satpy documentation

Release 0.6.2

SMHI

Jul 13, 2017

Contents

The SatPy package 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.

Get 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). In the base distribution, we provide support for the following readers:

Table 1: SatPy Readers

Description	Reader name	Status
MSG (Meteosat 8 to 11) Seviri data in HRIT format	*hrit_msg*	Nominal
MSG (Meteosat 8 to 11) SEVIRI data in native format	*native_msg*	No support for reading sub-section of the full disk. HRV data cannot be remapped.
Himawari 8 and 9 AHI data in HSD format	*ahi_hsd*	Nominal
Himawari 8 and 9 AHI data in HRIT format	*hrit_jma*	Nominal
GOES 16 imager data in netcdf format	*abi_l1b*	Nominal
GOES 11 to 15 imager data in HRIT format	*mipp_xrit*	Nominal
Electro-L N2 MSU-GS data in HRIT format	*hrit_electrol*	Nominal
NOAA 15 to 19, Metop A to C AVHRR data in AAPP format	*aapp_l1b*	Nominal
Metop A to C AVHRR in native level 1 format	*epsl1b*	Nominal
Tiros-N, NOAA 7 to 19 AVHRR data in GAC and LAC format	*gac_lac_l1b*	Nominal
NOAA 15 to 19 AVHRR data in raw HRPT format	*hrpt*	Nominal
GCOM-W1 AMSR2 data in HDF5 format	*amsr2_l1b*	Nominal
MTG FCI data in netcdf format	*fci_fdhsi*	Nominal
Callipso Caliop data in EOS-hdf4 format	*hdf4_caliopv3*	Nominal
Terra and Aqua MODIS data in EOS-hdf4 format	*hdfeos_l1b*	Nominal
NWCSAF MSG 2016 products in netCDF4 format	*nc_nwcsaf_msg*	Nominal
NWCSAF PPS 2014 products in netCDF4 format	*nc_nwcsaf_pps*	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*	Nominal
Sentinel-2 A and B MSI data in SAFE format	*safe_msi*	Nominal
Sentinel-3 A and B OLCI data in netCDF4 format	*nc_olci*	Nominal
Sentinel-3 A and B SLSTR data in netCDF4 format	*nc_slstr*	Nominal
OSISAF SST data in GHRSST (netcdf) format	*ghrsst_osisaf*	Nominal
NUCAPS EDR Retrieval in NetCDF4 format	*nucaps*	Nominal

Reprojection of data is also available through the use of pyresample.

CHAPTER 1

Installation instructions

Pip based installation

satpy is available from pypi. A sandbox environment for *satpy* can be created using Virtualenv.

Installation using pip:

```
$ pip install satpy
```

This will install *satpy* and all it's python dependendencies from pypi. However you have to asure, that all non python dependencies are installed on your machine.

Installation based on conda

The satpy repository contains an environment file to install *satpy* and all its dependencies (including non python dependencies) via conda. For now there is only one file available to install *satpy* on python 2.7 and numpy 1.11 (*satpy-env_np111py27.yml*). Environment files for other python or numpy versions can be created by adapting this file.

After downloading the file you can install *satpy* in an environment called *satpy-env* using *conda*:

```
$ conda env create -f satpy-env_np<xxx>py<yy>.yml
```

To activate this environment use

```
$ source activate satpy-env
```

CHAPTER 2

Quickstart

Loading data

Changed in version 2.0.0-alpha.1: New syntax

To work with weather satellite data, one has to create an instance of the `Scene` class. In order for satpy to get access to the data, either the current wording directory has to be set to the directory containing the data files, or the *base_dir* keyword argument has to be provided on scene creation:

```
>>> import os
>>> os.chdir("/home/a001673/data/satellite/Meteosat-10/seviri/lvl1.5/2015/04/20/HRIT")
>>> from satpy import Scene
>>> from datetime import datetime
>>> time_slot = datetime(2015, 4, 20, 10, 0)
>>> global_scene = Scene(platform_name="Meteosat-10", sensor="seviri", reader="hrit_
↪msg", start_time=datetime(2015, 4, 20, 10, 0))
```

or:

```
>>> from satpy.scene import Scene
>>> from datetime import datetime
>>> time_slot = datetime(2015, 4, 20, 10, 0)
>>> global_scene = Scene(platform_name="Meteosat-10", sensor="seviri", reader="hrit_
↪msg", start_time=datetime(2015, 4, 20, 10, 0), base_dir="/home/a001673/data/
↪satellite/Meteosat-10/seviri/lvl1.5/2015/04/20/HRIT")
>>>
```

