_NAME = instrument_name

SERIAL_NUMBER = serial_number

INSTRUMENT_DESCRPTION = 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

ATTS	Class that adds constants for interacting with tsdat data-model
VARS	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_DESCRPTION = location_description

INSTRUMENT_NAME = instrument_name

SERIAL_NUMBER = serial_number

INSTRUMENT_DESCRPTION = 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 contribututions 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

CsvHandler	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

AbstractFileHandler	Abstract class to define methods required by all FileHandlers. Classes
FileHandler	Class to provide methods to read and write files with a variety of

Functions

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

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.

FILEHANDLERS :Dict[str, AbstractFileHandler]

static _get_handler(*filename: str*) → *AbstractFileHandler*
Given the name of the file to read or write, this method applies a regular expression to match the name of the file with a handler that has been registered in its internal dictionary of FileHandler objects and returns the appropriate FileHandler, or None if a match is not found.

Parameters **filename** (*str*) – The name of the file whose handler should be retrieved.

Returns The FileHandler registered for use with the provided filename.

Return type *AbstractFileHandler*

static write(*ds: xarray.Dataset*, *filename: str*, *config:* tsdat.config.Config = *None*, ***kwargs*) → None
Saves the given dataset to file using the registered FileHandler for the provided filename.

Parameters

- **ds** (*xr.Dataset*) – The dataset ot save.
- **filename** (*str*) – The path to where the file should be written to.
- **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 using the registered FileHandler for the provided filename.

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

Returns A xr.Dataset object.

Return type xr.Dataset

static register_file_handler(*patterns: Union[str, List[str]]*, *handler:* AbstractFileHandler)
Static method to register an AbstractFileHandler for one or more file patterns. Once an AbstractFileHandler has been registered it may be used by this class to read or write files whose paths match one or more pattern(s) provided in registration.

Parameters

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

(continues on next page)

