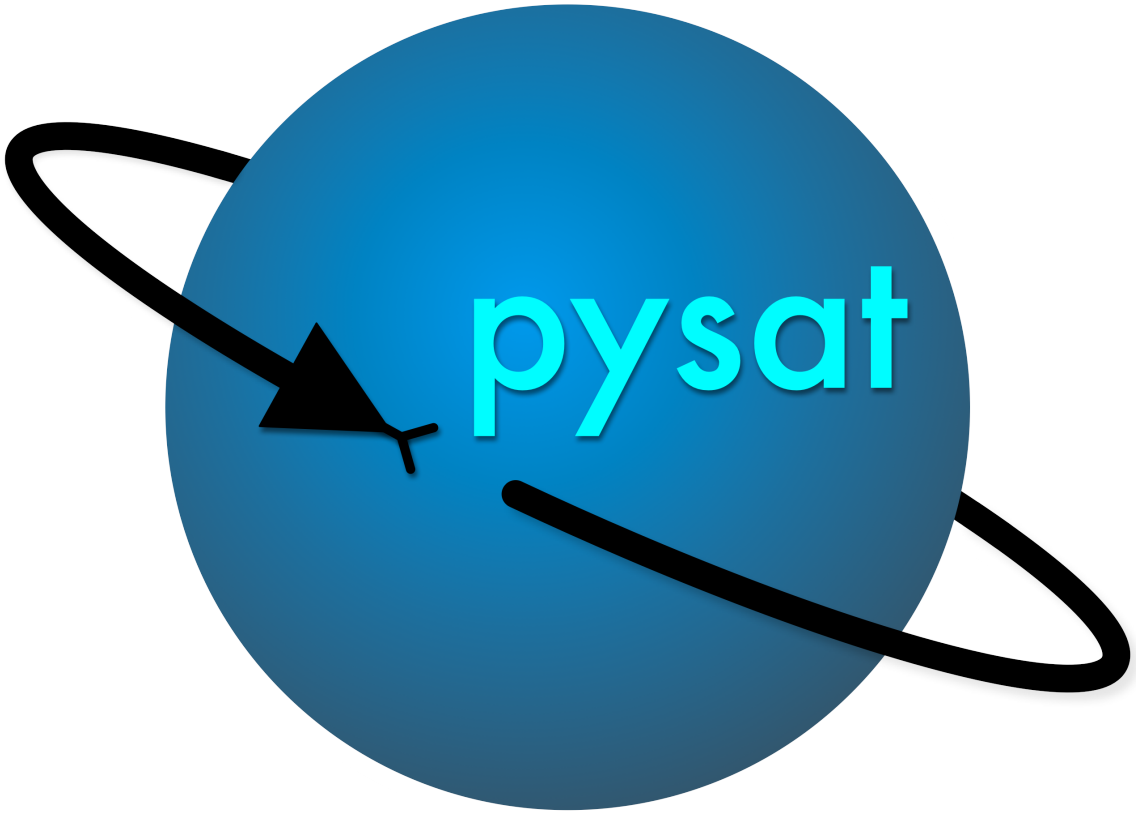


Python Satellite Data Analysis Toolkit (pysat) Documentation

Release 2.3.0

Russell Stoneback

Apr 01, 2021

Contents

1 Introduction **1**

2 Citations in the pysat ecosystem **3**

3 Installation **5**

4 Quick-Start **9**

5 Tutorial **11**
- 5.1 Basics . . . 11
- 5.2 Verbosity . . . 16
- 5.3 Custom Functions . . . 16
- 5.4 Initial Instrument Independence . . . 19
- 5.5 Iteration . . . 22
- 5.6 Orbit Support . . . 23
- 5.7 Iteration and Instrument Independent Analysis . . . 26
- 5.8 Summary Flow Charts . . . 27

6 Sample Scientific Analysis **29**
- 6.1 Seasonal Occurrence by Orbit . . . 29
- 6.2 Orbit-by-Orbit Plots . . . 31
- 6.3 Seasonal Averaging of Ion Drifts and Density Profiles . . . 35

7 Supported Instruments **41**
- 7.1 C/NOFS IVM . . . 41
- 7.2 C/NOFS PLP . . . 42
- 7.3 C/NOFS VEFI . . . 43
- 7.4 CHAMP-STAR . . . 44
- 7.5 COSMIC GPS . . . 44
- 7.6 DE2 LANG . . . 45
- 7.7 DE2 NACS . . . 46
- 7.8 DE2 RPA . . . 47
- 7.9 DE2 WATS . . . 48
- 7.10 Demeter IAP . . . 49
- 7.11 DMSP IVM . . . 50
- 7.12 ICON EUV . . . 53
- 7.13 ICON FUV . . . 53

7.14 ICON IVM . . . 54
7.15 ICON MIGHTI . . . 55
7.16 ISS-FPMU . . . 56
7.17 JRO ISR . . . 56
7.18 OMNI_HRO . . . 57
7.19 ROCSAT-1 IVM . . . 59
7.20 SPORT IVM . . . 59
7.21 SuperDARN . . . 59
7.22 SuperMAG . . . 60
7.23 SW Dst . . . 61
7.24 SW F107 . . . 61
7.25 SW Kp . . . 62
7.26 TIMED/SABER . . . 63
7.27 TIMED/SEE . . . 64
7.28 UCAR TIEGCM . . . 65

8 Adding a New Instrument **67**
8.1 Naming Conventions . . . 67
8.2 Required Routines . . . 68
8.3 Optional Routines and Support . . . 71
8.4 Logging . . . 72
8.5 Testing Support . . . 73
8.6 Data Acknowledgements . . . 73
8.7 Supported Instrument Templates . . . 74

9 API **77**
9.1 Instrument . . . 77
9.2 Instrument Methods . . . 84
9.3 Instrument Templates . . . 97
9.4 Constellation . . . 111
9.5 Custom . . . 113
9.6 Files . . . 115
9.7 Meta . . . 118
9.8 Orbits . . . 123
9.9 Seasonal Analysis . . . 125
9.10 Utilities . . . 129

10 Contributing **137**
10.1 Short version . . . 137
10.2 Bug reports . . . 137
10.3 Feature requests and feedback . . . 137
10.4 Development . . . 138
10.5 Pull Request Guidelines . . . 138

11 Frequently Asked Questions **139**

Python Module Index **141**

Index **143**

CHAPTER 1

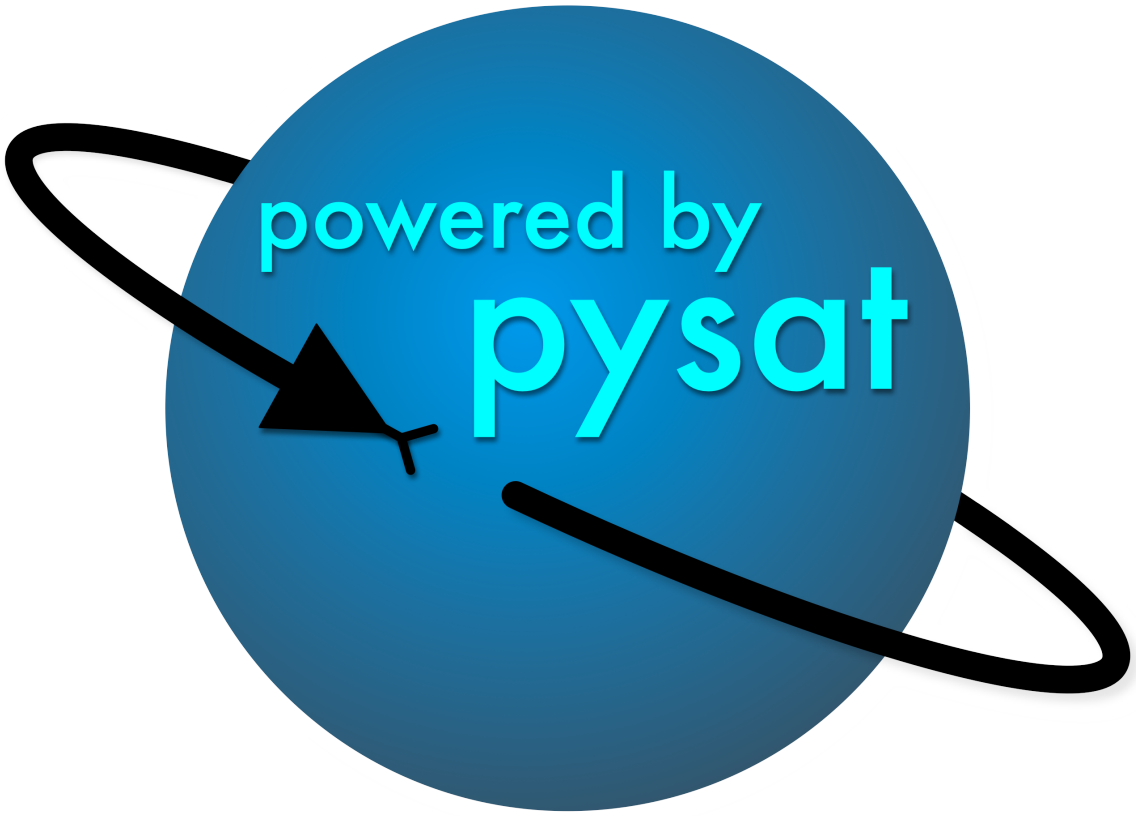
Introduction

Every scientific instrument has unique properties though the general process for science data analysis is independent of platform. Find and download the data, write code to load the data, clean the data, apply custom analysis functions, and plot the results. The Python Satellite Data Analysis Toolkit (pysat) provides a framework for this general process that builds upon these commonalities to simplify adding new instruments, reduce data management overhead, and enable instrument independent analysis routines. Though pysat was initially designed for in-situ satellite based measurements it aims to support all instruments in space science.

This document covers installation, a tutorial on pysat including demonstration code, coverage of supported instruments, an overview of adding new instruments to pysat, and an API reference.

Logos

Does your project use pysat? If so, grab a "powered by pysat" logo!


powered by
pysat

CHAPTER 2

Citations in the pysat ecosystem

When referring to this software package, please cite the original paper by Stoneback et al [2018] https://doi.org/10.1029/2018JA025297 as well as the package https://doi.org/10.5281/zenodo.1199703. Note that this doi will always point to the latest version of the code. A list of dois for all versions can be found at the Zenodo page.

Example for citation in BibTex for a generalized version:

```
@misc{pysat200,
  author       = {Stoneback, R.A. and
                  Klenzing, J.H. and
                  Burrell, A.G. and
                  Spence, C. and
                  Depew, M. and
                  Hargrave, N. and
                  von Bose, V. and
                  Luis, S. and
                  Iyer, G.},
  title        = {Python Satellite Data Analysis Toolkit (pysat) vX.Y.Z},
  month        = jul,
  year         = 2019,
  doi          = {10.5281/zenodo.1199703},
  url          = {https://doi.org/10.5281/zenodo.1199703}
}
```

Citing the publication:

```
@article{Stoneback2018,
  author    = {Stoneback, R. A. and
               Burrell, A. G. and
               Klenzing, J. and
               Depew, M. D.},
  doi       = {10.1029/2018JA025297},
  issn      = {21699402},
  journal   = {Journal of Geophysical Research: Space Physics},
  number    = {6},
```

(continues on next page)

(continued from previous page)

```
  pages     = {5271--5283},
  title     = {{PYSAT: Python Satellite Data Analysis Toolkit}},
  volume    = {123},
  year      = {2018}
}
```

To aid in scientific reproducibility, please include the version number in publications that use this code. This can be found by invoking `pysat.__version__`.

Information for appropriately acknowledging and citing the different instruments accessed through pysat is sometimes available in the metadata through `inst.meta.acknowledgements` and `inst.meta.references`. If this information is missing, please consider improving pysat by either submitting an issue or adding the information yourself.

CHAPTER 3

Installation

Starting from scratch

Python and associated packages for science are freely available. Convenient science python package setups are available from https://www.python.org/, Anaconda, and other locations (some platform specific). Anaconda also includes a developer environment that works well with pysat. Core science packages such as numpy, scipy, matplotlib, pandas and many others may also be installed directly via pip or your favorite package manager.

For maximum safety, pysat should be installed into its own virtual environment to ensure there are no conflicts with any system installed Python distributions.

For MacOS systems it is recommended that *gcc* is installed via HomeBrew for compatibility with Fortran code.

```
brew install gcc
```

For Windows systems, please see the `Windows` section below for setting up a POSIX compatible C/Fortran environment.

To use Anaconda's tools for creating a suitable virtual environment, for Python 2

```
conda create -n virt_env_name python=2.7
conda activate virt_env_name
conda install 'numpy<1.19' -c conda
```

and for Python 3

```
conda create -n virt_env_name python=3
conda activate virt_env_name
conda install 'numpy<1.19' -c conda
```

pysat

Pysat itself may be installed from a terminal command line via:

```
pip install pysat
```

Note that pysat requires a number of packages that will be installed automatically if not already present on a system. The default behavior for updating required libraries already on a system depends upon the version of pip present.

- beautifulsoup4
- h5py
- lxml
- madrigalWeb
- matplotlib
- netCDF4
- numpy (>=1.12)
- pandas (>=0.23, <0.25)
- PyForecastTools
- pysatCDF
- requests
- scipy
- xarray (<0.15)

The upper caps for packages above will be removed for the upcoming pysat 3.0.0 release.