For some platforms, it might be necessary to also specify an *end_time*:

```
>>> Scene(platform_name="SNPP", sensor="viirs", start_time=datetime(2015, 3, 11, 11,␣
↪20), end_time=datetime(2015, 3, 11, 11, 26))
```

Loading weather satellite data with satpy is as simple as calling the `Scene.load()` method:

```
>>> global_scene.load([0.6, 0.8, 10.8])
>>> print global_scene

seviri/IR_108:
        area: On-the-fly area
        start_time: 2015-04-20 10:00:00
        units: K
        wavelength_range: (9.8, 10.8, 11.8) μm
        shape: (3712, 3712)
seviri/VIS006:
        area: On-the-fly area
        start_time: 2015-04-20 10:00:00
        units: %
        wavelength_range: (0.56, 0.635, 0.71) μm
        shape: (3712, 3712)
seviri/VIS008:
        area: On-the-fly area
        start_time: 2015-04-20 10:00:00
        units: %
        wavelength_range: (0.74, 0.81, 0.88) μm
        shape: (3712, 3712)
```

As you can see, this loads the visible and IR channels provided as argument to the `load()` method as a list of wavelengths in micrometers. Another way to load the channels is to provide the names instead:

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

To have a look at the available bands you should be able to load with your *Scene* object, you can call the `available_datasets()` method:

```
>>> global_scene.available_dataset_names()

[u'HRV',
 u'IR_108',
 u'IR_120',
 u'VIS006',
 u'WV_062',
 u'IR_039',
 u'IR_134',
 u'IR_097',
 u'IR_087',
 u'VIS008',
 u'IR_016',
 u'WV_073']
```

To access the loaded data:

```
>>> print global_scene[0.6]
```

or:

```
>>> print global_scene["VIS006"]
```

To visualize it:

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

To combine them:

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

Generating composites

The easiest way to generate composites is to *load* them:

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

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

```
>>> global_scene.available_composites()

[u'overview_sun',
 u'airmass',
 u'natural',
 u'night_fog',
 u'overview',
 u'green_snow',
 u'dust',
 u'fog',
 u'natural_sun',
 u'cloudtop',
 u'convection',
 u'ash']
```

To save a composite to disk:

```
>>> global_scene.save_dataset('overview', 'my_nice_overview.png')
```

One can also specify which writer to use for filenames with non-standard extensions

```
>>> global_scene.save_dataset('overview', 'my_nice_overview.stupidextension', writer=
↪'geotiff')
```

Resampling

Until now, we have used the channels directly as provided by the satellite, that is in satellite projection. Generating composites thus produces views in satellite projection, *i.e.* as viewed by the satellite.

Most often however, we will want to resample the data onto a specific area so that only the area of interest is depicted in the RGB composites.

Here is how we do that:

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

Now we have resampled channel data and composites onto the "eurol" area in the *local_scene* variable and we can operate as before to display and save RGB composites:

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

The image is automatically saved here in GeoTiff format.

