satpy documentation

Release 0+unknown

SMHI

Mar 15, 2019

Contents

SatPy is a python library for reading and manipulating meteorological remote sensing data and writing it to various image and data file formats. SatPy comes with the ability to make various RGB composites directly from satellite instrument channel data or higher level processing output. The pyresample package is used to resample data to different uniform areas or grids. Various atmospheric corrections and visual enhancements are also provided, either directly in SatPy or from those in the PySpectral and TrollImage packages.

Go to the project page for source code and downloads.

It is designed to be easily extendable to support any meteorological satellite by the creation of plugins (readers, compositors, writers, etc). The table at the bottom of this page shows the input formats supported by the base SatPy installation.

Note: SatPy's interfaces are not guaranteed stable and may change until version 1.0 when backwards compatibility will be a main focus.

CHAPTER 1

Overview

SatPy is designed to provide easy access to common operations for processing meteorological remote sensing data. Any details needed to perform these operations are configured internally to SatPy meaning users should not have to worry about *how* something is done, only ask for what they want. Most of the features provided by SatPy can be configured by keyword arguments (see the *API Documentation* or other specific section for more details). For more complex customizations or added features SatPy uses a set of configuration files that can be modified by the user. The various components and concepts of SatPy are described below. The *Quickstart* guide also provides simple example code for the available features of SatPy.

1.1 Scene

SatPy provides most of its functionality through the `Scene` class. This acts as a container for the datasets being operated on and provides methods for acting on those datasets. It attempts to reduce the amount of low-level knowledge needed by the user while still providing a pythonic interface to the functionality underneath.

A Scene object represents a single geographic region of data, typically at a single continuous time range. It is possible to combine Scenes to form a Scene with multiple regions or multiple time observations, but it is not guaranteed that all functionality works in these situations.

1.2 DataArrays

SatPy's lower-level container for data is the `xarray.DataArray`. For historical reasons DataArrays are often referred to as "Datasets" in SatPy. These objects act similar to normal numpy arrays, but add additional metadata and attributes for describing the data. Metadata is stored in a `.attrs` dictionary and named dimensions can be accessed in a `.dims` attribute, along with other attributes. In most use cases these objects can be operated on like normal NumPy arrays with special care taken to make sure the metadata dictionary contains expected values. See the XArray documentation for more info on handling `xarray.DataArray` objects.

Additionally, SatPy uses a special form of DataArrays where data is stored in `dask.array.Array` objects which allows SatPy to perform multi-threaded lazy operations vastly improving the performance of processing. For help on developing with dask and xarray see *Migrating to xarray and dask* or the documentation for the specific project.

To uniquely identify `DataArray` objects SatPy uses *DatasetID*. A `DatasetID` consists of various pieces of available metadata. This usually includes *name* and *wavelength* as identifying metadata, but also includes *resolution*, *calibration*, *polarization*, and additional *modifiers* to further distinguish one dataset from another.

Warning: XArray includes other object types called "Datasets". These are different from the "Datasets" mentioned in SatPy.

1.3 Reading

One of the biggest advantages of using SatPy is the large number of input file formats that it can read. It encapsulates this functionality into individual *Readers*. SatPy Readers handle all of the complexity of reading whatever format they represent. Meteorological Satellite file formats can be extremely complex and formats are rarely reused across satellites or instruments. No matter the format, SatPy's Reader interface is meant to provide a consistent data loading interface while still providing flexibility to add new complex file formats.

1.4 Compositing

Many users of satellite imagery combine multiple sensor channels to bring out certain features of the data. This includes using one dataset to enhance another, combining 3 or more datasets in to an RGB image, or any other combination of datasets. SatPy comes with a lot of common composite combinations built-in and allows the user to request them like any other dataset. SatPy also makes it possible to create your own custom composites and have SatPy treat them like any other dataset. See *Composites* for more information.

1.5 Resampling

Satellite imagery data comes in two forms when it comes to geolocation, native satellite swath coordinates and uniform gridded projection coordinates. It is also common to see the channels from a single sensor in multiple resolutions, making it complicated to combine or compare the datasets. Many use cases of satellite data require the data to be in a certain projection other than the native projection or to have output imagery cover a specific area of interest. SatPy makes it easy to resample datasets to allow for users to combine them or grid them to these projections or areas of interest. SatPy uses the PyTroll *pyresample* package to provide nearest neighbor, bilinear, or elliptical weighted averaging resampling methods. See *Resampling* for more information.

1.6 Enhancements

When making images from satellite data the data has to be manipulated to be compatible with the output image format and still look good to the human eye. SatPy calls this functionality "enhancing" the data, also commonly called scaling or stretching the data. This process can become complicated not just because of how subjective the quality of an image can be, but also because of historical expectations of forecasters and other users for how the data should look. SatPy tries to hide the complexity of all the possible enhancement methods from the user and just provide the best looking image by default. SatPy still makes it possible to customize these procedures, but in most cases it shouldn't be necessary. See the documentation on *Writers* for more information on what's possible for output formats and enhancing images.

1.7 Writing

SatPy is designed to make data loading, manipulating, and analysis easy. However, the best way to get satellite imagery data out to as many users as possible is to make it easy to save it in multiple formats. SatPy allows users to save data in image formats like PNG or GeoTIFF as well as data file formats like NetCDF. Each format's complexity is hidden behind the interface of individual Writer objects and includes keyword arguments for accessing specific format features like compression and output data type. See the *Writers* documentation for the available writers and how to use them.

CHAPTER 2

Installation Instructions

2.1 Pip-based Installation

SatPy is available from the Python Packaging Index (PyPI). A sandbox environment for *satpy* can be created using Virtualenv.

To install the *satpy* package and the minimum amount of python dependencies:

```
$ pip install satpy
```

Additional dependencies can be installed as "extras" and are grouped by reader, writer, or feature added. Extras available can be found in the setup.py file. They can be installed individually:

```
$ pip install satpy[viirs_sdr]
```

Or all at once, although this isn't recommended due to the large number of dependencies:

```
$ pip install satpy[all]
```

2.2 Conda-based Installation

Starting with version 0.9, SatPy is available from the conda-forge channel. If you have not configured your conda environment to search conda-forge already then do:

```
$ conda config --add channels conda-forge
```

Then to install SatPy in to your current environment run:

```
$ conda install satpy
```

Note: SatPy only automatically installs the dependencies needed to process the most common use cases. Additional dependencies may need to be installed with conda or pip if import errors are encountered.

2.3 Ubuntu System Python Installation

To install SatPy on an Ubuntu system we recommend using virtual environments to separate SatPy and its dependencies from the rest of the system. Note that these instructions require using "sudo" privileges which may not be available to all users and can be very dangerous. The following instructions attempt to install some SatPy dependencies using the Ubuntu *apt* package manager to ease installation. Replace */path/to/pytroll-env* with the environment to be created.

```
$ sudo apt-get install python-pip python-gdal
$ sudo pip install virtualenv
$ virtualenv /path/to/pytroll-env
$ source /path/to/pytroll-env/bin/activate
$ pip install satpy
```

CHAPTER 3

Downloading Data

One of the main features of SatPy is its ability to read various satellite data formats. However, it does not currently provide any functionality for downloading data from any remote sources. SatPy assumes all data is available through the local system, either as a local directory or network mounted file systems. Certain readers that use `xarray` to open data files may be able to load files from remote systems by using OpenDAP or similar protocols.

As a user there are two options for getting access to data:

1. Download data to your local machine.
2. Connect to a remote system that already has access to data.

The most common case of a remote system having access to data is with a cloud computing service like Google Cloud Platform (GCP) or Amazon Web Services (AWS). Another possible case is an organization having direct broadcast antennas where they receive data directly from the satellite or satellite mission organization (NOAA, NASA, EUMETSAT, etc). In these cases data is usually available as a mounted network file system and can be accessed like a normal local path (with the added latency of network communications).

Below are some data sources that provide data that can be read by SatPy. If you know of others please let us know by either creating a GitHub issue or pull request.

3.1 NOAA GOES on Amazon Web Services

- Resource Description
- Data Browser
- Associated Readers: `abi_l1b`

In addition ot the pages above, Brian Blaylock has prepared some instructions for using the `rclone` tool for downloading AWS data to a local machine. The instructions can be found here.

3.2 NOAA GOES on Google Cloud Platform

3.2.1 GOES-16

- Resource Description
- Data Browser
- Associated Readers: `abi_l1b`

3.2.2 GOES-17

- Resource Description
- Data Browser
- Associated Readers: `abi_l1b`

3.3 NOAA CLASS

- Data Ordering
- Associated Readers: `viirs_sdr`

3.4 NASA VIIRS Atmosphere SIPS

- Resource Description
- Associated Readers: `viirs_l1b`

3.5 EUMETSAT Data Center

- Data Ordering

CHAPTER 4

Examples

SatPy examples are available as Jupyter Notebooks on the pytroll-examples git repository. They include python code, PNG images, and descriptions of what the example is doing. Below is a list of some of the examples and a brief summary. Additional example can be found at the repository mentioned above or as explanations in the various sections of this documentation.

Name	Description
Quickstart with MSG data	Satpy quickstart for loading and processing satellite data, with MSG data in this examples
Cartopy Plot	Plot a single VIIRS SDR granule using Cartopy and matplotlib
Himawari-8 AHI True Color	Generate and resample a rayleigh corrected true color RGB from Himawari-8 AHI data
Sentinel-3 OLCI True Color	Reading OLCI data from Sentinel 3 with Pytroll/Satpy
Sentinel 2 MSI true color	Reading MSI data from Sentinel 2 with Pytroll/SatPy
Suomi-NPP VIIRS SDR True Color	Generate a rayleigh corrected true color RGB from VIIRS I- and M-bands
Aqua/Terra MODIS True Color	Generate and resample a rayleigh corrected true color RGB from MODIS
Sentinel 1 SAR-C False Color	Generate a false color composite RGB from SAR-C polarized datasets
Level 2 EARS-NWC cloud products	Reading Level 2 EARS-NWC cloud products
Level 2 MAIA cloud products	Reading Level 2 MAIA cloud products

CHAPTER 5

Quickstart

5.1 Loading and accessing data

To work with weather satellite data you must create a *Scene* object. SatPy does not currently provide an interface to download satellite data, it assumes that the data is on a local hard disk already. In order for SatPy to get access to the data the Scene must be told what files to read and what *SatPy Reader* should read them:

```
>>> from satpy import Scene
>>> from glob import glob
>>> filenames = glob("/home/a001673/data/satellite/Meteosat-10/seviri/lvl1.5/2015/04/
↪20/HRIT/*201504201000*")
>>> global_scene = Scene(reader="seviri_l1b_hrit", filenames=filenames)
```

To load data from the files use the *Scene.load* method. Printing the Scene object will list each of the xarray.DataArray objects currently loaded:

```
>>> global_scene.load([0.6, 0.8, 10.8])
>>> print(global_scene)
<xarray.DataArray 'reshape-d66223a8e05819b890c4535bc7e74356' (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
  * x        (x) float64 5.567e+06 5.564e+06 5.561e+06 5.558e+06 5.555e+06 ...
  * y        (y) float64 -5.567e+06 -5.564e+06 -5.561e+06 -5.558e+06 ...
Attributes:
    satellite_longitude:  0.0
    sensor:               seviri
    satellite_altitude:   35785831.0
    platform_name:        Meteosat-11
    standard_name:        brightness_temperature
    units:                K
    wavelength:           (9.8, 10.8, 11.8)
    satellite_latitude:   0.0
    start_time:           2018-02-28 15:00:10.814000
    end_time:             2018-02-28 15:12:43.956000
```

(continues on next page)

(continued from previous page)

```
    area:                 Area ID: some_area_name\nDescription: On-the-fly ar...
    name:                 IR_108
    resolution:           3000.40316582
    calibration:          brightness_temperature
    polarization:         None
    level:                None
    modifiers:            ()
    ancillary_variables:  []
<xarray.DataArray 'reshape-1982d32298aca15acb42c481fd74a629' (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
  * x        (x) float64 5.567e+06 5.564e+06 5.561e+06 5.558e+06 5.555e+06 ...
  * y        (y) float64 -5.567e+06 -5.564e+06 -5.561e+06 -5.558e+06 ...
Attributes:
    satellite_longitude:  0.0
    sensor:               seviri
    satellite_altitude:   35785831.0
    platform_name:        Meteosat-11
    standard_name:        toa_bidirectional_reflectance
    units:                %
    wavelength:           (0.74, 0.81, 0.88)
    satellite_latitude:   0.0
    start_time:           2018-02-28 15:00:10.814000
    end_time:             2018-02-28 15:12:43.956000
    area:                 Area ID: some_area_name\nDescription: On-the-fly ar...
    name:                 VIS008
    resolution:           3000.40316582
    calibration:          reflectance
    polarization:         None
    level:                None
    modifiers:            ()
    ancillary_variables:  []
<xarray.DataArray 'reshape-e86d03c30ce754995ff9da484c0dc338' (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
  * x        (x) float64 5.567e+06 5.564e+06 5.561e+06 5.558e+06 5.555e+06 ...
  * y        (y) float64 -5.567e+06 -5.564e+06 -5.561e+06 -5.558e+06 ...
Attributes:
    satellite_longitude:  0.0
    sensor:               seviri
    satellite_altitude:   35785831.0
    platform_name:        Meteosat-11
    standard_name:        toa_bidirectional_reflectance
    units:                %
    wavelength:           (0.56, 0.635, 0.71)
    satellite_latitude:   0.0
    start_time:           2018-02-28 15:00:10.814000
    end_time:             2018-02-28 15:12:43.956000
    area:                 Area ID: some_area_name\nDescription: On-the-fly ar...
    name:                 VIS006
    resolution:           3000.40316582
    calibration:          reflectance
    polarization:         None
    level:                None
    modifiers:            ()
    ancillary_variables:  []
```

SatPy allows loading file data by wavelengths in micrometers (shown above) or by channel name:

```
>>> global_scene.load(["VIS006", "VIS008", "IR_108"])
```

To have a look at the available channels for loading from your *Scene* object use the *available_dataset_names()* method:

```
>>> global_scene.available_dataset_names()
['HRV',
 'IR_108',
 'IR_120',
 'VIS006',
 'WV_062',
 'IR_039',
 'IR_134',
 'IR_097',
 'IR_087',
 'VIS008',
 'IR_016',
 'WV_073']
```

To access the loaded data use the wavelength or name:

```
>>> print(global_scene[0.6])
```

5.2 Visualizing data

To visualize loaded data in a pop-up window:

```
>>> global_scene.show(0.6)
```

Alternatively if working in a Jupyter notebook the scene can be converted to a geoviews object using the *to_geoviews()* method. The geoviews package is not a requirement of the base satpy install so in order to use this feature the user needs to install the geoviews package himself.

```
>>> import holoviews as hv
>>> import geoviews as gv
>>> import geoviews.feature as gf
>>> gv.extension("bokeh", "matplotlib")
>>> %opts QuadMesh Image [width=600 height=400 colorbar=True] Feature [apply_
↪ranges=False]
>>> %opts Image QuadMesh (cmap='RdBu_r')
>>> gview = global_scene.to_geoviews(vdims=[0.6])
>>> gview[::5,::5] * gf.coastline * gf.borders
```

5.3 Creating new datasets

Calculations based on loaded datasets/channels can easily be assigned to a new dataset:

```
>>> global_scene["ndvi"] = (global_scene[0.8] - global_scene[0.6]) / (global_scene[0.
↪8] + global_scene[0.6])
>>> global_scene.show("ndvi")
```

For more information on loading datasets by resolution, calibration, or other advanced loading methods see the *Readers* documentation.

5.4 Generating composites

SatPy comes with many composite recipes built-in and makes them loadable like any other dataset:

```
>>> global_scene.load(['overview'])
```

To get a list of all available composites for the current scene:

```
>>> global_scene.available_composite_names()
['overview_sun',
 'airmass',
 'natural',
 'night_fog',
 'overview',
 'green_snow',
 'dust',
 'fog',
 'natural_sun',
 'cloudtop',
 'convection',
 'ash']
```

Loading composites will load all necessary dependencies to make that composite and unload them after the composite has been generated.

Note: Some composite require datasets to be at the same resolution or shape. When this is the case the Scene object must be resampled before the composite can be generated (see below).

5.5 Resampling

In certain cases it may be necessary to resample datasets whether they come from a file or are generated composites. Resampling is useful for mapping data to a uniform grid, limiting input data to an area of interest, changing from one projection to another, or for preparing datasets to be combined in a composite (see above). For more details on resampling, different resampling algorithms, and creating your own area of interest see the *Resampling* documentation. To resample a SatPy Scene:

```
>>> local_scene = global_scene.resample("eurol")
```

This creates a copy of the original `global_scene` with all loaded datasets resampled to the built-in "eurol" area. Any composites that were requested, but could not be generated are automatically generated after resampling. The new `local_scene` can now be used like the original `global_scene` for working with datasets, saving them to disk or showing them on screen:

```
>>> local_scene.show('overview')
>>> local_scene.save_dataset('overview', './local_overview.tif')
```

5.6 Saving to disk

To save all loaded datasets to disk as geotiff images:

```
>>> global_scene.save_datasets()
```

To save all loaded datasets to disk as PNG images:

```
>>> global_scene.save_datasets(writer='simple_image')
```

Or to save an individual dataset:

```
>>> global_scene.save_dataset('VIS006', 'my_nice_image.png')
```

Datasets are automatically scaled or "enhanced" to be compatible with the output format and to provide the best looking image. For more information on saving datasets and customizing enhancements see the documentation on *Writers*.

5.7 Troubleshooting

When something goes wrong, a first step to take is check that the latest Version of satpy and its dependencies are installed. Satpy drags in a few packages as dependencies per default, but each reader and writer has it's own dependencies which can be unfortunately easy to miss when just doing a regular *pip install*. To check the missing dependencies for the readers and writers, a utility function called *check_satpy* can be used:

```
>>> from satpy.config import check_satpy
>>> check_satpy()
```

Due to the way SatPy works, producing as many datasets as possible, there are times that behavior can be unexpected but with no exceptions raised. To help troubleshoot these situations log messages can be turned on. To do this run the following code before running any other SatPy code:

```
>>> from satpy.utils import debug_on
>>> debug_on()
```

CHAPTER 6

Readers

SatPy supports reading and loading data from many input file formats and schemes. The *Scene* object provides a simple interface around all the complexity of these various formats through its `load` method. The following sections describe the different way data can be loaded, requested, or added to a Scene object.

6.1 Available Readers

To get a list of available readers use the *available_readers* function:

```
>>> from satpy import available_readers
>>> available_readers()
```

6.2 Filter loaded files

Coming soon...

6.3 Load data

Datasets in SatPy are identified by certain pieces of metadata set during data loading. These include *name*, *wavelength*, *calibration*, *resolution*, *polarization*, and *modifiers*. Normally, once a `Scene` is created requesting datasets by *name* or *wavelength* is all that is needed:

```
>>> from satpy import Scene
>>> scn = Scene(reader="seviri_l1b_hrit", filenames=filenames)
>>> scn.load([0.6, 0.8, 10.8])
>>> scn.load(['IR_120', 'IR_134'])
```

However, in many cases datasets are available in multiple spatial resolutions, multiple calibrations (`brightness_temperature`, `reflectance`, `radiance`, etc), multiple polarizations, or have corrections or other modifiers already applied to them. By default SatPy will provide the version of the dataset with the highest resolution and the highest level of calibration (brightness temperature or reflectance over radiance). It is also possible to request one of these exact versions of a dataset by using the *DatasetID* class:

```
>>> from satpy import DatasetID
>>> my_channel_id = DatasetID(name='IR_016', calibration='radiance')
>>> scn.load([my_channel_id])
>>> print(scn['IR_016'])
```

Or request multiple datasets at a specific calibration, resolution, or polarization:

```
>>> scn.load([0.6, 0.8], resolution=1000)
```

Or multiple calibrations:

```
>>> scn.load([0.6, 10.8], calibrations=['brightness_temperature', 'radiance'])
```

In the above case SatPy will load whatever dataset is available and matches the specified parameters. So the above `load` call would load the `0.6` (a visible/reflectance band) radiance data and `10.8` (an IR band) brightness temperature data.

Note: If a dataset could not be loaded there is no exception raised. You must check the *scn.missing_datasets* property for any `DatasetID` that could not be loaded.

To find out what datasets are available from a reader from the files that were provided to the `Scene` use *available_dataset_ids()*:

```
>>> scn.available_dataset_ids()
```

Or *available_dataset_names()* for just the string names of Datasets:

```
>>> scn.available_dataset_names()
```

6.4 Search for local files

SatPy provides a utility *find_files_and_readers()* for searching for files in a base directory matching various search parameters. This function discovers files based on filename patterns. It returns a dictionary mapping reader name to a list of filenames supported. This dictionary can be passed directly to the *Scene* initialization.

```
>>> from satpy import find_files_and_readers, Scene
>>> from datetime import datetime
>>> my_files = find_files_and_readers(base_dir='/data/viirs_sdrs',
...                                   reader='viirs_sdr',
...                                   start_time=datetime(2017, 5, 1, 18, 1, 0),
...                                   end_time=datetime(2017, 5, 1, 18, 30, 0))
>>> scn = Scene(filenames=my_files)
```

See the *find_files_and_readers()* documentation for more information on the possible parameters.

6.5 Adding a Reader to SatPy

This is described in the developer guide, see *Adding a Custom Reader to SatPy*.

6.6 Implemented readers

6.6.1 xRIT-based readers

HRIT/LRIT format reader

This module is the base module for all HRIT-based formats. Here, you will find the common building blocks for hrit reading.

One of the features here is the on-the-fly decompression of hrit files. It needs a path to the xRITDecompress binary to be provided through the environment variable called XRIT_DECOMPRESS_PATH. When compressed hrit files are then encountered (files finishing with *.C_*), they are decompressed to the system's temporary directory for reading.

SEVIRI HRIT format reader

References

- MSG Level 1.5 Image Data Format Description
- Radiometric Calibration of MSG SEVIRI Level 1.5 Image Data in Equivalent Spectral Blackbody Radiance

HRIT format reader for JMA data

References

JMA HRIT - Mission Specific Implementation http://www.jma.go.jp/jma/jma-eng/satellite/introduction/4_2HRIT.pdf

GOES HRIT format reader

References

LRIT/HRIT Mission Specific Implementation, February 2012 GVARRDL98.pdf 05057_SPE_MSG_LRIT_HRI

HRIT format reader

References

ELECTRO-L GROUND SEGMENT MSU-GS INSTRUMENT, LRIT/HRIT Mission Specific Implementation, February 2012

CHAPTER 7

Composites

Documentation coming soon...

7.1 Modifiers

7.2 Making custom composites

Note: These features will be added to the `Scene` object in the future.

Building custom composites makes use of the `GenericCompositor` class. For example, building an overview composite can be done manually with:

```
>>> from satpy.composites import GenericCompositor
>>> compositor = GenericCompositor("myoverview", "bla", "")
>>> composite = compositor([local_scene[0.6],
...                         local_scene[0.8],
...                         local_scene[10.8]])
>>> from satpy.writers import to_image
>>> img = to_image(composite)
>>> img.invert([False, False, True])
>>> img.stretch("linear")
>>> img.gamma(1.7)
>>> img.show()
```

One important thing to notice is that there is an internal difference between a composite and an image. A composite is defined as a special dataset which may have several bands (like R, G, B bands). However, the data isn't stretched, or clipped or gamma filtered until an image is generated.

To save the custom composite, the following procedure can be used:

1. Create a custom directory for your custom configs.

2. Set it in the environment variable called PPP_CONFIG_DIR.
3. Write config files with your changes only (look at eg satpy/etc/composites/seviri.yaml for inspiration), pointing to the custom module containing your composites. Don't forget to add changes to the enhancement/generic.cfg file too.
4. Put your composites module on the python path.

With that, you should be able to load your new composite directly.

CHAPTER 8

Resampling

SatPy provides multiple resampling algorithms for resampling geolocated data to uniform projected grids. The easiest way to perform resampling in SatPy is through the *Scene* object's *resample()* method. Additional utility functions are also available to assist in resampling data. Below is more information on resampling with SatPy as well as links to the relevant API documentation for available keyword arguments.

8.1 Resampling algorithms

Table 1: Available Resampling Algorithms

Resampler	Description	Related
nearest	Nearest Neighbor	*KDTreeResampler*
ewa	Elliptical Weighted Averaging	*EWAResampler*
native	Native	*NativeResampler*
bilinear	Bilinear	*BilinearResampler*

The resampling algorithm used can be specified with the `resampler` keyword argument and defaults to `nearest`:

```
>>> scn = Scene(...)
>>> euro_scn = global_scene.resample('euro4', resampler='nearest')
```

Warning: Some resampling algorithms expect certain forms of data. For example, the EWA resampling expects polar-orbiting swath data and prefers if the data can be broken in to "scan lines". See the API documentation for a specific algorithm for more information.

8.2 Resampling for comparison and composites

While all the resamplers can be used to put datasets of different resolutions on to a common area, the 'native' resampler is designed to match datasets to one resolution in the dataset's original projection. This is extremely useful when generating composites between bands of different resolutions.

```
>>> new_scn = scn.resample(resampler='native')
```

By default this resamples to the *highest resolution area* (smallest footprint per pixel) shared between the loaded datasets. You can easily specify the lower resolution area:

```
>>> new_scn = scn.resample(scn.min_area(), resampler='native')
```

Providing an area that is neither the minimum or maximum resolution area may work, but behavior is currently undefined.

8.3 Caching for geostationary data

SatPy will do its best to reuse calculations performed to resample datasets, but it can only do this for the current processing and will lose this information when the process/script ends. Some resampling algorithms, like `nearest` and `bilinear`, can benefit by caching intermediate data on disk in the directory specified by *cache_dir* and using it next time. This is most beneficial with geostationary satellite data where the locations of the source data and the target pixels don't change over time.

```
>>> new_scn = scn.resample('euro4', cache_dir='/path/to/cache_dir')
```

See the documentation for specific algorithms to see availability and limitations of caching for that algorithm.

8.4 Create custom area definition

See `pyresample.geometry.AreaDefinition` for information on creating areas that can be passed to the resample method:

```
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> local_scene = global_scene.resample(my_area)
```

8.5 Create dynamic area definition

See `pyresample.geometry.DynamicAreaDefinition` for more information.