Development Installation

---

pysat may also be installed directly from the source repository on github:

```
git clone https://github.com/pysat/pysat.git
cd pysat
python setup.py install
```

An advantage to installing through github is access to the development branches. The latest bugfixes can be found in the `develop` branch. However, this branch is not stable (as the name implies). We recommend using this branch in a virtual environment and using:

```
git clone https://github.com/pysat/pysat.git
cd pysat
git checkout develop
python setup.py develop
```

The use of *develop* rather than *install* installs the code 'in-place', so any changes to the software do not have to be reinstalled to take effect.

The development version for v3.0 can be found in the `develop-3` branch (see above for caveats).

Windows

To get pysat installed in Windows you need a POSIX compatible C/ Fortran compiling environment. This is required to compile the pysatCDF package.

Python environment: Python 2.7.x

1. Install MSYS2 from http://repo.msys2.org. The distrib folder contains msys2-x86_64-latest.exe (64-bit version) to install MSYS2.
2. Assuming you installed it in its default location `C:\msys64`, launch MSYS2 environment from `C:\msys64\msys2.exe`. This launches a shell session.
3. Now you need to make sure everything is up to date. This terminal command will run updates:

```
pacman -Syuu
```

4. After running this command, you will be asked to close the terminal window using close button and not exit() command. Go ahead and do that.
5. Relaunch and run:

```
pacman -Syuu
```

   again.

6. After the second run, you should be up to date. If you run the update command again, you will be informed that there was nothing more to update. Now you need to install build tools and your compiler toolchains.:

```
pacman -S base-devel git mingw-w64-x86_64-toolchain
```

   If it prompts you to make a selection and says (default:all), just press enter. This install may take a bit.

7. Now you need to set up your MSYS2 environment to use whatever python interpreter you want to build pysatCDF for. In my case the path was `C:\Python27_64`, but yours will be wherever python.exe exists.
8. Update MSYS2 path to include the folders with python binary and Scripts. To do that, navigate to your home directory in MSYS2. Mine is `C:\msys64\home\gayui`.
9. Edit the .bash_profile file to add the below lines somewhere in the file.:

```
# Add System python
export PATH=$PATH:/c/Python27_64:/c/Python27_64/Scripts
```

   Note the unix-style paths. So `C:` becomes `/c/`. If your python was in `C:\foo\bar\python` you would put `/c/foo/bar/python` and `/c/foo/bar/python/Scripts`

10. Next step is to add the mingw64 bin folder to your windows system path. Right-click on computer, hit properties. Then click advanced system settings, then environment variables. Find the system variable (as opposed to user variables) named PATH. This is a semicolon delimited list of the OS search paths for binaries. Add another semicolon and the path `C:\msys64\mingw64\bin`
11. Now you should have access to Python from within your MSYS2 environment. And your windows path should have access to the mingw binaries. To verify this, launch the mingw64 MSYS2 environment.:

```
C:\msys64\mingw64.exe
```

    Run the command:

```
which python
```

    and confirm that it points to the correct python version you want to be using.

12. Microsoft Visual C++ 9.0 is required to compile C sources. Download and install the right version of Microsoft Visual C++ for Python 2.7 from http://aka.ms/vcpython27
13. We are now getting close to installing pysatCDF. Do the following in the shell environment that is already opened.:

```
mkdir src
cd src
git clone https://github.com/rstoneback/pysatCDF.git
cd pysatCDF
```

14. Using a text editor of your choice, create a file called setup.cfg in:

```
C:\msys64\home\gayui\src\pysatCDF
```

Put the following in the file before saving and closing it.:

```
[build]
compiler=mingw32
```

Note: gayui will be replaced with your username

15. In your MSYS2 MINGW64 environment, run:

```
python setup.py install
```

This should compile and install the package to your site-packages for the python you are using.

16. You should now be able to import pysatCDF in your Python environment. If you get an ImportError, restart Python and import again.

CHAPTER 4

Quick-Start

Set Data Directory

Pysat will maintain organization of data from various platforms. Upon the first

```
import pysat
```

pysat will remind you to set the top level directory that will hold the data,

```
pysat.utils.set_data_dir(path=path)
```

Note the directory path supplied must already exist or an error will be raised. To check the currently set data directory,

```
print(pysat.data_dir)
```

To check if pysat and required packages are working, instantiate one of the test instruments, and load a day of simulated data. Loading a day of data will ensure there is no problem with the underlying pandas installation.

```
inst = pysat.Instrument('pysat', 'testing')
inst.load(2009, 1)
print(inst.data)
```

To verify xarray is working

```
inst = pysat.Instrument('pysat', 'testing_xarray')
inst.load(2009, 1)
print(inst.data)
```

Note: pysat will not allow an Instrument to be instantiated without a data directory being specified.

CHAPTER 5

Tutorial

5.1 Basics

The core functionality of pysat is exposed through the pysat.Instrument object. The intent of the Instrument object is to offer a single interface for interacting with science data that is independent of measurement platform. The layer of abstraction presented by the Instrument object allows for things to occur in the background that can make science data analysis simpler and more rigorous.

To begin,

```
import pysat
```

The data directory pysat looks in for data (pysat_data_dir) needs to be set upon the first import,

```
pysat.utils.set_data_dir(path=path_to_existing_directory)
```

Note: A data directory must be set before any pysat.Instruments may be used or an error will be raised.

Basic Instrument Discovery

Support for each instrument in pysat is enabled by a suite of methods that interact with the particular files for that dataset and supply the data within in a pysat compatible format. A particular data set is identified using up to four parameters

Identifier	Description
platform	General platform instrument is on
name	Name of the instrument
tag	Label for a subset of total data
sat_id	Label for instrument sub-group

All supported pysat Instruments for v2.x are stored in the pysat.instruments submodule. A listing of all currently supported instruments is available via help,

```
help(pysat.instruments)
```

Each instrument listed will support one or more data sets for analysis. The submodules are named with the convention platform_name. To get a description of an instrument, along with the supported datasets, use help again,

```
help(pysat.instruments.dmsp_ivm)
```

Further, the dictionary:

```
pysat.instruments.dmsp_ivm.tags
```

is keyed by `tag` with a description of each type of data the `tag` parameter selects. The dictionary:

```
pysat.instruments.dmsp_ivm.sat_ids
```

indicates which instrument or satellite ids (`sat_id`) support which tag. The combination of `tag` and `sat_id` select the particular dataset a pysat.Instrument object will provide and interact with.

Instantiation

To create a pysat.Instrument object, select a `platform`, instrument `name`, and potentially a `tag` and `sat_id`, consistent with the desired data to be analyzed, from one the supported instruments.

To work with plasma data from the Ion Velocity Meter (IVM) onboard the Defense Meteorological Satellite Program (DMSP) constellation, use:

```
dmsp = pysat.Instrument(platform='dmsp', name='ivm', tag='utd', sat_id='f12')
```

Behind the scenes pysat uses a python module named dmsp_ivm that understands how to interact with 'utd' data for 'f12'.

Download

Let's download some data. DMSP data is hosted by the Madrigal database, a community resource for geospace data. The proper process for downloading DMSP and other Madrigal data is built into the open source tool madrigalWeb, which is invoked appropriately by pysat within the dmsp_ivm module. To get DMSP data specifically all we have to do is invoke the `.download()` method attached to the DMSP object. Madrigal requires that users provide their name and email address as their username and password.

```
# set user and password for Madrigal
user = 'Firstname+Lastname'
password = 'email@address.com'
# define date range to download data
start = pysat.datetime(2001, 1, 1)
stop = pysat.datetime(2001, 1, 2)
# download data to local system
dmsp.download(start, stop, user=user, password=password)
```

The data is downloaded to pysat_data_dir/platform/name/tag/, in this case pysat_data_dir/dmsp/ivm/utd/. At the end of the download, pysat will update the list of files associated with DMSP.

Some instruments support an improved download experience that ensures the local system is fully up to date compared to the data source. The command,

```
dmsp.download_updated_files()
```

will obtain the full set of files present on the server and compare the version and revision numbers for the server files with those on the local system. Any files missing or out of date on the local system are downloaded from the server. This command downloads, as needed, the entire dataset.

Note: Science data servers may not have the same reliability and bandwidth as commercial providers

---

Load Data

Data is loaded into a pysat.Instrument object, in this case dmsp, using the `.load` method using year, day of year; date; or filename.

```
# load by year, day of year
dmsp.load(2001, 1)
# load by datetime
dmsp.load(date=datetime.datetime(2001, 1, 1))
# load by filename
dmsp.load(fname='dms_ut_20010101_12.002.hdf5')
# load by filename
dmsp.load(fname=dmsp.files[0])
# load by filename
dmsp.load(fname=dmsp.files[datetime.datetime(2001, 1, 1)])
```

When the pysat load routine runs it stores the instrument data into dmsp.data. pysat supports the use of two different data structures, either a pandas DataFrame, a highly capable structure with labeled rows and columns, or an xarray DataSet for data sets with more dimensions. Either way, the full data structure is available at:

```
# all data
dmsp.data
```

providing full access to the underlying data library functionality. The type of data structure is flagged at the instrument level with the attribute `inst.pandas_format`, True if a DataFrame is returned by the corresponding instrument module load method.

In addition, convenience access to the data is also available at the instrument level.

```
# Convenience access
dmsp['ti']
# slicing
dmsp[0:10, 'ti']
# slicing by date time
dmsp[start:stop, 'ti']

# Convenience assignment
dmsp['ti'] = new_array
# exploit broadcasting, single value assigned to all times
dmsp['ti'] = single_value
# slicing
dmsp[0:10, 'ti'] = sub_array
# slicing by date time
dmsp[start:stop, 'ti'] = sub_array
```

See *Instrument* for more.

To load data over a season, pysat provides a convenience function that returns an array of dates over a season. The season need not be continuous.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas

# create empty series to hold result
mean_ti = pandas.Series()

# get list of dates between start and stop
start = dt.datetime(2001, 1, 1)
stop = dt.datetime(2001, 1, 10)
date_array = pysat.utils.time.create_date_range(start, stop)

# iterate over season, calculate the mean Ion Temperature
for date in date_array:
   # load data into dmsp.data
   dmsp.load(date=date)
   # check if data present
   if not dmsp.empty:
       # isolate data to locations near geomagnetic equator
       idx, = np.where((dmsp['mlat'] < 5) & (dmsp['mlat'] > -5))
       # downselect data
       dmsp.data = dmsp[idx]
       # compute mean ion temperature using pandas functions and store
       mean_ti[dmsp.date] = dmsp['ti'].abs().mean(skipna=True)

# plot the result using pandas functionality
mean_ti.plot(title='Mean Ion Temperature near Magnetic Equator')
plt.ylabel(dmsp.meta['ti', dmsp.desc_label] + ' (' +
           dmsp.meta['ti', dmsp.units_label] + ')')
```

Note, the numpy.where may be removed using the convenience access to the attached pandas data object.

```
idx, = np.where((dmsp['mlat'] < 5) & (dmsp['mlat'] > -5))
dmsp.data = dmsp[idx] = dmsp.data.iloc[idx
```

is equivalent to

```
dmsp.data = vefi[(dmsp['mlat'] < 5) & (dmsp['mlat'] > -5)]
```

Clean Data

Before data is available in .data it passes through an instrument specific cleaning routine. The amount of cleaning is set by the clean_level keyword, provided at instantiation. The level defaults to 'clean'.

```
dmsp = pysat.Instrument(platform='dmsp', name='ivm', tag='utd', sat_id='f12',
                        clean_level=None)
dmsp = pysat.Instrument(platform='dmsp', name='ivm', tag='utd', sat_id='f12',
                        clean_level='clean')
```

Four levels of cleaning may be specified,

clean_level	Result
clean	Generally good data
dusty	Light cleaning, use with care
dirty	Minimal cleaning, use with caution
none	No cleaning, use at your own risk

The user provided cleaning level is stored on the Instrument object at `dmsp.clean_level`. The details of the cleaning will generally vary greatly between instruments.

Metadata

Metadata is also stored along with the main science data. pysat presumes a minimum default set of metadata that may be arbitrarily expanded. The default parameters are driven by the attributes required by public science data files, like those produced by the Ionospheric Connections Explorer (ICON).

Metadata	Description
axis	Label for plot axes
desc	Description of variable
fill	Fill value for bad data points
label	Label used for plots
name	Name of variable, or long_name
notes	Notes about variable
min	Maximum valid value
max	Minimum valid value
scale	Axis scale, linear or log
units	Variable units

```
# all metadata
dmsp.meta.data
# variable metadata
dmsp.meta['ti']
# units using standard labels
dmsp.meta['ti'].units
# units using general labels
dmsp.meta['ti', dmsp.units_label]
# update units for ti
dmsp.meta['ti'] = {'units':'new_units'}
# update display name, long_name
dmsp.meta['ti'] = {'long_name':'Fancy Name'}
# add new meta data
dmsp.meta['new'] = {dmsp.units_label:'fake',
                    dmsp.name_label:'Display'}
```

The string values used within metadata to identify the parameters above are all attached to the instrument object as dmsp.*_label, or `dmsp.units_label`, `dmsp.min_label`, and `dmsp.notes_label`, etc.

All variables must have the same metadata parameters. If a new parameter is added for only one data variable, then the remaining data variables will get a null value for that metadata parameter.

Data may be assigned to the instrument, with or without metadata.