The default resampling method is nearest neighbour. Also bilinear interpolation is available, which can be used by adding *resampler="bilinear"* keyword:

```
>>> local_scene = global_scene.resample("euro4", resampler="bilinear")
>>>
```

To make resampling faster next time (when resampling geostationary satellite data), it is possible to save the resampling coefficients and use more CPUs when calculating the coefficients on the first go:

```
>>> local_scene = global_scene.resample("euro4", resampler="bilinear",
...                                     nprocs=4, cache_dir="/var/tmp")
>>>
```

Making custom composites

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

```
>>> from satpy.composites import RGBCompositor
>>> compositor = RGBCompositor("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 3

satpy package

Subpackages

satpy.composites package

satpy.composites module

Base classes for composite objects.

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

Bases: *satpy.composites.RGBCompositor*

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

Bases: *satpy.composites.CompositeBase*

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

Bases: *satpy.composites.ColormapCompositor*

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

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

Bases: *satpy.composites.RGBCompositor*

A compositor that uses colormaps.

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*, *metadata_requirements=None*, ***kwargs*)

Bases: satpy.dataset.InfoObject

apply_modifier_info(*origin*, *destination*)

class satpy.composites.**CompositorLoader**(*ppp_config_dir='/home/docs/checkouts/readthedocs.org/user_builds/satpy/che*
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*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.RGBCompositor*

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

class satpy.composites.**Dust**(*name*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.RGBCompositor*

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

Error raised upon compositing things of different shapes.

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

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

class satpy.composites.**PaletteCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.ColormapCompositor*

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

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

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

coszen = {}

satpy.composites.**show**(*data*, *filename=None*)
Show the stretched data.

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.

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.

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

Equalized DNB composite using the Zinke algorithm.

static gain_factor(*theta*)

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

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.

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

Bases: *satpy.composites.CompositeBase*

class satpy.composites.viirs.**VIIRSTrueColor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=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*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.RGBCompositor*

class satpy.composites.**CO2Corrector**(*name*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.CompositeBase*

class satpy.composites.**ColorizeCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.ColormapCompositor*

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

class satpy.composites.**ColormapCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.RGBCompositor*

A compositor that uses colormaps.

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*, *metadata_requirements=None*, ***kwargs*)
Bases: satpy.dataset.InfoObject

apply_modifier_info(*origin*, *destination*)

class satpy.composites.**CompositorLoader**(*ppp_config_dir='/home/docs/checkouts/readthedocs.org/user_builds/satpy/che*
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*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.RGBCompositor*

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

class satpy.composites.**Dust**(*name*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.RGBCompositor*

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

Error raised upon compositing things of different shapes.

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

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

class satpy.composites.**PaletteCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, *metadata_requirements=None*, ***kwargs*)