Examples coming soon...

8.6 Store area definitions

Area definitions can be added to a custom YAML file (see pyresample's documentation for more information) and loaded using pyresample's utility methods:

```
>>> from pyresample.utils import parse_area_file
>>> my_area = parse_area_file('my_areas.yaml', 'my_area')[0]
```

Examples coming soon...

CHAPTER 9

Writers

SatPy makes it possible to save datasets in multiple formats. For details on additional arguments and features available for a specific Writer see the table below. Most use cases will want to save datasets using the *save_datasets()* method:

```
>>> scn.save_datasets(writer='simple_image')
```

The `writer` parameter defaults to using the `geotiff` writer. One common parameter across almost all Writers is `filename` and `base_dir` to help automate saving files with custom filenames:

```
>>> scn.save_datasets(
...     filename='{name}_{start_time:%Y%m%d_%H%M%S}.tif',
...     base_dir='/tmp/my_ouput_dir')
```

Changed in version 0.10: The *file_pattern* keyword argument was renamed to *filename* to match the *save_dataset* method's keyword argument.

Table 1: SatPy Writers

Description	Writer name	Status
GeoTIFF	*geotiff*	Nominal
Simple Image (PNG, JPEG, etc)	*simple_image*	Nominal
NinJo TIFF (using `pyninjotiff` package)	*ninjotiff*	Nominal
NetCDF (Standard CF)	*cf*	Pre-alpha
AWIPS II Tiled SCMI NetCDF4	`scmi`	Beta

9.1 Available Writers

To get a list of available writers use the *available_writers* function:

```
>>> from satpy import available_writers
>>> available_writers()
```

9.2 Colorizing and Palettizing using user-supplied colormaps

Note: In the future this functionality will be added to the `Scene` object.

It is possible to create single channel "composites" that are then colorized using users' own colormaps. The colormaps are Numpy arrays with shape (num, 3), see the example below how to create the mapping file(s).

This example creates a 2-color colormap, and we interpolate the colors between the defined temperature ranges. Beyond those limits the image clipped to the specified colors.

```
>>> import numpy as np
>>> from satpy.composites import BWCompositor
>>> from satpy.enhancements import colorize
>>> from satpy.writers import to_image
>>> arr = np.array([[0, 0, 0], [255, 255, 255]])
>>> np.save("/tmp/binary_colormap.npy", arr)
>>> compositor = BWCompositor("test", standard_name="colorized_ir_clouds")
>>> composite = compositor((local_scene[10.8], ))
>>> img = to_image(composite)
>>> kwargs = {"palettes": [{"filename": "/tmp/binary_colormap.npy",
...           "min_value": 223.15, "max_value": 303.15}]}
>>> colorize(img, **kwargs)
>>> img.show()
```

Similarly it is possible to use discrete values without color interpolation using *palettize()* instead of *colorize()*.

You can define several colormaps and ranges in the *palettes* list and they are merged together. See trollimage documentation for more information how colormaps and color ranges are merged.

The above example can be used in enhancements YAML config like this:

```
hot_or_cold:
  standard_name: hot_or_cold
  operations:
    - name: colorize
      method: &colorizefun !!python/name:satpy.enhancements.colorize ''
      kwargs:
        palettes:
          - {filename: /tmp/binary_colormap.npy, min_value: 223.15, max_value: 303.15}
```

9.3 Saving multiple Scenes in one go

As mentioned earlier, it is possible to save *Scene* datasets directly using *`save_datasets()`* method. However, sometimes it is beneficial to collect more *Scene*s together and process and save them all at once.

```
>>> from satpy.writers import compute_writer_results
>>> res1 = scn.save_datasets(filename="/tmp/{name}.png",
...                          writer='simple_image',
...                          compute=False)
>>> res2 = scn.save_datasets(filename="/tmp/{name}.tif",
...                          writer='geotiff',
...                          compute=False)
>>> results = [res1, res2]
>>> compute_writer_results(results)
```

CHAPTER 10

MultiScene (Experimental)

Scene objects in SatPy are meant to represent a single geographic region at a specific single instant in time or range of time. This means they are not suited for handling multiple orbits of polar-orbiting satellite data, multiple time steps of geostationary satellite data, or other special data cases. To handle these cases SatPy provides the *MultiScene* class. The below examples will walk through some basic use cases of the MultiScene.

Warning: These features are still early in development and may change overtime as more user feedback is received and more features added.

10.1 Blending Scenes in MultiScene

Scenes contained in a MultiScene can be combined in different ways.

10.1.1 Stacking scenes

The code below uses the *blend()* method of the `MultiScene` object to stack two separate orbits from a VIIRS sensor. By default the `blend` method will use the *stack()* function which uses the first dataset as the base of the image and then iteratively overlays the remaining datasets on top.

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> scenes = [
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_1/*t180*.h5')),
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_2/*t180*.h5'))
... ]
>>> mscn = MultiScene(scenes)
>>> mscn.load(['I04'])
>>> new_mscn = mscn.resample(my_area)
```

(continues on next page)

(continued from previous page)

```
>>> blended_scene = new_mscn.blend()
>>> blended_scene.save_datasets()
```

10.1.2 Timeseries

Using the *blend()* method with the *timeseries()* function will combine multiple scenes from different time slots by time. A single *Scene* with each dataset/channel extended by the time dimension will be returned. If used together with the *to_geoviews()* method, creation of interactive timeseries Bokeh plots is possible.

```
>>> from satpy import Scene, MultiScene
>>> from satpy.multiscene import timeseries
>>> from glob import glob
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> scenes = [
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_1/*t180*.h5')),
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_2/*t180*.h5'))
... ]
>>> mscn = MultiScene(scenes)
>>> mscn.load(['I04'])
>>> new_mscn = mscn.resample(my_area)
>>> blended_scene = new_mscn.blend(blend_function=timeseries)
>>> blended_scene['I04']
<xarray.DataArray (time: 2, y: 1536, x: 6400)>
dask.array<shape=(2, 1536, 6400), dtype=float64, chunksize=(1, 1536, 4096)>
Coordinates:
  * time     (time) datetime64[ns] 2012-02-25T18:01:24.570942 2012-02-25T18:02:49.
↪975797
Dimensions without coordinates: y, x
```

10.2 Saving frames of an animation

The MultiScene can take "frames" of data and join them together in a single animation movie file. Saving animations requires the *imageio* python library and for most available formats the `ffmpeg` command line tool suite should also be installed. The below example saves a series of GOES-EAST ABI channel 1 and channel 2 frames to MP4 movie files. We can use the *MultiScene.from_files* class method to create a *MultiScene* from a series of files. This uses the *group_files()* utility function to group files by start time.

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> mscn = MultiScene.from_files(glob('/data/abi/day_1/*C0[12]*.nc'), reader='abi_l1b
↪')
>>> mscn.load(['C01', 'C02'])
>>> mscn.save_animation('{name}_{start_time:%Y%m%d_%H%M%S}.mp4', fps=2)
```

New in version 0.12: The `from_files` and `group_files` functions were added in SatPy 0.12. See below for an alternative solution.

This will compute one video frame (image) at a time and write it to the MPEG-4 video file. For users with more powerful systems it is possible to use the `client` and `batch_size` keyword arguments to compute multiple frames in parallel using the dask `distributed` library (if installed). See the dask distributed documentation for information on creating a `Client` object. If working on a cluster you may want to use dask jobqueue to take advantage of multiple nodes at a time.

For older versions of SatPy we can manually create the *Scene* objects used. The `glob()` function and for loops are used to group files into Scene objects that, if used individually, could load the data we want. The code below is equivalent to the `from_files` code above:

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> scene_files = []
>>> for time_step in ['1800', '1810', '1820', '1830']:
...     scene_files.append(glob('/data/abi/day_1/*C0[12]*s???????{}*.nc'.format(time_
↪step)))
>>> scenes = [
...     Scene(reader='abi_l1b', filenames=files) for files in sorted(scene_files)
... ]
>>> mscn = MultiScene(scenes)
>>> mscn.load(['C01', 'C02'])
>>> mscn.save_animation('{name}_{start_time:%Y%m%d_%H%M%S}.mp4', fps=2)
```

Warning: GIF images, although supported, are not recommended due to the large file sizes that can be produced from only a few frames.

10.3 Saving multiple scenes

The `MultiScene` object includes a *`save_datasets()`* method for saving the data from multiple Scenes to disk. By default this will operate on one Scene at a time, but similar to the `save_animation` method above this method can accept a dask distributed `Client` object via the `client` keyword argument to compute scenes in parallel (see documentation above). Note however that some writers, like the `geotiff` writer, do not support multi-process operations at this time and will fail when used with dask distributed. To save multiple Scenes use:

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> mscn = MultiScene.from_files(glob('/data/abi/day_1/*C0[12]*.nc'), reader='abi_l1b
↪')
>>> mscn.load(['C01', 'C02'])
>>> mscn.save_datasets(base_dir='/path/for/output')
```

CHAPTER 11

Developer's Guide

The below sections will walk through how to set up a development environment, make changes to the code, and test that they work. See the *How to contribute* section for more information on getting started and contributor expectations. Additional information for developer's can be found at the pages listed below.

11.1 How to contribute

Thank you for considering contributing to SatPy! SatPy's development team is made up of volunteers so any help we can get is very appreciated.

Contributions from users are what keep this community going. We welcome any contributions including bug reports, documentation fixes or updates, bug fixes, and feature requests. By contributing to SatPy you are providing code that everyone can use and benefit from.

The following guidelines will describe how the SatPy project structures its code contributions from discussion to code to package release.

For more information on contributing to open source projects see GitHub's Guide.

11.1.1 What can I do?

- Make sure you have a GitHub account.
- Submit a ticket for your issue, assuming one does not already exist.
- If you're uncomfortable using Git/GitHub, see Learn Git Branching or other online tutorials.
- If you are uncomfortable contributing to an open source project see:
  - How to Contribute to an Open Source Project on GitHub video series
  - Aaron Meurer's Git Workflow
  - How to Contribute to Open Source
- See what issues already exist. Issues marked good first issue or help wanted can be good issues to start with.

- Read the *Developer's Guide* for more details on contributing code.
- Fork the repository on GitHub and install the package in development mode.
- Update the SatPy documentation to make it clearer and more detailed.
- Contribute code to either fix a bug or add functionality and submit a Pull Request.
- Make an example Jupyter Notebook and add it to the available examples.

11.1.2 What if I break something?

Not possible. If something breaks because of your contribution it was our fault. When you submit your changes to be merged as a GitHub Pull Request they will be automatically tested and checked against coding style rules. Before they are merged they are reviewed by at least one maintainer of the SatPy project. If anything needs updating, we'll let you know.

11.1.3 What is expected?

You can expect the SatPy maintainers to help you. We are all volunteers, have jobs, and occasionally go on vacations. We will try our best to answer your questions as soon as possible. We will try our best to understand your use case and add the features you need. Although we strive to make SatPy useful for everyone there may be some feature requests that we can't allow if they would require breaking existing features. Other features may be best for a different package, PyTroll or otherwise. Regardless, we will help you find the best place for your feature and to make it possible to do what you want.

We, the SatPy maintainers, expect you to be patient, understanding, and respectful of both developers and users. SatPy can only be successful if everyone in the community feels welcome. We also expect you to put in as much work as you expect out of us. There is no dedicated PyTroll or SatPy support team, so there may be times when you need to do most of the work to solve your problem (trying different test cases, environments, etc).

Being respectful includes following the style of the existing code for any code submissions. Please follow PEP8 style guidelines and limit lines of code to 80 characters whenever possible and when it doesn't hurt readability. SatPy follows Google Style Docstrings for all code API documentation. When in doubt use the existing code as a guide for how coding should be done.

11.1.4 How do I get help?

The SatPy developers (and all other PyTroll package developers) monitor the:

- Mailing List
- Slack chat (get an invitation)
- GitHub issues

11.1.5 How do I submit my changes?

Any contributions should start with some form of communication (see above) to let the SatPy maintainers know how you plan to help. The larger the contribution the more important direct communication is so everyone can avoid duplicate code and wasted time. After talking to the SatPy developers any additional work like code or documentation changes can be provided as a GitHub Pull Request.

11.1.6 Code of Conduct

SatPy follows the same code of conduct as the PyTroll project. For reference it is copied to this repository in CODE_OF_CONDUCT.md.

As stated in the PyTroll home page, this code of conduct applies to the project space (GitHub) as well as the public space online and offline when an individual is representing the project or the community. Online examples of this include the PyTroll Slack team, mailing list, and the PyTroll twitter account. This code of conduct also applies to in-person situations like PyTroll Contributor Weeks (PCW), conference meet-ups, or any other time when the project is being represented.

Any violations of this code of conduct will be handled by the core maintainers of the project including David Hoese, Martin Raspaud, and Adam Dybbroe. If you wish to report one of the maintainers for a violation and are not comfortable with them seeing it, please contact one or more of the other maintainers to report the violation. Responses to violations will be determined by the maintainers and may include one or more of the following:

- Verbal warning
- Ask for public apology
- Temporary or permanent ban from in-person events
- Temporary or permanent ban from online communication (Slack, mailing list, etc)

For details see the official code of conduct document.

11.2 Migrating to xarray and dask

Many python developers dealing with meteorologic satellite data begin with using NumPy arrays directly. This work usually involves masked arrays, boolean masks, index arrays, and reshaping. Due to the libraries used by SatPy these operations can't always be done in the same way. This guide acts as a starting point for new SatPy developers in transitioning from NumPy's array operations to SatPy's operations, although they are very similar.

To provide the most functionality for users, SatPy uses the xarray library's `DataArray` object as the main representation for its data. DataArray objects can also benefit from the dask library. The combination of these libraries allow SatPy to easily distribute operations over multiple workers, lazy evaluate operations, and keep track additional metadata and coordinate information.

11.2.1 XArray

```
import xarray as xr
```

`XArray's DataArray` is now the standard data structure for arrays in satpy. They allow the array to define dimensions, coordinates, and attributes (that we use for metadata).

To create such an array, you can do for example

```
my_dataarray = xr.DataArray(my_data, dims=['y', 'x'],
                            coords={'x': np.arange(...)},
                            attrs={'sensor': 'olci'})
```

where `my_data` can be a regular numpy array, a numpy memmap, or, if you want to keep things lazy, a dask array (more on dask later). SatPy uses dask arrays with all of its DataArrays.

Dimensions

In satpy, the dimensions of the arrays should include:

- *x* for the x or column or pixel dimension
- *y* for the y or row or line dimension
- *bands* for composites
- *time* can also be provided, but we have limited support for it at the moment. Use metadata for common cases (*start_time*, *end_time*)

Dimensions are accessible through `my_dataarray.dims`. To get the size of a given dimension, use `sizes`:

```
my_dataarray.sizes['x']
```

Coordinates

Coordinates can be defined for those dimensions when it makes sense:

- *x* and *y*: Usually defined when the data's area is an `AreaDefinition`, and they contain the projection coordinates in x and y.
- *bands*: Contain the letter of the color they represent, eg `['R', 'G', 'B']` for an RGB composite.

This allows then to select for example a single band like this:

```
red = my_composite.sel(bands='R')
```

or even multiple bands:

```
red_and_blue = my_composite.sel(bands=['R', 'B'])
```

To access the coordinates of the data array, use the following syntax:

```
x_coords = my_dataarray['x']
my_dataarray['y'] = np.arange(...)
```

Most of the time, satpy will fill the coordinates for you, so you just need to provide the dimension names.

Attributes

To save metadata, we use the `attrs` dictionary.

```
my_dataarray.attrs['platform_name'] = 'Sentinel-3A'
```

Some metadata that should always be present in our dataarrays:

- `area` the area of the dataset. This should be handled in the reader.
- `start_time`, `end_time`
- `sensor`

Operations on DataArrays

DataArrays work with regular arithmetic operation as one would expect of eg numpy arrays, with the exception that using an operator on two DataArrays requires both arrays to share the same dimensions, and coordinates if those are defined.

For mathematical functions like cos or log, use the ufuncs module:

```
import xarray.ufuncs as xu
cos_zen = xu.cos(zen_xarray)
```

Note that the `xu.something` function also work on numpy arrays.

Masking data

In DataArrays, masked data is represented with NaN values. Hence the default type is `float64`, but `float32` works also in this case. XArray can't handle masked data for integer data, but in satpy we try to use the special `_FillValue` attribute (in `.attrs`) to handle this case. If you come across a case where this isn't handled properly, contact us.

Masking data from a condition can be done with:

```
result = my_dataarray.where(my_dataarray > 5)
```

Result is then analogous to my_dataarray, with values lower or equal to 5 replaced by NaNs.

Further reading

http://xarray.pydata.org/en/stable/generated/xarray.DataArray.html#xarray.DataArray

11.2.2 Dask

```
import dask.array as da
```

The data part of the DataArrays we use in satpy are mostly dask Arrays. That allows lazy and chunked operations for efficient processing.

Creation

From a numpy array

To create a dask array from a numpy array, one can call the `from_array()` function:

```
darr = da.from_array(my_numpy_array, chunks=4096)
```

The *chunks* keyword tells dask the size of a chunk of data. If the numpy array is 3-dimensional, the chunk size provide above means that one chunk will be 4096x4096x4096 elements. To prevent this, one can provide a tuple:

```
darr = da.from_array(my_numpy_array, chunks=(4096, 1024, 2))
```

meaning a chunk will be 4096x1024x2 elements in size.

Even more detailed sizes for the chunks can be provided if needed, see the dask documentation.

From memmaps or other lazy objects

To avoid loading the data into memory when creating a dask array, other kinds of arrays can be passed to `from_array()`. For example, a numpy memmap allows dask to know where the data is, and will only be loaded when the actual values need to be computed. Another example is a hdf5 variable read with h5py.

Procedural generation of data

Some procedural generation function are available in dask, eg `meshgrid()`, `arange()`, or `random.random`.

From XArray to Dask and back

Certain operations are easiest to perform on dask arrays by themselves, especially when certain functions are only available from the dask library. In these cases you can operate on the dask array beneath the DataArray and create a new DataArray when done. Note dask arrays do not support in-place operations. In-place operations on xarray DataArrays will reassign the dask array automatically.

```
dask_arr = my_dataarray.data
dask_arr = dask_arr + 1
# ... other non-xarray operations ...
new_dataarr = xr.DataArray(dask_arr, dims=my_dataarray.dims, attrs=my_dataarray.attrs.
↪copy())
```

Or if the operation should be assigned back to the original DataArray (if and only if the data is the same size):

```
my_dataarray.data = dask_arr
```

Operations and how to get actual results

Regular arithmetic operations are provided, and generate another dask array.

```
>>> arr1 = da.random.uniform(0, 1000, size=(1000, 1000), chunks=100)
>>> arr2 = da.random.uniform(0, 1000, size=(1000, 1000), chunks=100)
>>> arr1 + arr2
dask.array<add, shape=(1000, 1000), dtype=float64, chunksize=(100, 100)>
```

In order to compute the actual data during testing, use the `compute()` method. In normal SatPy operations you will want the data to be evaluated as late as possible to improve performance so *compute* should only be used when needed.

```
>>> (arr1 + arr2).compute()
array([[  898.08811639,  1236.96107629,  1154.40255292, ...,
         1537.50752674,  1563.89278664,   433.92598566],
       [ 1657.43843608,  1063.82390257,  1265.08687916, ...,
         1103.90421234,  1721.73564104,  1276.5424228 ],
       [ 1620.11393216,   212.45816261,   771.99348555, ...,
         1675.6561068 ,   585.89123159,   935.04366354],
       ...,
       [ 1533.93265862,  1103.33725432,   191.30794159, ...,
          520.00434673,   426.49238283,  1090.61323471],
       [  816.6108554 ,  1526.36292498,   412.91953023, ...,
          982.71285721,   699.087645  ,  1511.67447362],
       [ 1354.6127365 ,  1671.24591983,  1144.64848757, ...,
         1247.37586051,  1656.50487092,   978.28184726]])
```

Dask also provides *cos*, *log* and other mathematical function, that you can use with `da.cos` and `da.log`. However, since satpy uses xarrays as standard data structure, prefer the xarray functions when possible (they call in turn the dask counterparts when possible).

Wrapping non-dask friendly functions

Some operations are not supported by dask yet or are difficult to convert to take full advantage of dask's multithreaded operations. In these cases you can wrap a function to run on an entire dask array when it is being computed and pass on the result. Note that this requires fully computing all of the dask inputs to the function and are passed as a numpy array or in the case of an XArray DataArray they will be a DataArray with a numpy array underneath. You should *NOT* use dask functions inside the delayed function.

```
import dask
import dask.array as da

def _complex_operation(my_arr1, my_arr2):
    return my_arr1 + my_arr2

delayed_result = dask.delayed(_complex_operation)(my_dask_arr1, my_dask_arr2)
# to create a dask array to use in the future
my_new_arr = da.from_delayed(delayed_result, dtype=my_dask_arr1.dtype, shape=my_dask_
↪arr1.shape)
```

Dask Delayed objects can also be computed `delayed_result.compute()` if the array is not needed or if the function doesn't return an array.

http://dask.pydata.org/en/latest/array-api.html#dask.array.from_delayed

Map dask blocks to non-dask friendly functions

If the complicated operation you need to perform can be vectorized and does not need the entire data array to do its operations you can use `da.map_blocks` to get better performance than creating a delayed function. Similar to delayed functions the inputs to the function are fully computed DataArrays or numpy arrays, but only the individual chunks of the dask array at a time. Note that `map_blocks` must be provided dask arrays and won't function properly on XArray DataArrays. It is recommended that the function object passed to `map_blocks` **not** be an internal function (a function defined inside another function) or it may be unserializable and can cause issues in some environments.

```
my_new_arr = da.map_blocks(_complex_operation, my_dask_arr1, my_dask_arr2, dtype=my_
↪dask_arr1.dtype)
```

Helpful functions

- `map_blocks()`
- `map_overlap()`
- `atop()`
- `store()`
- `tokenize()`
- `compute()`
- Delayed
- `rechunk()`

- vindex

11.3 Adding a Custom Reader to SatPy

In order to add a reader to satpy, you will need to create two files:

- a YAML file for describing the files to read and the datasets that are available
- a python file implementing the actual reading of the datasets and metadata

For this tutorial, we will implement a reader for the Eumetsat NetCDF format for SEVIRI data

11.3.1 Naming your reader

SatPy tries to follow a standard scheme for naming its readers. These names are used in filenames, but are also used by users so it is important that the name be recognizable and clear. Although some special cases exist, most fit in to the following naming scheme:

```
<sensor>[_<processing level>[_<level detail>]][_<file format>]
```

All components of the name should be lowercase and use underscores as the main separator between fields. Hyphens should be used as an intra-field separator if needed (ex. goes-imager).

sensor The first component of the name represents the sensor or instrument that observed the data stored in the files being read. If the files are the output of a specific processing software or a certain algorithm implementation that supports multiple sensors then a lowercase version of that software's name should be used (e.g. clavrx for CLAVR-x, nucaps for NUCAPS). The `sensor` field is the only required field of the naming scheme. If it is actually an instrument name then the reader name should include one of the other optional fields. If sensor is a software package then that may be enough without any additional information to uniquely identify the reader.

processing level This field marks the specific level of processing or calibration that has been performed to produce the data in the files being read. Common values of this field include: `sdr` for Sensor Data Record (SDR), `edr` for Environmental Data Record (EDR), `l1b` for Level 1B, and `l2` for Level 2.

level detail In cases where the processing level is not enough to completely define the reader this field can be used to provide a little more context. For example, some VIIRS EDR products are specific to a particular field of study or type of scientific event, like a flood or cloud product. In these cases the detail field can be added to produce a name like `viirs_edr_flood`. This field shouldn't be used unless processing level is also specified.

file format If the file format of the files is informative to the user or can distinguish one reader from another then this field should be specified. Common format names should be abbreviated following existing abbreviations like *nc* for NetCDF3 or NetCDF4, *hdf* for HDF4, *h5* for HDF5.

The existing *reader's table* can be used for reference. When in doubt, reader names can be discussed in the github pull request when this reader is added to SatPy or a github issue.

11.3.2 The YAML file

The yaml file is composed of three sections:

- the `reader` section, that provides basic parameters for the reader
- the `file_types` section, which gives the patterns of the files this reader can handle

- the `datasets` section, describing the datasets available from this reader

The `reader` section

The `reader` section, that provides basic parameters for the reader.

The parameters to provide in this section are:

- description: General description of the reader
- name: this is the name of the reader, it should be the same as the filename (without the .yaml extension). The naming convention for this is described above in the *Naming your reader* section above.
- sensors: the list of sensors this reader will support
- reader: the metareader to use, in most cases the `FileYAMLReader` is a good choice.

```
reader:
  description: NetCDF4 reader for the Eumetsat MSG format
  name: nc_seviri_l1b
  sensors: [seviri]
  reader: !!python/name:satpy.readers.yaml_reader.FileYAMLReader
```

The `file_types` section

Each file type needs to provide:

- `file_reader`, the class that will handle the files for this reader, that you will implement in the corresponding python file (see next section)
- `file_patterns`, the patterns to match to find files this reader can handle. The syntax to use is basically the same as `format` with the addition of time. See the trollsift package documentation for more details.
- Optionally, a file type can have a `requires` field: it is a list of file types that the current file types needs to function. For example, the HRIT MSG format segment files each need a prologue and epilogue file to be read properly, hence in this case we have added `requires: [HRIT_PRO, HRIT_EPI]` to the file type definition.

```
file_types:
    nc_seviri_l1b:
        file_reader: !!python/name:satpy.readers.nc_seviri_l1b.NCSEVIRIFileHandler
        file_patterns: ['W_XX-EUMETSAT-Darmstadt,VIS+IR+IMAGERY,{satid:4s}+SEVIRI_C_
↪EUMG_{processing_time:%Y%m%d%H%M%S}.nc']
    nc_seviri_l1b_hrv:
        file_reader: !!python/name:satpy.readers.nc_seviri_l1b.NCSEVIRIHRVFileHandler
        file_patterns: ['W_XX-EUMETSAT-Darmstadt,HRV+IMAGERY,{satid:4s}+SEVIRI_C_EUMG_
↪{processing_time:%Y%m%d%H%M%S}.nc']
```

The `datasets` section

The datasets section describes each dataset available in the files. The parameters provided are made available to the methods of the implementing class.