```
# assign data alone
dmsp['new_data'] = new_data
```

(continues on next page)

(continued from previous page)

```
# assign data with metadata
# the data must be keyed under 'data'
# all other dictionary inputs are presumed to be metadata
dmsp['new_data'] = {'data': new_data,
                    dmsp.units_label: new_unit,
                    'new_meta_data': new_value}
# alter assigned metadata
dmsp.meta['new_data', 'new_meta_data'] = even_newer_value
```

The labels used for identifying metadata may be provided by the user at Instrument instantiation and do not need to conform with what is in the file:

```
dmsp = pysat.Instrument(platform='dmsp', name='ivm', tag='utd', sat_id='f12',
                        clean_level='dirty', units_label='new_units')
dmsp.load(2001, 1)
dmsp.meta['ti', 'new_units']
dmsp.meta['ti', dmsp.units_label]
```

While this feature doesn't require explicit support on the part of an instrument module developer, code that does not use the metadata labels may not always work when a user invokes this functionality.

pysat's metadata object is case insensitive but case preserving. Thus, if a particular Instrument uses 'units' for units metadata, but a separate package that operates via pysat but uses 'Units' or even 'UNITS', the code will still function:

```
# the following are all equivalent
dmsp.meta['TI', 'Long_Name']
dmsp.meta['Ti', 'long_Name']
dmsp.meta['ti', 'Long_NAME']
```

Note: While metadata access is case-insensitive, data access is case-sensitive.

5.2 Verbosity

Pysat uses Python's standard logging tools to control the verbosity of output. By default, only logger.warning messages are shown. For more detailed instrument output, you may change the logging level.

```
from pysat import logger, logging
logger.set_level(logging.INFO)
```

The logging level will be applied to all instruments loaded by pysat.

5.3 Custom Functions

Science analysis is built upon custom data processing. To simplify this task and enable instrument independent analysis, custom functions may be attached to the Instrument object. Each function is run automatically when new data is loaded before it is made available in `inst.data`.

This feature enables a user to hand an Instrument object to an independent routine and ensure any desired customizations required are performed without any additional user intervention. This feature enables for the transparent modification of a dataset in between its state at rest on disk and when the data becomes available for use at `inst.data`.

Warning: Custom arguments and keywords are supported for these methods. However, these arguments and keywords are only evaluated initially when the method is attached to an Instrument object. Thus the objects passed in must be static or capable of updating themselves from within the custom method itself.

Modify Functions

The instrument object is passed to function in place, there is no Instrument copy made in memory. The method is expected to modify the supplied Instrument object directly. 'Modify' methods are not allowed to return any information via the method itself.

```
def custom_func_modify(inst, optional_param=False):
    """Modify a pysat.Instrument object in place

    Parameters
    ----------
    inst : pysat.Instrument
        Object to be modified
    optional_param : stand-in
        Placeholder to indicate support for custom keywords
        and arguments
    """

    if optional_param:
        inst['double_mlt'] = 2.0 * inst['mlt']
    else:
        inst['double_mlt'= -2.0 * inst['mlt']
    return
```

Add Functions

A copy of the Instrument is passed to the method thus any changes made directly to the object are lost. The data to be added must be returned via 'return' in the method and is added to the true Instrument object by pysat. Multiple return types are supported.

Type	Notes
tuple	(data_name, data_to_be_added)
dict	Data to be added keyed by data_name
Iterable	((name1, name2, ...), (data1, data2, ...))
Series	Variable name must be in .name
DataFrame	Columns used as variable names
DataArray	Variable name must be in .name

```
def custom_func_add(inst, optional_param=False):
    """Calculate data to be added to pysat.Instrument object

    Parameters
    ----------
    inst : pysat.Instrument
        pysat will add returned data to this object
    optional_param : stand-in
        Placeholder indicated support for custom keywords
        and arguments
    """

    return ('double_mlt', 2.0 * inst['mlt'])
```

Add Function Including Metadata

Metadata may also be returned when using a dictionary object as the return type. In this case, the data must be in 'data', with other keys interpreted as metadata parameters. Multiple data variables may be added in this case only when using the DataFrame.

```
def custom_func_add(inst, optional_param1=False, optional_param2=False):
    return {'data': 2.*inst['mlt'], 'name': 'double_mlt',
            'long_name': 'doubledouble', 'units': 'hours'}
```

Attaching Custom Function

Custom methods must be attached to an Instrument object for pysat to automatically apply the method upon ever load.

```
ivm.custom.attach(custom_func_modify, 'modify', optional_param2=True)
ivm.load(2009, 1)
print(ivm['double_mlt'])
ivm.custom.attach(custom_func_add, 'add', optional_param2=True)
# can also set via a string name for method
ivm.custom.attach('custom_func_add', 'add', optional_param2=False)
# set bounds limiting the file/date range the Instrument will iterate over
ivm.bounds = (start, stop)
# perform analysis. Whatever modifications are enabled by the custom
# methods are automatically available within the custom analysis
custom_complicated_analysis_over_season(ivm)
```

The output of custom_func_modify will always be available from instrument object, regardless of what level the science analysis is performed.

We can repeat the earlier DMSP example, this time using nano-kernel functionality.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas

# create custom function
def filter_dmsp(inst, limit=None):
    # isolate data to locations near geomagnetic equator
    idx, = np.where((dmsp['mlat'] < 5) & (dmsp['mlat'] > -5))
    # downselect data
    dmsp.data = dmsp[idx]

# get list of dates between start and stop
start = dt.datetime(2001, 1, 1)
stop = dt.datetime(2001, 1, 10)
date_array = pysat.utils.time.create_date_range(start, stop)

# create empty series to hold result
mean_ti = pandas.Series()

# instantiate pysat.Instrument
dmsp = pysat.Instrument(platform='dmsp', name='ivm', tag='utd',
                        sat_id='f12')
# attach custom method from above
dmsp.custom.attach(filter_dmsp, 'modify')

# iterate over season, calculate the mean Ion Temperature
for date in date_array:
   # load data into dmsp.data
```

(continues on next page)

(continued from previous page)

```
    dmsp.load(date=date)
    # check if data present
    if not dmsp.empty:
        # compute mean ion temperature using pandas functions and store
        mean_ti[dmsp.date] = dmsp['ti'].mean(skipna=True)

# plot the result using pandas functionality
mean_ti.plot(title='Mean Ion Temperature near Magnetic Equator')
plt.ylabel(dmsp.meta['ti', dmsp.desc_label] + ' (' +
           dmsp.meta['ti', dmsp.units_label] + ')')
```

Note the same result is obtained. The DMSP instrument object and analysis are performed at the same level, so there is no strict gain by using the pysat nano-kernel in this simple demonstration. However, we can use the nano-kernel to translate this daily mean into an versatile instrument independent function.

5.4 Initial Instrument Independence

Adding Instrument Independence

pysat features enable the development of instrument independent methods, code that can work on many if not all pysat supported datasets. This section continues the evolution of the simple DMSP temperature averaging method presented earlier towards greater instrument independence as well as application to non-DMSP data sets.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas

def daily_mean(inst, start, stop, data_label):
    """Perform daily mean of data_label over season

    Parameters
    ----------
    inst : pysat.Instrument
        Instrument object
    start : datetime.datetime
        Start date
    stop : datetime.datetime
        Stop date
    data_label : string
        Identifier for variable to be averaged
    """

    # create empty series to hold result
    mean_val = pandas.Series()

    # get list of dates between start and stop
    date_array = pysat.utils.time.create_date_range(start, stop)

    # iterate over season, calculate the mean
    for date in date_array:
            inst.load(date=date)
                if not inst.data.empty:
                # compute absolute mean using pandas functions and store
                mean_val[inst.date] = inst[data_label].abs().mean(skipna=True)
```

(continues on next page)

(continued from previous page)

```
    return mean_val

# instantiate pysat.Instrument object to get access to data
vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b')

# define custom filtering method
def filter_inst(inst, data_label, data_gate):
    # select data within +/- data gate
    min_gate = -np.abs(data_gate)
    max_gate = np.abs(data_gate)
    idx, = np.where((inst[data_label] < max_gate) &
                    (inst[data_label] > min_gate))
    inst.data = inst[idx]
    return

# attach filter to vefi object, function is run upon every load
vefi.custom.add(filter_inst, 'modify', 'latitude', 5.)

# make a plot of daily mean of 'db_mer'
mean_dB = daily_mean(vefi, start, stop, 'dB_mer')

# plot the result using pandas functionality
mean_dB.plot(title='Absolute Daily Mean of '
             + vefi.meta['dB_mer'].long_name)
plt.ylabel('Absolute Daily Mean (' + vefi.meta['dB_mer'].units + ')')
```

The pysat nano-kernel lets you modify any data set as needed so that you can get the daily mean you desire, without having to modify the daily_mean function.

Check the instrument independence using a different instrument. Whatever instrument is supplied may be modified in arbitrary ways by the nano-kernel.

Note: Downloading data for COSMIC requires an account at the Cosmic Data Analysis and Archive Center (CDAAC).

```
cosmic = pysat.Instrument('cosmic', 'gps', tag='ionprf', clean_level='clean',
                          altitude_bin=3)
# attach filter method
cosmic.custom.add(filter_inst, 'modify', 'edmaxlat', 15.)
# perform average
mean_max_dens = daily_mean(cosmic, start, stop, 'edmax')

# plot the result using pandas functionality
long_name = cosmic.meta[data_label, cosmic.name_label]
units = cosmic.meta[data_label, cosmic.units_label]
mean_max_dens.plot(title='Absolute Daily Mean of ' + long_name)
plt.ylabel('Absolute Daily Mean (' + units + ')')
```

`daily_mean` now works for any instrument, as long as the data to be averaged is 1D. This can be fixed.

Partial Independence from Dimensionality

This section continues the evolution of the daily_mean method presented earlier towards greater instrument independence by supporting more than 1D datasets.

```
import pandas
import pysat

def daily_mean(inst, start, stop, data_label):

    # create empty series to hold result
    mean_val = pandas.Series()
    # get list of dates between start and stop
    date_array = pysat.utils.time.create_date_range(start, stop)
    # iterate over season, calculate the mean
    for date in date_array:
        inst.load(date=date)
        if not inst.data.empty:
            # compute mean absolute using pandas functions and store
            # data could be an image, or lower dimension, account for 2D and lower
            data = inst[data_label]
            if isinstance(data.iloc[0], pandas.DataFrame):
                # 3D data, 2D data at every time
                data_panel = pandas.Panel.from_dict(dict([(i, data.iloc[i]) for i in␣
↪xrange(len(data))]))
                mean_val[inst.date] = data_panel.abs().mean(axis=0,skipna=True)
            elif isinstance(data.iloc[0], pandas.Series):
                # 2D data, 1D data for each time
                data_frame = pandas.DataFrame(data.tolist())
                data_frame.index = data.index
                mean_val[inst.date] = data_frame.abs().mean(axis=0, skipna=True)
            else:
               # 1D data
                mean_val[inst.date] = inst[data_label].abs().mean(axis=0,skipna=True)

return mean_val
```

This code works for 1D, 2D, and 3D datasets, regardless of instrument platform, with only some minor changes from the initial VEFI specific code. In-situ measurements, remote profiles, and remote images. It is true the nested if statements aren't the most elegant. Particularly the 3D case. However this code puts the data into an appropriate structure for pandas to align each of the profiles/images by their respective indices before performing the average. Note that the line to obtain the arithmetic mean is the same in all cases, .mean(axis=0, skipna=True). There is an opportunity here for pysat to clean up the little mess caused by dimensionality.

```
import pandas
import pysat

def daily_mean(inst, start, stop, data_label):

    # create empty series to hold result
    mean_val = pandas.Series()
    # get list of dates between start and stop
    date_array = pysat.utils.time.create_date_range(start, stop)
    # iterate over season, calculate the mean
    for date in date_array:
        inst.load(date=date)
        if not inst.data.empty:
            # compute mean absolute using pandas functions and store
            # data could be an image, or lower dimension, account for 2D and lower
            data = inst[data_label]
            data = pysat.ssnl.computational_form(data)
            mean_val[inst.date] = data.abs().mean(axis=0, skipna=True)
```

(continues on next page)

(continued from previous page)

```
return mean_val
```

5.5 Iteration

The seasonal analysis loop is commonly repeated in data analysis:

```
vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b')
date_array = pysat.utils.time.create_date_range(start, stop)
for date in date_array:
    vefi.load(date=date)
    print('Maximum meridional magnetic perturbation ', vefi['dB_mer'].max())
```

Iteration support is built into the Instrument object to support this and similar cases. The whole of a data set may be iterated over on a daily basis using

```
for vefi in vefi:
        print('Maximum meridional magnetic perturbation ', vefi['dB_mer'].max())
```

Each loop of the python for iteration initiates a vefi.load() for the next date, starting with the first available date. By default the instrument instance will iterate over all available data. To control the range, set the instrument bounds,

```
# multi-season season
vefi.bounds = ([start1, start2], [stop1, stop2])
# continuous season
vefi.bounds = (start, stop)
# iterate over custom season
for vefi in vefi:
    print('Maximum meridional magnetic perturbation ', vefi['dB_mer'].max())
```

The output is,

```
Returning cnofs vefi dc_b data for 05/09/10
Maximum meridional magnetic perturbation  19.3937
Returning cnofs vefi dc_b data for 05/10/10
Maximum meridional magnetic perturbation  23.745
Returning cnofs vefi dc_b data for 05/11/10
Maximum meridional magnetic perturbation  25.673
Returning cnofs vefi dc_b data for 05/12/10
Maximum meridional magnetic perturbation  26.583
```

So far, the iteration support has only saved a single line of code, the .load line. However, this line in the examples above is tied to loading by date. What if we wanted to load by file instead? This would require changing the code. However, with the abstraction provided by the Instrument iteration, that is no longer the case.