Bases: *satpy.composites.ColormapCompositor*

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

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

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

coszen = {}

satpy.composites.**show**(*data*, *filename=None*)
Show the stretched data.

satpy.readers package

satpy.readers module

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.

get_best_choice(*key*, *choices*)

get_item(*name_or_wl*, *resolution=None*, *polarization=None*, *calibration=None*, *modifiers=None*)

get_key(*key*)

get_keys(*name_or_wl*, *resolution=None*, *polarization=None*, *calibration=None*, *modifiers=None*)

get_keys_by_datasetid(*did*)

keys(*names=False*, *wavelengths=False*)

exception satpy.readers.**MalformedConfigError**
Bases: exceptions.Exception

class satpy.readers.**ReaderFinder**(*ppp_config_dir='/home/docs/checkouts/readthedocs.org/user_builds/satpy/checkouts/late*
base_dir=None, *start_time=None*, *end_time=None*, *area=None*)
Bases: object

Find readers given a scene, filenames, sensors, and/or a reader_name

config_files()

satpy.readers.**load_reader**(*reader_configs*, ***reader_kwargs*)
Import and setup the reader from *reader_info*

satpy.readers.**read_reader_config**(*config_files*)
Read the reader *config_files* and return the info extracted.

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_ids*, *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.acspo module

ACSPO SST Reader

See the following page for more information:

https://podaac.jpl.nasa.gov/dataset/VIIRS_NPP-OSPO-L2P-v2.3

class satpy.readers.acspo.**ACSPOFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*)

Bases: satpy.readers.netcdf_utils.NetCDF4FileHandler

ACSPO L2P SST File Reader

end_time

get_dataset(*dataset_id*, *ds_info*, *out=None*, *xslice=slice(None, None, None)*, *yslice=slice(None, None, None)*)

Load data array and metadata from file on disk.

get_shape(*ds_id*, *ds_info*)

Get numpy array shape for the specified dataset.

Parameters

- **ds_id** (*DatasetID*) – ID of dataset that will be loaded
- **ds_info** (*dict*) – Dictionary of dataset information from config file

Returns (rows, cols)

Return type tuple

platform_name

sensor_name

start_time

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_dataset(*key*, *info*)

Get calibrated channel data.

get_full_angles()

Get the interpolated lons/lats.

get_full_lonlats()

Get the interpolated lons/lats.

get_lonlat(*row*, *col*)

Get lons/lats for given indices. WARNING: if the lon/lats were not expanded, this will refer to the tiepoint data.

get_lonlats()

keys()
List of reader's keys.

platform_name

sensor_name

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

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

start_time

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.mipp_xrit module

satpy.readers.viirs_l1b module

Interface to VIIRS L1B format

class satpy.readers.viirs_l1b.**VIIRSL1BFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*)
Bases: satpy.readers.netcdf_utils.NetCDF4FileHandler

VIIRS L1B File Reader

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

end_orbit_number

end_time

get_dataset(*dataset_id*, *ds_info*, *out=None*, *xslice=slice(None, None, None)*, *yslice=slice(None, None, None)*)

get_shape(*ds_id*, *ds_info*)

platform_name

sensor_name

start_orbit_number

start_time

satpy.readers.viirs_sdr module

Interface to VIIRS SDR format

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*)
Bases: satpy.readers.hdf5_utils.HDF5FileHandler

VIIRS HDF5 File Reader

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

end_orbit_number

end_time

get_dataset(*dataset_id*, *ds_info*, *out=None*)

get_file_units(*dataset_id*, *ds_info*)

get_shape(*ds_id*, *ds_info*)

platform_name

scale_swath_data(*data*, *mask*, *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=True*, ***kwargs*)
Bases: satpy.readers.yaml_reader.FileYAMLReader

Custom file reader for finding VIIRS SDR geolocation at runtime.

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.

get_best_choice(*key*, *choices*)

get_item(*name_or_wl*, *resolution=None*, *polarization=None*, *calibration=None*, *modifiers=None*)

get_key(*key*)

get_keys(*name_or_wl*, *resolution=None*, *polarization=None*, *calibration=None*, *modifiers=None*)

get_keys_by_datasetid(*did*)

keys(*names=False*, *wavelengths=False*)

exception satpy.readers.**MalformedConfigError**
Bases: exceptions.Exception

class satpy.readers.**ReaderFinder**(*ppp_config_dir='/home/docs/checkouts/readthedocs.org/user_builds/satpy/checkouts/late*