Parameters you can define for example are:

- name
- sensor

- resolution
- wavelength
- polarization
- standard_name: the name used for the dataset, that will be used for knowing what kind of data it is and handle it appropriately
- units: the units of the data, important to get consistent processing across multiple platforms/instruments
- modifiers: what modification have already been applied to the data, eg `sunz_corrected`
- file_type
- coordinates: this tells which datasets to load to navigate the current dataset
- and any other field that is relevant for the reader

This section can be copied and adapted simply from existing seviri readers, like for example the `msg_native` reader.

```
datasets:
  HRV:
    name: HRV
    resolution: 1000.134348869
    wavelength: [0.5, 0.7, 0.9]
    calibration:
      reflectance:
        standard_name: toa_bidirectional_reflectance
        units: "%"
      radiance:
        standard_name: toa_outgoing_radiance_per_unit_wavelength
        units: W m-2 um-1 sr-1
      counts:
        standard_name: counts
        units: count
    file_type: nc_seviri_l1b_hrv

  IR_016:
    name: IR_016
    resolution: 3000.403165817
    wavelength: [1.5, 1.64, 1.78]
    calibration:
      reflectance:
        standard_name: toa_bidirectional_reflectance
        units: "%"
      radiance:
        standard_name: toa_outgoing_radiance_per_unit_wavelength
        units: W m-2 um-1 sr-1
      counts:
        standard_name: counts
        units: count
    file_type: nc_seviri_l1b
    nc_key: 'ch3'

  IR_039:
    name: IR_039
    resolution: 3000.403165817
    wavelength: [3.48, 3.92, 4.36]
    calibration:
      brightness_temperature:
```

(continues on next page)

(continued from previous page)

```
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b
  nc_key: 'ch4'

IR_087:
  name: IR_087
  resolution: 3000.403165817
  wavelength: [8.3, 8.7, 9.1]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

IR_097:
  name: IR_097
  resolution: 3000.403165817
  wavelength: [9.38, 9.66, 9.94]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

IR_108:
  name: IR_108
  resolution: 3000.403165817
  wavelength: [9.8, 10.8, 11.8]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
```

(continues on next page)

(continued from previous page)

```
    file_type: nc_seviri_l1b

  IR_120:
    name: IR_120
    resolution: 3000.403165817
    wavelength: [11.0, 12.0, 13.0]
    calibration:
      brightness_temperature:
        standard_name: toa_brightness_temperature
        units: K
      radiance:
        standard_name: toa_outgoing_radiance_per_unit_wavelength
        units: W m-2 um-1 sr-1
      counts:
        standard_name: counts
        units: count
    file_type: nc_seviri_l1b

  IR_134:
    name: IR_134
    resolution: 3000.403165817
    wavelength: [12.4, 13.4, 14.4]
    calibration:
      brightness_temperature:
        standard_name: toa_brightness_temperature
        units: K
      radiance:
        standard_name: toa_outgoing_radiance_per_unit_wavelength
        units: W m-2 um-1 sr-1
      counts:
        standard_name: counts
        units: count
    file_type: nc_seviri_l1b

  VIS006:
    name: VIS006
    resolution: 3000.403165817
    wavelength: [0.56, 0.635, 0.71]
    calibration:
      reflectance:
        standard_name: toa_bidirectional_reflectance
        units: "%"
      radiance:
        standard_name: toa_outgoing_radiance_per_unit_wavelength
        units: W m-2 um-1 sr-1
      counts:
        standard_name: counts
        units: count
    file_type: nc_seviri_l1b

  VIS008:
    name: VIS008
    resolution: 3000.403165817
    wavelength: [0.74, 0.81, 0.88]
    calibration:
      reflectance:
        standard_name: toa_bidirectional_reflectance
```

(continues on next page)

(continued from previous page)

```
      units: "%"
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

WV_062:
  name: WV_062
  resolution: 3000.403165817
  wavelength: [5.35, 6.25, 7.15]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: "K"
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

WV_073:
  name: WV_073
  resolution: 3000.403165817
  wavelength: [6.85, 7.35, 7.85]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: "K"
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b
```

The YAML file is now ready, let's go on with the corresponding python file.

11.3.3 The python file

The python files needs to implement a file handler class for each file type that we want to read. Such a class needs to implement a few methods:

- the `__init__` method, that takes as arguments
  - the filename (string)
  - the filename info (dict) that we get by parsing the filename using the pattern defined in the yaml file
  - the filetype info that we get from the filetype definition in the yaml file

This method can also receive other file handler instances as parameter if the filetype at hand has requirements. (See the explanation in the YAML file filetype section above)

- the `get_dataset` method, which takes as arguments
  - the dataset ID of the dataset to load
  - the dataset info that is the description of the channel in the YAML file

  This method has to return an xarray.DataArray instance if the loading is successful, containing the data and metadata of the loaded dataset, or return None if the loading was unsuccessful.
- the `get_area_def` method, that takes as single argument the dataset ID for which we want the area. For the data that cannot be geolocated with an area definition, the pixel coordinates need to be loadable from `get_dataset` for the resulting scene to be navigated. That is, if the data cannot be geolocated with an area definition then the dataset section should specify `coordinates: [longitude_dataset, latitude_dataset]`
- Optionally, the `get_bounding_box` method can be implemented if filtering files by area is desirable for this data type

On top of that, two attributes need to be defined: `start_time` and `end_time`, that define the start and end times of the sensing.

```
# this is nc_seviri_l1b.py
class NCSEVIRIFileHandler():
    def __init__(self, filename, filename_info, filetype_info):
        super(NCSEVIRIFileHandler, self).__init__(filename, filename_info, filetype_
↪info)
        self.nc = None

    def get_dataset(self, dataset_id, dataset_info):
        if dataset_id.calibration != 'radiance':
            # TODO: implement calibration to reflectance or brightness temperature
            return
        if self.nc is None:
            self.nc = xr.open_dataset(self.filename,
                                      decode_cf=True,
                                      mask_and_scale=True,
                                      chunks={'num_columns_vis_ir': CHUNK_SIZE,
                                              'num_rows_vis_ir': CHUNK_SIZE})
            self.nc = self.nc.rename({'num_columns_vir_ir': 'x', 'num_rows_vir_ir': 'y
↪'})
        dataset = self.nc[dataset_info['nc_key']]
        dataset.attrs.update(dataset_info)
        return dataset

    def get_area_def(self, dataset_id):
        # TODO
        pass

class NCSEVIRIHRVFileHandler():
  # left as an exercise to the reader :)
```

11.4 Coding guidelines

SatPy tries to follow PEP8 style guidelines for all of its python code. We also try to limit lines of code to 80 characters whenever possible and when it doesn't hurt readability. SatPy follows Google Style Docstrings for all code API

documentation. When in doubt use the existing code as a guide for how coding should be done.

SatPy currently supports Python 2.7 and 3.4+. All code should be written to be compatible with these versions.

11.5 Development installation

See the *Installation Instructions* section for basic installation instructions. When it comes time to install SatPy it should be installed from a clone of the git repository and in development mode so that local file changes are automatically reflected in the python environment. We highly recommend making a separate conda environment or virtualenv for development.

First, if you plan on contributing back to the project you should fork the repository and clone your fork. The package can then be installed in development by doing:

```
pip install -e .
```

11.6 Running tests

SatPy tests are written using the python `unittest` module and the tests can be executed by running:

```
python setup.py test
```

11.7 Documentation

SatPy's documentation is built using Sphinx. All documentation lives in the `doc/` directory of the project repository. After editing the source files there the documentation can be generated locally:

```
cd doc
make html
```

The output of the make command should be checked for warnings and errors. If code has been changed (new functions or classes) then the API documentation files should be regenerated before running the above command:

```
sphinx-apidoc -f -T -o source/api ../satpy ../satpy/tests
```

CHAPTER 12

satpy package

12.1 Subpackages

12.1.1 satpy.composites package

Submodules

satpy.composites.abi module

Composite classes for the AHI instrument.

class satpy.composites.abi.**SimulatedGreen**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.GenericCompositor*

A single-band dataset resembles a Green (0.55 µm).

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

satpy.composites.ahi module

Composite classes for the AHI instrument.

class satpy.composites.ahi.**GreenCorrector**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.GenericCompositor*

Corrector of the AHI green band to compensate for the deficit of chlorophyl signal.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

satpy.composites.cloud_products module

Compositors for cloud products.

class satpy.composites.cloud_products.**CloudTopHeightCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.ColormapCompositor*

Colorize with a palette, put cloud-free pixels as black.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

static build_colormap(*palette*, *info*)
Create the colormap from the *raw_palette* and the valid_range.

satpy.composites.crefl_utils module

Shared utilities for correcting reflectance data using the 'crefl' algorithm.

Original code written by Ralph Kuehn with modifications by David Hoese and Martin Raspaud. Ralph's code was originally based on the C crefl code distributed for VIIRS and MODIS.

satpy.composites.crefl_utils.**G_calc**(*zenith*, *a_coeff*)

satpy.composites.crefl_utils.**atm_variables_finder**(*mus*, *muv*, *phi*, *height*, *tau*, *tO3*, *tH2O*, *taustep4sphalb*, *tO2=1.0*)

satpy.composites.crefl_utils.**chand**(*phi*, *muv*, *mus*, *taur*)

satpy.composites.crefl_utils.**csalbr**(*tau*)

satpy.composites.crefl_utils.**find_coefficient_index**(*sensor*, *wavelength_range*, *resolution=0*)

Return index in to coefficient arrays for this band's wavelength.

This function search through the *COEFF_INDEX_MAP* dictionary and finds the first key where the nominal wavelength of *wavelength_range* falls between the minimum wavelength and maximum wavelength of the key. *wavelength_range* can also be the standard name of the band. For example, "M05" for VIIRS or "1" for MODIS.

Parameters

- **sensor** – sensor of band to be corrected
- **wavelength_range** – 3-element tuple of (min wavelength, nominal wavelength, max wavelength)
- **resolution** – resolution of the band to be corrected

Returns index in to coefficient arrays like *aH2O*, *aO3*, etc. None is returned if no matching wavelength is found

satpy.composites.crefl_utils.**get_atm_variables**(*mus*, *muv*, *phi*, *height*, *ah2o*, *bh2o*, *ao3*, *tau*)

satpy.composites.crefl_utils.**get_atm_variables_abi**(*mus*, *muv*, *phi*, *height*, *G_O3*, *G_H2O*, *G_O2*, *ah2o*, *ao2*, *ao3*, *tau*)

satpy.composites.crefl_utils.**get_coefficients**(*sensor*, *wavelength_range*, *resolution=0*)

Parameters

- **sensor** – sensor of the band to be corrected
- **wavelength_range** – 3-element tuple of (min wavelength, nominal wavelength, max wavelength)
- **resolution** – resolution of the band to be corrected

Returns aH2O, bH2O, aO3, taur0 coefficient values

satpy.composites.crefl_utils.**run_crefl**(*refl*, *coeffs*, *lon*, *lat*, *sensor_azimuth*, *sensor_zenith*, *solar_azimuth*, *solar_zenith*, *avg_elevation=None*, *percent=False*, *use_abi=False*)

Run main crefl algorithm.

All input parameters are per-pixel values meaning they are the same size and shape as the input reflectance data, unless otherwise stated.

Parameters

- **reflectance_bands** – tuple of reflectance band arrays
- **coefficients** – tuple of coefficients for each band (see *get_coefficients*)
- **lon** – input swath longitude array
- **lat** – input swath latitude array
- **sensor_azimuth** – input swath sensor azimuth angle array
- **sensor_zenith** – input swath sensor zenith angle array
- **solar_azimuth** – input swath solar azimuth angle array
- **solar_zenith** – input swath solar zenith angle array
- **avg_elevation** – average elevation (usually pre-calculated and stored in CMG-DEM.hdf)
- **percent** – True if input reflectances are on a 0-100 scale instead of 0-1 scale (default: False)

satpy.composites.sar module

Composite classes for the VIIRS instrument.

class satpy.composites.sar.**SARIce**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.GenericCompositor*

The SAR Ice composite.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.sar.**SARIceLegacy**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

The SAR Ice composite, legacy version with dynamic stretching.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.sar.**SARQuickLook**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

The SAR QuickLook composite.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.sar.**SARRGB**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

The SAR RGB composite.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

satpy.composites.sar.**overlay**(*top*, *bottom*, *maxval=None*)
Blending two layers.

from: https://docs.gimp.org/en/gimp-concepts-layer-modes.html

satpy.composites.viirs module

Composite classes for the VIIRS instrument.

class satpy.composites.viirs.**AdaptiveDNB**(**args*, ***kwargs*)
Bases: *satpy.composites.viirs.HistogramDNB*

Adaptive histogram equalized DNB composite.

The logic for this code was taken from Polar2Grid and was originally developed by Eva Schiffer (SSEC).

This composite separates the DNB data in to 3 main regions: Day, Night, and Mixed. Each region is equalized separately to bring out the most information from the region due to the high dynamic range of the DNB data. Optionally, the mixed region can be separated in to multiple smaller regions by using the *mixed_degree_step* keyword.

Initialize the compositor with values from the user or from the configuration file.

Adaptive histogram equalization and regular histogram equalization can be configured independently for each region: day, night, or mixed. A region can be set to use adaptive equalization "always", or "never", or only when there are multiple regions in a single scene "multiple" via the *adaptive_X* keyword arguments (see below).

Parameters

- **adaptive_day** – one of ("always", "multiple", "never") meaning when adaptive equalization is used.
- **adaptive_mixed** – one of ("always", "multiple", "never") meaning when adaptive equalization is used.

- **adaptive_night** – one of ("always", "multiple", "never") meaning when adaptive equalization is used.

class satpy.composites.viirs.**ERFDNB**(**args*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

Equalized DNB composite using the error function (erf).

The logic for this code was taken from Polar2Grid and was originally developed by Curtis Seaman and Steve Miller. The original code was written in IDL and is included as comments in the code below.

class satpy.composites.viirs.**HistogramDNB**(**args*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

Histogram equalized DNB composite.

The logic for this code was taken from Polar2Grid and was originally developed by Eva Schiffer (SSEC).

This composite separates the DNB data in to 3 main regions: Day, Night, and Mixed. Each region is equalized separately to bring out the most information from the region due to the high dynamic range of the DNB data. Optionally, the mixed region can be separated in to multiple smaller regions by using the *mixed_degree_step* keyword.

Initialize the compositor with values from the user or from the configuration file.

Parameters

- **high_angle_cutoff** – solar zenith angle threshold in degrees, values above this are considered "night"
- **low_angle_cutoff** – solar zenith angle threshold in degrees, values below this are considered "day"
- **mixed_degree_step** – Step interval to separate "mixed" region in to multiple parts by default does whole mixed region

class satpy.composites.viirs.**NCCZinke**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

Equalized DNB composite using the Zinke algorithm[1].

References

gain_factor(*theta*)

class satpy.composites.viirs.**ReflectanceCorrector**(**args*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

CREFL modifier

Uses a python rewrite of the C CREFL code written for VIIRS and MODIS.

Initialize the compositor with values from the user or from the configuration file.

If *dem_filename* can't be found or opened then correction is done assuming TOA or sealevel options.

Parameters

- **dem_filename** – path to the ancillary 'averaged heights' file default: CMGDEM.hdf environment override: os.path.join(<SATPY_ANCPATH>, <CREFL_ANCFILENAME>)

[1] Stephan Zinke (2017), A simplified high and near-constant contrast approach for the display of VIIRS day/night band imagery DOI:10.1080/01431161.2017.1338838

- **dem_sds** – variable name to load from the ancillary file

get_angles(*vis*)

class satpy.composites.viirs.**SnowAge**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

Create RGB snow product.

Product is based on method presented at the second CSPP/IMAPP users' meeting at Eumetsat in Darmstadt on 14-16 April 2015

Bernard Bellec snow Look-Up Tables V 1.0 (c) Meteo-France # These Look-up Tables allow you to create the RGB snow product # for SUOMI-NPP VIIRS Imager according to the algorithm # presented at the second CSPP/IMAPP users' meeting at Eumetsat # in Darmstadt on 14-16 April 2015 # The algorithm and the product are described in this # presentation : # http://www.ssec.wisc.edu/meetings/cspp/2015/Agenda%20PDF/Wednesday/Roquet_snow_product_cspp2015.pdf # For further information you may contact # Bernard Bellec at Bernard.Bellec@meteo.fr # or # Pascale Roquet at Pascale.Roquet@meteo.fr

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.viirs.**VIIRSFog**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

satpy.composites.viirs.**histogram_equalization**(*data*, *mask_to_equalize*, *number_of_bins=1000*, *std_mult_cutoff=4.0*, *do_zerotoone_normalization=True*, *valid_data_mask=None*, *clip_limit=None*, *slope_limit=None*, *do_log_scale=False*, *log_offset=None*, *local_radius_px=None*, *out=None*)
Perform a histogram equalization on the data selected by mask_to_equalize. The data will be separated into number_of_bins levels for equalization and outliers beyond +/- std_mult_cutoff*std will be ignored.

If do_zerotoone_normalization is True the data selected by mask_to_equalize will be returned in the 0 to 1 range. Otherwise the data selected by mask_to_equalize will be returned in the 0 to number_of_bins range.

Note: the data will be changed in place.

satpy.composites.viirs.**local_histogram_equalization**(*data*, *mask_to_equalize*, *valid_data_mask=None*, *number_of_bins=1000*, *std_mult_cutoff=3.0*, *do_zerotoone_normalization=True*, *local_radius_px=300*, *clip_limit=60.0*, *slope_limit=3.0*, *do_log_scale=True*, *log_offset=1e-05*, *out=None*)
Equalize the provided data (in the mask_to_equalize) using adaptive histogram equalization.

tiles of width/height (2 * local_radius_px + 1) will be calculated and results for each pixel will be bilinerarly interpolated from the nearest 4 tiles when pixels fall near the edge of the image (there is no adjacent tile) the resultant interpolated sum from the available tiles will be multipled to account for the weight of any missing tiles:

```
pixel total interpolated value = pixel available interpolated value / (1 -␣
↪missing interpolation weight)
```

if `do_zerotoone_normalization` is True the data will be scaled so that all data in the mask_to_equalize falls between 0 and 1; otherwise the data in mask_to_equalize will all fall between 0 and number_of_bins

Returns The equalized data

`satpy.composites.viirs.`**`make_day_night_masks`**(*solarZenithAngle*, *good_mask*, *highAngleCutoff*, *lowAngleCutoff*, *stepsDegrees=None*)

given information on the solarZenithAngle for each point, generate masks defining where the day, night, and mixed regions are

optionally provide the highAngleCutoff and lowAngleCutoff that define the limits of the terminator region (if no cutoffs are given the DEFAULT_HIGH_ANGLE and DEFAULT_LOW_ANGLE will be used)

optionally provide the stepsDegrees that define how many degrees each "mixed" mask in the terminator region should be (if no stepsDegrees is given, the whole terminator region will be one mask)

Module contents

Base classes for composite objects.

class `satpy.composites.`**`Airmass`**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters **`common_channel_mask`** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class `satpy.composites.`**`BWCompositor`**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters **`common_channel_mask`** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class `satpy.composites.`**`CO2Corrector`**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *satpy.composites.CompositeBase*

class `satpy.composites.`**`CloudCompositor`**(*transition_min=258.15*, *transition_max=298.15*, *transition_gamma=3.0*, ***kwargs*)

Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters

- **`transition_min`** (*float*) – Values below or equal to this are clouds -> opaque white
- **`transition_max`** (*float*) – Values above this are cloud free -> transparent
- **`transition_gamma`** (*float*) – Gamma correction to apply at the end

class `satpy.composites.`**`ColorizeCompositor`**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.ColormapCompositor*

A compositor colorizing the data, interpolating the palette colors when needed.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**ColormapCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.GenericCompositor*

A compositor that uses colormaps.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

static build_colormap(*palette*, *dtype*, *info*)

Create the colormap from the *raw_palette* and the valid_range.

class satpy.composites.**CompositeBase**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *satpy.dataset.MetadataObject*

apply_modifier_info(*origin*, *destination*)

check_areas(*data_arrays*)

class satpy.composites.**CompositorLoader**(*ppp_config_dir=None*)

Bases: object

Read composites using the configuration files on disk.

get_compositor(*key*, *sensor_names*)

get_modifier(*key*, *sensor_names*)

load_compositors(*sensor_names*)

Load all compositor configs for the provided sensors.

Parameters **sensor_names** (*list of strings*) – Sensor names that have matching `sensor_name.yaml` config files.

Returns

Where *comps* is a dictionary:

sensor_name -> composite ID -> compositor object

And *mods* is a dictionary:

sensor_name -> modifier name -> (modifier class, modifiers options)

Note that these dictionaries are copies of those cached in this object.

Return type (comps, mods)

load_sensor_composites(*sensor_name*)

Load all compositor configs for the provided sensor.

class satpy.composites.**Convection**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**DayNightCompositor**(*lim_low=85.0*, *lim_high=95.0*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

A compositor that blends a day data with night data.

Collect custom configuration values.

Parameters

- **lim_low** (*float*) – lower limit of Sun zenith angle for the blending of the given channels
- **lim_high** (*float*) – upper limit of Sun zenith angle for the blending of the given channels

class satpy.composites.**DifferenceCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

class satpy.composites.**Dust**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**EffectiveSolarPathLengthCorrector**(*correction_limit=88.0*, ***kwargs*)
Bases: *satpy.composites.SunZenithCorrectorBase*

Special sun zenith correction with the method proposed by Li and Shibata.

(2006): https://doi.org/10.1175/JAS3682.1

In addition to adjusting the provided reflectances by the cosine of the solar zenith angle, this modifier forces all reflectances beyond a solar zenith angle of *max_sza* to 0 to reduce noise in the final data. It also gradually reduces the amount of correction done between `correction_limit` and `max_sza`. If `max_sza` is `None` then a constant correction is applied to zenith angles beyond `correction_limit`.

To set `max_sza` to `None` in a YAML configuration file use:

```
effective_solar_pathlength_corrected:
  compositor: !!python/name:satpy.composites.EffectiveSolarPathLengthCorrector
  max_sza: !!null
  optional_prerequisites:
  - solar_zenith_angle
```

Collect custom configuration values.

Parameters

- **correction_limit** (*float*) – Maximum solar zenith angle to apply the correction in degrees. Pixels beyond this limit have a constant correction applied. Default 88.
- **max_sza** (*float*) – Maximum solar zenith angle in degrees that is considered valid and correctable. Default 95.0.

class satpy.composites.**FillingCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

Make a regular RGB, filling the RGB bands with the first provided dataset's values.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**GenericCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

modes = {1: 'L', 2: 'LA', 3: 'RGB', 4: 'RGBA'}

exception satpy.composites.**IncompatibleAreas**
Bases: Exception

Error raised upon compositing things of different shapes.

exception satpy.composites.**IncompatibleTimes**
Bases: Exception

Error raised upon compositing things from different times.

class satpy.composites.**LuminanceSharpeningCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**NIREmissivePartFromReflectance**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)
Bases: *satpy.composites.NIRReflectance*

class satpy.composites.**NIRReflectance**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

class satpy.composites.**PSPAtmosphericalCorrection**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

class satpy.composites.**PSPRayleighReflectance**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

get_angles(*vis*)

class satpy.composites.**PaletteCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.ColormapCompositor*

A compositor colorizing the data, not interpolating the palette colors.

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**RGBCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**RatioSharpenedRGB**(**args*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

Sharpen RGB bands with ratio of a high resolution band to a lower resolution version.

Any pixels where the ratio is computed to be negative or infinity, it is reset to 1. Additionally, the ratio is limited to 1.5 on the high end to avoid high changes due to small discrepancies in instrument detector footprint. Note that the input data to this compositor must already be resampled so all data arrays are the same shape.

Example

R_lo - 1000m resolution - shape=(2000, 2000) G - 1000m resolution - shape=(2000, 2000) B - 1000m resolution - shape=(2000, 2000) R_hi - 500m resolution - shape=(4000, 4000)

ratio = R_hi / R_lo new_R = R_hi new_G = G * ratio new_B = B * ratio

class satpy.composites.**RealisticColors**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**SandwichCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)
Bases: *satpy.composites.GenericCompositor*

Collect custom configuration values.

Parameters **common_channel_mask** (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**SelfSharpenedRGB**(**args*, ***kwargs*)
Bases: *satpy.composites.RatioSharpenedRGB*

Sharpen RGB with ratio of a band with a strided-version of itself.

Example

R - 500m resolution - shape=(4000, 4000) G - 1000m resolution - shape=(2000, 2000) B - 1000m resolution - shape=(2000, 2000)

ratio = R / four_element_average(R) new_R = R new_G = G * ratio new_B = B * ratio

static four_element_average_dask(*d*)
Average every 4 elements (2x2) in a 2D array

class satpy.composites.**SunZenithCorrector**(*correction_limit=88.0*, ***kwargs*)
Bases: *satpy.composites.SunZenithCorrectorBase*

Standard sun zenith correction using `1 / cos(sunz)`.

In addition to adjusting the provided reflectances by the cosine of the solar zenith angle, this modifier forces all reflectances beyond a solar zenith angle of `max_sza` to 0. It also gradually reduces the amount of correction

done between `correction_limit` and `max_sza`. If `max_sza` is `None` then a constant correction is applied to zenith angles beyond `correction_limit`.

To set `max_sza` to `None` in a YAML configuration file use:

```
sunz_corrected:
  compositor: !!python/name:satpy.composites.SunZenithCorrector
  max_sza: !!null
  optional_prerequisites:
  - solar_zenith_angle
```

Collect custom configuration values.

Parameters

- **correction_limit** (*float*) – Maximum solar zenith angle to apply the correction in degrees. Pixels beyond this limit have a constant correction applied. Default 88.
- **max_sza** (*float*) – Maximum solar zenith angle in degrees that is considered valid and correctable. Default 95.0.

class satpy.composites.**SunZenithCorrectorBase**(*max_sza=95.0*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

Base class for sun zenith correction.

Collect custom configuration values.