(continued from previous page)

```
    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

NetCdfHandler	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

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

Functions

register_filehandler(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.

FILEHANDLERS :Dict[str, AbstractFileHandler]

static _get_handler(*filename: str*) → *AbstractFileHandler*

Given the name of the file to read or write, this method applies a regular expression to match the name of

the file with a handler that has been registered in its internal dictionary of FileHandler objects and returns the appropriate FileHandler, or None if a match is not found.

Parameters **filename** (*str*) – The name of the file whose handler should be retrieved.

Returns The FileHandler registered for use with the provided filename.

Return type *AbstractFileHandler*

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

Saves the given dataset to file using the registered FileHandler for the provided filename.

Parameters

- **ds** (*xr.Dataset*) – The dataset ot save.
- **filename** (*str*) – The path to where the file should be written to.
- **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 using the registered FileHandler for the provided filename.

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

Returns A xr.Dataset object.

Return type xr.Dataset

static register_file_handler(*patterns: Union[str, List[str]]*, *handler:* AbstractFileHandler)

Static method to register an AbstractFileHandler for one or more file patterns. Once an AbstractFileHandler has been registered it may be used by this class to read or write files whose paths match one or more pattern(s) provided in registration.

Parameters

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

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

Attributes

SEPARATOR

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 fileystem 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 int 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 intantiated 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 fileystem 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

FilesystemTemporaryStorage	Class used to store temporary files or perform
FilesystemStorage	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 fileystem 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]]*) → *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: 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 int 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 intantiated 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 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 fileystem 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.

tsdat.io.storage

Module Contents

Classes

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

Functions

_is_image(x)	
_is_raw(x)	

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 fileystem 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.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 fileystem 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 int 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

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

Functions

register_filehandler(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.

FILEHANDLERS :Dict[str, AbstractFileHandler]

static _get_handler (*filename: str*) → *AbstractFileHandler*
Given the name of the file to read or write, this method applies a regular expression to match the name of the file with a handler that has been registered in its internal dictionary of FileHandler objects and returns the appropriate FileHandler, or None if a match is not found.

Parameters **filename** (*str*) – The name of the file whose handler should be retrieved.

Returns The FileHandler registered for use with the provided filename.

Return type *AbstractFileHandler*

static write (*ds: xarray.Dataset*, *filename: str*, *config:* tsdat.config.Config = *None*, ***kwargs*) → None
Saves the given dataset to file using the registered FileHandler for the provided filename.

Parameters

- **ds** (*xr.Dataset*) – The dataset ot save.
- **filename** (*str*) – The path to where the file should be written to.
- **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 using the registered FileHandler for the provided filename.

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

Returns A xr.Dataset object.

Return type xr.Dataset

static register_file_handler (*patterns: Union[str, List[str]]*, *handler:* AbstractFileHandler)
Static method to register an AbstractFileHandler for one or more file patterns. Once an AbstractFileHandler has been registered it may be used by this class to read or write files whose paths match one or more pattern(s) provided in registration.

Parameters

- **patterns** (*Union[str, List[str]]*) – The patterns (regex) that should be used to match a filepath to the AbstractFileHandler provided.
- **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 fileystem 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 fileystem 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 int 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 intantiated 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 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 fileystem 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.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 intantiated 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 fileystem 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]]*) → None

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 it 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

Pipeline	This class serves as the base class for all tsdat data pipelines.

class tsdat.pipeline.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 definiton 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 hte 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 provdided 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

Pipeline	This class serves as the base class for all tsdat data pipelines.
IngestPipeline	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 definiton 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 hte 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 provdided 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]]*) → None

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 it 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 contribututions to increase this default list.

Submodules

tsdat.qc.checkers

Module Contents

Classes

QualityChecker	Class containing the code to perform a single Quality Check on a
CheckMissing	Checks if any values are assigned to _FillValue or 'NaN' (for non-time
CheckMin	Check that no values for the specified variable are less than
CheckMax	Check that no values for the specified variable are greater than
CheckValidMin	Check that no values for the specified variable are less than