base_dir=None, *start_time=None*, *end_time=None*, *area=None*)
Bases: object

Find readers given a scene, filenames, sensors, and/or a reader_name

config_files()

satpy.readers.**load_reader**(*reader_configs*, ***reader_kwargs*)
Import and setup the reader from *reader_info*

satpy.readers.**read_reader_config**(*config_files*)
Read the reader *config_files* and return the info extracted.

satpy.writers package

satpy.writers module

Shared objects of the various writer classes.

For now, this includes enhancement configuration utilities.

class satpy.writers.**EnhancementDecisionTree**(**config_files*, ***kwargs*)
Bases: object

add_config_to_tree(**config_files*)

any_key = None

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.

add_sensor_enhancements(*sensor*)

apply(*img*, ***info*)

get_sensor_enhancement_config(*sensor*)

class satpy.writers.**ImageWriter**(*name=None*, *fill_value=None*, *file_pattern=None*, *enhancement_config=None*, *base_dir=None*, ***kwargs*)
Bases: *satpy.writers.Writer*

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *overlay=None*, ***kwargs*)
Saves the *dataset* to a given *filename*.

save_image(*img*, *filename=None*, ***kwargs*)

class satpy.writers.**Writer**(*name=None*, *fill_value=None*, *file_pattern=None*, *base_dir=None*, ***kwargs*)
Bases: *satpy.plugin_base.Plugin*

Writer plugins. They must implement the *save_image* method. This is an abstract class to be inherited.

create_filename_parser(*base_dir*)

get_filename(***kwargs*)

load_section_writer(*section_name*, *section_options*)

save_dataset(*dataset*, *filename=None*, *fill_value=None*, ***kwargs*)
Saves the *dataset* to a given *filename*.

save_datasets(*datasets*, ***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.

satpy.writers.**add_overlay**(*orig*, *area*, *coast_dir*, *color=(0, 0, 0)*, *width=0.5*, *resolution=None*)
Add coastline and political borders to image, using *color* (tuple of integers between 0 and 255). Warning: Loses

the masks ! *resolution* is chosen automatically if None (default), otherwise it should be one of: +—–+————————-+————+ | ‘f’ | Full resolution | 0.04 km | | ‘h’ | High resolution | 0.2 km | | ‘i’ | Intermediate resolution | 1.0 km | | ‘l’ | Low resolution | 5.0 km | | ‘c’ | Crude resolution | 25 km | +—–+————————-+————+

satpy.writers.**get_enhanced_image**(*dataset*, *enhancer=None*, *fill_value=None*, *ppp_config_dir=None*, *enhancement_config_file=None*, *overlay=None*)

satpy.writers.**show**(*dataset*, ***kwargs*)
Display the dataset as an image.

satpy.writers.**to_image**(*dataset*, *copy=True*, ***kwargs*)

satpy.writers.geotiff module

GeoTIFF writer objects for creating GeoTIFF files from *Dataset* objects.

class satpy.writers.geotiff.**GeoTIFFWriter**(*floating_point=False*, *tags=None*, ***kwargs*)
Bases: *satpy.writers.ImageWriter*

GDAL_OPTIONS = (‘tfw’, ‘rpb’, ‘rpctxt’, ‘interleave’, ‘tiled’, ‘blockxsize’, ‘blockysize’, ‘nbits’, ‘compress’, ‘num_threads’

save_image(*img*, *filename=None*, *floating_point=False*, ***kwargs*)
Save the image to the given *filename* in geotiff format. *floating_point* allows the saving of ‘L’ mode images in floating point format if set to True.

satpy.writers.simple_image module

class satpy.writers.simple_image.**PillowWriter**(***kwargs*)
Bases: *satpy.writers.ImageWriter*

save_image(*img*, *filename=None*, ***kwargs*)

Module contents

Shared objects of the various writer classes.

For now, this includes enhancement configuration utilities.

class satpy.writers.**EnhancementDecisionTree**(**config_files*, ***kwargs*)
Bases: object

add_config_to_tree(**config_files*)

any_key = None

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.

add_sensor_enhancements(*sensor*)

apply(*img*, ***info*)

get_sensor_enhancement_config(*sensor*)

class satpy.writers.**ImageWriter**(*name=None*, *fill_value=None*, *file_pattern=None*, *enhancement_config=None*, *base_dir=None*, ***kwargs*)

Bases: *satpy.writers.Writer*

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *overlay=None*, ***kwargs*)

Saves the *dataset* to a given *filename*.

save_image(*img*, *filename=None*, ***kwargs*)

class satpy.writers.**Writer**(*name=None*, *fill_value=None*, *file_pattern=None*, *base_dir=None*, ***kwargs*)

Bases: *satpy.plugin_base.Plugin*

Writer plugins. They must implement the *save_image* method. This is an abstract class to be inherited.

create_filename_parser(*base_dir*)

get_filename(***kwargs*)

load_section_writer(*section_name*, *section_options*)

save_dataset(*dataset*, *filename=None*, *fill_value=None*, ***kwargs*)