Parameters **max_sza** (*float*) – Maximum solar zenith angle in degrees that is considered valid and correctable. Default 95.0.

coszen = <WeakValueDictionary>

satpy.composites.**add_bands**(*data*, *bands*)
Add bands so that they match *bands*

satpy.composites.**check_times**(*projectables*)

satpy.composites.**enhance2dataset**(*dset*)
Apply enhancements to dataset *dset* and return the resulting data array of the image.

satpy.composites.**sub_arrays**(*proj1*, *proj2*)
Substract two DataArrays and combine their attrs.

satpy.composites.**zero_missing_data**(*data1*, *data2*)
Replace NaN values with zeros in data1 if the data is valid in data2.

12.1.2 satpy.enhancements package

Module contents

Enhancements.

satpy.enhancements.**apply_enhancement**(*data*, *func*, *exclude=None*, *separate=False*, *pass_dask=False*)
Apply *func* to the provided data.

Parameters

- **data** (*xarray.DataArray*) – Data to be modified inplace.
- **func** (*callable*) – Function to be applied to an xarray

- **exclude** (*iterable*) – Bands in the 'bands' dimension to not include in the calculations.
- **separate** (*bool*) – Apply *func* one band at a time. Default is False.
- **pass_dask** (*bool*) – Pass the underlying dask array instead of the xarray.DataArray.

satpy.enhancements.**btemp_threshold**(*img*, *min_in*, *max_in*, *threshold*, *threshold_out=None*, ***kwargs*)

Scale data linearly in two separate regions.

This enhancement scales the input data linearly by splitting the data into two regions; min_in to threshold and threshold to max_in. These regions are mapped to 1 to threshold_out and threshold_out to 0 respectively, resulting in the data being "flipped" around the threshold. A default threshold_out is set to *176.0 / 255.0* to match the behavior of the US National Weather Service's forecasting tool called AWIPS.

Parameters

- **img** (*XRImage*) – Image object to be scaled
- **min_in** (*float*) – Minimum input value to scale
- **max_in** (*float*) – Maximum input value to scale
- **threshold** (*float*) – Input value where to split data in to two regions
- **threshold_out** (*float*) – Output value to map the input *threshold* to. Optional, defaults to 176.0 / 255.0.

satpy.enhancements.**cira_stretch**(*img*, ***kwargs*)

Logarithmic stretch adapted to human vision.

Applicable only for visible channels.

satpy.enhancements.**colorize**(*img*, ***kwargs*)

Colorize the given image.

satpy.enhancements.**create_colormap**(*palette*)

Create colormap of the given numpy file, color vector or colormap.

satpy.enhancements.**crefl_scaling**(*img*, ***kwargs*)

satpy.enhancements.**gamma**(*img*, ***kwargs*)

Perform gamma correction.

satpy.enhancements.**invert**(*img*, **args*)

Perform inversion.

satpy.enhancements.**lookup**(*img*, ***kwargs*)

Assign values to channels based on a table.

satpy.enhancements.**palettize**(*img*, ***kwargs*)

Palettize the given image (no color interpolation).

satpy.enhancements.**stretch**(*img*, ***kwargs*)

Perform stretch.

satpy.enhancements.**three_d_effect**(*img*, ***kwargs*)

Create 3D effect using convolution

12.1.3 satpy.readers package

Submodules

satpy.readers.aapp_l1b module

Reader for aapp level 1b data.

Options for loading:

- pre_launch_coeffs (False): use pre-launch coefficients if True, operational otherwise (if available).

http://research.metoffice.gov.uk/research/interproj/nwpsaf/aapp/ NWPSAF-MF-UD-003_Formats.pdf

class satpy.readers.aapp_l1b.**AVHRRAAPPL1BFile**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

calibrate(*dataset_id*, *pre_launch_coeffs=False*, *calib_coeffs=None*)
Calibrate the data

end_time

get_angles(*angle_id*)
Get sun-satellite viewing angles

get_dataset(*key*, *info*)
Get a dataset from the file.

navigate()
Return the longitudes and latitudes of the scene.

read()
Read the data.

shape()

start_time

satpy.readers.aapp_l1b.**create_xarray**(*arr*)

satpy.readers.abi_l1b module

Advance Baseline Imager reader for the Level 1b format

The files read by this reader are described in the official PUG document:

https://www.goes-r.gov/users/docs/PUG-L1b-vol3.pdf

class satpy.readers.abi_l1b.**NC_ABI_L1B**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_area_def(*key*)
Get the area definition of the data at hand.

get_dataset(*key*, *info*)
Load a dataset.

get_shape(*key*, *info*)
Get the shape of the data.

start_time

satpy.readers.acspo module

satpy.readers.ahi_hsd module

Advanced Himawari Imager (AHI) standard format data reader.

References

- Himawari-8/9 Himawari Standard Data User's Guide
- http://www.data.jma.go.jp/mscweb/en/himawari89/space_segment/spsg_ahi.html

Time Information

AHI observations use the idea of a "scheduled" time and an "observation time. The "scheduled" time is when the instrument was told to record the data, usually at a specific and consistent interval. The "observation" time is when the data was actually observed. Scheduled time can be accessed from the *scheduled_time* metadata key and observation time from the *start_time* key.

class satpy.readers.ahi_hsd.**AHIHSDFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.file_handlers.BaseFileHandler*

AHI standard format reader.

Initialize the reader.

calibrate(*data*, *calibration*)

Calibrate the data

convert_to_radiance(*data*)

Calibrate to radiance.

end_time

get_area_def(*dsid*)

get_dataset(*key*, *info*)

read_band(*key*, *info*)

Read the data.

scheduled_time

Time this band was scheduled to be recorded.

start_time

satpy.readers.amsr2_l1b module

Reader for AMSR2 L1B files in HDF5 format.

class satpy.readers.amsr2_l1b.**AMSR2L1BFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.hdf5_utils.HDF5FileHandler*

get_dataset(*ds_id*, *ds_info*)

Get output data and metadata of specified dataset.

get_metadata(*ds_id*, *ds_info*)

get_shape(*ds_id*, *ds_info*)
Get output shape of specified dataset.

satpy.readers.avhrr_l1b_gaclac module

Reading and calibrating GAC and LAC avhrr data.

class satpy.readers.avhrr_l1b_gaclac.**GACLACFile**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

Reader for GAC and LAC data.

end_time

get_dataset(*key*, *info*)

start_time

satpy.readers.caliop_l2_cloud module

class satpy.readers.caliop_l2_cloud.**HDF4BandReader**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

CALIOP v3 HDF4 reader.

end_time

get_dataset(*key*, *info*)
Read data from file and return the corresponding projectables.

get_end_time()
Get observation end time from file metadata.

get_filehandle()
Get HDF4 filehandle.

get_lonlats()
Get longitude and latitude arrays from the file.

get_sds_variable(*name*)
Read variable from the HDF4 file.

static parse_metadata_string(*metadata_string*)
Grab end time with regular expression.

start_time

satpy.readers.clavrx module

satpy.readers.electrol_hrit module

HRIT format reader

References

ELECTRO-L GROUND SEGMENT MSU-GS INSTRUMENT, LRIT/HRIT Mission Specific Implementation, February 2012

class satpy.readers.electrol_hrit.**HRITGOMSEpilogueFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.hrit_base.HRITFileHandler*

GOMS HRIT format reader.

Initialize the reader.

read_epilogue()
Read the prologue metadata.

class satpy.readers.electrol_hrit.**HRITGOMSFileHandler**(*filename*, *filename_info*, *filetype_info*, *prologue*, *epilogue*)

Bases: *satpy.readers.hrit_base.HRITFileHandler*

GOMS HRIT format reader.

Initialize the reader.

calibrate(*data*, *calibration*)
Calibrate the data.

get_area_def(*dsid*)
Get the area definition of the band.

get_dataset(*key*, *info*)
Get the data from the files.

class satpy.readers.electrol_hrit.**HRITGOMSPrologueFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.hrit_base.HRITFileHandler*

GOMS HRIT format reader.

Initialize the reader.

process_prologue()
Reprocess prologue to correct types.

read_prologue()
Read the prologue metadata.

satpy.readers.electrol_hrit.**recarray2dict**(*arr*)

satpy.readers.eps_l1b module

Reader for eps level 1b data. Uses xml files as a format description.

class satpy.readers.eps_l1b.**EPSAVHRRFile**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.file_handlers.BaseFileHandler*

Eps level 1b reader for AVHRR data.

end_time

get_bounding_box()
Get the bounding box of the files, as a (lons, lats) tuple.

The tuple return should a lons and lats list of coordinates traveling clockwise around the points available in the file.

get_dataset(*key*, *info*)
Get calibrated channel data.

get_full_angles()
Get the interpolated lons/lats.

get_full_lonlats()
Get the interpolated lons/lats.

get_lonlats()

keys()
List of reader's keys.

platform_name

sensor_name

sensors = {'AVHR': 'avhrr-3'}

spacecrafts = {'M01': 'Metop-B', 'M02': 'Metop-A', 'M03': 'Metop-C'}

start_time

satpy.readers.eps_l1b.**create_xarray**(*arr*)

satpy.readers.eps_l1b.**radiance_to_bt**(*arr*, *wc_*, *a__*, *b__*)
Convert to BT.

satpy.readers.eps_l1b.**radiance_to_refl**(*arr*, *solar_flux*)
Convert to reflectances.

satpy.readers.eps_l1b.**read_raw**(*filename*)
Read *filename* without scaling it afterwards.

satpy.readers.eum_base module

Utilities for EUMETSAT satellite data (HRIT/NATIVE)

satpy.readers.eum_base.**recarray2dict**(*arr*)

satpy.readers.eum_base.**timecds2datetime**(*tcds*)
Method for converting time_cds-variables to datetime-objectsself. Works both with a dictionary and a numpy record_array.

satpy.readers.fci_l1c_fdhsi module

Interface to MTG-FCI Retrieval NetCDF files

class satpy.readers.fci_l1c_fdhsi.**FCIFDHSIFileHandler**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

MTG FCI FDHSI File Reader

calc_area_extent(*key*)
Calculate area extent for a dataset.

calibrate(*data*, *key*)
Data calibration.

end_time

get_area_def(*key*, *info=None*)
Calculate on-fly area definition for 0 degree geos-projection for a dataset.

get_dataset(*key*, *info=None*)
Load a dataset.

start_time

satpy.readers.file_handlers module

class satpy.readers.file_handlers.**BaseFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: object

available_datasets()
Get information of available datasets in file.

This is used for dynamically specifying what datasets are available from a file instead of those listed in a YAML configuration file.

Returns: Iterator of (DatasetID, dict) pairs where dict is the dataset's metadata, similar to that specified in the YAML configuration files.

combine_info(*all_infos*)
Combine metadata for multiple datasets.

When loading data from multiple files it can be non-trivial to combine things like start_time, end_time, start_orbit, end_orbit, etc.

By default this method will produce a dictionary containing all values that were equal across **all** provided info dictionaries.

Additionally it performs the logical comparisons to produce the following if they exist:

- start_time
- end_time
- start_orbit
- end_orbit
- satellite_altitude
- satellite_latitude
- satellite_longitude

Also, concatenate the areas.

end_time

get_area_def(*dsid*)

get_bounding_box()
Get the bounding box of the files, as a (lons, lats) tuple.

The tuple return should a lons and lats list of coordinates traveling clockwise around the points available in the file.

get_dataset(*dataset_id*, *ds_info*)

sensor_names
List of sensors represented in this file.

start_time

satpy.readers.generic_image module

Reader for generic image (e.g. gif, png, jpg, tif, geotiff, ...).

Returns a dataset without calibration. Includes coordinates if available in the file (eg. geotiff).

class satpy.readers.generic_image.**GenericImageFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_area_def(*dsid*)

get_dataset(*key*, *info*)
Get a dataset from the file.

get_geotiff_area_def(*crs*)
Get extents from GeoTIFF file

read()
Read the image

start_time

satpy.readers.generic_image.**get_geotiff_area_def**(*filename*, *crs*)
Read area definition from a geotiff.

satpy.readers.generic_image.**mask_image_data**(*data*)
Mask image data if alpha channel is present.

satpy.readers.geocat module

satpy.readers.ghrsst_l3c_sst module

satpy.readers.goes_imager_hrit module

GOES HRIT format reader

References

LRIT/HRIT Mission Specific Implementation, February 2012 GVARRDL98.pdf 05057_SPE_MSG_LRIT_HRI

exception satpy.readers.goes_imager_hrit.**CalibrationError**
Bases: Exception

class satpy.readers.goes_imager_hrit.**HRITGOESFileHandler**(*filename*, *filename_info*, *filetype_info*, *prologue*)

Bases: *satpy.readers.hrit_base.HRITFileHandler*

GOES HRIT format reader.

Initialize the reader.

calibrate(*data*, *calibration*)
Calibrate the data.

get_area_def(*dsid*)
Get the area definition of the band.

get_dataset(*key*, *info*)
Get the data from the files.

class satpy.readers.goes_imager_hrit.**HRITGOESPrologueFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.hrit_base.HRITFileHandler*

GOES HRIT format reader

Initialize the reader.

process_prologue()
Reprocess prologue to correct types.

read_prologue()
Read the prologue metadata.

satpy.readers.goes_imager_hrit.**make_gvar_float**(*float_val*)

satpy.readers.goes_imager_hrit.**make_sgs_time**(*sgs_time_array*)

satpy.readers.goes_imager_nc module

Reader for GOES 8-15 imager data in netCDF format from NOAA CLASS Also handles GOES 15 data in netCDF format reformated by Eumetsat

GOES Imager netCDF files contain geolocated detector counts. If ordering via NOAA CLASS, select 16 bits/pixel. The instrument oversamples the viewed scene in E-W direction by a factor of 1.75: IR/VIS pixels are 112/28 urad on a side, but the instrument samples every 64/16 urad in E-W direction (see [BOOK-I] and [BOOK-N]).

Important note: Some essential information are missing in the netCDF files, which might render them inappropriate for certain applications. The unknowns are:

1. Subsatellite point
2. Calibration coefficients
3. Detector-scanline assignment, i.e. information about which scanline was recorded by which detector

Items 1. and 2. are not critical because the images are geo-located and NOAA provides static calibration coefficients ([VIS], [IR]). The detector-scanline assignment however cannot be reconstructed properly. This is where an approximation has to be applied (see below).

Calibration

Calibration is performed according to [VIS] and [IR], but with an average calibration coefficient applied to all detectors in a certain channel. The reason for and impact of this approximation is described below.

The GOES imager simultaneously records multiple scanlines per sweep using multiple detectors per channel. The VIS channel has 8 detectors, the IR channels have 1-2 detectors (see e.g. Figures 3-5a/b, 3-6a/b and 3-7/a-b in [BOOK-N]). Each detector has its own calibration coefficients, so in order to perform an accurate calibration, the detector-scanline assignment is needed.

In theory it is known which scanline was recorded by which detector (VIS: 5,6,7,8,1,2,3,4; IR: 1,2). However, the plate on which the detectors are mounted flexes due to thermal gradients in the instrument which leads to a N-S shift

of +/- 8 visible or +/- 2 IR pixels. This shift is compensated in the GVAR scan formation process, but in a way which is hard to reconstruct properly afterwards. See [GVAR], section 3.2.1. for details.

Since the calibration coefficients of the detectors in a certain channel only differ slightly, a workaround is to calibrate each scanline with the average calibration coefficients. A worst case estimate of the introduced error can be obtained by calibrating all possible counts with both the minimum and the maximum calibration coefficients and computing the difference. The maximum differences are:

GOES-8		
Channel	Diff	Unit
00_7	0.0	% # Counts are normalized
03_9	0.187	K
06_8	0.0	K # only one detector
10_7	0.106	K
12_0	0.036	K

GOES-9		
Channel	Diff	Unit
00_7	0.0	% # Counts are normalized
03_9	0.0	K # coefs identical
06_8	0.0	K # only one detector
10_7	0.021	K
12_0	0.006	K

GOES-10		
Channel	Diff	Unit
00_7	1.05	%
03_9	0.0	K # coefs identical
06_8	0.0	K # only one detector
10_7	0.013	K
12_0	0.004	K

GOES-11		
Channel	Diff	Unit
00_7	1.25	%
03_9	0.0	K # coefs identical
06_8	0.0	K # only one detector
10_7	0.0	K # coefs identical
12_0	0.065	K

GOES-12		
Channel	Diff	Unit
00_7	0.8	%
03_9	0.0	K # coefs identical
06_5	0.044	K
10_7	0.0	K # coefs identical
13_3	0.0	K # only one detector

GOES-13		
Channel	Diff	Unit
00_7	1.31	%
03_9	0.0	K # coefs identical
06_5	0.085	K
10_7	0.008	K
13_3	0.0	K # only one detector

GOES-14		
Channel	Diff	Unit
00_7	0.66	%
03_9	0.0	K # coefs identical
06_5	0.043	K
10_7	0.006	K
13_3	0.003	K

GOES-15		
Channel	Diff	Unit
00_7	0.86	%
03_9	0.0	K # coefs identical
06_5	0.02	K
10_7	0.009	K
13_3	0.008	K

References:

- [GVAR] https://goes.gsfc.nasa.gov/text/GVARRDL98.pdf
- [BOOK-N] https://goes.gsfc.nasa.gov/text/GOES-N_Databook/databook.pdf
- [BOOK-I] https://goes.gsfc.nasa.gov/text/databook/databook.pdf
- [IR] https://www.ospo.noaa.gov/Operations/GOES/calibration/gvar-conversion.html
- [VIS] https://www.ospo.noaa.gov/Operations/GOES/calibration/goes-vis-ch-calibration.html
- [FAQ] https://www.ncdc.noaa.gov/sites/default/files/attachments/Satellite-Frequently-Asked-Questions_2.pdf
- [SCHED-W] http://www.ospo.noaa.gov/Operations/GOES/west/imager-routine.html
- [SCHED-E] http://www.ospo.noaa.gov/Operations/GOES/east/imager-routine.html

Eumetsat formated netCDF data:

The main differences are:

1. The geolocation is in a separate file, used for all bands
2. VIS data is calibrated to Albedo (or reflectance)
3. IR data is calibrated to radiance.
4. VIS data is downsampled to IR resolution (4km)
5. File name differs also slightly
6. Data is received via EumetCast

class satpy.readers.goes_imager_nc.**GOESCoefficientReader**(*ir_url*, *vis_url*)
Bases: object

Read GOES Imager calibration coefficients from NOAA reference HTMLs

get_coefs(*platform*, *channel*)

gvar_channels = {'GOES-10': {'00_7': 1, '03_9': 2, '06_8': 3, '10_7': 4, '12_0':

ir_tables = {'GOES-10': '2-3', 'GOES-11': '2-4', 'GOES-12': '2-5a', 'GOES-13': '2-

vis_tables = {'GOES-10': 'Table 2.', 'GOES-11': 'Table 3.', 'GOES-12': 'Table 4.',

class satpy.readers.goes_imager_nc.**GOESEUMGEONCFileHandler**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

File handler for GOES Geolocation data in EUM netCDF format

Initialize the reader.

get_dataset(*key*, *info*)
Load dataset designated by the given key from file

resolution
Specify the spatial resolution of the dataset.

In the EUMETSAT format VIS data is downsampled to IR resolution (4km).

class satpy.readers.goes_imager_nc.**GOESEUMNCFileHandler**(*filename*, *filename_info*, *filetype_info*, *geo_data*)
Bases: *satpy.readers.goes_imager_nc.GOESNCBaseFileHandler*

File handler for GOES Imager data in EUM netCDF format

TODO: Remove datasets which are not available in the file (counts, VIS radiance) via available_datasets() -> See #434

Initialize the reader.

calibrate(*data*, *calibration*, *channel*)
Perform calibration

get_dataset(*key*, *info*)
Load dataset designated by the given key from file

ir_sectors = {(566, 3464): 'Southern Hemisphere (GOES-East)', (1062, 2760): 'Souther

vis_sectors = {(566, 3464): 'Southern Hemisphere (GOES-East)', (1062, 2760): 'Southe

class satpy.readers.goes_imager_nc.**GOESNCBaseFileHandler**(*filename*, *filename_info*, *filetype_info*, *geo_data=None*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

File handler for GOES Imager data in netCDF format

Initialize the reader.

calibrate(*data*, *calibration*, *channel*)
Perform calibration

end_time
End timestamp of the dataset

get_dataset(*key*, *info*)
Load dataset designated by the given key from file

get_shape(*key*, *info*)
Get the shape of the data

Returns Number of lines, number of columns

ir_sectors

meta
Derive metadata from the coordinates

resolution
Specify the spatial resolution of the dataset.

Channel 13_3's spatial resolution changes from one platform to another while the wavelength and file format remain the same. In order to avoid multiple YAML reader definitions for the same file format, read the channel's resolution from the file instead of defining it in the YAML dataset. This information will then be used by the YAML reader to complement the YAML definition of the dataset.

Returns Spatial resolution in kilometers

start_time
Start timestamp of the dataset

vis_sectors

class satpy.readers.goes_imager_nc.**GOESNCFileHandler**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.goes_imager_nc.GOESNCBaseFileHandler*

File handler for GOES Imager data in netCDF format

Initialize the reader.

calibrate(*counts*, *calibration*, *channel*)
Perform calibration

get_dataset(*key*, *info*)
Load dataset designated by the given key from file

ir_sectors = {(566, 3464): 'Southern Hemisphere (GOES-East)', (1062, 2760): 'Southern

vis_sectors = {(2267, 13852): 'Southern Hemisphere (GOES-East)', (4251, 11044): 'Sou

satpy.readers.goes_imager_nc.**test_coefs**(*ir_url*, *vis_url*)
Test calibration coefficients against NOAA reference pages

Currently the reference pages are:

ir_url = https://www.ospo.noaa.gov/Operations/GOES/calibration/gvar-conversion.html vis_url = https://www.ospo.noaa.gov/Operations/GOES/calibration/goes-vis-ch-calibration.html

Parameters

- **ir_url** – Path or URL to HTML page with IR coefficients
- **vis_url** – Path or URL to HTML page with VIS coefficients

Raises ValueError if coefficients don't match the reference

satpy.readers.grib module

satpy.readers.hdf4_utils module

Helpers for reading hdf4-based files.

class satpy.readers.hdf4_utils.**HDF4FileHandler**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

Small class for inspecting a HDF5 file and retrieve its metadata/header data.

collect_metadata(*name*, *obj*)

get(*item*, *default=None*)

satpy.readers.hdf4_utils.**from_sds**(*var*, **args*, ***kwargs*)
Create a dask array from a SD dataset.

satpy.readers.hdf5_utils module

Helpers for reading hdf5-based files.

class satpy.readers.hdf5_utils.**HDF5FileHandler**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

Small class for inspecting a HDF5 file and retrieve its metadata/header data.

collect_metadata(*name*, *obj*)

get(*item*, *default=None*)

satpy.readers.hrit_base module

HRIT/LRIT format reader

This module is the base module for all HRIT-based formats. Here, you will find the common building blocks for hrit reading.

One of the features here is the on-the-fly decompression of hrit files. It needs a path to the xRITDecompress binary to be provided through the environment variable called XRIT_DECOMPRESS_PATH. When compressed hrit files are then encountered (files finishing with .C_), they are decompressed to the system's temporary directory for reading.

class satpy.readers.hrit_base.**HRITFileHandler**(*filename*, *filename_info*, *filetype_info*, *hdr_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

HRIT standard format reader.

Initialize the reader.

end_time

get_area_def(*dsid*)
Get the area definition of the band.

get_area_extent(*size*, *offsets*, *factors*, *platform_height*)
Get the area extent of the file.

get_dataset(*key*, *info*)
Load a dataset.

get_shape(*dsid*, *ds_info*)

get_xy_from_linecol(*line*, *col*, *offsets*, *factors*)
Get the intermediate coordinates from line & col.

Intermediate coordinates are actually the instruments scanning angles.

read_band(*key*, *info*)
Read the data.

start_time

satpy.readers.hrit_base.**decompress**(*infile*, *outdir='.'*)
Decompress an XRIT data file and return the path to the decompressed file.

It expect to find Eumetsat's xRITDecompress through the environment variable XRIT_DECOMPRESS_PATH.

satpy.readers.hrit_base.**get_xritdecompress_cmd**()
Find a valid binary for the xRITDecompress command.

satpy.readers.hrit_base.**get_xritdecompress_outfile**(*stdout*)
Analyse the output of the xRITDecompress command call and return the file.

satpy.readers.hrit_jma module

HRIT format reader for JMA data

References

JMA HRIT - Mission Specific Implementation http://www.jma.go.jp/jma/jma-eng/satellite/introduction/4_2HRIT.pdf

class satpy.readers.hrit_jma.**HRITJMAFileHandler**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.hrit_base.HRITFileHandler*

JMA HRIT format reader.

Initialize the reader.

calibrate(*data*, *calibration*)
Calibrate the data.

get_area_def(*dsid*)
Get the area definition of the band.

get_dataset(*key*, *info*)
Get the dataset designated by *key*.

satpy.readers.hrpt module

Reading and calibrating hrpt avhrr data. Todo: - AMSU - Compare output with AAPP

Reading: http://www.ncdc.noaa.gov/oa/pod-guide/ncdc/docs/klm/html/c4/sec4-1.htm#t413-1

Calibration: http://www.ncdc.noaa.gov/oa/pod-guide/ncdc/docs/klm/html/c7/sec7-1.htm

class satpy.readers.hrpt.**HRPTFile**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

Reader for HRPT Minor Frame, 10 bits data expanded to 16 bits.

end_time

get_dataset(*key*, *info*)

get_lonlats()

get_telemetry()

read()

start_time

satpy.readers.hrpt.**bfield**(*array*, *bit*)
return the bit array.

satpy.readers.hrpt.**geo_interpolate**(*lons32km*, *lats32km*)

satpy.readers.hrpt.**time_seconds**(*tc_array*, *year*)
Return the time object from the timecodes

satpy.readers.iasi_l2 module

IASI L2 HDF5 files.

class satpy.readers.iasi_l2.**IASIL2HDF5**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

File handler for IASI L2 HDF5 files.

end_time

get_dataset(*key*, *info*)
Load a dataset

start_time

satpy.readers.iasi_l2.**read_dataset**(*fid*, *key*)
Read dataset

satpy.readers.iasi_l2.**read_geo**(*fid*, *key*)
Read geolocation and related datasets.

satpy.readers.li_l2 module

Interface to MTG-LI L2 product NetCDF files

The reader is based on preliminary test data provided by EUMETSAT. The data description is described in the "LI L2 Product User Guide [LIL2PUG] Draft version" documentation.

class satpy.readers.li_l2.**LIFileHandler**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

MTG LI File Reader.

end_time

get_area_def(*key*, *info=None*)
Create AreaDefinition for specified product.