```
vefi.bounds('filename1', 'filename2')
for vefi in vefi:
    print('Maximum meridional magnetic perturbation ', vefi['dB_mer'].max())
```

For VEFI there is only one file per day so there is no practical difference between the previous example. However, for instruments that have more than one file a day, there is a difference.

Building support for this iteration into the mean_day example is easy.

```
import pandas
import pysat

def daily_mean(inst, data_label):

    # create empty series to hold result
    mean_val = pandas.Series()

    for inst in inst:
        if not inst.data.empty:
            # compute mean absolute using pandas functions and store
            # data could be an image, or lower dimension, account for 2D and lower
            data = inst[data_label]
            data = pysat.ssnl.computational_form(data)
            mean_val[inst.date] = data.abs().mean(axis=0, skipna=True)

    return mean_val
```

Since bounds are attached to the Instrument object, the start and stop dates for the season are no longer required as inputs. If a user forgets to specify the bounds, the loop will start on the first day of data and end on the last day.

```
# make a plot of daily dB_mer
vefi.bounds = (start, stop)
mean_dB = daily_mean(vefi, 'dB_mer')

# plot the result using pandas functionality
variable_str = vefi.meta['dB_mer', vefi.name_label]
units_str = vefi.meta['dB_mer', vefi.units_label]
mean_dB.plot(title='Absolute Daily Mean of ' + variable_str)
plt.ylabel('Absolute Daily Mean ('+ units_str +')')
```

The abstraction provided by the iteration support is also used for the next section on orbit data.

5.6 Orbit Support

Pysat has functionality to determine orbits on the fly from loaded data. These orbits will span day breaks as needed (generally). To use any of these orbit features, information about the orbit needs to be provided at initialization. The 'index' is the name of the data to be used for determining orbits, and 'kind' indicates type of orbit. See *pysat.Orbits* for latest inputs.

There are several orbits to choose from,

kind	method
local time	Uses negative gradients to delineate orbits
longitude	Uses negative gradients to delineate orbits
polar	Uses sign changes to delineate orbits
orbit	Uses any change in value to delineate orbits

Changes in universal time are also used to delineate orbits. Pysat compares any gaps to the supplied orbital period, nominally assumed to be 97 minutes. As orbit periods aren't constant, a 100% success rate is not be guaranteed.

```
info = {'index': 'mlt', 'kind': 'local time'}
ivm = pysat.Instrument(platform='cnofs', name='ivm', orbit_info=info,
                       clean_level='None')
```

Orbit determination acts upon data loaded in the ivm object, so to begin we must load some data.

```
ivm.load(date=start)
```

Orbits may be selected directly from the attached `ivm.orbit` class. The data for the orbit is stored in `ivm.data`.

```
In [50]: ivm.orbits[1]
Out[50]:
Returning cnofs ivm  data for 12/27/12
Returning cnofs ivm  data for 12/28/12
Loaded Orbit:0
```

Note that getting the first orbit caused pysat to load the day previous, and then back to the current day. Orbits are zero indexed. pysat is checking here if the first orbit for 12/28/2012 actually started on 12/27/2012. In this case it does.

```
In [51]: ivm[0:5, 'mlt']
Out[51]:
2012-12-27 23:05:14.584000    0.002449
2012-12-27 23:05:15.584000    0.006380
2012-12-27 23:05:16.584000    0.010313
2012-12-27 23:05:17.584000    0.014245
2012-12-27 23:05:18.584000    0.018178
Name: mlt, dtype: float32

In [52]: ivm[-5:, 'mlt']
Out[52]:
2012-12-28 00:41:50.563000    23.985415
2012-12-28 00:41:51.563000    23.989031
2012-12-28 00:41:52.563000    23.992649
2012-12-28 00:41:53.563000    23.996267
2012-12-28 00:41:54.563000    23.999886
Name: mlt, dtype: float32
```

Let's go back an orbit.

```
In [53]: ivm.orbits.prev()
Out[53]:
Returning cnofs ivm  data for 12/27/12
Loaded Orbit:15

In [54]: ivm[-5:, 'mlt']
Out[54]:
2012-12-27 23:05:09.584000    23.982796
2012-12-27 23:05:10.584000    23.986725
2012-12-27 23:05:11.584000    23.990656
2012-12-27 23:05:12.584000    23.994587
2012-12-27 23:05:13.584000    23.998516
Name: mlt, dtype: float32
```

pysat loads the previous day, as needed, and returns the last orbit for 12/27/2012 that does not (or should not) extend into 12/28.

If we continue to iterate orbits using

```
ivm.orbits.next()
```

eventually the next day will be loaded to try and form a complete orbit. You can skip the iteration and just go for the last orbit of a day,

```
In[] : ivm.orbits[-1]
Out[]:
Returning cnofs ivm  data for 12/29/12
Loaded Orbit:1
```

```
In[72] : ivm[:5, 'mlt']
Out[72]:
2012-12-28 23:03:34.160000    0.003109
2012-12-28 23:03:35.152000    0.007052
2012-12-28 23:03:36.160000    0.010996
2012-12-28 23:03:37.152000    0.014940
2012-12-28 23:03:38.160000    0.018884
Name: mlt, dtype: float32

In[73] : ivm[-5:, 'mlt']
Out[73]:
2012-12-29 00:40:13.119000    23.982937
2012-12-29 00:40:14.119000    23.986605
2012-12-29 00:40:15.119000    23.990273
2012-12-29 00:40:16.119000    23.993940
2012-12-29 00:40:17.119000    23.997608
Name: mlt, dtype: float32
```

Pysat loads the next day of data to see if the last orbit on 12/28/12 extends into 12/29/12, which it does. Note that the last orbit of 12/28/12 is the same as the first orbit of 12/29/12. Thus, if we ask for the next orbit,

```
In[] : ivm.orbits.next()
Loaded Orbit:2
```

pysat will indicate it is the second orbit of the day. Going back an orbit gives us orbit 16, but referenced to a different day. Earlier, the same orbit was labeled orbit 1.

```
In[] : ivm.orbits.prev()
Returning cnofs ivm  data for 12/28/12
Loaded Orbit:16
```

Orbit iteration is built into ivm.orbits just like iteration by day is built into ivm.

```
start = [pandas.datetime(2009, 1, 1), pandas.datetime(2010, 1, 1)]
stop = [pandas.datetime(2009, 4, 1), pandas.datetime(2010, 4, 1)]
ivm.bounds = (start, stop)
for ivm in ivm.orbits:
    print 'next available orbit ', ivm.data
```

5.6.1 Ground Based Instruments

The nominal breakdown of satellite data into discrete orbits isn't typically as applicable for ground based instruments, each of which makes exactly one orbit per day. However, as the orbit iterator triggers off of negative gradients in a variable, a change in sign, or any change in a value, this functionality may be used to break a ground based data set into alternative groupings, as appropriate and desired.

As the orbit iterator defaults to an orbit period consistent with Low Earth Orbit, the expected period of the 'orbits' must be provided at Instrument instantiation. Given the orbit heritage, it is assumed that there is a small amount of variation in the orbit period. pysat will actively filter 'orbits' that are inconsistent with the prescribed orbit period.

5.7 Iteration and Instrument Independent Analysis

The combination of iteration and instrument independence supports generalizing `daily_mean` method introduced earlier in the tutorial into two functions, one that averages by day, the other by orbit. Strictly speaking, the daily_mean above already does this with the right input, as shown

```
mean_daily_val = daily_mean(vefi, 'dB_mer')
mean_orbit_val = daily_mean(vefi.orbits, 'dB_mer')
```

However, the output of the by_orbit attempt gets rewritten for most orbits since the output from daily_mean is stored by date. Though this could be fixed, supplying an instrument object/iterator in one case and an orbit iterator in the other might be a bit inconsistent. Even if not, let's try another route.

We also don't want to maintain two code bases that do almost the same thing. So instead, let's create three functions, two of which simply call a hidden third.

Iteration Independence

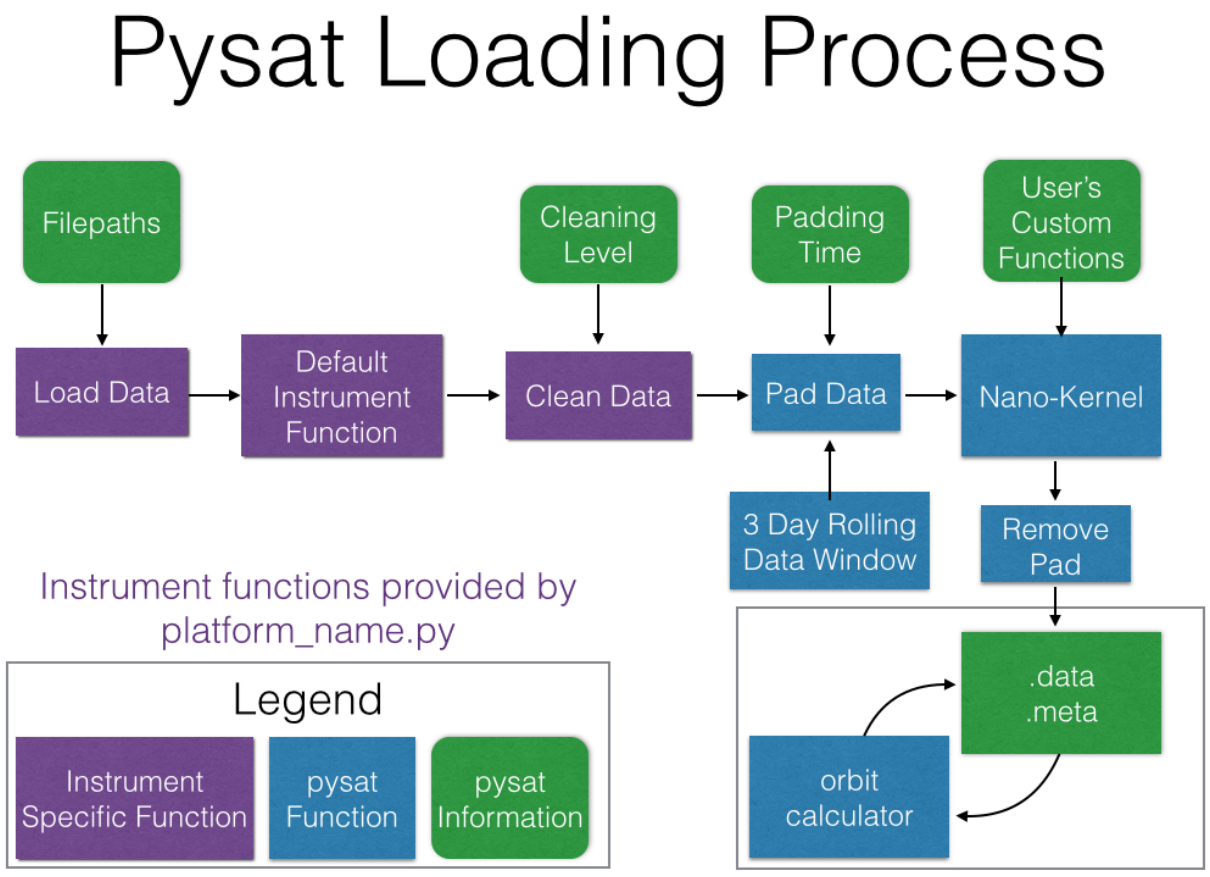
```
def daily_mean(inst, data_label):
    """Mean of data_label by day/file over Instrument.bounds"""
    return _core_mean(inst, data_label, by_day=True)

def by_orbit_mean(inst, data_label):
    """Mean of data_label by orbit over Instrument.bounds"""
    return _core_mean(inst, data_label, by_orbit=True)

def _core_mean(inst, data_label, by_orbit=False, by_day=False):

    if by_orbit:
        iterator = inst.orbits
    elif by_day:
        iterator = inst
    else:
        raise ValueError('A choice must be made, by day/file, or by orbit')
    if by_orbit and by_day:
        raise ValueError('A choice must be made, by day/file, or by orbit')