Saves the *dataset* to a given *filename*.

save_datasets(*datasets*, ***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.

satpy.writers.**add_overlay**(*orig*, *area*, *coast_dir*, *color=(0, 0, 0)*, *width=0.5*, *resolution=None*)

Add coastline and political borders to image, using *color* (tuple of integers between 0 and 255). Warning: Loses the masks ! *resolution* is chosen automatically if None (default), otherwise it should be one of: +—–+————————-+———+ | ‘f’ | Full resolution | 0.04 km | | ‘h’ | High resolution | 0.2 km | | ‘i’ | Intermediate resolution | 1.0 km | | ‘l’ | Low resolution | 5.0 km | | ‘c’ | Crude resolution | 25 km | +—–+————————-+———+

satpy.writers.**get_enhanced_image**(*dataset*, *enhancer=None*, *fill_value=None*, *ppp_config_dir=None*, *enhancement_config_file=None*, *overlay=None*)

satpy.writers.**show**(*dataset*, ***kwargs*)

Display the dataset as an image.

satpy.writers.**to_image**(*dataset*, *copy=True*, ***kwargs*)

satpy module

SatPy Package initializer.

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

The base plugin class. It is not to be used as is, it has to be inherited by other classes.

get_section_type(*section_name*)

load_config(*conf*)

satpy.projectable module

satpy.resample module

satpy.scene module

Scene objects to hold satellite data.

class satpy.scene.**Scene**(*filenames=None*, *ppp_config_dir='/home/docs/checkouts/readthedocs.org/user_builds/satpy/checkouts/*
reader=None, *base_dir=None*, *sensor=None*, *start_time=None*, *end_time=None*, *area=None*, *reader_kwargs=None*, ***metadata*)

Bases: satpy.dataset.InfoObject

The almighty scene class.

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()

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*)

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.

compute(*nodes=None*)
Compute all the composites contained in *requirements*.

create_reader_instances(*filenames=None*, *base_dir=None*, *reader=None*, *reader_kwargs=None*)
Find readers and return their instanciations.

end_time
Return the end time of the file.

get_writer(*writer='geotiff'*, ***kwargs*)

get_writer_by_ext(*extension*, ***kwargs*)

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*, *compute=True*, *unload=True*, ***kwargs*)
Read, compute and unload.

load_writer_config(*config_files*, ***kwargs*)

missing_datasets

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

read_composites(*compositor_nodes*)
Read (generate) composites.

read_datasets(*dataset_nodes*, ***kwargs*)
Read the given datasets from file.

resample(*destination*, *datasets=None*, *compute=True*, *unload=True*, ***resample_kwargs*)
Resample the datasets and return a new scene.

save_dataset(*dataset_id*, *filename=None*, *writer=None*, *overlay=None*, ***kwargs*)
Save the *dataset_id* to file using *writer* (geotiff by default).

save_datasets(*writer='geotiff'*, *datasets=None*, ***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.

start_time
Return the start time of the file.

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 compute composites that have yet to be computed.

Parameters **keepables** (*iterable*) – DatasetIDs to keep whether they are needed or not.

satpy.tools module

Helper functions for eg. performing Sun zenith angle correction.

satpy.tools.**sunzen_corr_cos**(*data*, *cos_zen*, *limit=88.0*)
Perform Sun zenith angle correction to the given *data* using 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*. Both *data* and *cos_zen* are given as 2-dimensional Numpy arrays or Numpy MaskedArrays, and they should have equal shapes.

satpy.utils module

Module defining various utilities.

class satpy.utils.**NullHandler**(*level=0*)
Bases: logging.Handler

Empty handler.

emit(*record*)
Record a message.

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.**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 handle

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.**strftime**(*utctime*, *format_string*)
Like datetime.strftime, except it works with string formatting conversion specifier items on windows, making the assumption that all conversion specifiers use mapping keys.

E.g.: >>> from datetime import datetime >>> t = datetime.utcnow() >>> a = "blabla%Y%d%m-%H%M%S-%(value)s" >>> strftime(t, a) 'blabla20120911-211448-%(value)s'

satpy.utils.**xyz2angle**(*x*, *y*, *z*)
Convert cartesian to azimuth and zenith.

satpy.utils.**xyz2lonlat**(*x*, *y*, *z*)
Convert cartesian to lon lat.

Module contents

SatPy Package initializer.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X