CheckValidMax	Check that no values for the specified variable are greater than
CheckFailMin	Check that no values for the specified variable are less than
CheckFailMax	Check that no values for the specified variable greater less than
CheckWarnMin	Check that no values for the specified variable are less than
CheckWarnMax	Check that no values for the specified variable are greater than
CheckValidDelta	Check that the difference between any two consecutive
CheckMonotonic	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 monitonic 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 arithmectic 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 arithmectic 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 arithmectic 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 arithmectic 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 posess 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 posess 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 posess 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 posess 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 posess 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 posess 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 posess 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 arithmectic 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 arithmectic 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

QCParamKeys	Symbolic constants used for referencing QC-related
QualityHandler	Class containing code to be executed if a particular quality check fails.
RecordQualityResults	Record the results of the quality check in an ancillary qc variable.
RemoveFailedValues	Replace all the failed values with _FillValue
SortDatasetByCoordinate	Sort coordinate data using xr.Dataset.sortby(). Accepts the following

continues on next page

Table 29 – continued from previous page

SendEmailAWS	Send an email to the recipients using AWS services.
FailPipeline	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:* 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 persion, 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:* 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 persion, 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: 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 persion, 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: tsdat.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 persion, 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: 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 persion, 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: 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 persion, 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

QualityManagement	Class that provides static helper functions for providing quality
QualityManager	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 monitonic 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: 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 monitonic 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

QualityManagement	Class that provides static helper functions for providing quality
QualityManager	Applies a single Quality Manager to the given Dataset, as defined by
QualityChecker	Class containing the code to perform a single Quality Check on a
CheckWarnMax	Check that no values for the specified variable are greater than
CheckFailMax	Check that no values for the specified variable greater less than
CheckFailMin	Check that no values for the specified variable are less than
CheckMax	Check that no values for the specified variable are greater than

continues on next page

Table 31 – continued from previous page

CheckMin	Check that no values for the specified variable are less than
CheckMissing	Checks if any values are assigned to _FillValue or 'NaN' (for non-time
CheckMonotonic	Checks that all values for the specified variable are either
CheckValidDelta	Check that the difference between any two consecutive
CheckValidMax	Check that no values for the specified variable are greater than
CheckValidMin	Check that no values for the specified variable are less than
CheckWarnMin	Check that no values for the specified variable are less than
QualityHandler	Class containing code to be executed if a particular quality check fails.
QCParamKeys	Symbolic constants used for referencing QC-related
FailPipeline	Throw an exception, halting the pipeline & indicating a critical error
RecordQualityResults	Record the results of the quality check in an ancillary qc variable.
RemoveFailedValues	Replace all the failed values with _FillValue
SendEmailAWS	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 monitonic 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 monitonic 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 monitonic 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 arithmectic 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:* 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 posess the 'warn_range' attribute, this check will be skipped.

class tsdat.qc.**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 posess the 'fail_range' attribute, this check will be skipped.

class tsdat.qc.**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 posess the 'fail_range' attribute, this check will be skipped.

class tsdat.qc.**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 arithmectic 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: 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 arithmectic 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 arithmectic 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 arithmectic 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 posess 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 arithmectic 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 vaue set by the 'valid_range' attribute. If the variable in question does not posess 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 vaue set by the 'valid_range' attribute. If the variable in question does not posess 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 vaue set by the 'warn_range' attribute. If the variable in question does not posess 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 persion, 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 persion, 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 persion, 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 persion, 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 persion, 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

Converter	Base class for converting data arrays from one units to another.
DefaultConverter	Default class for converting units on data arrays. This class utilizes
StringTimeConverter	Convert a time string to a np.datetime64, which is needed for xarray.
TimestampTimeConverter	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

DSUtil	Provides helper functions for xarray.Dataset