    # create empty series to hold result
    mean_val = pandas.Series()
    # iterate over season, calculate the mean
    for inst in iterator:
           if not inst.data.empty:
               # compute mean absolute using pandas functions and store
               # data could be an image, or lower dimension, account for 2D and lower
               data = inst[data_label]
               data.dropna(inplace=True)

               if by_orbit:
                   date = inst.data.index[0]
               else:
                   date = inst.date

               data = pysat.ssnl.computational_form(data)
               mean_val[date] = data.abs().mean(axis=0, skipna=True)

    del iterator
    return mean_val
```

The addition of a few more lines to the daily_mean function adds support for averages by orbit, or by day, for any platform with data 3D or less. The date issue and the type of iteration are solved with simple if else checks. From a practical perspective, the code doesn't really deviate from the first solution of simply passing in vefi.orbits, except for the fact that the .orbits switch is 'hidden' in the code. NaN values are also dropped from the data. If the first element is a NaN, it isn't handled by the simple instance check.

A name change and a couple more dummy functions separates out the orbit vs daily iteration clearly, without having multiple codebases. Iteration by file and by date are handled by the same Instrument iterator, controlled by the settings in Instrument.bounds. A by_file_mean was not created because bounds could be set by date and then by_file_mean applied. Of course this could set up to produce an error. However, the settings on Instrument.bounds controls the iteration type between files and dates, so we maintain this view with the expressed calls. Similarly, the orbit iteration is a separate iterator, with a separate call. This technique above is used by other seasonal analysis routines in pysat.

You may notice that the mean call could also easily be replaced by a median, or even a mode. We might also want to return the standard deviation, or appropriate measure. Perhaps another level of generalization is needed?

5.8 Summary Flow Charts



CHAPTER 6

Sample Scientific Analysis

Pysat tends to reduce certain science data investigations to the construction of a routine(s) that makes that investigation unique, a call to a seasonal analysis routine, and some plotting commands. Several demonstrations are offered in this section. The full code for each example is available in the repository in the demo folder.

6.1 Seasonal Occurrence by Orbit

How often does a particular thing occur on a orbit-by-orbit basis? Let's find out. For VEFI, let us calculate the occurrence of a positive perturbation in the meridional component of the geomagnetic field.

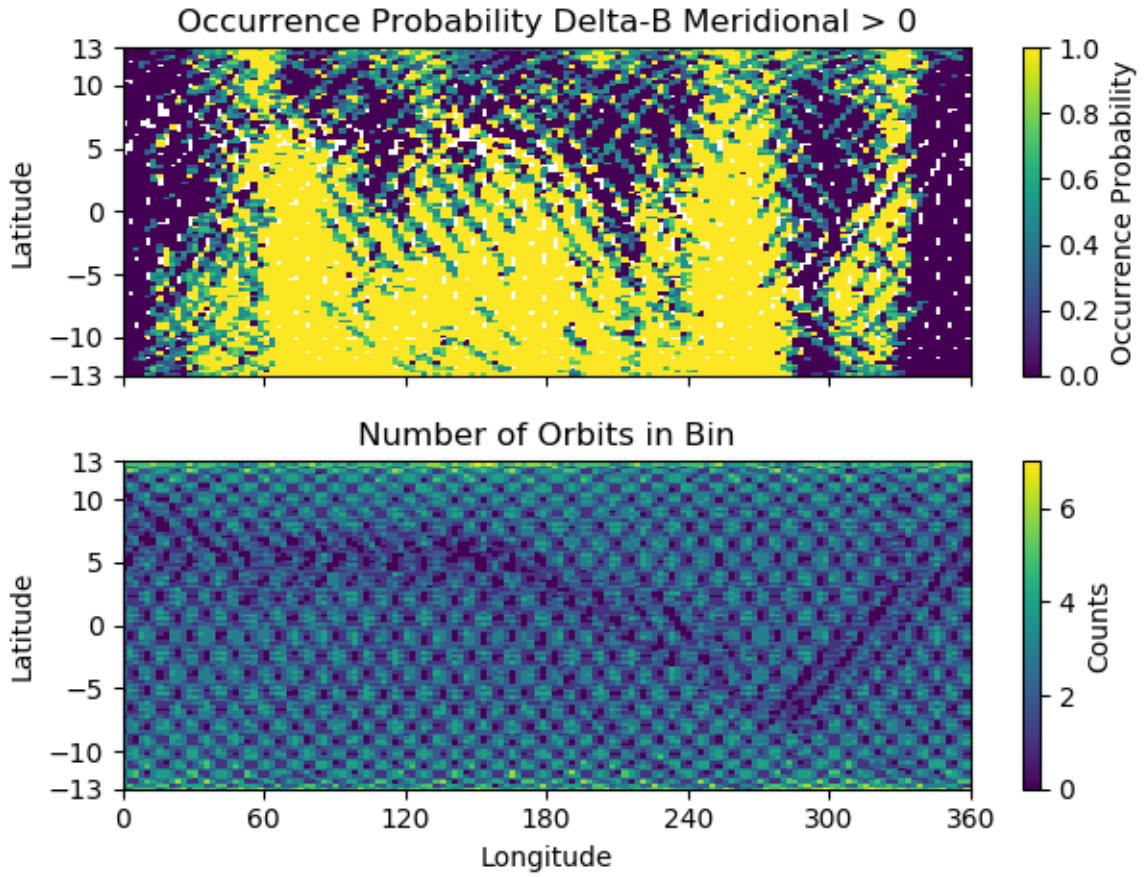
```
"""
Demonstrates iterating over an instrument data set by orbit and determining
the occurrent probability of an event occurring.
"""

import os
import pysat
import matplotlib.pyplot as plt
import numpy as np

# set the directory to save plots to
results_dir = ''

# select vefi dc magnetometer data, use longitude to determine where
# there are changes in the orbit (local time info not in file)
orbit_info = {'index': 'longitude', 'kind': 'longitude'}
vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b',
                        clean_level=None, orbit_info=orbit_info)


# define function to remove flagged values
def filter_vefi(inst):
    idx, = np.where(inst['B_flag'] == 0)
```

(continues on next page)

(continued from previous page)

```
    inst.data = inst.data.iloc[idx]
    return


vefi.custom.add(filter_vefi, 'modify')
# set limits on dates analysis will cover, inclusive
start = pysat.datetime(2010, 5, 9)
stop = pysat.datetime(2010, 5, 15)

# if there is no vefi dc magnetometer data on your system, then run command
# below where start and stop are pandas datetimes (from above)
# pysat will automatically register the addition of this data at the end of
# download
vefi.download(start, stop)

# leave bounds unassigned to cover the whole dataset (comment out lines below)
vefi.bounds = (start, stop)

# perform occurrence probability calculation
# any data added by custom functions is available within routine below
ans = pysat.ssnl.occur_prob.by_orbit2D(vefi, [0, 360, 144], 'longitude',
                                       [-13, 13, 104], 'latitude', ['dB_mer'],
                                       [0.], returnBins=True)

# a dict indexed by data_label is returned
# in this case, only one, we'll pull it out
ans = ans['dB_mer']

# plot occurrence probability
f, axarr = plt.subplots(2, 1, sharex=True, sharey=True)
masked = np.ma.array(ans['prob'], mask=np.isnan(ans['prob']))
im = axarr[0].pcolor(ans['bin_x'], ans['bin_y'], masked)
axarr[0].set_title('Occurrence Probability Delta-B Meridional > 0')
axarr[0].set_ylabel('Latitude')
axarr[0].set_yticks((-13, -10, -5, 0, 5, 10, 13))
axarr[0].set_ylim((ans['bin_y'][0], ans['bin_y'][-1]))
plt.colorbar(im, ax=axarr[0], label='Occurrence Probability')

im = axarr[1].pcolor(ans['bin_x'], ans['bin_y'], ans['count'])
axarr[1].set_title('Number of Orbits in Bin')
axarr[1].set_xlabel('Longitude')
axarr[1].set_xticks((0, 60, 120, 180, 240, 300, 360))
axarr[1].set_xlim((ans['bin_x'][0], ans['bin_x'][-1]))
axarr[1].set_ylabel('Latitude')
plt.colorbar(im, ax=axarr[1], label='Counts')

f.tight_layout()
plt.savefig(os.path.join(results_dir, 'ssnl_occurrence_by_orbit_demo'))
plt.close()
```

Result



The top plot shows the occurrence probability of a positive magnetic field perturbation as a function of geographic longitude and latitude. The bottom plot shows the number of times the satellite was in each bin with data (on per orbit basis). Individual orbit tracks may be seen. The hatched pattern is formed from the satellite traveling North to South and vice-versa. At the latitudinal extremes of the orbit the latitudinal velocity goes through zero providing a greater coverage density. The satellite doesn't return to the same locations on each pass so there is a reduction in counts between orbit tracks. All local times are covered by this plot, over-representing the coverage of a single satellite.

The horizontal blue band that varies in latitude as a function of longitude is the location of the magnetic equator. Torque rod firings that help C/NOFS maintain proper attitude are performed at the magnetic equator. Data during these firings is excluded by the custom function attached to the vefi instrument object.

6.2 Orbit-by-Orbit Plots

Plotting a series of orbit-by-orbit plots is a great way to become familiar with a data set. If the data set doesn't come with orbit information, this can be a challenge. Orbits also go past day breaks, so if data comes in daily files this requires loading multiple files at once, joining the data together, etc. pysat goes through that trouble for you.

```
import os
import pysat
import matplotlib.pyplot as plt

# set the directory to save plots to
results_dir = ''
```

(continues on next page)

(continued from previous page)

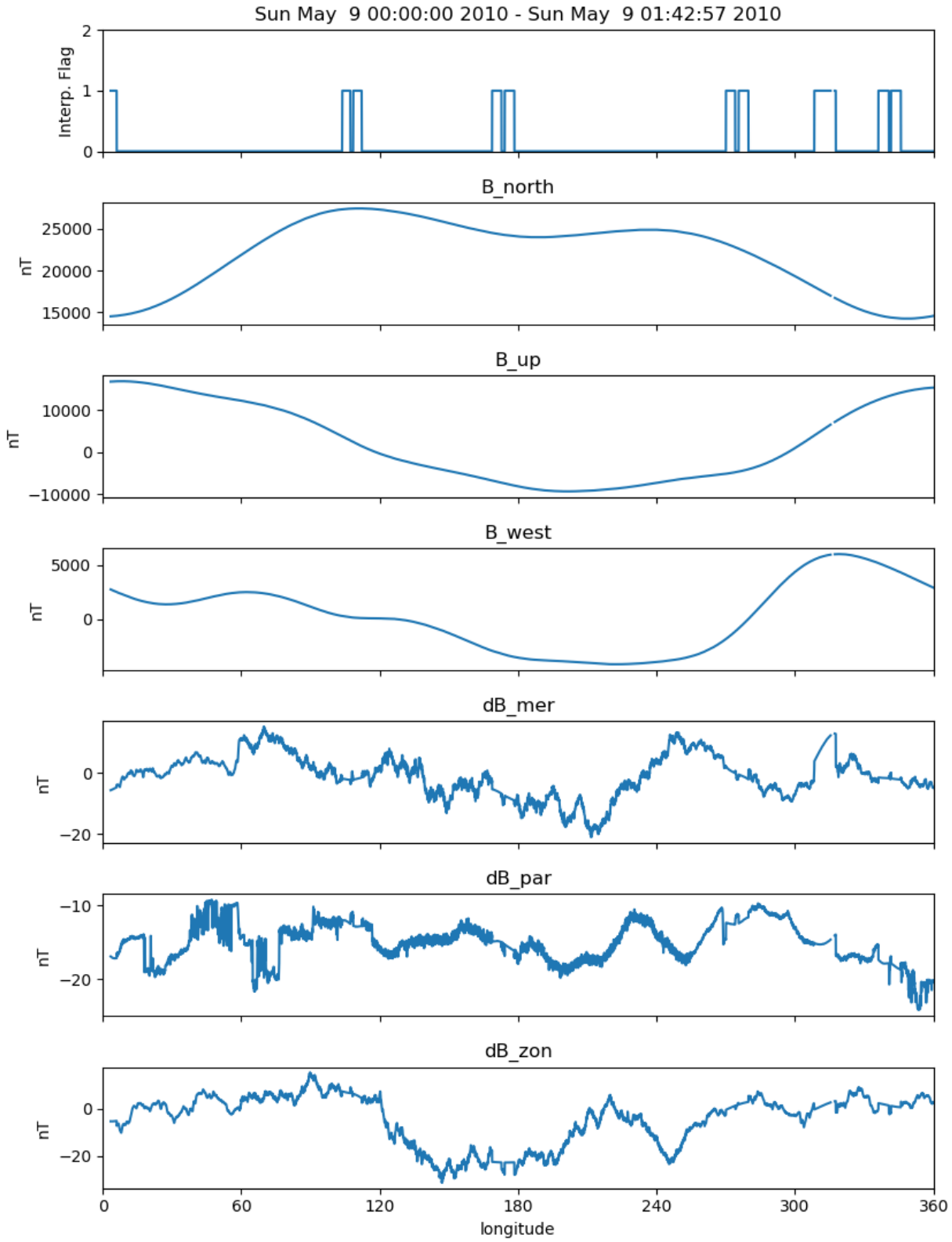
```
# select vefi dc magnetometer data, use longitude to determine where
# there are changes in the orbit (local time info not in file)
orbit_info = {'index': 'longitude', 'kind': 'longitude'}
vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b',
                        clean_level=None, orbit_info=orbit_info)

# set limits on dates analysis will cover, inclusive
start = pysat.datetime(2010, 5, 9)
stop = pysat.datetime(2010, 5, 12)

# if there is no vefi dc magnetometer data on your system
# then run command below
# where start and stop are pandas datetimes (from above)
# pysat will automatically register the addition of this
# data at the end of download
vefi.download(start, stop)

# leave bounds unassigned to cover the whole dataset
vefi.bounds = (start, stop)

for orbit_count, vefi in enumerate(vefi.orbits):
    # for each loop pysat puts a copy of the next available
    # orbit into   vefi.data
    # changing .data at this level does not alter other orbits
    # reloading the same orbit will erase any changes made