Projection information are hard coded for 0 degree geos projection Test dataset doesn't provide the values in the file container. Only fill values are inserted.

get_dataset(*key*, *info=None*, *out=None*, *xslice=None*, *yslice=None*)
Load a dataset

start_time

satpy.readers.maia module

Reader for NWPSAF AAPP MAIA Cloud product.

https://nwpsaf.eu/site/software/aapp/

Documentation reference:

> [NWPSAF-MF-UD-003] DATA Formats [NWPSAF-MF-UD-009] MAIA version 4 Scientific User Manual

class satpy.readers.maia.**MAIAFileHandler**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_dataset(*key*, *info*, *out=None*)
Get a dataset from the file.

get_platform(*platform*)

read(*filename*)

start_time

satpy.readers.modis_l1b module

satpy.readers.msi_safe module

SAFE MSI L1C reader.

class satpy.readers.msi_safe.**SAFEMSIL1C**(*filename*, *filename_info*, *filetype_info*, *mda*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_area_def(*dsid*)

get_dataset(*key*, *info*)
Load a dataset.

start_time

class satpy.readers.msi_safe.**SAFEMSIMDXML**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_area_def(*dsid*)
Get the area definition of the dataset.

get_dataset(*key*, *info*)
Get the dataset refered to by *key*.

interpolate_angles(*angles*, *resolution*)

start_time

satpy.readers.netcdf_utils module

satpy.readers.nucaps module

satpy.readers.nwcsaf_nc module

Nowcasting SAF common PPS&MSG NetCDF/CF format reader

class satpy.readers.nwcsaf_nc.**NcNWCSAF**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

NWCSAF PPS&MSG NetCDF reader.

Init method.

end_time
Return the end time of the object.

get_area_def(*dsid*)
Get the area definition of the datasets in the file.

Only applicable for MSG products!

get_dataset(*dsid*, *info*)
Load a dataset.

remove_timedim(*var*)
Remove time dimension from dataset

scale_dataset(*dsid*, *variable*, *info*)
Scale the data set, applying the attributes from the netCDF file

start_time
Return the start time of the object.

upsample_geolocation(*dsid*, *info*)
Upsample the geolocation (lon,lat) from the tiepoint grid

satpy.readers.nwcsaf_nc.**remove_empties**(*variable*)
Remove empty objects from the *variable*'s attrs.

satpy.readers.olci_nc module

Sentinel-3 OLCI reader

class satpy.readers.olci_nc.**BitFlags**(*value*)
Bases: object

Manipulate flags stored bitwise.

flag_list = ['INVALID', 'WATER', 'LAND', 'CLOUD', 'SNOW_ICE', 'INLAND_WATER', 'TIDAL',

meaning = {'AC_FAIL': 17, 'ADJAC': 14, 'BPAC_ON': 25, 'CLOUD': 3, 'CLOUD_AMBIGUOUS': 2

class satpy.readers.olci_nc.**NCOLCI1B**(*filename*, *filename_info*, *filetype_info*, *cal*)
Bases: *satpy.readers.olci_nc.NCOLCIChannelBase*

get_dataset(*key*, *info*)
Load a dataset.

class satpy.readers.olci_nc.**NCOLCI2**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.olci_nc.NCOLCIChannelBase*

get_dataset (*key*, *info*)
Load a dataset

getbitmask (*wqsf*, *items=[]*)

class satpy.readers.olci_nc.**NCOLCIAngles** (*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

datasets = {'satellite_azimuth_angle': 'OAA', 'satellite_zenith_angle': 'OZA', 'sola

end_time

get_dataset (*key*, *info*)
Load a dataset.

start_time

class satpy.readers.olci_nc.**NCOLCIBase** (*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_dataset (*key*, *info*)
Load a dataset.

start_time

class satpy.readers.olci_nc.**NCOLCICal** (*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.olci_nc.NCOLCIBase*

class satpy.readers.olci_nc.**NCOLCIChannelBase** (*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.olci_nc.NCOLCIBase*

class satpy.readers.olci_nc.**NCOLCIGeo** (*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.olci_nc.NCOLCIBase*

satpy.readers.omps_edr module

Interface to OMPS EDR format

class satpy.readers.omps_edr.**EDREOSFileHandler** (*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.omps_edr.EDRFileHandler*

class satpy.readers.omps_edr.**EDRFileHandler** (*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.hdf5_utils.HDF5FileHandler*

adjust_scaling_factors (*factors*, *file_units*, *output_units*)

end_orbit_number

get_dataset (*dataset_id*, *ds_info*)

get_metadata (*dataset_id*, *ds_info*)

get_shape (*ds_id*, *ds_info*)

platform_name

sensor_name

start_orbit_number

satpy.readers.sar_c_safe module

satpy.readers.scatsat1_l2b module

ScatSat-1 L2B Reader, distributed by Eumetsat in HDF5 format

class satpy.readers.scatsat1_l2b.**SCATSAT1L2BFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.file_handlers.BaseFileHandler*

get_dataset(*key*, *info*)

satpy.readers.scmi module

SCMI NetCDF4 Reader

SCMI files are typically used for data for the ABI instrument onboard the GOES-16/17 satellites. It is the primary format used for providing ABI data to the AWIPS visualization clients used by the US National Weather Service forecasters. The python code for this reader may be reused by other readers as NetCDF schemes/metadata change for different products. The initial reader using this code is the "scmi_abi" reader (see *abi_l1b_scmi.yaml* for more information).

There are two forms of these files that this reader supports:

1. **Official SCMI format: NetCDF4 files where the main data variable is stored** in a variable called "Sectorized_CMI". This variable name can be configured in the YAML configuration file.
2. **SatPy/Polar2Grid SCMI format: NetCDF4 files based on the official SCMI** format created for the Polar2Grid project. This format was migrated to SatPy as part of Polar2Grid's adoption of SatPy for the majority of its features. This format is what is produced by SatPy's *scmi* writer. This format can be identified by a single variable named "data" and a global attribute named `"awips_id"` that is set to a string starting with `"AWIPS_"`.

class satpy.readers.scmi.**SCMIFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.file_handlers.BaseFileHandler*

Handle a single SCMI NetCDF4 file.

end_time

get_area_def(*key*)
Get the area definition of the data at hand.

get_dataset(*key*, *info*)
Load a dataset.

get_shape(*key*, *info*)
Get the shape of the data.

sensor_names
List of sensors represented in this file.

start_time

satpy.readers.seviri_base module

Utilities and eventually also base classes for MSG HRIT/Native data reading

class satpy.readers.seviri_base.**SEVIRICalibrationHandler**
Bases: object

Calibration handler for SEVIRI HRIT- and native-formats.

satpy.readers.seviri_base.**dec10216**(*inbuf*)
Decode 10 bits data into 16 bits words.

```
/*
 * pack 4 10-bit words in 5 bytes into 4 16-bit words
 *
 * 0       1       2       3       4       5
 * 0123456789012345678901234567890123456789
 * 0         1         2         3         4
 */
ip = &in_buffer[i];
op = &out_buffer[j];
op[0] = ip[0]*4 + ip[1]/64;
op[1] = (ip[1] & 0x3F)*16 + ip[2]/16;
op[2] = (ip[2] & 0x0F)*64 + ip[3]/4;
op[3] = (ip[3] & 0x03)*256 +ip[4];
```

satpy.readers.seviri_base.**get_cds_time**(*days*, *msecs*)
Get the datetime object of the time since epoch given in days and milliseconds of day

satpy.readers.seviri_l1b_hrit module

SEVIRI HRIT format reader

References

- MSG Level 1.5 Image Data Format Description
- Radiometric Calibration of MSG SEVIRI Level 1.5 Image Data in Equivalent Spectral Blackbody Radiance

class satpy.readers.seviri_l1b_hrit.**HRITMSGEpilogueFileHandler**(*filename*, *filename_info*, *filetype_info*, *calib_mode='nominal'*, *ext_calib_coefs=None*)
Bases: *satpy.readers.hrit_base.HRITFileHandler*

SEVIRI HRIT epilogue reader.

Initialize the reader.

read_epilogue()
Read the epilogue metadata.

class satpy.readers.seviri_l1b_hrit.**HRITMSGFileHandler**(*filename*, *filename_info*, *filetype_info*, *prologue*, *epilogue*, *calib_mode='nominal'*, *ext_calib_coefs=None*)
Bases: *satpy.readers.hrit_base.HRITFileHandler*, *satpy.readers.seviri_base.SEVIRICalibrationHandler*

SEVIRI HRIT format reader

It is possible to choose between two file-internal calibration coefficients for the conversion from counts to radiances:

- Nominal for all channels (default)
- GSICS for IR channels and nominal for VIS channels

In order to change the default behaviour, use the `reader_kwargs` upon Scene creation:

```
import satpy
import glob

filenames = glob.glob('H-000-MSG3*')
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_hrit',
                    reader_kwargs={'calib_mode': 'GSICS'})
scene.load(['VIS006', 'IR_108'])
```

Furthermore, it is possible to specify external calibration coefficients for the conversion from counts to radiances. They must be specified in [mW m-2 sr-1 (cm-1)-1]. External coefficients take precedence over internal coefficients. If external calibration coefficients are specified for only a subset of channels, the remaining channels will be calibrated using the chosen file-internal coefficients (nominal or GSICS).

In the following example we use external calibration coefficients for the `VIS006` & `IR_108` channels, and nominal coefficients for the remaining channels:

```
coefs = {'VIS006': {'gain': 0.0236, 'offset': -1.20},
         'IR_108': {'gain': 0.2156, 'offset': -10.4}}
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_hrit',
                    reader_kwargs={'ext_calib_coefs': coefs})
scene.load(['VIS006', 'VIS008', 'IR_108', 'IR_120'])
```

In the next example we use we use external calibration coefficients for the `VIS006` & `IR_108` channels, nominal coefficients for the remaining VIS channels and GSICS coefficients for the remaining IR channels:

```
coefs = {'VIS006': {'gain': 0.0236, 'offset': -1.20},
         'IR_108': {'gain': 0.2156, 'offset': -10.4}}
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_hrit',
                    reader_kwargs={'calib_mode': 'GSICS',
                                   'ext_calib_coefs': coefs})
scene.load(['VIS006', 'VIS008', 'IR_108', 'IR_120'])
```

Initialize the reader.

calibrate (*data*, *calibration*)
Calibrate the data.

end_time

get_area_def (*dsid*)
Get the area definition of the band.

get_area_extent (*size*, *offsets*, *factors*, *platform_height*)
Get the area extent of the file.

get_dataset (*key*, *info*)
Load a dataset.

get_xy_from_linecol(*line*, *col*, *offsets*, *factors*)
Get the intermediate coordinates from line & col.

Intermediate coordinates are actually the instruments scanning angles.

start_time

class satpy.readers.seviri_l1b_hrit.**HRITMSGPrologueFileHandler**(*filename*, *filename_info*, *filetype_info*, *calib_mode='nominal'*, *ext_calib_coefs=None*)

Bases: *satpy.readers.hrit_base.HRITFileHandler*

SEVIRI HRIT prologue reader.

Initialize the reader.

read_prologue()
Read the prologue metadata.

satpy.readers.seviri_l1b_hrit.**show**(*data*, *negate=False*)
Show the stretched data.

satpy.readers.seviri_l1b_native module

SEVIRI native format reader.

References

MSG Level 1.5 Native Format File Definition https://www.eumetsat.int/website/wcm/idc/idcplg?IdcService=GET_FILE&dDocName=PDF_FG15_MSG-NATIVE-FORMAT-15&RevisionSelectionMethod=LatestReleased&Rendition=Web MSG Level 1.5 Image Data Format Description https://www.eumetsat.int/website/wcm/idc/idcplg?IdcService=GET_FILE&dDocName=PDF_TEN_05105_MSG_IMG_DATA&RevisionSelectionMethod=LatestReleased&Rendition=Web

class satpy.readers.seviri_l1b_native.**NativeMSGFileHandler**(*filename*, *filename_info*, *filetype_info*, *calib_mode='nominal'*)

Bases: *satpy.readers.file_handlers.BaseFileHandler*, *satpy.readers.seviri_base.SEVIRICalibrationHandler*

SEVIRI native format reader. The Level1.5 Image data calibration method can be changed by adding the required mode to the Scene object instantiation kwargs eg kwargs = {“calib_mode”: “gsics”,}

Initialize the reader.

calibrate(*data*, *dsid*)
Calibrate the data.

end_time

get_area_def(*dsid*)

get_area_extent(*dsid*)

get_dataset(*dsid*, *info*, *xslice=slice(None, None, None)*, *yslice=slice(None, None, None)*)

start_time

satpy.readers.seviri_l1b_native.**get_available_channels**(*header*)
Get the available channels from the header information

satpy.readers.seviri_l1b_native_hdr module

Header and trailer records of SEVIRI native format.

class satpy.readers.seviri_l1b_native_hdr.**GSDTRecords**
Bases: object

MSG Ground Segment Data Type records.

Reference Document (EUM/MSG/SPE/055): MSG Ground Segment Design Specification (GSDS)

gp_cpu_address = [('Qualifier_1', <class 'numpy.uint8'>), ('Qualifier_2', <class 'numpy

gp_fac_env
alias of numpy.uint8

gp_fac_id
alias of numpy.uint8

gp_pk_header = [('HeaderVersionNo', <class 'numpy.uint8'>), ('PacketType', <class 'num

gp_pk_sh1 = [('SubHeaderVersionNo', <class 'numpy.uint8'>), ('ChecksumFlag', <class 'b

gp_sc_id
alias of numpy.uint16

gp_su_id
alias of numpy.uint32

gp_svce_type
alias of numpy.uint8

class satpy.readers.seviri_l1b_native_hdr.**HritPrologue**
Bases: *satpy.readers.seviri_l1b_native_hdr.L15DataHeaderRecord*

get()

class satpy.readers.seviri_l1b_native_hdr.**L15DataHeaderRecord**
Bases: object

Reference Document (EUM/MSG/ICD/105): MSG Level 1.5 Image Data Format Description

celestial_events

geometric_processing

get()

image_acquisition

image_description

impf_configuration

radiometric_processing

satellite_status

class satpy.readers.seviri_l1b_native_hdr.**L15MainProductHeaderRecord**
Bases: object

Reference Document: MSG Level 1.5 Native Format File Definition

get()

class satpy.readers.seviri_l1b_native_hdr.**L15PhData**
Bases: object

l15_ph_data = [('Name', 'S30'), ('Value', 'S50')]

class satpy.readers.seviri_l1b_native_hdr.**L15SecondaryProductHeaderRecord**
Bases: object

Reference Document: MSG Level 1.5 Native Format File Definition

get()

class satpy.readers.seviri_l1b_native_hdr.**Msg15NativeHeaderRecord**
Bases: object

SEVIRI Level 1.5 header for native-format

get()

class satpy.readers.seviri_l1b_native_hdr.**Msg15NativeTrailerRecord**
Bases: object

SEVIRI Level 1.5 trailer for native-format

Reference Document (EUM/MSG/ICD/105): MSG Level 1.5 Image Data Format Description

geometric_quality

get()

image_production_stats

navigation_extraction_results

radiometric_quality

seviri_l15_trailer

timeliness_and_completeness

satpy.readers.seviri_l1b_nc module

satpy.readers.slstr_l1b module

Compact viirs format.

class satpy.readers.slstr_l1b.**NCSLSTR1B**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_dataset(*key*, *info*)
Load a dataset.

start_time

class satpy.readers.slstr_l1b.**NCSLSTRAngles**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_dataset(*key*, *info*)
Load a dataset

start_time

class satpy.readers.slstr_l1b.**NCSLSTRFlag**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_dataset(*key*, *info*)
Load a dataset.

start_time

class satpy.readers.slstr_l1b.**NCSLSTRGeo**(*filename*, *filename_info*, *filetype_info*)
Bases: *satpy.readers.file_handlers.BaseFileHandler*

end_time

get_dataset(*key*, *info*)
Load a dataset

start_time

satpy.readers.utils module

Helper functions for area extent calculations.

satpy.readers.utils.**bbox**(*img*)
Find the bounding box around nonzero elements in the given array

Copied from https://stackoverflow.com/a/31402351/5703449 .

Returns rowmin, rowmax, colmin, colmax

satpy.readers.utils.**get_area_slices**(*data_area*, *area_to_cover*)
Compute the slice to read from an *area* based on an *area_to_cover*.

satpy.readers.utils.**get_geostationary_angle_extent**(*geos_area*)
Get the max earth (vs space) viewing angles in x and y.

satpy.readers.utils.**get_geostationary_bounding_box**(*geos_area*, *nb_points=50*)
Get the bbox in lon/lats of the valid pixels inside *geos_area*.

Parameters **nb_points** – Number of points on the polygon

satpy.readers.utils.**get_geostationary_mask**(*area*)
Compute a mask of the earth's shape as seen by a geostationary satellite

Parameters **area** (*pyresample.geometry.AreaDefinition*) – Corresponding area definition

Returns Boolean mask, True inside the earth's shape, False outside.

satpy.readers.utils.**get_sub_area**(*area*, *xslice*, *yslice*)
Apply slices to the area_extent and size of the area.

satpy.readers.utils.**np2str**(*value*)
Convert an *numpy.string_* to str.

Parameters **value** (*ndarray*) – scalar or 1-element numpy array to convert

Raises ValueError – if value is array larger than 1-element or it is not of type *numpy.string_* or it is not a numpy array

satpy.readers.utils.**unzip_file**(*filename*)
Unzip the file if file is bzipped = ending with 'bz2'

satpy.readers.viirs_compact module

Compact viirs format.

class satpy.readers.viirs_compact.**VIIRSCompactFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.file_handlers.BaseFileHandler*

angles(*azi_name*, *zen_name*)

end_time

get_bounding_box()

Get the bounding box of the files, as a (lons, lats) tuple.

The tuple return should a lons and lats list of coordinates traveling clockwise around the points available in the file.

get_dataset(*key*, *info*)

Load a dataset

navigate()

read_dataset(*dataset_key*, *info*)

read_geo(*key*, *info*)

Read angles.

start_time

satpy.readers.viirs_compact.**expand_array**(*data*, *scans*, *c_align*, *c_exp*, *scan_size=16*, *tpz_size=16*, *nties=200*, *track_offset=0.5*, *scan_offset=0.5*)

Expand *data* according to alignment and expansion.

satpy.readers.viirs_compact.**navigate_dnb**(*h5f*)

satpy.readers.viirs_compact.**read_dnb**(*h5f*)

satpy.readers.viirs_edr_flood module

class satpy.readers.viirs_edr_flood.**VIIRSEDRFlood**(*filename*, *filename_info*, *filetype_info*)

Bases: *satpy.readers.hdf4_utils.HDF4FileHandler*

end_time

get_area_def(*ds_id*)

get_dataset(*ds_id*, *ds_info*)

get_metadata(*data*, *ds_info*)

platform_name

sensor_name

start_time

satpy.readers.viirs_l1b module

satpy.readers.viirs_sdr module

Interface to VIIRS SDR format

This reader implements the support of VIIRS SDR files as produced by CSPP and CLASS. It is comprised of two parts:

- A subclass of the YAMLFileReader class to allow handling all the files
- A filehandler class to implement the actual reading

Format documentation:

- http://npp.gsfc.nasa.gov/science/sciencedocuments/082012/474-00001-03_CDFCBVolIII_RevC.pdf

class satpy.readers.viirs_sdr.**VIIRSSDRFileHandler**(*filename*, *filename_info*, *filetype_info*, *use_tc=None*, ***kwargs*)

Bases: *satpy.readers.hdf5_utils.HDF5FileHandler*

VIIRS HDF5 File Reader

adjust_scaling_factors(*factors*, *file_units*, *output_units*)

concatenate_dataset(*dataset_group*, *var_path*)

end_orbit_number

end_time

static expand_single_values(*var*, *scans*)
Expand single valued variable to full scan lengths.

get_bounding_box()
Get the bounding box of this file.

get_dataset(*dataset_id*, *ds_info*)

get_file_units(*dataset_id*, *ds_info*)

mask_fill_values(*data*, *ds_info*)

platform_name

scale_swath_data(*data*, *scaling_factors*)
Scale swath data using scaling factors and offsets.

Multi-granule (a.k.a. aggregated) files will have more than the usual two values.

sensor_name

start_orbit_number

start_time

class satpy.readers.viirs_sdr.**VIIRSSDRReader**(*config_files*, *use_tc=None*, ***kwargs*)

Bases: *satpy.readers.yaml_reader.FileYAMLReader*

Custom file reader for finding VIIRS SDR geolocation at runtime.

Initialize file reader and adjust geolocation preferences.

Parameters

- **config_files** (*iterable*) – yaml config files passed to base class

- **use_tc** (*boolean*) – If *True* use the terrain corrected files. If *False*, switch to non-TC files. If *None* (default), use TC if available, non-TC otherwise.

filter_filenames_by_info(*filename_items*)
Filter out file using metadata from the filenames.

This sorts out the different lon and lat datasets depending on TC is desired or not.

get_right_geo_fhs(*dsid*, *fhs*)
Find the right geographical file handlers for given dataset ID *dsid*.

satpy.readers.viirs_sdr.**split_desired_other**(*fhs*, *req_geo*, *rem_geo*)
Split the provided filehandlers *fhs* into desired filehandlers and others.

satpy.readers.xmlformat module

Reads a format from an xml file to create dtypes and scaling factor arrays.

class satpy.readers.xmlformat.**XMLFormat**(*filename*)
Bases: object

XMLFormat object.

apply_scales(*array*)
Apply scales to *array*.

dtype(*key*)
Get the dtype for the format object.

satpy.readers.xmlformat.**parse_format**(*xml_file*)
Parse the xml file to create types, scaling factor types, and scales.

satpy.readers.xmlformat.**process_array**(*elt*, *ascii=False*)
Process an 'array' tag.

satpy.readers.xmlformat.**process_delimiter**(*elt*, *ascii=False*)
Process a 'delimiter' tag.

satpy.readers.xmlformat.**process_field**(*elt*, *ascii=False*)
Process a 'field' tag.

satpy.readers.xmlformat.**to_dtype**(*val*)
Parse *val* to return a dtype.

satpy.readers.xmlformat.**to_scaled_dtype**(*val*)
Parse *val* to return a dtype.

satpy.readers.xmlformat.**to_scales**(*val*)
Parse *val* to return an array of scale factors.

satpy.readers.yaml_reader module

class satpy.readers.yaml_reader.**AbstractYAMLReader**(*config_files*)
Bases: object

all_dataset_ids

all_dataset_names

available_dataset_ids

available_dataset_names

end_time
End time of the reader.

filter_selected_filenames(*filenames*)
Filter provided filenames by parameters in reader configuration.

Returns: iterable of usable files

get_dataset_key(*key*, ***kwargs*)
Get the fully qualified *DatasetID* matching *key*.

See *satpy.readers.get_key* for more information about kwargs.

load(*dataset_keys*)
Load *dataset_keys*.

load_ds_ids_from_config()
Get the dataset ids from the config.

select_files_from_directory(*directory=None*)
Find files for this reader in *directory*.

If directory is None or '', look in the current directory.

select_files_from_pathnames(*filenames*)
Select the files from *filenames* this reader can handle.

sensor_names

start_time
Start time of the reader.

supports_sensor(*sensor*)
Check if *sensor* is supported.

Returns True is *sensor* is None.

class satpy.readers.yaml_reader.**FileYAMLReader**(*config_files*, *filter_parameters=None*, *filter_filenames=True*, ***kwargs*)
Bases: *satpy.readers.yaml_reader.AbstractYAMLReader*

Implementation of the YAML reader.

add_ds_ids_from_files()
Check files for more dynamically discovered datasets.

available_dataset_ids

static check_file_covers_area(*file_handler*, *check_area*)
Checks if the file covers the current area.

If the file doesn't provide any bounding box information or 'area' was not provided in *filter_parameters*, the check returns True.

create_filehandlers(*filenames*, *fh_kwargs=None*)
Organize the filenames into file types and create file handlers.

end_time
End time of the reader.

static filename_items_for_filetype(*filenames*, *filetype_info*)
Iterator over the filenames matching *filetype_info*.

filter_fh_by_metadata(*filehandlers*)
Filter out filehandlers using provide filter parameters.

filter_filenames_by_info(*filename_items*)
Filter out file using metadata from the filenames.

Currently only uses start and end time. If only start time is available from the filename, keep all the filename that have a start time before the requested end time.

filter_selected_filenames(*filenames*)
Filter provided filenames by parameters in reader configuration.

Returns: iterable of usable files

find_required_filehandlers(*requirements*, *filename_info*)
Find the necessary file handlers for the given requirements.

We assume here requirements are available.

Raises

- KeyError, if no handler for the given requirements is available.
- RuntimeError, if there is a handler for the given requirements,
- but it doesn't match the filename info.

load(*dataset_keys*, *previous_datasets=None*)
Load *dataset_keys*.

If *previous_datasets* is provided, do not reload those.

metadata_matches(*sample_dict*, *file_handler=None*)

new_filehandler_instances(*filetype_info*, *filename_items*, *fh_kwargs=None*)
Generate new filehandler instances.

new_filehandlers_for_filetype(*filetype_info*, *filenames*, *fh_kwargs=None*)
Create filehandlers for a given filetype.

sensor_names

sorted_filetype_items()
Sort the instance's filetypes in using order.

start_time
Start time of the reader.

time_matches(*fstart*, *fend*)

update_ds_ids_from_file_handlers()
Update DatasetIDs with information from loaded files.

This is useful, for example, if dataset resolution may change depending on what files were loaded.

satpy.readers.yaml_reader.**get_filebase**(*path*, *pattern*)
Get the end of *path* of same length as *pattern*.

satpy.readers.yaml_reader.**listify_string**(*something*)
Takes *something* and make it a list.

something is either a list of strings or a string, in which case the function returns a list containing the string. If *something* is None, an empty list is returned.

satpy.readers.yaml_reader.**match_filenames**(*filenames*, *pattern*)
Get the filenames matching *pattern*.