class tsdat.utils.dsutils.**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 formated 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 'yyyymmdd' 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*) → 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*) → 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 datstream 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

DSUtil	Provides helper functions for xarray.Dataset
Converter	Base class for converting data arrays from one units to another.
DefaultConverter	Default class for converting units on data arrays. This class utilizes
StringTimeConverter	Convert a time string to a np.datetime64, which is needed for xarray.
TimestampTimeConverter	Convert a numeric UTC timestamp to a np.datetime64, 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 formated 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 'yyyymmdd' 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*) → 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*) → 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 datstream 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

Config	Wrapper for the pipeline configuration file.
PipelineDefinition	Wrapper for the pipeline portion of the pipeline config file.
DatasetDefinition	Wrapper for the dataset_definition portion of the pipeline config
DimensionDefinition	Class to represent dimensions defined in the pipeline config file.
VariableDefinition	Class to encode variable definitions from the config file. Also provides
ATTS	Class that adds constants for interacting with tsdat data-model
VARS	Class that adds keywords for referring to variables.
DatastreamStorage	DatastreamStorage is the base class for providing
AwsStorage	DatastreamStorage subclass for an AWS S3-based filesystem.
FilesystemStorage	Datastreamstorage subclass for a local Linux-based filesystem.
AbstractFileHandler	Abstract class to define methods required by all File-Handlers. Classes
FileHandler	Class to provide methods to read and write files with a variety of
CsvHandler	FileHandler to read from and write to CSV files. Takes a number of
NetCdfHandler	FileHandler to read from and write to netCDF files. Takes a number of
Pipeline	This class serves as the base class for all tsdat data pipelines.
IngestPipeline	The IngestPipeline class is designed to read in raw, non-standardized
DSUtil	Provides helper functions for xarray.Dataset
Converter	Base class for converting data arrays from one units to another.
DefaultConverter	Default class for converting units on data arrays. This class utilizes
StringTimeConverter	Convert a time string to a np.datetime64, which is needed for xarray.
TimestampTimeConverter	Convert a numeric UTC timestamp to a np.datetime64, which is needed for

Functions

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

Attemps 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` 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

class tsdat.**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 much 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 intialize 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_DESCRPTION = location_description

INSTRUMENT_NAME = instrument_name

SERIAL_NUMBER = serial_number

INSTRUMENT_DESCRPTION = 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

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 fileystem 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 intantiated 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 fileystem 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.**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 intantiated 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 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 fileystem 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.

FILEHANDLERS :Dict[str, AbstractFileHandler]

static _get_handler (*filename: str*) → *AbstractFileHandler*
Given the name of the file to read or write, this method applies a regular expression to match the name of the file with a handler that has been registered in its internal dictionary of FileHandler objects and returns the appropriate FileHandler, or None if a match is not found.

Parameters **filename** (*str*) – The name of the file whose handler should be retrieved.

Returns The FileHandler registered for use with the provided filename.

Return type *AbstractFileHandler*

static write (*ds: xarray.Dataset*, *filename: str*, *config:* tsdat.config.Config = *None*, ***kwargs*) → None
Saves the given dataset to file using the registered FileHandler for the provided filename.

Parameters

- **ds** (*xr.Dataset*) – The dataset ot save.
- **filename** (*str*) – The path to where the file should be written to.
- **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 using the registered FileHandler for the provided filename.

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

Returns A xr.Dataset object.

Return type xr.Dataset

static register_file_handler (*patterns: Union[str, List[str]]*, *handler:* AbstractFileHandler)
Static method to register an AbstractFileHandler for one or more file patterns. Once an AbstractFileHandler has been registered it may be used by this class to read or write files whose paths match one or more pattern(s) provided in registration.

Parameters

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

- **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 definiton 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 hte 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 provdided 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]]*) → None
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 it 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 QualityManagement 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]

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.**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 formated 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 'yyyymmdd' 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*) → 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*) → 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 datstream 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

CHAPTER

EIGHT

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

CHAPTER

NINE

ACKNOWLEDGEMENTS

tsdat was developed by Carina Lansing[1] and Maxwell Levin[1] with support and management from Chitra Sivaraman[1] 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 Fao[2], Calum Kenny[2], Raghavendra Krishnamurthy[1], Yangchao (Nino) Lin[1], and Eddie Schuman[1] for their feedback, testing, and support early on during the development of tsdat.

[1] Pacific Northwest National Laboratory

[2] National Renewable Energy Laboratory

CHAPTER

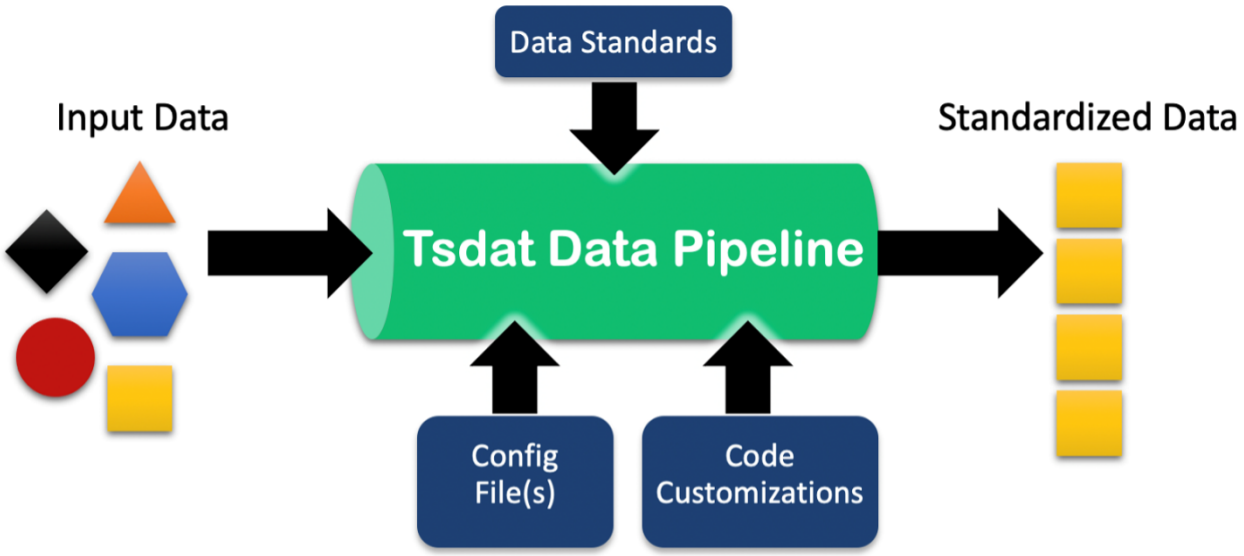
TEN

TSDAT

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.

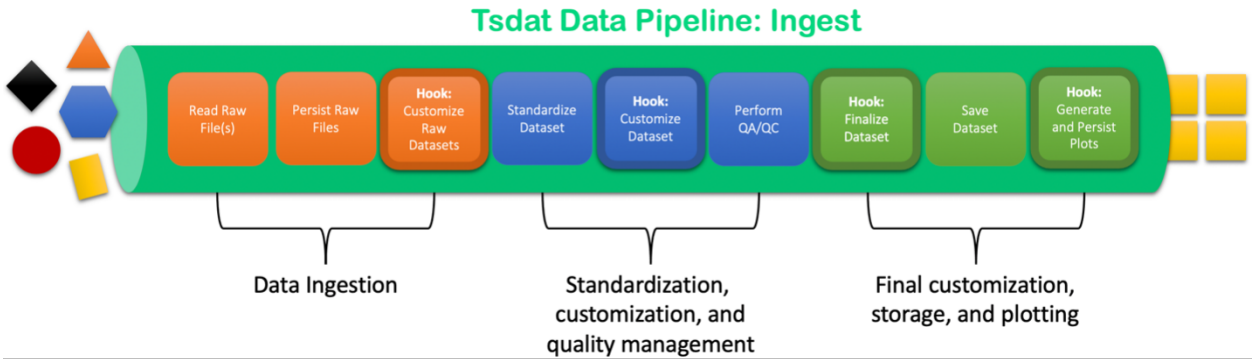
10.1 Quick Overview

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

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W