    # satellite data can have time gaps, which leads to plots
    # with erroneous lines connecting measurements on
    # both sides of the gap
    # command below fills in any data gaps using a
    # 1-second cadence with NaNs
    # see pandas documentation for more info
    vefi.data = vefi.data.resample('1S',  fill_method='ffill',
                                   limit=1, label='left')

    f, ax = plt.subplots(7, sharex=True, figsize=(8.5,11))

    ax[0].plot(vefi['longitude'], vefi['B_flag'])
    ax[0].set_title(' '.join((vefi.data.index[0].ctime(),'-',
                              vefi.data.index[-1].ctime())))
    ax[0].set_ylabel('Interp. Flag')
    ax[0].set_ylim((0, 2))

    p_params = ['B_north', 'B_up', 'B_west',
                'dB_mer', 'dB_par', 'dB_zon']
    for a, param in zip(ax[1:], p_params):
       a.plot(vefi['longitude'], vefi[param])
       a.set_title(vefi.meta[param].long_name)
       a.set_ylabel(vefi.meta[param].units)

    ax[6].set_xlabel(vefi.meta['longitude'].long_name)
    ax[6].set_xticks([0, 60, 120, 180, 240, 300, 360])
    ax[6].set_xlim((0, 360))

    f.tight_layout()
    fname = 'orbit_{num:05}.png'.format(num=orbit_count)
```

(continues on next page)

(continued from previous page)

```
    plt.savefig(os.path.join(results_dir, fname))
    plt.close()
```

Sample Output (first orbit only)


Sun May 9 00:00:00 2010 - Sun May 9 01:42:57 2010
Interp. Flag
B_north
nT
B_up
nT
B_west
nT
dB_mer
nT
dB_par
nT
dB_zon
nT
longitude

6.3 Seasonal Averaging of Ion Drifts and Density Profiles

In-situ measurements of the ionosphere by the Ion Velocity Meter onboard C/NOFS provides information on plasma density, composition, ion temperature, and ion drifts. This provides a great deal of information on the ionosphere though this information is limited to the immediate vicinity of the satellite. COSMIC GPS measurements, with some processing, provide information on the vertical electron density distribution in the ionosphere. The vertical motion of ions measured by IVM should be reflected in the vertical plasma densities measured by COSMIC. To look at this relationship over all longitudes and local times, for magnetic latitudes near the geomagnetic equator, the code excerpts below provides a framework for the user. The full code can be found at *<https://github.com/pysat/pysat/blob/main/demo/cosmic_and_ivm_demo.py>*_

Note the same averaging routine is used for both COSMIC and IVM, and that both 1D and 2D data are handled correctly.

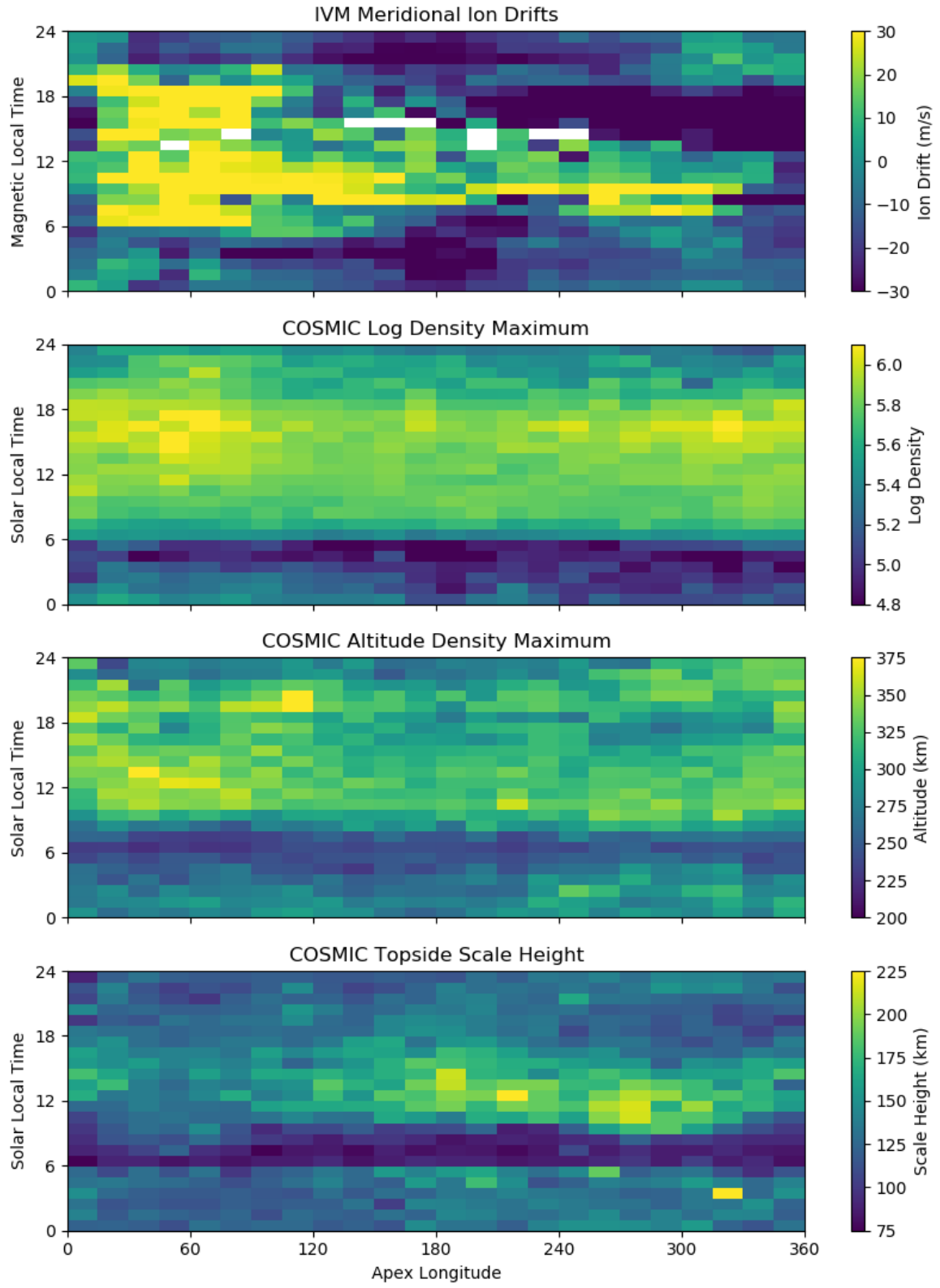
Note: Downloading data for COSMIC requires an account at the Cosmic Data Analysis and Archive Center (CDAAC).

```
# instantiate IVM Object
ivm = pysat.Instrument(platform='cnofs',
                       name='ivm', tag='',
                       clean_level='clean')
# restrict measurements to those near geomagnetic equator
ivm.custom.add(restrictMLAT, 'modify', maxMLAT=25.)
# perform seasonal average
ivm.bounds = (startDate, stopDate)
ivmResults = pysat.ssnl.avg.median2D(ivm, [0, 360, 24], 'alon',
                                     [0, 24, 24], 'mlt', ['ionVelmeridional'])

# create COSMIC instrument object
cosmic = pysat.Instrument(platform='cosmic',
                          name='gps', tag='ionprf',
                          clean_level='clean',
                          altitude_bin=3)

# apply custom functions to all data that is loaded through cosmic
cosmic.custom.add(addApexLong, 'add')

# select locations near the magnetic equator
cosmic.custom.add(filterMLAT, 'modify', mlatRange=(0., 10.))

# take the log of NmF2 and add to the dataframe
cosmic.custom.add(addlogNm, 'add')

# calculates the height above hmF2 to reach Ne < NmF2/e
cosmic.custom.add(addTopsideScaleHeight, 'add')

# do an average of multiple COSMIC data products
# from startDate through stopDate
# a mixture of 1D and 2D data is averaged
cosmic.bounds = (startDate, stopDate)
cosmicResults = pysat.ssnl.avg.median2D(cosmic, [0, 360, 24], 'apex_long',
                                        [0, 24, 24], 'edmaxlct',
                                        ['profiles', 'edmaxalt',
                                         'lognm', 'thf2'])
```

(continues on next page)

(continued from previous page)

```
# the work is done, plot the results
```



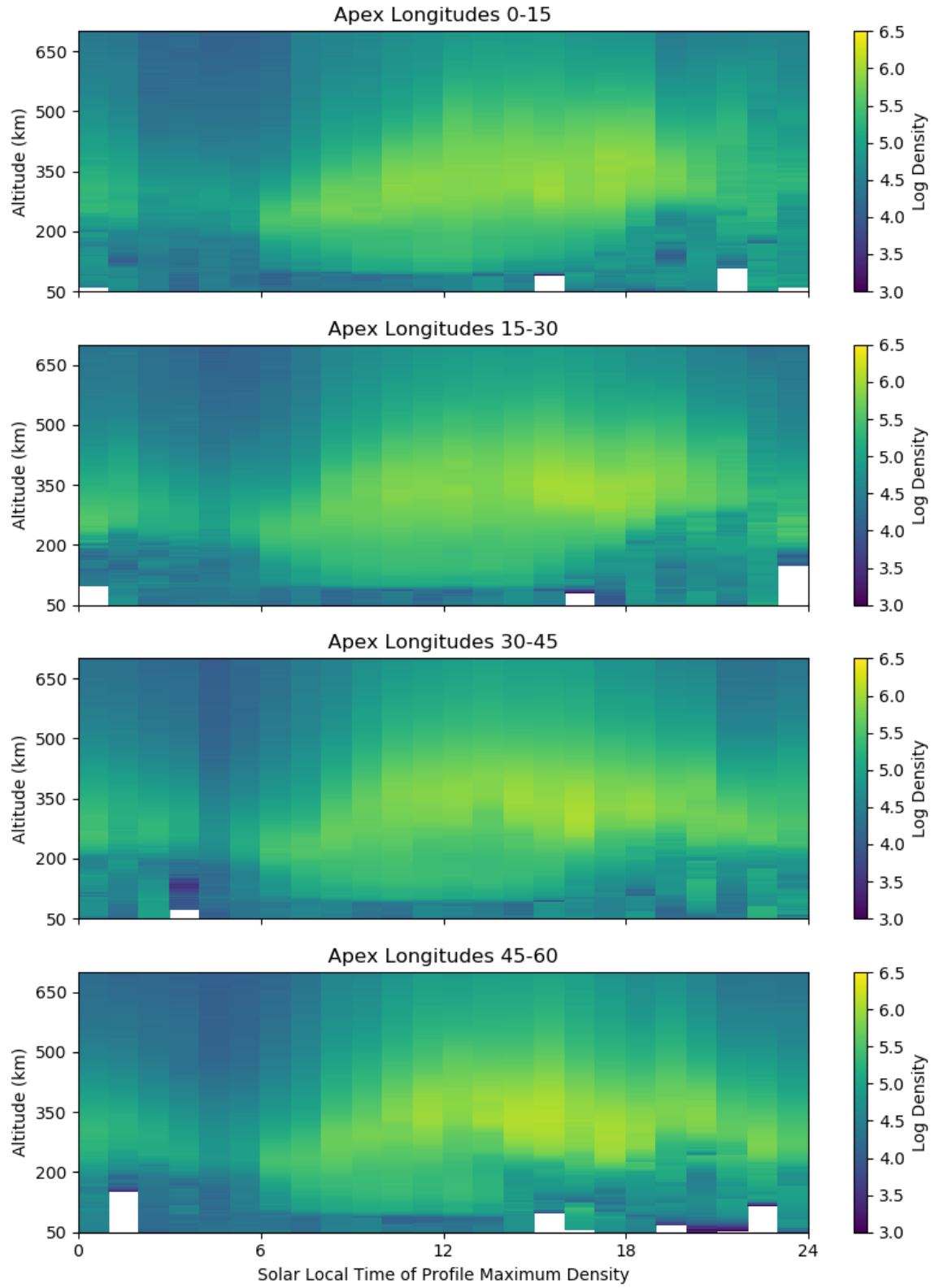
The top image is the median ion drift from the IVM, while the remaining plots are derived from the COSMIC density profiles. COSMIC data does not come with the location of the profiles in magnetic coordinates, so this information is added using the nano-kernel.