Module contents

Shared objects of the various reader classes.

class satpy.readers.**DatasetDict**(**args*, ***kwargs*)
Bases: dict

Special dictionary object that can handle dict operations based on dataset name, wavelength, or DatasetID.

Note: Internal dictionary keys are *DatasetID* objects.

contains(*item*)
Check contains when we know the *exact* DatasetID.

get(*key*, *default=None*)
Get value with optional default.

get_key(*match_key*, *num_results=1*, *best=True*, ***dfilter*)
Get multiple fully-specified keys that match the provided query.

Parameters

- **key** (DatasetID) – DatasetID of query parameters to use for searching. Any parameter that is *None* is considered a wild card and any match is accepted. Can also be a string representing the dataset name or a number representing the dataset wavelength.
- **num_results** (*int*) – Number of results to return. If *0* return all, if *1* return only that element, otherwise return a list of matching keys.
- ****dfilter** (*dict*) – See *get_key* function for more information.

getitem(*item*)
Get Node when we know the *exact* DatasetID.

keys() → a set-like object providing a view on D's keys

exception satpy.readers.**TooManyResults**
Bases: KeyError

satpy.readers.**available_readers**(*as_dict=False*)
Available readers based on current configuration.

Parameters **as_dict** (*bool*) – Optionally return reader information as a dictionary. Default: False

Returns: List of available reader names. If *as_dict* is *True* then a list of dictionaries including additionally reader information is returned.

satpy.readers.**configs_for_reader**(*reader=None*, *ppp_config_dir=None*)
Generator of reader configuration files for one or more readers

Parameters

- **reader** (*Optional[str]*) – Yield configs only for this reader
- **ppp_config_dir** (*Optional[str]*) – Additional configuration directory to search for reader configuration files.

Returns: Generator of lists of configuration files

satpy.readers.**filter_keys_by_dataset_id**(*did*, *key_container*)
Filer provided key iterable by the provided *DatasetID*.

Note: The *modifiers* attribute of *did* should be *None* to allow for any modifier in the results.

Parameters

- **did** (DatasetID) – Query parameters to match in the *key_container*.
- **key_container** (*iterable*) – Set, list, tuple, or dict of *DatasetID* keys.

Returns (list): List of keys matching the provided parameters in no specific order.

satpy.readers.**find_files_and_readers**(*start_time=None*, *end_time=None*, *base_dir=None*, *reader=None*, *sensor=None*, *ppp_config_dir=None*, *filter_parameters=None*, *reader_kwargs=None*)

Find on-disk files matching the provided parameters.

Use *start_time* and/or *end_time* to limit found filenames by the times in the filenames (not the internal file metadata). Files are matched if they fall anywhere within the range specified by these parameters.

Searching is **NOT** recursive.

The returned dictionary can be passed directly to the *Scene* object through the *filenames* keyword argument.

The behaviour of time-based filtering depends on whether or not the filename contains information about the end time of the data or not:

- if the end time is not present in the filename, the start time of the filename is used and has to fall between (inclusive) the requested start and end times
- otherwise, the timespan of the filename has to overlap the requested timespan

Parameters

- **start_time** (*datetime*) – Limit used files by starting time.
- **end_time** (*datetime*) – Limit used files by ending time.
- **base_dir** (*str*) – The directory to search for files containing the data to load. Defaults to the current directory.
- **reader** (*str or list*) – The name of the reader to use for loading the data or a list of names.
- **sensor** (*str or list*) – Limit used files by provided sensors.
- **ppp_config_dir** (*str*) – The directory containing the configuration files for SatPy.
- **filter_parameters** (*dict*) – Filename pattern metadata to filter on. *start_time* and *end_time* are automatically added to this dictionary. Shortcut for *reader_kwargs['filter_parameters']*.
- **reader_kwargs** (*dict*) – Keyword arguments to pass to specific reader instances to further configure file searching.

Returns: Dictionary mapping reader name string to list of filenames

satpy.readers.**get_best_dataset_key**(*key*, *choices*)

Choose the "best" *DatasetID* from *choices* based on *key*.

The best key is chosen based on the follow criteria:

1. Central wavelength is nearest to the *key* wavelength if specified.
2. Least modified dataset if *modifiers* is *None* in *key*. Otherwise, the modifiers are ignored.
3. Highest calibration if *calibration* is *None* in *key*. Calibration priority is chosen by *satpy.CALIBRATION_ORDER*.

4. Best resolution (smallest number) if *resolution* is *None* in *key*. Otherwise, the resolution is ignored.

This function assumes *choices* has already been filtered to only include datasets that match the provided *key*.

Parameters

- **key** (DatasetID) – Query parameters to sort *choices* by.
- **choices** (*iterable*) – *DatasetID* objects to sort through to determine the best dataset.

Returns: List of best *DatasetID's from 'choices*. If there is more than one element this function could not choose between the available datasets.

satpy.readers.**get_key**(*key*, *key_container*, *num_results=1*, *best=True*, *resolution=None*, *calibration=None*, *polarization=None*, *level=None*, *modifiers=None*)

Get the fully-specified key best matching the provided key.

Only the best match is returned if *best* is *True* (default). See *get_best_dataset_key* for more information on how this is determined.

The *resolution* and other identifier keywords are provided as a convenience to filter by multiple parameters at once without having to filter by multiple *key* inputs.

Parameters

- **key** (DatasetID) – DatasetID of query parameters to use for searching. Any parameter that is *None* is considered a wild card and any match is accepted.
- **key_container** (*dict or set*) – Container of DatasetID objects that uses hashing to quickly access items.
- **num_results** (*int*) – Number of results to return. Use *0* for all matching results. If *1* then the single matching key is returned instead of a list of length 1. (default: 1)
- **best** (*bool*) – Sort results to get "best" result first (default: True). See *get_best_dataset_key* for details.
- **resolution** (*float, int, or list*) – Resolution of the dataset in dataset units (typically meters). This can also be a list of these numbers.
- **calibration** (*str or list*) – Dataset calibration (ex.'reflectance'). This can also be a list of these strings.
- **polarization** (*str or list*) – Dataset polarization (ex.'V'). This can also be a list of these strings.
- **level** (*number or list*) – Dataset level (ex. 100). This can also be a list of these numbers.
- **modifiers** (*list*) – Modifiers applied to the dataset. Unlike resolution and calibration this is the exact desired list of modifiers for one dataset, not a list of possible modifiers.

Returns (list or DatasetID): Matching key(s)

Raises: KeyError if no matching results or if more than one result is found when *num_results* is *1*.

satpy.readers.**group_files**(*files_to_sort*, *reader=None*, *time_threshold=10*, *group_keys=None*, *ppp_config_dir=None*, *reader_kwargs=None*)

Group series of files by file pattern information.

By default this will group files by their filename `start_time` assuming it exists in the pattern. By passing the individual dictionaries returned by this function to the Scene classes' `filenames`, a series *Scene* objects can be easily created.

New in version 0.12.

Parameters

- **files_to_sort** (*iterable*) – File paths to sort in to group
- **reader** (*str*) – Reader whose file patterns should be used to sort files. This
- **time_threshold** (*int*) – Number of seconds used to consider time elements in a group as being equal. For example, if the 'start_time' item is used to group files then any time within *time_threshold* seconds of the first file's 'start_time' will be seen as occurring at the same time.
- **group_keys** (*list or tuple*) – File pattern information to use to group files. Keys are sorted in order and only the first key is used when comparing datetime elements with *time_threshold* (see above). This means it is recommended that datetime values should only come from the first key in `group_keys`. Otherwise, there is a good chance that files will not be grouped properly (datetimes being barely unequal). Defaults to a reader's `group_keys` configuration (set in YAML), otherwise `('start_time',)`.
- **ppp_config_dir** (*str*) – Root usser configuration directory for SatPy. This will be deprecated in the future, but is here for consistency with other SatPy features.
- **reader_kwargs** (*dict*) – Additional keyword arguments to pass to reader creation.

Returns List of dictionaries mapping 'reader' to a list of filenames. Each of these dictionaries can be passed as `filenames` to a *Scene* object.

`satpy.readers.`**load_reader**(*reader_configs*, ***reader_kwargs*)
Import and setup the reader from *reader_info*.

`satpy.readers.`**load_readers**(*filenames=None*, *reader=None*, *reader_kwargs=None*, *ppp_config_dir=None*)
Create specified readers and assign files to them.

Parameters

- **filenames** (*iterable or dict*) – A sequence of files that will be used to load data from. A `dict` object should map reader names to a list of filenames for that reader.
- **reader** (*str or list*) – The name of the reader to use for loading the data or a list of names.
- **filter_parameters** (*dict*) – Specify loaded file filtering parameters. Shortcut for *reader_kwargs['filter_parameters']*.
- **reader_kwargs** (*dict*) – Keyword arguments to pass to specific reader instances.
- **ppp_config_dir** (*str*) – The directory containing the configuration files for satpy.

Returns: Dictionary mapping reader name to reader instance

`satpy.readers.`**read_reader_config**(*config_files*, *loader=<class 'yaml.loader.UnsafeLoader'>*)
Read the reader *config_files* and return the info extracted.

12.1.4 satpy.writers package

Submodules

satpy.writers.cf_writer module

Writer for netCDF4/CF.

class satpy.writers.cf_writer.**CFWriter**(*name=None*, *filename=None*, *base_dir=None*, ***kwargs*)

Bases: *satpy.writers.Writer*

Writer producing NetCDF/CF compatible datasets.

Initialize the writer object.

Parameters

- **name** (*str*) – A name for this writer for log and error messages. If this writer is configured in a YAML file its name should match the name of the YAML file. Writer names may also appear in output file attributes.
- **filename** (*str*) – Filename to save data to. This filename can and should specify certain python string formatting fields to differentiate between data written to the files. Any attributes provided by the `.attrs` of a DataArray object may be included. Format and conversion specifiers provided by the `trollsift` package may also be used. Any directories in the provided pattern will be created if they do not exist. Example:

```
{platform_name}_{sensor}_{name}_{start_time:%Y%m%d_%H%M%S.tif
```

- **base_dir** (*str*) – Base destination directories for all created files.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the *Plugin* class.

static da2cf(*dataarray*, *epoch='seconds since 1970-01-01 00:00:00'*)
Convert the dataarray to something cf-compatible.

save_dataset(*dataset*, *filename=None*, *fill_value=None*, ***kwargs*)
Save the *dataset* to a given *filename*.

save_datasets(*datasets*, *filename=None*, ***kwargs*)
Save all datasets to one or more files.

satpy.writers.cf_writer.**area2cf**(*dataarray*, *strict=False*)

satpy.writers.cf_writer.**area2gridmapping**(*dataarray*)

satpy.writers.cf_writer.**area2lonlat**(*dataarray*)
Convert an area to longitudes and latitudes.

satpy.writers.cf_writer.**create_grid_mapping**(*area*)
Create the grid mapping instance for *area*.

satpy.writers.cf_writer.**geos2cf**(*area*)
Return the cf grid mapping for the geos projection.

satpy.writers.cf_writer.**get_extra_ds**(*dataset*)
Get the extra datasets associated to *dataset*.

satpy.writers.cf_writer.**laea2cf**(*area*)
Return the cf grid mapping for the laea projection.

satpy.writers.cf_writer.**make_time_bounds**(*dataarray*, *start_times*, *end_times*)

satpy.writers.cf_writer.**omerc2cf**(*area*)
Return the cf grid mapping for the omerc projection.

satpy.writers.geotiff module

GeoTIFF writer objects for creating GeoTIFF files from *Dataset* objects.

class satpy.writers.geotiff.**GeoTIFFWriter**(*dtype=None*, *tags=None*, ***kwargs*)
Bases: *satpy.writers.ImageWriter*

Writer to save GeoTIFF images.

Basic example from Scene:

scn.save_datasets(writer='geotiff')

Un-enhanced float geotiff with NaN for fill values:

scn.save_datasets(writer='geotiff', dtype=np.float32, enhance=False)

GDAL_OPTIONS = ('tfw', 'rpb', 'rpctxt', 'interleave', 'tiled', 'blockxsize', 'blockysi

save_image(*img*, *filename=None*, *dtype=None*, *fill_value=None*, *floating_point=None*, *compute=True*, ***kwargs*)
Save the image to the given `filename` in geotiff format.

Note for faster output and reduced memory usage the `rasterio` library must be installed. This writer currently falls back to using `gdal` directly, but that will be deprecated in the future.

Parameters

- **img** (*xarray.DataArray*) – Data to save to geotiff.
- **filename** (*str*) – Filename to save the image to. Defaults to `filename` passed during writer creation. Unlike the creation `filename` keyword argument, this filename does not get formatted with data attributes.
- **dtype** (*numpy.dtype*) – Numpy data type to save the image as. Defaults to 8-bit unsigned integer (`np.uint8`). If the `dtype` argument is provided during writer creation then that will be used as the default.
- **fill_value** (*int or float*) – Value to use where data values are NaN/null. If this is specified in the writer configuration file that value will be used as the default.
- **floating_point** (*bool*) – Deprecated. Use `dtype=np.float64` instead.
- **compute** (*bool*) – Compute dask arrays and save the image immediately. If `False` then the return value can be passed to *compute_writer_results()* to do the computation. This is useful when multiple images may share input calculations where dask can benefit from not repeating them multiple times. Defaults to `True` in the writer by itself, but is typically passed as `False` by callers where calculations can be combined.

classmethod separate_init_kwargs(*kwargs*)
Helper class method to separate arguments between init and save methods.

Currently the *Scene* is passed one set of arguments to represent the Writer creation and saving steps. This is not preferred for Writer structure, but provides a simpler interface to users. This method splits the provided keyword arguments between those needed for initialization and those needed for the `save_dataset` and `save_datasets` method calls.

Writer subclasses should try to prefer keyword arguments only for the save methods only and leave the init keyword arguments to the base classes when possible.

satpy.writers.mitiff module

MITIFF writer objects for creating MITIFF files from *Dataset* objects.

class satpy.writers.mitiff.**MITIFFWriter**(*name=None*, *tags=None*, ***kwargs*)
Bases: *satpy.writers.ImageWriter*

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *compute=True*, ***kwargs*)
Saves the `dataset` to a given `filename`.

This method creates an enhanced image using *get_enhanced_image*. The image is then passed to *save_image*. See both of these functions for more details on the arguments passed to this method.

save_datasets(*datasets*, *filename=None*, *fill_value=None*, *compute=True*, ***kwargs*)
Save all datasets to one or more files.

save_image()
Save Image object to a given `filename`.

Parameters

- **img** (*trollimage.xrimage.XRImage*) – Image object to save to disk.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. It may include string formatting patterns that will be filled in by dataset attributes.
- **compute** (*bool*) – If *True* (default), compute and save the dataset. If *False* return either a *dask.delayed.Delayed* object or tuple of (source, target). See the return values below for more information.
- ****kwargs** – Other keyword arguments to pass to this writer.

Returns Value returned depends on *compute*. If *compute* is *True* then the return value is the result of computing a *dask.delayed.Delayed* object or running *dask.array.store*. If *compute* is *False* then the returned value is either a *dask.delayed.Delayed* object that can be computed using *delayed.compute()* or a tuple of (source, target) that should be passed to *dask.array.store*. If target is provided the the caller is responsible for calling *target.close()* if the target has this method.

satpy.writers.ninjotiff module

GeoTIFF writer objects for creating GeoTIFF files from *Dataset* objects.

class satpy.writers.ninjotiff.**NinjoTIFFWriter**(*tags=None*, ***kwargs*)
Bases: *satpy.writers.ImageWriter*

save_image(*img*, *filename=None*, ***kwargs*)
Save the image to the given *filename* in ninjotiff format.

satpy.writers.scmi module

satpy.writers.simple_image module

class satpy.writers.simple_image.**PillowWriter**(***kwargs*)
Bases: *satpy.writers.ImageWriter*

save_image(*img*, *filename=None*, *compute=True*, ***kwargs*)
Save Image object to a given `filename`.

Parameters

- **img** (*trollimage.xrimage.XRImage*) – Image object to save to disk.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. It may include string formatting patterns that will be filled in by dataset attributes.

- **compute** (*bool*) – If *True* (default), compute and save the dataset. If *False* return either a *dask.delayed.Delayed* object or tuple of (source, target). See the return values below for more information.
- ****kwargs** – Keyword arguments to pass to the images *save* method.

Returns Value returned depends on *compute*. If *compute* is *True* then the return value is the result of computing a *dask.delayed.Delayed* object or running *dask.array.store*. If *compute* is *False* then the returned value is either a *dask.delayed.Delayed* object that can be computed using *delayed.compute()* or a tuple of (source, target) that should be passed to *dask.array.store*. If target is provided the the caller is responsible for calling *target.close()* if the target has this method.

Module contents

Shared objects of the various writer classes.

For now, this includes enhancement configuration utilities.

class satpy.writers.**DecisionTree**(*decision_dicts*, *attrs*, ***kwargs*)

Bases: object

add_config_to_tree(**decision_dicts*)

any_key = None

find_match(***kwargs*)

class satpy.writers.**EnhancementDecisionTree**(**decision_dicts*, ***kwargs*)

Bases: *satpy.writers.DecisionTree*

add_config_to_tree(**decision_dict*)

find_match(***kwargs*)

class satpy.writers.**Enhancer**(*ppp_config_dir=None*, *enhancement_config_file=None*)

Bases: object

Helper class to get enhancement information for images.

Initialize an Enhancer instance.

Parameters

- **ppp_config_dir** – Points to the base configuration directory
- **enhancement_config_file** – The enhancement configuration to apply, False to leave as is.

add_sensor_enhancements(*sensor*)

apply(*img*, ***info*)

get_sensor_enhancement_config(*sensor*)

class satpy.writers.**ImageWriter**(*name=None*, *filename=None*, *base_dir=None*, *enhance=None*, *enhancement_config=None*, ***kwargs*)

Bases: *satpy.writers.Writer*

Base writer for image file formats.

Initialize image writer object.

Parameters

- **name** (*str*) – A name for this writer for log and error messages. If this writer is configured in a YAML file its name should match the name of the YAML file. Writer names may also appear in output file attributes.
- **filename** (*str*) – Filename to save data to. This filename can and should specify certain python string formatting fields to differentiate between data written to the files. Any attributes provided by the `.attrs` of a DataArray object may be included. Format and conversion specifiers provided by the `trollsift` package may also be used. Any directories in the provided pattern will be created if they do not exist. Example:

```
{platform_name}_{sensor}_{name}_{start_time:%Y%m%d_%H%M%S.tif
```

- **base_dir** (*str*) – Base destination directories for all created files.
- **enhance** (*bool or Enhancer*) – Whether to automatically enhance data to be more visually useful and to fit inside the file format being saved to. By default this will default to using the enhancement configuration files found using the default *Enhancer* class. This can be set to *False* so that no enhancments are performed. This can also be an instance of the *Enhancer* class if further custom enhancement is needed.
- **enhancement_config** (*str*) – Deprecated.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the `Writer` base class.

Changed in version 0.10: Deprecated *enhancement_config_file* and 'enhancer' in favor of *enhance*. Pass an instance of the *Enhancer* class to *enhance* instead.

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *overlay=None*, *decorate=None*, *compute=True*, ***kwargs*)

Saves the `dataset` to a given `filename`.

This method creates an enhanced image using *get_enhanced_image*. The image is then passed to *save_image*. See both of these functions for more details on the arguments passed to this method.

save_image(*img*, *filename=None*, *compute=True*, ***kwargs*)

Save Image object to a given `filename`.

Parameters

- **img** (*trollimage.xrimage.XRImage*) – Image object to save to disk.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. It may include string formatting patterns that will be filled in by dataset attributes.
- **compute** (*bool*) – If *True* (default), compute and save the dataset. If *False* return either a *dask.delayed.Delayed* object or tuple of (source, target). See the return values below for more information.
- ****kwargs** – Other keyword arguments to pass to this writer.

Returns Value returned depends on *compute*. If *compute* is *True* then the return value is the result of computing a *dask.delayed.Delayed* object or running *dask.array.store*. If *compute* is *False* then the returned value is either a *dask.delayed.Delayed* object that can be computed using *delayed.compute()* or a tuple of (source, target) that should be passed to *dask.array.store*. If target is provided the the caller is responsible for calling *target.close()* if the target has this method.

classmethod separate_init_kwargs(*kwargs*)

Helper class method to separate arguments between init and save methods.

Currently the *Scene* is passed one set of arguments to represent the Writer creation and saving steps. This is not preferred for Writer structure, but provides a simpler interface to users. This method

splits the provided keyword arguments between those needed for initialization and those needed for the `save_dataset` and `save_datasets` method calls.

Writer subclasses should try to prefer keyword arguments only for the save methods only and leave the init keyword arguments to the base classes when possible.

class satpy.writers.**Writer**(*name=None*, *filename=None*, *base_dir=None*, ***kwargs*)

Bases: *satpy.plugin_base.Plugin*

Base Writer class for all other writers.

A minimal writer subclass should implement the *save_dataset* method.

Initialize the writer object.

Parameters

- **name** (*str*) – A name for this writer for log and error messages. If this writer is configured in a YAML file its name should match the name of the YAML file. Writer names may also appear in output file attributes.
- **filename** (*str*) – Filename to save data to. This filename can and should specify certain python string formatting fields to differentiate between data written to the files. Any attributes provided by the `.attrs` of a DataArray object may be included. Format and conversion specifiers provided by the `trollsift` package may also be used. Any directories in the provided pattern will be created if they do not exist. Example:

```
{platform_name}_{sensor}_{name}_{start_time:%Y%m%d_%H%M%S.tif
```

- **base_dir** (*str*) – Base destination directories for all created files.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the *Plugin* class.

create_filename_parser(*base_dir*)

Create a `trollsift.parser.Parser` object for later use.

get_filename(***kwargs*)

Create a filename where output data will be saved.

Parameters **kwargs** (*dict*) – Attributes and other metadata to use for formatting the previously provided *filename*.

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *compute=True*, ***kwargs*)

Saves the `dataset` to a given `filename`.

This method must be overloaded by the subclass.

Parameters

- **dataset** (*xarray.DataArray*) – Dataset to save using this writer.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. If not provided then *filename* which can be provided to the init method will be used and formatted by dataset attributes.
- **fill_value** (*int or float*) – Replace invalid values in the dataset with this fill value if applicable to this writer.
- **compute** (*bool*) – If *True* (default), compute and save the dataset. If *False* return either a *dask.delayed.Delayed* object or tuple of (source, target). See the return values below for more information.
- ****kwargs** – Other keyword arguments for this particular writer.

Returns Value returned depends on *compute*. If *compute* is *True* then the return value is the result of computing a *dask.delayed.Delayed* object or running *dask.array.store*. If *compute* is *False* then the returned value is either a *dask.delayed.Delayed* object that can be computed using *delayed.compute()* or a tuple of (source, target) that should be passed to *dask.array.store*. If target is provided the the caller is responsible for calling *target.close()* if the target has this method.

save_datasets(*datasets*, *compute=True*, ***kwargs*)

Save all datasets to one or more files.

Subclasses can use this method to save all datasets to one single file or optimize the writing of individual datasets. By default this simply calls *save_dataset* for each dataset provided.

Parameters

- **datasets** (*iterable*) – Iterable of *xarray.DataArray* objects to save using this writer.
- **compute** (*bool*) – If *True* (default), compute all of the saves to disk. If *False* then the return value is either a *dask.delayed.Delayed* object or two lists to be passed to a *dask.array.store* call. See return values below for more details.
- ****kwargs** – Keyword arguments to pass to *save_dataset*. See that documentation for more details.

Returns Value returned depends on *compute* keyword argument. If *compute* is *True* the value is the result of a either a *dask.array.store* operation or a *dask.delayed.Delayed* compute, typically this is *None*. If *compute* is *False* then the result is either a *dask.delayed.Delayed* object that can be computed with *delayed.compute()* or a two element tuple of sources and targets to be passed to *dask.array.store*. If *targets* is provided then it is the caller's responsibility to close any objects that have a "close" method.

classmethod separate_init_kwargs(*kwargs*)

Helper class method to separate arguments between init and save methods.

Currently the *Scene* is passed one set of arguments to represent the Writer creation and saving steps. This is not preferred for Writer structure, but provides a simpler interface to users. This method splits the provided keyword arguments between those needed for initialization and those needed for the `save_dataset` and `save_datasets` method calls.

Writer subclasses should try to prefer keyword arguments only for the save methods only and leave the init keyword arguments to the base classes when possible.

satpy.writers.**add_decorate**(*orig*, *fill_value=None*, ***decorate*)

Decorate an image with text and/or logos/images.

This call adds text/logos in order as given in the input to keep the alignment features available in pydecorate.

An example of the decorate config:

```
decorate = {
    'decorate': [
        {'logo': {'logo_path': <path to a logo>, 'height': 143, 'bg': 'white',
↪'bg_opacity': 255}},
        {'text': {'txt': start_time_txt,
                  'align': {'top_bottom': 'bottom', 'left_right': 'right'},
                  'font': <path to ttf font>,
                  'font_size': 22,
                  'height': 30,
                  'bg': 'black',
                  'bg_opacity': 255,
                  'line': 'white'}}
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Any numbers of text/logo in any order can be added to the decorate list, but the order of the list is kept as described above.

Note that a feature given in one element, eg. bg (which is the background color) will also apply on the next elements unless a new value is given.

align is a special keyword telling where in the image to start adding features, top_bottom is either top or bottom and left_right is either left or right.

satpy.writers.**add_logo**(*orig*, *dc*, *img*, *logo=None*)

Add logos or other images to an image using the pydecorate package.

All the features of pydecorate's `add_logo` are available. See documentation of Welcome to the Pydecorate documentation! for more info.

satpy.writers.**add_overlay**(*orig*, *area*, *coast_dir*, *color=(0, 0, 0)*, *width=0.5*, *resolution=None*, *level_coast=1*, *level_borders=1*, *fill_value=None*)

Add coastline and political borders to image.

Uses `color` for feature colors where `color` is a 3-element tuple of integers between 0 and 255 representing (R, G, B).

Warning: This function currently loses the data mask (alpha band).

`resolution` is chosen automatically if None (default), otherwise it should be one of:

'f' 'h' 'i' 'l' 'c'	Full resolution High resolution Intermediate resolution Low resolution Crude resolution	0.04 km 0.2 km 1.0 km 5.0 km 25 km

satpy.writers.**add_text**(*orig*, *dc*, *img*, *text=None*)

Add text to an image using the pydecorate package.

All the features of pydecorate's `add_text` are available. See documentation of Welcome to the Pydecorate documentation! for more info.

satpy.writers.**available_writers**(*as_dict=False*)

Available writers based on current configuration.

Parameters **as_dict** (*bool*) – Optionally return writer information as a dictionary. Default: False

Returns: List of available writer names. If *as_dict* is *True* then a list of dictionaries including additionally writer information is returned.

satpy.writers.**compute_writer_results**(*results*)

Compute all the given dask graphs *results* so that the files are saved.

Parameters **results** (*iterable*) – Iterable of dask graphs resulting from calls to *scn.save_datasets(..., compute=False)*

satpy.writers.**configs_for_writer**(*writer=None*, *ppp_config_dir=None*)

Generator of writer configuration files for one or more writers

Parameters

- **writer** (*Optional[str]*) – Yield configs only for this writer
- **ppp_config_dir** (*Optional[str]*) – Additional configuration directory to search for writer configuration files.

Returns: Generator of lists of configuration files

satpy.writers.**get_enhanced_image**(*dataset*, *ppp_config_dir=None*, *enhance=None*, *enhancement_config_file=None*, *overlay=None*, *decorate=None*, *fill_value=None*)

Get an enhanced version of *dataset* as an XRImage instance.

Parameters

- **dataset** (*xarray.DataArray*) – Data to be enhanced and converted to an image.
- **ppp_config_dir** (*str*) – Root configuration directory.
- **enhance** (*bool or Enhancer*) – Whether to automatically enhance data to be more visually useful and to fit inside the file format being saved to. By default this will default to using the enhancement configuration files found using the default *Enhancer* class. This can be set to *False* so that no enhancments are performed. This can also be an instance of the *Enhancer* class if further custom enhancement is needed.
- **enhancement_config_file** (*str*) – Deprecated.
- **overlay** (*dict*) – Options for image overlays. See *add_overlay()* for available options.
- **decorate** (*dict*) – Options for decorating the image. See *add_decorate()* for available options.
- **fill_value** (*int or float*) – Value to use when pixels are masked or invalid. Default of *None* means to create an alpha channel. See finalize() for more details. Only used when adding overlays or decorations. Otherwise it is up to the caller to "finalize" the image before using it except if calling img.show() or providing the image to a writer as these will finalize the image.

Changed in version 0.10: Deprecated *enhancement_config_file* and 'enhancer' in favor of *enhance*. Pass an instance of the *Enhancer* class to *enhance* instead.

satpy.writers.**load_writer**(*writer*, *ppp_config_dir=None*, ***writer_kwargs*)

Find and load writer *writer* in the available configuration files.

satpy.writers.**load_writer_configs**(*writer_configs*, *ppp_config_dir*, ***writer_kwargs*)

Load the writer from the provided *writer_configs*.

satpy.writers.**read_writer_config**(*config_files*, *loader=<class 'yaml.loader.UnsafeLoader'>*)

Read the writer *config_files* and return the info extracted.

satpy.writers.**show**(*dataset*, ***kwargs*)

Display the dataset as an image.

satpy.writers.**split_results**(*results*)

Get sources, targets and delayed objects to separate lists from a list of results collected from (multiple) writer(s).

satpy.writers.**to_image**(*dataset*)

12.2 Submodules

12.3 satpy.config module

SatPy Configuration directory and file handling

satpy.config.**check_satpy**()
Check the satpy readers and writers for correct installation.

satpy.config.**check_yaml_configs**(*configs*, *key*, *hdr_len*)
Get a diagnostic for the yaml *configs*.

key is the section to look for to get a name for the config at hand. *hdr_len* is the number of lines that can be safely read from the config to get a name.

satpy.config.**config_search_paths**(*filename*, **search_dirs*, ***kwargs*)

satpy.config.**get_config**(*filename*, **search_dirs*, ***kwargs*)
Blends the different configs, from package defaults to .

satpy.config.**get_config_path**(*filename*, **search_dirs*)
Get the appropriate path for a filename, in that order: filename, ., PPP_CONFIG_DIR, package's etc dir.

satpy.config.**get_environ_ancpath**(*default='.'*)

satpy.config.**get_environ_config_dir**(*default=None*)

satpy.config.**glob_config**(*pattern*, **search_dirs*)
Return glob results for all possible configuration locations.

Note: This method does not check the configuration "base" directory if the pattern includes a subdirectory.
This is done for performance since this is usually used to find *all* configs for a certain component.

satpy.config.**recursive_dict_update**(*d*, *u*)
Recursive dictionary update.

Copied from:

http://stackoverflow.com/questions/3232943/update-value-of-a-nested-dictionary-of-varying-depth

satpy.config.**runtime_import**(*object_path*)
Import at runtime.

12.4 satpy.dataset module

Dataset objects.

class satpy.dataset.**Dataset**
Bases: object

Placeholder for the deprecated class.

class satpy.dataset.**DatasetID**
Bases: *satpy.dataset.DatasetID*

Identifier for all *Dataset* objects.

DatasetID is a namedtuple that holds identifying and classifying information about a Dataset. There are two identifying elements, `name` and `wavelength`. These can be used to generically refer to a Dataset. The other elements of a DatasetID are meant to further distinguish a Dataset from the possible variations it may have. For

example multiple Datasets may be called by one `name` but may exist in multiple resolutions or with different calibrations such as "radiance" and "reflectance". If an element is *None* then it is considered not applicable.

A DatasetID can also be used in SatPy to query for a Dataset. This way a fully qualified DatasetID can be found even if some of the DatasetID elements are unknown. In this case a *None* signifies something that is unknown or not applicable to the requested Dataset.

Parameters

- **name** (*str*) – String identifier for the Dataset
- **wavelength** (*float, tuple*) – Single float wavelength when querying for a Dataset. Otherwise 3-element tuple of floats specifying the minimum, nominal, and maximum wavelength for a Dataset. *None* if not applicable.
- **resolution** (*int, float*) – Per data pixel/area resolution. If resolution varies across the Dataset then nadir view resolution is preferred. Usually this is in meters, but for lon/lat gridded data angle degrees may be used.
- **polarization** (*str*) – 'V' or 'H' polarizations of a microwave channel. *None* if not applicable.
- **calibration** (*str*) – String identifying the calibration level of the Dataset (ex. 'radiance', 'reflectance', etc). *None* if not applicable.
- **level** (*int, float*) – Pressure/altitude level of the dataset. This is typically in hPa, but may be in inverse meters for altitude datasets (1/meters).
- **modifiers** (*tuple*) – Tuple of strings identifying what corrections or other modifications have been performed on this Dataset (ex. 'sunz_corrected', 'rayleigh_corrected', etc). *None* or empty tuple if not applicable.

classmethod from_dict(*d*, ***kwargs*)
Convert a dict to an ID.

static name_match(*a*, *b*)
Return if two string names are equal.

Parameters

- **a** (*str*) – DatasetID.name or other string
- **b** (*str*) – DatasetID.name or other string

to_dict(*trim=True*)
Convert the ID to a dict.

static wavelength_match(*a*, *b*)
Return if two wavelengths are equal.

Parameters

- **a** (*tuple or scalar*) – (min wl, nominal wl, max wl) or scalar wl
- **b** (*tuple or scalar*) – (min wl, nominal wl, max wl) or scalar wl

class satpy.dataset.**MetadataObject**(***attributes*)
Bases: object

A general metadata object.

Initialize the class with *attributes*.

id
Return the DatasetID of the object.

satpy.dataset.**average_datetimes**(*dt_list*)
Average a series of datetime objects.

Note: This function assumes all datetime objects are naive and in the same time zone (UTC).

Parameters **dt_list** (*iterable*) – Datetime objects to average

Returns: Average datetime as a datetime object

satpy.dataset.**combine_metadata**(**metadata_objects*, ***kwargs*)
Combine the metadata of two or more Datasets.

If any keys are not equal or do not exist in all provided dictionaries then they are not included in the returned dictionary. By default any keys with the word 'time' in them and consisting of datetime objects will be averaged. This is to handle cases where data were observed at almost the same time but not exactly.

Parameters

- ***metadata_objects** – MetadataObject or dict objects to combine
- **average_times** (*bool*) – Average any keys with 'time' in the name

Returns the combined metadata

Return type dict

satpy.dataset.**dataset_walker**(*datasets*)
Walk through *datasets* and their ancillary data.

Yields datasets and their parent.

satpy.dataset.**replace_anc**(*dataset*, *parent_dataset*)
Replace *dataset* the *parent_dataset*'s *ancillary_variables* field.

12.5 satpy.multiscene module

MultiScene object to blend satellite data.

class satpy.multiscene.**MultiScene**(*scenes=None*)
Bases: object

Container for multiple *Scene* objects.

Initialize MultiScene and validate sub-scenes.

Parameters **scenes** (*iterable*) – *Scene* objects to operate on (optional)

Note: If the *scenes* passed to this object are a generator then certain operations performed will try to preserve that generator state. This may limit what properties or methods are available to the user. To avoid this behavior compute the passed generator by converting the passed scenes to a list first: `MultiScene(list(scenes))`.

all_same_area

blend(*blend_function=<function stack>*)
Blend the datasets into one scene.

Note: Blending is not currently optimized for generator-based MultiScene.

crop (**args*, ***kwargs*)
Crop the multiscene and return a new cropped multiscene.

first_scene
First Scene of this MultiScene object.

classmethod from_files (*files_to_sort*, *reader=None*, ***kwargs*)
Create multiple Scene objects from multiple files.

This uses the *satpy.readers.group_files()* function to group files. See this function for more details on possible keyword arguments.

New in version 0.12.

is_generator
Contained Scenes are stored as a generator.

load (**args*, ***kwargs*)
Load the required datasets from the multiple scenes.

loaded_dataset_ids
Union of all Dataset IDs loaded by all children.

resample (*destination=None*, ***kwargs*)
Resample the multiscene.

save_animation (*filename*, *datasets=None*, *fps=10*, *fill_value=None*, *batch_size=1*, *ignore_missing=False*, *client=True*, ***kwargs*)
Helper method for saving to movie (MP4) or GIF formats.

Supported formats are dependent on the *imageio* library and are determined by filename extension by default.

Note: Starting with `imageio` 2.5.0, the use of FFMPEG depends on a separate `imageio-ffmpeg` package.

By default all datasets available will be saved to individual files using the first Scene's datasets metadata to format the filename provided. If a dataset is not available from a Scene then a black array is used instead (np.zeros(shape)).

This function can use the `dask.distributed` library for improved performance by computing multiple frames at a time (see *batch_size* option below). If the distributed library is not available then frames will be generated one at a time, one product at a time.

Parameters

- **filename** (*str*) – Filename to save to. Can include python string formatting keys from dataset `.attrs` (ex. "{name}_{start_time:%Y%m%d_%H%M%S.gif")
- **datasets** (*list*) – DatasetIDs to save (default: all datasets)
- **fps** (*int*) – Frames per second for produced animation
- **fill_value** (*int*) – Value to use instead creating an alpha band.
- **batch_size** (*int*) – Number of frames to compute at the same time. This only has effect if the *dask.distributed* package is installed. This will default to 1. Setting this to 0 or less will attempt to process all frames at once. This option should be used with

care to avoid memory issues when trying to improve performance. Note that this is the total number of frames for all datasets, so when saving 2 datasets this will compute `(batch_size / 2)` frames for the first dataset and `(batch_size / 2)` frames for the second dataset.

- **ignore_missing** (*bool*) – Don't include a black frame when a dataset is missing from a child scene.
- **client** (*bool or dask.distributed.Client*) – Dask distributed client to use for computation. If this is `True` (default) then any existing clients will be used. If this is `False` or `None` then a client will not be created and `dask.distributed` will not be used. If this is a dask `Client` object then it will be used for distributed computation.
- **kwargs** – Additional keyword arguments to pass to *imageio.get_writer*.

save_datasets(*client=True*, *batch_size=1*, ***kwargs*)

Run save_datasets on each Scene.

Note that some writers may not be multi-process friendly and may produce unexpected results or fail by raising an exception. In these cases `client` should be set to `False`. This is currently a known issue for basic 'geotiff' writer work loads.

Parameters

- **batch_size** (*int*) – Number of scenes to compute at the same time. This only has effect if the *dask.distributed* package is installed. This will default to 1. Setting this to 0 or less will attempt to process all scenes at once. This option should be used with care to avoid memory issues when trying to improve performance.
- **client** (*bool or dask.distributed.Client*) – Dask distributed client to use for computation. If this is `True` (default) then any existing clients will be used. If this is `False` or `None` then a client will not be created and `dask.distributed` will not be used. If this is a dask `Client` object then it will be used for distributed computation.
- **kwargs** – Additional keyword arguments to pass to *save_datasets()*. Note `compute` can not be provided.

scenes

Get list of Scene objects contained in this MultiScene.

Note: If the Scenes contained in this object are stored in a generator (not list or tuple) then accessing this property will load/iterate through the generator possibly

---

shared_dataset_ids

Dataset IDs shared by all children.

satpy.multiscene.**stack**(*datasets*)

First dataset at the bottom.

satpy.multiscene.**timeseries**(*datasets*)

Expands dataset with and concats by time dimension

12.6 satpy.node module

Nodes to build trees.

class satpy.node.**DependencyTree**(*readers*, *compositors*, *modifiers*)

Bases: *satpy.node.Node*

Structure to discover and store *Dataset* dependencies

Used primarily by the *Scene* object to organize dependency finding. Dependencies are stored used a series of *Node* objects which this class is a subclass of.

Collect Dataset generating information.

Collect the objects that generate and have information about Datasets including objects that may depend on certain Datasets being generated. This includes readers, compositors, and modifiers.

Parameters

- **readers** (*dict*) – Reader name -> Reader Object
- **compositors** (*dict*) – Sensor name -> Composite ID -> Composite Object
- **modifiers** (*dict*) – Sensor name -> Modifier name -> (Modifier Class, modifier options)

add_child(*parent*, *child*)
Add a child to the node.

add_leaf(*ds_id*, *parent=None*)

contains(*item*)
Check contains when we know the *exact* DatasetID.

copy()
Copy the this node tree

Note all references to readers are removed. This is meant to avoid tree copies accessing readers that would return incompatible (Area) data. Theoretically it should be possible for tree copies to request compositor or modifier information as long as they don't depend on any datasets not already existing in the dependency tree.

find_dependencies(*dataset_keys*, ***dfilter*)
Create the dependency tree.

Parameters

- **dataset_keys** (*iterable*) – Strings or DatasetIDs to find dependencies for
- ****dfilter** (*dict*) – Additional filter parameters. See *satpy.readers.get_key* for more details.

Returns Root node of the dependency tree and a set of unknown datasets

Return type (*Node*, set)

get_compositor(*key*)

get_modifier(*comp_id*)

getitem(*item*)
Get Node when we know the *exact* DatasetID.

leaves(*nodes=None*, *unique=True*)
Get the leaves of the tree starting at this root.

Parameters

- **nodes** (*iterable*) – limit leaves for these node names
- **unique** – only include individual leaf nodes once

Returns list of leaf nodes

trunk(*nodes=None*, *unique=True*)
Get the trunk nodes of the tree starting at this root.

Parameters

- **nodes** (*iterable*) – limit trunk nodes to the names specified or the children of them that are also trunk nodes.
- **unique** – only include individual trunk nodes once

Returns list of trunk nodes

class satpy.node.**Node**(*name*, *data=None*)
Bases: object

A node object.

Init the node object.

add_child(*obj*)
Add a child to the node.

copy(*node_cache=None*)

display(*previous=0*, *include_data=False*)
Display the node.

flatten(*d=None*)
Flatten tree structure to a one level dictionary.

Parameters **d** (*dict, optional*) – output dictionary to update

Returns

Node.name -> Node. The returned dictionary includes the current Node and all its children.

Return type dict

is_leaf

leaves(*unique=True*)
Get the leaves of the tree starting at this root.

trunk(*unique=True*)
Get the trunk of the tree starting at this root.

12.7 satpy.plugin_base module

The *satpy.plugin_base* module defines the plugin API.

class satpy.plugin_base.**Plugin**(*ppp_config_dir=None*, *default_config_filename=None*, *config_files=None*, ***kwargs*)
Bases: object

Base plugin class for all dynamically loaded and configured objects.

Load configuration files related to this plugin.

This initializes a *self.config* dictionary that can be used to customize the subclass.

Parameters

- **ppp_config_dir** (*str*) – Base "etc" directory for all configuration files.

- **default_config_filename** (*str*) – Configuration filename to use if no other files have been specified with *config_files*.
- **config_files** (*list or str*) – Configuration files to load instead of those automatically found in *ppp_config_dir* and other default configuration locations.
- **kwargs** (*dict*) – Unused keyword arguments.

load_yaml_config(*conf*)
Load a YAML configuration file and recursively update the overall configuration.

12.8 satpy.resample module

SatPy provides multiple resampling algorithms for resampling geolocated data to uniform projected grids. The easiest way to perform resampling in SatPy is through the *Scene* object's *resample()* method. Additional utility functions are also available to assist in resampling data. Below is more information on resampling with SatPy as well as links to the relevant API documentation for available keyword arguments.

12.8.1 Resampling algorithms

Table 1: Available Resampling Algorithms

Resampler	Description	Related
nearest	Nearest Neighbor	*KDTreeResampler*
ewa	Elliptical Weighted Averaging	*EWAResampler*
native	Native	*NativeResampler*
bilinear	Bilinear	*BilinearResampler*

The resampling algorithm used can be specified with the `resampler` keyword argument and defaults to `nearest`:

```
>>> scn = Scene(...)
>>> euro_scn = global_scene.resample('euro4', resampler='nearest')
```

Warning: Some resampling algorithms expect certain forms of data. For example, the EWA resampling expects polar-orbiting swath data and prefers if the data can be broken in to "scan lines". See the API documentation for a specific algorithm for more information.

12.8.2 Resampling for comparison and composites

While all the resamplers can be used to put datasets of different resolutions on to a common area, the 'native' resampler is designed to match datasets to one resolution in the dataset's original projection. This is extremely useful when generating composites between bands of different resolutions.

```
>>> new_scn = scn.resample(resampler='native')
```

By default this resamples to the *highest resolution area* (smallest footprint per pixel) shared between the loaded datasets. You can easily specify the lower resolution area:

```
>>> new_scn = scn.resample(scn.min_area(), resampler='native')
```

Providing an area that is neither the minimum or maximum resolution area may work, but behavior is currently undefined.

12.8.3 Caching for geostationary data

SatPy will do its best to reuse calculations performed to resample datasets, but it can only do this for the current processing and will lose this information when the process/script ends. Some resampling algorithms, like `nearest` and `bilinear`, can benefit by caching intermediate data on disk in the directory specified by *cache_dir* and using it next time. This is most beneficial with geostationary satellite data where the locations of the source data and the target pixels don't change over time.

```
>>> new_scn = scn.resample('euro4', cache_dir='/path/to/cache_dir')
```

See the documentation for specific algorithms to see availability and limitations of caching for that algorithm.

12.8.4 Create custom area definition

See `pyresample.geometry.AreaDefinition` for information on creating areas that can be passed to the resample method:

```
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> local_scene = global_scene.resample(my_area)
```

12.8.5 Create dynamic area definition

See `pyresample.geometry.DynamicAreaDefinition` for more information.

Examples coming soon...

12.8.6 Store area definitions

Area definitions can be added to a custom YAML file (see pyresample's documentation for more information) and loaded using pyresample's utility methods:

```
>>> from pyresample.utils import parse_area_file
>>> my_area = parse_area_file('my_areas.yaml', 'my_area')[0]
```

Examples coming soon...

class satpy.resample.**BaseResampler**(*source_geo_def*, *target_geo_def*)

Bases: `object`

Base abstract resampler class.

Initialize resampler with geolocation information.

Parameters

- **source_geo_def** (*SwathDefinition, AreaDefinition*) – Geolocation definition for the data to be resampled
- **target_geo_def** (*CoordinateDefinition, AreaDefinition*) – Geolocation definition for the area to resample data to.

compute(*data*, ***kwargs*)
Do the actual resampling.

This must be implemented by subclasses.

get_hash(*source_geo_def=None*, *target_geo_def=None*, ***kwargs*)
Get hash for the current resample with the given *kwargs*.

precompute(***kwargs*)
Do the precomputation.

This is an optional step if the subclass wants to implement more complex features like caching or can share some calculations between multiple datasets to be processed.

resample(*data*, *cache_dir=None*, *mask_area=None*, ***kwargs*)
Resample *data* by calling *precompute* and *compute* methods.

Only certain resampling classes may use *cache_dir* and the *mask* provided when *mask_area* is True. The return value of calling the *precompute* method is passed as the *cache_id* keyword argument of the *compute* method, but may not be used directly for caching. It is up to the individual resampler subclasses to determine how this is used.

Parameters

- **data** (*xarray.DataArray*) – Data to be resampled
- **cache_dir** (*str*) – directory to cache precomputed results (default False, optional)
- **mask_area** (*bool*) – Mask geolocation data where data values are invalid. This should be used when data values may affect what neighbors are considered valid.

Returns (xarray.DataArray): Data resampled to the target area

class satpy.resample.**BilinearResampler**(*source_geo_def*, *target_geo_def*)
Bases: *satpy.resample.BaseResampler*

Resample using bilinear.

compute(*data*, *fill_value=None*, ***kwargs*)
Resample the given data using bilinear interpolation

load_bil_info(*cache_dir*, ***kwargs*)

precompute(*mask=None*, *radius_of_influence=50000*, *epsilon=0*, *reduce_data=True*, *nprocs=1*, *cache_dir=False*, ***kwargs*)
Create bilinear coefficients and store them for later use.

Note: The *mask* keyword should be provided if geolocation may be valid where data points are invalid. This defaults to the *mask* attribute of the *data* numpy masked array passed to the *resample* method.

save_bil_info(*cache_dir*, ***kwargs*)

class satpy.resample.**EWAResampler**(*source_geo_def*, *target_geo_def*)
Bases: *satpy.resample.BaseResampler*

Resample using an elliptical weighted averaging algorithm.

This algorithm does **not** use caching or any externally provided data mask (unlike the 'nearest' resampler).

This algorithm works under the assumption that the data is observed one scan line at a time. However, good results can still be achieved for non-scan based data provided *rows_per_scan* is set to the number of rows in the entire swath or by setting it to *None*.

Parameters

- **rows_per_scan** (*int, None*) – Number of data rows for every observed scanline. If None then the entire swath is treated as one large scanline.
- **weight_count** (*int*) – number of elements to create in the gaussian weight table. Default is 10000. Must be at least 2
- **weight_min** (*float*) – the minimum value to store in the last position of the weight table. Default is 0.01, which, with a *weight_distance_max* of 1.0 produces a weight of 0.01 at a grid cell distance of 1.0. Must be greater than 0.
- **weight_distance_max** (*float*) – distance in grid cell units at which to apply a weight of *weight_min*. Default is 1.0. Must be greater than 0.
- **weight_delta_max** (*float*) – maximum distance in grid cells in each grid dimension over which to distribute a single swath cell. Default is 10.0.
- **weight_sum_min** (*float*) – minimum weight sum value. Cells whose weight sums are less than *weight_sum_min* are set to the grid fill value. Default is EPSILON.
- **maximum_weight_mode** (*bool*) – If False (default), a weighted average of all swath cells that map to a particular grid cell is used. If True, the swath cell having the maximum weight of all swath cells that map to a particular grid cell is used. This option should be used for coded/category data, i.e. snow cover.

compute(*data*, *cache_id=None*, *fill_value=0*, *weight_count=10000*, *weight_min=0.01*, *weight_distance_max=1.0*, *weight_delta_max=1.0*, *weight_sum_min=-1.0*, *maximum_weight_mode=False*, *grid_coverage=0*, ***kwargs*)
Resample the data according to the precomputed X/Y coordinates.

precompute(*cache_dir=None*, *swath_usage=0*, ***kwargs*)
Generate row and column arrays and store it for later use.

resample(**args*, ***kwargs*)
Run precompute and compute methods.

Note: This sets the default of 'mask_area' to False since it is not needed in EWA resampling currently.

---

class satpy.resample.**KDTreeResampler**(*source_geo_def*, *target_geo_def*)
Bases: *satpy.resample.BaseResampler*

Resample using a KDTree-based nearest neighbor algorithm.

This resampler implements on-disk caching when the *cache_dir* argument is provided to the *resample* method. This should provide significant performance improvements on consecutive resampling of geostationary data. It is not recommended to provide *cache_dir* when the *mask* keyword argument is provided to *precompute* which occurs by default for *SwathDefinition* source areas.

Parameters

- **cache_dir** (*str*) – Long term storage directory for intermediate results. By default only 10 different source/target combinations are cached to save space.
- **mask_area** (*bool*) – Force resampled data's invalid pixel mask to be used when searching for nearest neighbor pixels. By default this is True for SwathDefinition source areas and False for all other area definition types.
- **radius_of_influence** (*float*) – Search radius cut off distance in meters
- **epsilon** (*float*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time.

compute(*data*, *weight_funcs=None*, *fill_value=nan*, *with_uncert=False*, ***kwargs*)
Do the actual resampling.

This must be implemented by subclasses.

load_neighbour_info(*cache_dir*, *mask=None*, ***kwargs*)
Read index arrays from either the in-memory or disk cache.

precompute(*mask=None*, *radius_of_influence=None*, *epsilon=0*, *cache_dir=None*, ***kwargs*)
Create a KDTree structure and store it for later use.

Note: The *mask* keyword should be provided if geolocation may be valid where data points are invalid.

save_neighbour_info(*cache_dir*, *mask=None*, ***kwargs*)
Cache resampler's index arrays if there is a cache dir.

class satpy.resample.**NativeResampler**(*source_geo_def*, *target_geo_def*)
Bases: *satpy.resample.BaseResampler*

Expand or reduce input datasets to be the same shape.

If data is higher resolution (more pixels) than the destination area then data is averaged to match the destination resolution.

If data is lower resolution (less pixels) than the destination area then data is repeated to match the destination resolution.

This resampler does not perform any caching or masking due to the simplicity of the operations.

Initialize resampler with geolocation information.

Parameters

- **source_geo_def** (*SwathDefinition, AreaDefinition*) – Geolocation definition for the data to be resampled
- **target_geo_def** (*CoordinateDefinition, AreaDefinition*) – Geolocation definition for the area to resample data to.

static aggregate(*d*, *y_size*, *x_size*)
Average every 4 elements (2x2) in a 2D array

compute(*data*, *expand=True*, ***kwargs*)
Do the actual resampling.

This must be implemented by subclasses.

classmethod expand_reduce(*d_arr*, *repeats*)

resample(*data*, *cache_dir=None*, *mask_area=False*, ***kwargs*)
Resample *data* by calling *precompute* and *compute* methods.

Only certain resampling classes may use *cache_dir* and the *mask* provided when *mask_area* is True. The return value of calling the *precompute* method is passed as the *cache_id* keyword argument of the *compute* method, but may not be used directly for caching. It is up to the individual resampler subclasses to determine how this is used.

Parameters

- **data** (*xarray.DataArray*) – Data to be resampled
- **cache_dir** (*str*) – directory to cache precomputed results (default False, optional)
- **mask_area** (*bool*) – Mask geolocation data where data values are invalid. This should be used when data values may affect what neighbors are considered valid.

Returns (xarray.DataArray): Data resampled to the target area

satpy.resample.**get_area_def**(*area_name*)
Get the definition of *area_name* from file.

The file is defined to use is to be placed in the $PPP_CONFIG_DIR directory, and its name is defined in satpy's configuration file.

satpy.resample.**get_area_file**()
Find area file(s) to use.

The files are to be named *areas.yaml* or *areas.def*.

satpy.resample.**get_fill_value**(*dataset*)
Get the fill value of the *dataset*, defaulting to np.nan.

satpy.resample.**hash_dict**(*the_dict*, *the_hash=None*)

satpy.resample.**prepare_resampler**(*source_area*, *destination_area*, *resampler=None*, ***resample_kwargs*)
Instantiate and return a resampler.

satpy.resample.**resample**(*source_area*, *data*, *destination_area*, *resampler=None*, ***kwargs*)
Do the resampling.

satpy.resample.**resample_dataset**(*dataset*, *destination_area*, ***kwargs*)
Resample *dataset* and return the resampled version.

Parameters

- **dataset** (*xarray.DataArray*) – Data to be resampled.
- **destination_area** – The destination onto which to project the data, either a full blown area definition or a string corresponding to the name of the area as defined in the area file.
- ****kwargs** – The extra parameters to pass to the resampler objects.

Returns A resampled DataArray with updated `.attrs["area"]` field. The dtype of the array is preserved.

12.9 satpy.scene module

Scene objects to hold satellite data.

class satpy.scene.**Scene**(*filenames=None*, *reader=None*, *filter_parameters=None*, *reader_kwargs=None*, *ppp_config_dir=None*, *base_dir=None*, *sensor=None*, *start_time=None*, *end_time=None*, *area=None*)
Bases: *satpy.dataset.MetadataObject*

The Almighty Scene Class.

Example usage:

```
from satpy import Scene
from glob import glob

# create readers and open files
scn = Scene(filenames=glob('/path/to/files/*'), reader='viirs_sdr')

# load datasets from input files
scn.load(['I01', 'I02'])
```

(continues on next page)

(continued from previous page)

```
# resample from satellite native geolocation to builtin 'eurol' Area
new_scn = scn.resample('eurol')

# save all resampled datasets to geotiff files in the current directory
new_scn.save_datasets()
```

Initialize Scene with Reader and Compositor objects.

To load data *filenames* and preferably *reader* must be specified. If *filenames* is provided without *reader* then the available readers will be searched for a Reader that can support the provided files. This can take a considerable amount of time so it is recommended that *reader* always be provided. Note without *filenames* the Scene is created with no Readers available requiring Datasets to be added manually:

```
scn = Scene()
scn['my_dataset'] = Dataset(my_data_array, **my_info)
```

Parameters

- **filenames** (*iterable or dict*) – A sequence of files that will be used to load data from. A `dict` object should map reader names to a list of filenames for that reader.
- **reader** (*str or list*) – The name of the reader to use for loading the data or a list of names.
- **filter_parameters** (*dict*) – Specify loaded file filtering parameters. Shortcut for *reader_kwargs['filter_parameters']*.
- **reader_kwargs** (*dict*) – Keyword arguments to pass to specific reader instances.
- **ppp_config_dir** (*str*) – The directory containing the configuration files for satpy.
- **base_dir** (*str*) – (DEPRECATED) The directory to search for files containing the data to load. If *filenames* is also provided, this is ignored.
- **sensor** (*list or str*) – (DEPRECATED: Use *find_files_and_readers* function) Limit used files by provided sensors.
- **area** (*AreaDefinition*) – (DEPRECATED: Use *filter_parameters*) Limit used files by geographic area.
- **start_time** (*datetime*) – (DEPRECATED: Use *filter_parameters*) Limit used files by starting time.
- **end_time** (*datetime*) – (DEPRECATED: Use *filter_parameters*) Limit used files by ending time.

all_composite_ids(*sensor_names=None*)
Get all composite IDs that are configured.

Returns: generator of configured composite names

all_composite_names(*sensor_names=None*)

all_dataset_ids(*reader_name=None*, *composites=False*)
Get names of all datasets from loaded readers or *reader_name* if specified..

Returns list of all dataset names

all_dataset_names(*reader_name=None*, *composites=False*)

all_modifier_names()

all_same_area
All contained data arrays are on the same area.

all_same_proj
All contained data array are in the same projection.

available_composite_ids (*available_datasets=None*)
Get names of compositors that can be generated from the available datasets.

Returns: generator of available compositor's names

available_composite_names (*available_datasets=None*)
All configured composites known to this Scene.

available_dataset_ids (*reader_name=None*, *composites=False*)
Get names of available datasets, globally or just for *reader_name* if specified, that can be loaded.

Available dataset names are determined by what each individual reader can load. This is normally determined by what files are needed to load a dataset and what files have been provided to the scene/reader.

Returns list of available dataset names

available_dataset_names (*reader_name=None*, *composites=False*)
Get the list of the names of the available datasets.

copy (*datasets=None*)
Create a copy of the Scene including dependency information.

Parameters **datasets** (*list, tuple*) – *DatasetID* objects for the datasets to include in the new Scene object.

create_reader_instances (*filenames=None*, *reader=None*, *reader_kwargs=None*)
Find readers and return their instances.

crop (*area=None*, *ll_bbox=None*, *xy_bbox=None*, *dataset_ids=None*)
Crop Scene to a specific Area boundary or bounding box.

Parameters

- **area** (*AreaDefinition*) – Area to crop the current Scene to
- **ll_bbox** (*tuple, list*) – 4-element tuple where values are in lon/lat degrees. Elements are `(xmin, ymin, xmax, ymax)` where X is longitude and Y is latitude.
- **xy_bbox** (*tuple, list*) – Same as *ll_bbox* but elements are in projection units.
- **dataset_ids** (*iterable*) – DatasetIDs to include in the returned *Scene*. Defaults to all datasets.

This method will attempt to intelligently slice the data to preserve relationships between datasets. For example, if we are cropping two DataArrays of 500m and 1000m pixel resolution then this method will assume that exactly 4 pixels of the 500m array cover the same geographic area as a single 1000m pixel. It handles these cases based on the shapes of the input arrays and adjusting slicing indexes accordingly. This method will have trouble handling cases where data arrays seem related but don't cover the same geographic area or if the coarsest resolution data is not related to the other arrays which are related.

It can be useful to follow cropping with a call to the native resampler to resolve all datasets to the same resolution and compute any composites that could not be generated previously:

```
>>> cropped_scn = scn.crop(ll_bbox=(-105., 40., -95., 50.))
>>> remapped_scn = cropped_scn.resample(resampler='native')
```

Note: The *resample* method automatically crops input data before resampling to save time/memory.

end_time
Return the end time of the file.

generate_composites(*nodes=None*)
Compute all the composites contained in *requirements*.

get(*key*, *default=None*)
Return value from DatasetDict with optional default.

classmethod get_writer_by_ext(*extension*)
Find the writer matching the *extension*.

images()
Generate images for all the datasets from the scene.

iter_by_area()
Generate datasets grouped by Area.

Returns generator of (area_obj, list of dataset objects)

keys(***kwargs*)

load(*wishlist*, *calibration=None*, *resolution=None*, *polarization=None*, *level=None*, *generate=True*, *unload=True*, ***kwargs*)
Read and generate requested datasets.

When the *wishlist* contains *DatasetID* objects they can either be fully-specified *DatasetID* objects with every parameter specified or they can not provide certain parameters and the "best" parameter will be chosen. For example, if a dataset is available in multiple resolutions and no resolution is specified in the wishlist's DatasetID then the highest (smallest number) resolution will be chosen.

Loaded *DataArray* objects are created and stored in the Scene object.

Parameters

- **wishlist** (*iterable*) – Names (str), wavelengths (float), or DatasetID objects of the requested datasets to load. See *available_dataset_ids()* for what datasets are available.
- **calibration** (*list, str*) – Calibration levels to limit available datasets. This is a shortcut to having to list each DatasetID in *wishlist*.
- **resolution** (*list | float*) – Resolution to limit available datasets. This is a shortcut similar to calibration.
- **polarization** (*list | str*) – Polarization ('V', 'H') to limit available datasets. This is a shortcut similar to calibration.
- **level** (*list | str*) – Pressure level to limit available datasets. Pressure should be in hPa or mb. If an altitude is used it should be specified in inverse meters (1/m). The units of this parameter ultimately depend on the reader.
- **generate** (*bool*) – Generate composites from the loaded datasets (default: True)
- **unload** (*bool*) – Unload datasets that were required to generate the requested datasets (composite dependencies) but are no longer needed.

max_area(*datasets=None*)
Get highest resolution area for the provided datasets.

Parameters **datasets** (*iterable*) – Datasets whose areas will be compared. Can be either *xarray.DataArray* objects or identifiers to get the DataArrays from the current Scene. Defaults to all datasets.

min_area(*datasets=None*)
Get lowest resolution area for the provided datasets.

Parameters **datasets** (*iterable*) – Datasets whose areas will be compared. Can be either *xarray.DataArray* objects or identifiers to get the DataArrays from the current Scene. Defaults to all datasets.

missing_datasets
DatasetIDs that have not been loaded.

read(*nodes=None*, ***kwargs*)
Load datasets from the necessary reader.

Parameters

- **nodes** (*iterable*) – DependencyTree Node objects
- ****kwargs** – Keyword arguments to pass to the reader's *load* method.

Returns DatasetDict of loaded datasets

resample(*destination=None*, *datasets=None*, *generate=True*, *unload=True*, *resampler=None*, *reduce_data=True*, ***resample_kwargs*)
Resample datasets and return a new scene.

Parameters

- **destination** (*AreaDefinition, GridDefinition*) – area definition to resample to. If not specified then the area returned by *Scene.max_area()* will be used.
- **datasets** (*list*) – Limit datasets to resample to these specified *DatasetID* objects . By default all currently loaded datasets are resampled.
- **generate** (*bool*) – Generate any requested composites that could not be previously due to incompatible areas (default: True).
- **unload** (*bool*) – Remove any datasets no longer needed after requested composites have been generated (default: True).
- **resampler** (*str*) – Name of resampling method to use. By default, this is a nearest neighbor KDTree-based resampling ('nearest'). Other possible values include 'native', 'ewa', etc. See the *resample* documentation for more information.
- **reduce_data** (*bool*) – Reduce data by matching the input and output areas and slicing the data arrays (default: True)
- **resample_kwargs** – Remaining keyword arguments to pass to individual resampler classes. See the individual resampler class documentation *here* for available arguments.

save_dataset(*dataset_id*, *filename=None*, *writer=None*, *overlay=None*, *compute=True*, ***kwargs*)
Save the *dataset_id* to file using *writer* (default: geotiff).

save_datasets(*writer='geotiff'*, *datasets=None*, *compute=True*, ***kwargs*)
Save all the datasets present in a scene to disk using *writer*.

show(*dataset_id*, *overlay=None*)
Show the *dataset* on screen as an image.

slice(*key*)
Slice Scene by dataset index.

Note: DataArrays that do not have an `area` attribute will not be sliced.

start_time
Return the start time of the file.

to_geoviews(*gvtype=None*, *datasets=None*, *kdims=None*, *vdims=None*, *dynamic=False*)
Convert satpy Scene to geoviews.

Parameters

- **gvtype** (*gv plot type*) – One of gv.Image, gv.LineContours, gv.FilledContours, gv.Points Default to `geoviews.Image`. See Geoviews documentation for details.
- **datasets** (*list*) – Limit included products to these datasets
- **kdims** (*list of str*) – Key dimensions. See geoviews documentation for more information.
- **vdims** – list of str, optional Value dimensions. See geoviews documentation for more information. If not given defaults to first data variable
- **dynamic** – boolean, optional, default False

Returns: geoviews object

to_xarray_dataset(*datasets=None*)
Merge all xr.DataArrays of a scene to a xr.DataSet.

Parameters **datasets** (*list*) – List of products to include in the `xarray.Dataset`

Returns: `xarray.Dataset`

unload(*keepables=None*)
Unload all unneeded datasets.

Datasets are considered unneeded if they weren't directly requested or added to the Scene by the user or they are no longer needed to generate composites that have yet to be generated.

Parameters **keepables** (*iterable*) – DatasetIDs to keep whether they are needed or not.

values()

12.10 satpy.utils module

Module defining various utilities.

class satpy.utils.**OrderedConfigParser**(**args*, ***kwargs*)
Bases: `object`

Intercepts read and stores ordered section names. Cannot use inheritance and super as ConfigParser use old style classes.

read(*filename*)
Reads config file

sections()
Get sections from config file

satpy.utils.**angle2xyz**(*azi*, *zen*)
Convert azimuth and zenith to cartesian.

satpy.utils.**atmospheric_path_length_correction**(*data*, *cos_zen*, *limit=88.0*, *max_sza=95.0*)
Perform Sun zenith angle correction.

This function uses the correction method proposed by Li and Shibata (2006): https://doi.org/10.1175/JAS3682.1

The correction is limited to `limit` degrees (default: 88.0 degrees). For larger zenith angles, the correction is the same as at the `limit` if `max_sza` is *None*. The default behavior is to gradually reduce the correction past `limit` degrees up to `max_sza` where the correction becomes 0. Both `data` and `cos_zen` should be 2D arrays of the same shape.

satpy.utils.**debug_on**()
Turn debugging logging on.

satpy.utils.**ensure_dir**(*filename*)
Checks if the dir of f exists, otherwise create it.

satpy.utils.**get_logger**(*name*)
Return logger with null handler added if needed.

satpy.utils.**in_ipynb**()
Are we in a jupyter notebook?

satpy.utils.**logging_off**()
Turn logging off.

satpy.utils.**logging_on**(*level=30*)
Turn logging on.

satpy.utils.**lonlat2xyz**(*lon*, *lat*)
Convert lon lat to cartesian.

satpy.utils.**proj_units_to_meters**(*proj_str*)
Convert projection units from kilometers to meters.

satpy.utils.**sunzen_corr_cos**(*data*, *cos_zen*, *limit=88.0*, *max_sza=95.0*)
Perform Sun zenith angle correction.

The correction is based on the provided cosine of the zenith angle (`cos_zen`). The correction is limited to `limit` degrees (default: 88.0 degrees). For larger zenith angles, the correction is the same as at the `limit` if `max_sza` is *None*. The default behavior is to gradually reduce the correction past `limit` degrees up to `max_sza` where the correction becomes 0. Both `data` and `cos_zen` should be 2D arrays of the same shape.

satpy.utils.**trace_on**()
Turn trace logging on.

satpy.utils.**xyz2angle**(*x*, *y*, *z*)
Convert cartesian to azimuth and zenith.

satpy.utils.**xyz2lonlat**(*x*, *y*, *z*)
Convert cartesian to lon lat.

12.11 satpy.version module

Git implementation of _version.py.

exception satpy.version.**NotThisMethod**
Bases: Exception

Exception raised if a method is not valid for the current scenario.

class satpy.version.**VersioneerConfig**
Bases: object

Container for Versioneer configuration parameters.

satpy.version.**get_config**()
Create, populate and return the VersioneerConfig() object.

satpy.version.**get_keywords**()
Get the keywords needed to look up the version information.

satpy.version.**get_versions**()
Get version information or return default if unable to do so.

satpy.version.**git_get_keywords**(*versionfile_abs*)
Extract version information from the given file.

satpy.version.**git_pieces_from_vcs**(*tag_prefix*, *root*, *verbose*, *run_command=<function run_command>*)
Get version from 'git describe' in the root of the source tree.

This only gets called if the git-archive 'subst' keywords were *not* expanded, and _version.py hasn't already been rewritten with a short version string, meaning we're inside a checked out source tree.

satpy.version.**git_versions_from_keywords**(*keywords*, *tag_prefix*, *verbose*)
Get version information from git keywords.

satpy.version.**plus_or_dot**(*pieces*)
Return a + if we don't already have one, else return a .

satpy.version.**register_vcs_handler**(*vcs*, *method*)
Decorator to mark a method as the handler for a particular VCS.

satpy.version.**render**(*pieces*, *style*)
Render the given version pieces into the requested style.

satpy.version.**render_git_describe**(*pieces*)
TAG[-DISTANCE-gHEX][-dirty].

Like 'git describe –tags –dirty –always'.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

satpy.version.**render_git_describe_long**(*pieces*)
TAG-DISTANCE-gHEX[-dirty].

Like 'git describe –tags –dirty –always -long'. The distance/hash is unconditional.

Exceptions: 1: no tags. HEX[-dirty] (note: no 'g' prefix)

satpy.version.**render_pep440**(*pieces*)
Build up version string, with post-release "local version identifier".

Our goal: TAG[+DISTANCE.gHEX[.dirty]] . Note that if you get a tagged build and then dirty it, you'll get TAG+0.gHEX.dirty

Exceptions: 1: no tags. git_describe was just HEX. 0+untagged.DISTANCE.gHEX[.dirty]

satpy.version.**render_pep440_old**(*pieces*)
TAG[.postDISTANCE[.dev0]] .

The ".dev0" means dirty.

Eexceptions: 1: no tags. 0.postDISTANCE[.dev0]

satpy.version.**render_pep440_post**(*pieces*)
TAG[.postDISTANCE[.dev0]+gHEX] .

The ".dev0" means dirty. Note that .dev0 sorts backwards (a dirty tree will appear "older" than the corresponding clean one), but you shouldn't be releasing software with -dirty anyways.

Exceptions: 1: no tags. 0.postDISTANCE[.dev0]

satpy.version.**render_pep440_pre**(*pieces*)
TAG[.post.devDISTANCE] – No -dirty.

Exceptions: 1: no tags. 0.post.devDISTANCE

satpy.version.**run_command**(*commands*, *args*, *cwd=None*, *verbose=False*, *hide_stderr=False*, *env=None*)
Call the given command(s).

satpy.version.**versions_from_parentdir**(*parentdir_prefix*, *root*, *verbose*)
Try to determine the version from the parent directory name.

Source tarballs conventionally unpack into a directory that includes both the project name and a version string. We will also support searching up two directory levels for an appropriately named parent directory

12.12 Module contents

SatPy Package initializer.

CHAPTER 13

FAQ

Below you'll find frequently asked questions, performance tips, and other topics that don't really fit in to the rest of the SatPy documentation.

If you have any other questions that aren't answered here feel free to make an issue on GitHub or talk to us on the Slack team or mailing list. See the *contributing* documentation for more information.

Topics

13.1 Why is SatPy slow on my powerful machine?

SatPy depends heavily on the dask library for its performance. However, on some systems dask's default settings can actually hurt performance. By default dask will create a "worker" for each logical core on your system. In most systems you have twice as many logical cores (also known as threaded cores) as physical cores. Managing and communicating with all of these workers can slow down dask, especially when they aren't all being used by most SatPy calculations. One option is to limit the number of workers by doing the following at the **top** of your python code:

```
import dask
from multiprocessing.pool import ThreadPool
dask.config.set(pool=ThreadPool(8))
# all other SatPy imports and code
```

This will limit dask to using 8 workers. Typically numbers between 4 and 8 are good starting points. Number of workers can also be set from an environment variable before running the python script, so code modification isn't necessary:

```
DASK_NUM_WORKERS=4 python myscript.py
```

Similarly, if you have many workers processing large chunks of data you may be using much more memory than you expect. If you limit the number of workers *and* the size of the data chunks being processed by each worker you can reduce the overall memory usage. Default chunk size can be configured in SatPy by setting the following environment variable:

```
export PYTROLL_CHUNK_SIZE=2048
```

This could also be set inside python using `os.environ`, but must be set **before** SatPy is imported. This value defaults to 4096, meaning each chunk of data will be 4096 rows by 4096 columns. In the future setting this value will change to be easier to set in python.

13.2 Why multiple CPUs are used even with one worker?

Many of the underlying Python libraries use math libraries like BLAS and LAPACK written in C or FORTRAN, and they are often compiled to be multithreaded. If necessary, it is possible to force the number of threads they use by setting an environment variable:

```
OMP_NUM_THREADS=2 python myscript.py
```

13.3 What is the difference between number of workers and number of threads?

The above questions handle two different stages of parallellization: Dask workers and math library threading.

The number of Dask workers affect how many separate tasks are started, effectively telling how many chunks of the data are processed at the same time. The more workers are in use, the higher also the memory usage will be.

The number of threads determine how much parallel computations are run for the chunk handled by each worker. This has minimal effect on memory usage.

The optimal setup is often a mix of these two settings, for example

```
DASK_NUM_WORKERS=2 OMP_NUM_THREADS=4 python myscript.py
```

would create two workers, and each of them would process their chunk of data using 4 threads when calling the underlying math libraries.

13.4 How do I avoid memory errors?

If your environment is using many dask workers, it may be using more memory than it needs to be using. See the "Why is SatPy slow on my powerful machine?" question above for more information on changing SatPy's memory usage.

Table 1: SatPy Readers

Description	Reader name	Status
MSG (Meteosat 8 to 11) SEVIRI data in HRIT format	*seviri_l1b_hrit*	Nominal
MSG (Meteosat 8 to 11) SEVIRI data in native format	*seviri_l1b_native*	HRV full disk data cannot be remapped.
MSG (Meteosat 8 to 11) SEVIRI data in netCDF format	*seviri_l1b_nc*	HRV channel not supported, incomplete metadata in the files. EUMETSAT has been notified.
Himawari 8 and 9 AHI data in HSD format	*ahi_hsd*	Nominal
Himawari 8 and 9 AHI data in HRIT format	*ahi_hrit*	Nominal
MTSAT-1R JAMI data in JMA HRIT format	*jami_hrit*	Beta
MTSAT-2 Imager data in JMA HRIT format	*mtsat2-imager_hrit*	Beta
GOES 16 imager data in netcdf format	*abi_l1b*	Nominal
GOES 11 to 15 imager data in HRIT format	*goes-imager_hrit*	Nominal
GOES 8 to 15 imager data in netCDF format (from NOAA CLASS)	*goes-imager_nc*	Beta
Electro-L N2 MSU-GS data in HRIT format	*electrol_hrit*	Nominal
NOAA 15 to 19, Metop A to C AVHRR data in AAPP format	*avhrr_l1b_aapp*	Nominal
Metop A to C AVHRR in native level 1 format	*avhrr_l1b_eps*	Nominal
Tiros-N, NOAA 7 to 19 AVHRR data in GAC and LAC format	*avhrr_l1b_gaclac*	Nominal
NOAA 15 to 19 AVHRR data in raw HRPT format	*avhrr_l1b_hrpt*	In development
GCOM-W1 AMSR2 data in HDF5 format	*amsr2_l1b*	Nominal
MTG FCI Level 1C data for Full Disk High Spectral Imagery (FDHSI) in netcdf format	*fci_l1c_fdhsi*	In development
Callipso Caliop Level 2 Cloud Layer data (v3) in EOS-hdf4 format	*caliop_l2_cloud*	In development
Terra and Aqua MODIS data in EOS-hdf4 level-1 format as produced by IMAPP and IPOPP or downloaded from LAADS	*modis_l1b*	Nominal
NWCSAF GEO 2016 products in netCDF4 format (limited to SEVIRI)	*nwcsaf-geo*	In development
NWCSAF PPS 2014, 2018 products in netCDF4 format	*nwcsaf-pps_nc*	Not yet support for remapped netCDF products. Only the standard swath based output is supported. CPP products not supported yet
Sentinel-1 A and B SAR-C data in SAFE format	*sar-c_safe*	Nominal
Sentinel-2 A and B MSI data in SAFE format	*msi_safe*	Nominal
Sentinel-3 A and B OLCI Level 1B data in netCDF4 format	*olci_l1b*	Nominal
Sentinel-3 A and B OLCI Level 2 data in netCDF4 format	*olci_l2*	Nominal

Continued on next page

Table 1 – continued from previous page

Description	Reader name	Status
Sentinel-3 A and B SLSTR data in netCDF4 format	*slstr_l1b*	In development
OSISAF SST data in GHRSST (netcdf) format	*ghrsst_l3c_sst*	In development
NUCAPS EDR Retrieval in NetCDF4 format	*nucaps*	Nominal
NOAA Level 2 ACSPO SST data in netCDF4 format	*acspo*	Nominal
GEOstationary Cloud Algorithm Test-bed (GEOCAT)	*geocat*	Nominal
The Clouds from AVHRR Extended (CLAVR-x)	*clavrx*	Nominal
SNPP VIIRS data in HDF5 SDR format	*viirs_sdr*	Nominal
SNPP VIIRS data in netCDF4 L1B format	*viirs_l1b*	Nominal
SNPP VIIRS SDR data in HDF5 Compact format	*viirs_compact*	Nominal
AAPP MAIA VIIRS and AVHRR products in hdf5 format	*maia*	Nominal
VIIRS EDR Flood data in hdf4 format	*viirs_edr_flood*	Beta
GRIB2 format	*grib*	Beta
SCMI ABI L1B format	*abi_l1b_scmi*	Beta

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

A

B

C

D

E

F

G

H

I

O

P

R

S

T

U

V

W

X

Z