```
cosmic.custom.add(addApexLong, 'add')
```

call runs a routine that adds the needed information. This routine is currently only using a simple titled dipole model. Similarly, using custom functions, locations away from the magnetic equator are filtered out and a couple new quantities are added.

There is a strong correspondence between the distribution of downward drifts between noon and midnight and a reduction in the height of the peak ionospheric density around local sunset. There isn't the same strong correspondence with the other parameters but ion density profiles are also affected by production and loss processes, not measured by IVM.

The median averaging routine also produced a series a median altitude profiles as a function of longitude and local time. A selection are shown below.



There is a gradient in the altitude distribution over longitude near sunset. Between 0-15 longitude an upward slope is seen in bottom-side density levels with local time though higher altitudes have a flatter gradient. This is consistent with the upward ion drifts reported by IVM. Between 45-60 the bottom-side ionosphere is flat with local time, while

densities at higher altitudes drop steadily. Ion drifts in this sector become downward at night. Downward drifts lower plasma into exponentially higher neutral densities, rapidly neutralizing plasma and producing an effective flat bottom. Thus, the COSMIC profile in this sector is also consistent with the IVM drifts.

Between 15-30 degrees longitude, ion drifts are upward, but less than the 0-15 sector. Similarly, the density profile in the same sector has a weaker upward gradient with local time than the 0-15 sector. Between 30-45 longitude, drifts are mixed, then transition into weaker downward drifts than between 45-60 longitude. The corresponding profiles have a flatter bottom-side gradient than sectors with upward drift (0-30), and a flatter top-side gradient than when drifts are more downward (45-60), consistent with the ion drifts.

CHAPTER 7

Supported Instruments

7.1 C/NOFS IVM

Supports the Ion Velocity Meter (IVM) onboard the Communication and Navigation Outage Forecasting System (C/NOFS) satellite, part of the Coupled Ion Netural Dynamics Investigation (CINDI). Downloads data from the NASA Coordinated Data Analysis Web (CDAWeb) in CDF format.

The IVM is composed of the Retarding Potential Analyzer (RPA) and Drift Meter (DM). The RPA measures the energy of plasma along the direction of satellite motion. By fitting these measurements to a theoretical description of plasma the number density, plasma composition, plasma temperature, and plasma motion may be determined. The DM directly measures the arrival angle of plasma. Using the reported motion of the satellite the angle is converted into ion motion along two orthogonal directions, perpendicular to the satellite track.

References

A brief discussion of the C/NOFS mission and instruments can be found at de La Beaujardière, O., et al. (2004), C/NOFS: A mission to forecast scintillations, J. Atmos. Sol. Terr. Phys., 66, 1573–1591, doi:10.1016/j.jastp.2004.07.030.

Discussion of cleaning parameters for ion drifts can be found in: Burrell, Angeline G., Equatorial topside magnetic field-aligned ion drifts at solar minimum, The University of Texas at Dallas, ProQuest Dissertations Publishing, 2012. 3507604.

Discussion of cleaning parameters for ion temperature can be found in: Hairston, M. R., W. R. Coley, and R. A. Heelis (2010), Mapping the duskside topside ionosphere with CINDI and DMSP, J. Geophys. Res.,115, A08324, doi:10.1029/2009JA015051.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.1.1 Properties

platform 'cnofs'

name 'ivm'

tag None supported

sat_id None supported

Warning:

- The sampling rate of the instrument changes on July 29th, 2010. The rate is attached to the instrument object as .sample_rate.
- The cleaning parameters for the instrument are still under development.

7.2 C/NOFS PLP

Supports the Planar Langmuir Probe (PLP) onboard the Communication and Navigation Outage Forecasting System (C/NOFS) satellite. Downloads data from the NASA Coordinated Data Analysis Web (CDAWeb).

Description from CDAWeb:

The Planar Langmuir Probe on C/NOFS is a suite of 2 current measuring sensors mounted on the ram facing surface of the spacecraft. The primary sensor is an Ion Trap (conceptually similar to RPAs flown on many other spacecraft) capable of measuring ion densities as low as 1 cm-3 with a 12 bit log electrometer. The secondary senor is a swept bias planar Langmuir probe (Surface Probe) capable of measuring Ne, Te, and spacecraft potential.

The ion number density is the one second average of the ion density sampled at either 32, 256, 512, or 1024 Hz (depending on the mode).

The ion density standard deviation is the standard deviation of the samples used to produce the one second average number density.

DeltaN/N is the detrened ion number density 1 second standard deviation divided by the mean 1 sec density.

The electron density, electron temperature, and spacecraft potential are all derived from a least squares fit to the current-bias curve from the Surface Probe.

The data is PRELIMINARY, and as such, is intended for BROWSE PURPOSES ONLY.

References

A brief discussion of the C/NOFS mission and instruments can be found at de La Beaujardière, O., et al. (2004), C/NOFS: A mission to forecast scintillations, J. Atmos. Sol. Terr. Phys., 66, 1573–1591, doi:10.1016/j.jastp.2004.07.030.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.2.1 Properties

platform 'cnofs'

name 'plp'

tag None supported

sat_id None supported

Warning:

- The data are PRELIMINARY, and as such, are intended for BROWSE PURPOSES ONLY.
- Currently no cleaning routine.
- Module not written by PLP team.

7.3 C/NOFS VEFI

Supports the Vector Electric Field Instrument (VEFI) onboard the Communication and Navigation Outage Forecasting System (C/NOFS) satellite. Downloads data from the NASA Coordinated Data Analysis Web (CDAWeb).

Description from CDAWeb:

The DC vector magnetometer on the CNOFS spacecraft is a three axis, fluxgate sensor with active thermal control situated on a 0.6m boom. This magnetometer measures the Earth's magnetic field using 16 bit A/D converters at 1 sample per sec with a range of .. 45,000 nT. Its primary objective on the CNOFS spacecraft is to enable an accurate V x B measurement along the spacecraft trajectory. In order to provide an in-flight calibration of the magnetic field data, we compare the most recent POMME model (the POtsdam Magnetic Model of the Earth, http://geomag.org/models/pomme5.html) with the actual magnetometer measurements to help determine a set of calibration parameters for the gains, offsets, and non-orthogonality matrix of the sensor axes. The calibrated magnetic field measurements are provided in the data file here. The VEFI magnetic field data file currently contains the following variables: B_north Magnetic field in the north direction B_up Magnetic field in the up direction B_west Magnetic field in the west direction

The data is PRELIMINARY, and as such, is intended for BROWSE PURPOSES ONLY.

References

A brief discussion of the C/NOFS mission and instruments can be found at de La Beaujardière, O., et al. (2004), C/NOFS: A mission to forecast scintillations, J. Atmos. Sol. Terr. Phys., 66, 1573–1591, doi:10.1016/j.jastp.2004.07.030.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.3.1 Properties

platform 'cnofs'

name 'vefi'

tag Select measurement type, one of {'dc_b'}

sat_id None supported

Note:

- tag = 'dc_b': 1 second DC magnetometer data

Warning:

- The data is PRELIMINARY, and as such, is intended for BROWSE PURPOSES ONLY.
- Limited cleaning routine.
- Module not written by VEFI team.

7.4 CHAMP-STAR

Supports the Spatial Triaxial Accelerometer for Research (STAR) instrument onboard the Challenging Minipayload (CHAMP) satellite. Accesses local data in ASCII format.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatIncubator (https://github.com/pysat/pysatIncubator)

7.4.1 Properties

platform 'champ'

name 'star'

tag None supported

sat_id None supported

Warning:

- The cleaning parameters for the instrument are still under development.

7.4.2 Authors

Angeline G. Burrell, Feb 22, 2016, University of Leicester

7.5 COSMIC GPS

Loads data from the COSMIC satellite.

The Constellation Observing System for Meteorology, Ionosphere, and Climate (COSMIC) is comprised of six satellites in LEO with GPS receivers. The occultation of GPS signals by the atmosphere provides a measurement of atmospheric parameters. Data downloaded from the COSMIC Data Analaysis and Archival Center.

Default behavior is to search for the 2013 re-processed data first, then the post-processed data as recommended on https://cdaac-www.cosmic.ucar.edu/cdaac/products.html

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatCDAAC (https://github.com/pysat/pysatCDAAC)

7.5.1 Properties

platform 'cosmic'

name 'gps' for Radio Occultation profiles

tag Select profile type, or scintillation, one of: {'ionprf', 'sonprf', 'wetprf', 'atmprf', 'scnlv1'}

sat_id None supported

altitude_bin Number of kilometers to bin altitude profiles by when loading. Currently only supported for tag='ionprf'.

Note:

- 'ionprf: 'ionPrf' ionosphere profiles
- 'sonprf': 'sonPrf' files
- 'wetprf': 'wetPrf' files
- 'atmprf': 'atmPrf' files
- 'scnlv1': 'scnLv1' files

Warning:

- Routine was not produced by COSMIC team
- More recent versions of netCDF4 and numpy limit the casting of some variable types into others. This issue could prevent data loading for some variables such as 'MSL_Altitude' in the 'sonprf' and 'wetprf' files. The default UserWarning when this occurs is

  'UserWarning: WARNING: missing_value not used since it cannot be safely cast to variable data type'

7.6 DE2 LANG

Supports the Langmuir Probe (LANG) instrument on Dynamics Explorer 2 (DE2).

From CDAWeb:

The Langmuir Probe Instrument (LANG) was a cylindrical electrostatic probe that obtained measurements of electron temperature, Te, and electron or ion concentration, Ne or Ni, respectively, and spacecraft potential. Data from this investigation were used to provide temperature and density measurements along magnetic field lines related to thermal energy and particle flows within the magnetosphere-ionosphere system, to provide thermal plasma conditions for wave-particle interactions, and to measure large-scale and fine-structure ionospheric effects of energy deposition in the ionosphere. The Langmuir Probe instrument was identical to that used on the AE satellites and the Pioneer Venus Orbiter. Two independent sensors were connected to individual adaptive sweep voltage circuits which continuously tracked the changing electron temperature and spacecraft potential, while autoranging electrometers adjusted their gain in response to the changing plasma density. The control signals used to achieve this automatic tracking provided a continuous monitor of the ionospheric parameters without telemetering each volt-ampere (V-I) curve. Furthermore, internal data storage circuits permitted high resolution, high data rate sampling of selected V-I curves for transmission to ground to verify or correct the inflight processed data. Time resolution was 0.5 seconds.

References

J. P. Krehbiel, L. H. Brace, R. F. Theis, W. H. Pinkus, and R. B. Kaplan, The Dynamics Explorer 2 Langmuir Probe (LANG), Space Sci. Instrum., v. 5, n. 4, p. 493, 1981.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.6.1 Properties

platform 'de2'

name 'lang'

sat_id None Supported

tag None Supported

7.6.2 Authors

J. Klenzing

7.7 DE2 NACS

Supports the Neutral Atmosphere Composition Spectrometer (NACS) instrument on Dynamics Explorer 2 (DE2).

From CDAWeb:

The Neutral Atmosphere Composition Spectrometer (NACS) was designed to obtain in situ measurements of the neutral atmospheric composition and to study the variations of the neutral atmosphere in response to energy coupled into it from the magnetosphere. Because temperature enhancements, large-scale circulation cells, and wave propagation are produced by energy input (each of which posseses a specific signature in composition variation), the measurements permitted the study of the partition, flow, and deposition of energy from the magnetosphere. Specifically, the investigation objective was to characterize the composition of the neutral atmosphere with particular emphasis on variability in constituent densities driven by interactions in the atmosphere, ionosphere, and magnetosphere system. The quadrupole mass spectrometer used was nearly identical to those flown on the AE-C, -D, and -E missions. The electron- impact ion source was used in a closed mode. Atmospheric particles entered an antechamber through a knife-edged orifice, where they were thermalized to the instrument temperature. The ions with the selected charge-to-mass ratios had stable trajectories through the hyperbolic electric field, exited the analyzer, and entered the detection system. An off-axis beryllium-copper dynode multiplier operating at a gain of 2.E6 provided an output pulse of electrons for each ion arrival. The detector output had a pulse rate proportional to the neutral density in the ion source of the selected mass. The instrument also included two baffles that scanned across the input orifice for optional measurement of the zonal and vertical components of the neutral wind. The mass select system provided for 256 mass values between 0 and 51 atomic mass units (u) or each 0.2 u. It was possible to call any one of these mass numbers into each of eight 0.016-s intervals. This sequence was repeated each 0.128 s.

This data set includes daily files of the PI-provided DE-2 NACS 1-second data and corresponding orbit parameters. The data set was generated at NSSDC from the original PI-provided data and software (SPTH-00010) and from the orbit/attitude database and software that is part of the DE-2 UA data set (SPIO-00174). The original NACS data were provided by the PI team in a highly compressed VAX/VMS binary format on magnetic tapes. The data set covers the whole DE-2 mission time period. Each data point is an average over the normally 8 measurements per second. Densities and relative errors are provided for atomic oxygen (O), molecular nitrogen (N2), helium (He), atomic nitrogen (N), and argon (Ar). The data quality is generally quite good below 500 km, but deteriorates towards higher altitudes as oxygen and molecular nitrogen approach their background values (which could only be determined

from infrequent spinning orbits) and the count rate for Ar becomes very low. The difference between minimum (background) and maximum count rate for atomic nitrogen (estimated from mass 30) was so small that results are generally poor. Data were lost between 12 March 1982 and 31 March 1982 when the counter overflowed.

References

G. R. Carrignan, B. P. Block, J. C. Maurer, A. E. Hedin, C. A. Reber, N. W. Spencer The neutral mass spectrometer on Dynamics Explorer B Space Sci. Instrum., v. 5, n. 4, p. 429, 1981.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.7.1 Properties

platform 'de2'

name 'nacs'

sat_id None Supported

tag None Supported

7.7.2 Authors

J. Klenzing

7.8 DE2 RPA

Supports the Retarding Potential Analyzer (RPA) instrument on Dynamics Explorer 2 (DE2).

From CDAWeb:

The Retarding Potential Analyzer (RPA) measured the bulk ion velocity in the direction of the spacecraft motion, the constituent ion concentrations, and the ion temperature along the satellite path. These parameters were derived from a least squares fit to the ion number flux vs energy curve obtained by sweeping or stepping the voltage applied to the internal retarding grids of the RPA. In addition, a separate wide aperture sensor, a duct sensor, was flown to measure the spectral characteristics of iregularities in the total ion concentration. The measured parameters obtained from this investigation were important to the understanding of mechanisms that influence the plasma; i.e., to understand the coupling between the solar wind and the earth's atmosphere. The measurements were made with a multigridded planar retarding potential analyzer very similar in concept and geometry to the instruments carried on the AE satellites. The retarding potential was variable in the range from approximately +32 to 0 V. The details of this voltage trace, and whether it was continuous or stepped, depended on the operating mode of the instrument. Specific parameters deduced from these measurements were ion temperature; vehicle potential; ram component of the ion drift velocity; the ion and electron concentration irregularity spectrum; and the concentration of H+, He+, O+, and Fe+, and of molecular ions near perigee.

It includes the DUCT portion of the high resolutiondata from the Dynamics Explorer 2 (DE-2) Retarding Potential Analyzer (RPA) for the whole DE-2 mission time period in ASCII format. This version was generated at NSSDC from the PI-provided binary data (SPIO-00232). The DUCT files include RPA measurements ofthe total ion concentration every 64 times per second. Due to a failure in the instrument memory system RPA data are not available from 81317 06:26:40 UT to 82057 13:16:00 UT. This data set is based on the revised version of the RPA files that was submitted by the PI team in June of 1995. The revised RPA data include a correction to the spacecraft potential.

References

W. B. Hanson, R. A. Heelis, R. A. Power, C. R. Lippincott, D. R. Zuccaro, B. J. Holt, L. H. Harmon, and S. Sanatani, "The retarding potential analyzer for dynamics explorer-B," Space Sci. Instrum. 5, 503–510 (1981).

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.8.1 Properties

platform 'de2'

name 'rpa'

sat_id ''

tag None Supported

7.8.2 Authors

J. Klenzing

7.9 DE2 WATS

Supports the Wind and Temperature Spectrometer (WATS) instrument on Dynamics Explorer 2 (DE2).

From CDAWeb:

The Wind and Temperature Spectrometer (WATS) measured the in situ neutral winds, the neutral particle temperatures, and the concentrations of selected gases. The objective of this investigation was to study the interrelationships among the winds, temperatures, plasma drift, electric fields, and other properties of the thermosphere that were measured by this and other instruments on the spacecraft. Knowledge of how these properties are interrelated contributed to an understanding of the consequences of the acceleration of neutral particles by the ions in the ionosphere, the acceleration of ions by neutrals creating electric fields, and the related energy transfer between the ionosphere and the magnetosphere. Three components of the wind, one normal to the satellite velocity vector in the horizontal plane, one vertical, and one in the satellite direction were measured. A retarding potential quadrupole mass spectrometer, coupled to the atmosphere through a precisely orificed antechamber, was used. It was operated in either of two modes: one employed the retarding capability and the other used the ion source as a conventional nonretarding source. Two scanning baffles were used in front of the mass spectrometer: one moved vertically and the other moved horizontally. The magnitudes of the horizontal and vertical components of the wind normal to the spacecraft velocity vector were computed from measurements of the angular relationship between the neutral particle stream and the sensor. The component of the total stream velocity in the satellite direction was measured directly by the spectrometer system through determination of the required retarding potential. At altitudes too high for neutral species measurements, the planned operation required the instrument to measure the thermal ion species only. A series of four sequentially occurring slots –each a 2-s long measurement interval– was adapted for the basic measurement format of the instrument. Different functions were commanded into these slots in any combination, one per measurement interval. Thus the time resolution can be 2, 4, 6, or 8 seconds. Further details are found in This data set consists of the high-resolution data of the Dynamics Explorer 2 Wind and Temperature Spectrometer (WATS) experiment. The files contain the neutral density, temperature and horizontal (zonal) wind velocity, and orbital parameters in ASCII format. The time resolution is typically 2 seconds. Data are given as daily files (typically a few 100 Kbytes each). PI-provided software (WATSCOR) was used to correct the binary data set. NSSDC-developed software was used to add the orbit parameters, to convert the binary into ASCII format and to combine the (PI-provided) orbital files into daily files. For more on DE-2, WATS, and the binary data, see the WATS_VOLDESC_SFDU_DE.DOC and WATS_FORMAT_SFDU_DE.DOC files. More information about the processing done at NSSDC is given in WATS_NSSDC_PRO_DE.DOC.

References

N. W. Spencer, L. E. Wharton, H. B. Niemann, A. E. Hedin, G. R. Carrignan, J. C. Maurer The Dynamics Explorer Wind and Temperature Spectrometer Space Sci. Instrum., v. 5, n. 4, p. 417, 1981.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.9.1 Properties

platform 'de2'

name 'wats'

sat_id None Supported

tag None Supported

7.9.2 Authors

J. Klenzing

7.10 Demeter IAP

Supports the Plasma Analyzer Instrument (Instrument Analyseur de Plasma, or IAP) onboard the Detection of Electro-Magnetic Emissions Transmitted from Earthquake Regions (DEMETER) Microsatellite.

The IAP consists of a Velocity Analyzer (ADV) and Retarding potential analyzer (APR) to provide plasma velocities, ion density and temperature, and satellite potential. The computation of the ion plasma parameters works well when there are at least two ions being considered. Also, the ADV requires currents of at least 1 nA to produce believable measurements. The IAP was run in both survey and burst mode.

Downloads data from the Plasma physics data center (Centre de donees de la physique des plasmas, CDPP), the French national data center for natural plasmas of the solar system. This data product requires registration and user initiated downloading after ordering a data product.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatIncubator (https://github.com/pysat/pysatIncubator)

7.10.1 Properties

platform 'demeter'

name 'iap'

tag 'survey' or 'burst'

sat_id None supported

Examples

```
import pysat
demeter = pysat.Instrument('demeter', 'iap', 'survey', clean_level='none')
demeter.load(2009, 363)
```

7.10.2 Custom Functions

add_drift_geo_coord Calcuate the ion velocity in geographic coordinates

add_drift_lgm_coord Calcuate the ion velocity in local geomagneic coordinates

add_drift_sat_coord Calculate the ion velocity in satellite x, y, z coordinates

pysat.instruments.demeter_iap.**add_drift_sat_coord**(*inst*)
Calculate the ion velocity in satellite x,y,z coordinates

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in pysatIncubator

Parameters **inst** (pysat.Instrument) – DEMETER IAP instrument class object

Returns

Return type Adds data values iv_Ox, iv_Oy

pysat.instruments.demeter_iap.**add_drift_lgm_coord**(*inst*)
Calcuate the ion velocity in local geomagneic coordinates

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in pysatIncubator

Parameters **inst** (pysat.Instrument) – DEMETER IAP instrument class object

Returns

- *Adds data values iv_par (parallel to B vector at satellite),*
- *iv_pos (perpendictular to B, in the plane of the satellite),*
- *iv_perp (completes the coordinate system). If iv_Ox and iv_Oy*
- *do not exist yet, adds them as well*

pysat.instruments.demeter_iap.**add_drift_geo_coord**(*inst*)
Calcuate the ion velocity in geographic coordinates

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in pysatIncubator

Parameters **inst** (pysat.Instrument) – DEMETER IAP instrument class object

Returns

- *Adds data values iv_geo_x (towards the intersection of equator and*
- *Grennwich meridian), iv_geo_y (completes coordinate system),*
- *iv_geo_z (follows Earth's rotational axis, positive Northward).*
- *If iv_Ox,y do not exist yet, adds them as well*

7.11 DMSP IVM

Supports the Ion Velocity Meter (IVM) onboard the Defense Meteorological Satellite Program (DMSP).

The IVM is comprised of the Retarding Potential Analyzer (RPA) and Drift Meter (DM). The RPA measures the energy of plasma along the direction of satellite motion. By fitting these measurements to a theoretical description of plasma the number density, plasma composition, plasma temperature, and plasma motion may be determined. The

DM directly measures the arrival angle of plasma. Using the reported motion of the satellite the angle is converted into ion motion along two orthogonal directions, perpendicular to the satellite track.

Downloads data from the National Science Foundation Madrigal Database. The routine is configured to utilize data files with instrument performance flags generated at the Center for Space Sciences at the University of Texas at Dallas.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatMadrigal (https://github.com/pysat/pysatMadrigal)

7.11.1 Properties

platform 'dmsp'

name 'ivm'

tag 'utd', None

sat_id ['f11', 'f12', 'f13', 'f14', 'f15', 'f16', 'f17', 'f18']

Examples

```
import pysat
dmsp = pysat.Instrument('dmsp', 'ivm', 'utd', 'f15', clean_level='clean')
dmsp.download(pysat.datetime(2017, 12, 30), pysat.datetime(2017, 12, 31),
              user='Firstname+Lastname', password='email@address.com')
dmsp.load(2017, 363)
```

Note: Please provide name and email when downloading data with this routine.

Code development supported by NSF grant 1259508

7.11.2 Custom Functions

add_drift_unit_vectors Add unit vectors for the satellite velocity

add_drifts_polar_cap_x_y Add polar cap drifts in cartesian coordinates

smooth_ram_drifts Smooth the ram drifts using a rolling mean

update_DMSP_ephemeris Updates DMSP instrument data with DMSP ephemeris

pysat.instruments.dmsp_ivm.**smooth_ram_drifts**(*inst*, *rpa_flag_key=None*, *rpa_vel_key='ion_v_sat_for'*)

Smooth the ram drifts using a rolling mean

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatMadrigal.instruments.methods.dmsp.smooth_ram_drifts*

Parameters

- **rpa_flag_key** (*string or NoneType*) – RPA flag key, if None will not select any data. The UTD RPA flag key is 'rpa_flag_ut' (default=None)
- **rpa_vel_key** (*string*) – RPA velocity data key (default='ion_v_sat_for')

Returns

Return type RPA data in instrument object

pysat.instruments.dmsp_ivm.**update_DMSP_ephemeris**(*inst*, *ephem=None*)
Updates DMSP instrument data with DMSP ephemeris

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatMadrigal.instruments.methods.dmsp.update_DMSP_ephemeris*

Parameters **ephem** (*pysat.Instrument or NoneType*) – dmsp_ivm_ephem instrument object

Returns

Return type Updates 'mlt' and 'mlat'

pysat.instruments.dmsp_ivm.**add_drift_unit_vectors**(*inst*)
Add unit vectors for the satellite velocity

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatMadrigal.instruments.methods.dmsp.add_drift_unit_vectors*

Returns

- *Adds unit vectors in cartesian and polar coordinates for RAM and*
- *cross-track directions –*
  - 'unit_ram_x', 'unit_ram_y', 'unit_ram_r', 'unit_ram_theta'
  - 'unit_cross_x', 'unit_cross_y', 'unit_cross_r', 'unit_cross_theta'

Notes

Assumes that the RAM vector is pointed perfectly forward

pysat.instruments.dmsp_ivm.**add_drifts_polar_cap_x_y**(*inst*, *rpa_flag_key=None*, *rpa_vel_key='ion_v_sat_for'*, *cross_vel_key='ion_v_sat_left'*)
Add polar cap drifts in cartesian coordinates

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatMadrigal.instruments.methods.dmsp.add_drifts_polar_cap_x_y*

Parameters

- **rpa_flag_key** (*string or NoneType*) – RPA flag key, if None will not select any data. The UTD RPA flag key is 'rpa_flag_ut' (default=None)
- **rpa_vel_key** (*string*) – RPA velocity data key (default='ion_v_sat_for')
- **cross_vel_key** (*string*) – Cross-track velocity data key (default='ion_v_sat_left')

Returns

- *Adds 'ion_vel_pc_x', 'ion_vel_pc_y', and 'partial'. The last data key*
- *indicates whether RPA data was available (False) or not (True).*

Notes

Polar cap drifts assume there is no vertical component to the X-Y velocities

7.12 ICON EUV

Supports the Extreme Ultraviolet (EUV) imager onboard the Ionospheric CONnection Explorer (ICON) satellite. Accesses local data in netCDF format.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA). Note that the ICON files are retrieved from different servers here and in pysatNASA, resulting in a difference in local file names. Please see the migration guide there for more details.

7.12.1 Properties

platform 'icon'

name 'euv'

tag None supported

Warning:

- The cleaning parameters for the instrument are still under development.
- Only supports level-2 data.

Examples

```
import pysat
euv = pysat.Instrument(platform='icon', name='euv')
euv.download(dt.datetime(2020, 1, 1), dt.datetime(2020, 1, 31))
euv.load(2020, 1)
```

By default, pysat removes the ICON level tags from variable names, ie, ICON_L27_Ion_Density becomes Ion_Density. To retain the original names, use

```
euv = pysat.Instrument(platform='icon', name='euv',
                       keep_original_names=True)
```

7.12.2 Authors

Jeff Klenzing, Mar 17, 2018, Goddard Space Flight Center Russell Stoneback, Mar 23, 2018, University of Texas at Dallas

7.13 ICON FUV

Supports the Far Ultraviolet (FUV) imager onboard the Ionospheric CONnection Explorer (ICON) satellite. Accesses local data in netCDF format.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA). Note that the ICON files are retrieved from different servers here and in pysatNASA, resulting in a difference in local file names. Please see the migration guide there for more details.

7.13.1 Properties

platform 'icon'

name 'fuv'

tag None supported

Warning:

- The cleaning parameters for the instrument are still under development.
- Only supports level-2 data.

Example

```
import pysat
fuv = pysat.Instrument(platform='icon', name='fuv', tag='day')
fuv.download(dt.datetime(2020, 1, 1), dt.datetime(2020, 1, 31))
fuv.load(2020, 1)
```

By default, pysat removes the ICON level tags from variable names, ie, ICON_L27_Ion_Density becomes Ion_Density. To retain the original names, use

```
fuv = pysat.Instrument(platform='icon', name='fuv', tag=day',
                       keep_original_names=True)
```

7.13.2 Authors

Originated from EUV support. Jeff Klenzing, Mar 17, 2018, Goddard Space Flight Center Russell Stoneback, Mar 23, 2018, University of Texas at Dallas Conversion to FUV, Oct 8th, 2028, University of Texas at Dallas

7.14 ICON IVM

Supports the Ion Velocity Meter (IVM) onboard the Ionospheric Connections (ICON) Explorer.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA). Note that the ICON files are retrieved from different servers here and in pysatNASA, resulting in a difference in local file names. Please see the migration guide there for more details.

7.14.1 Properties

platform 'icon'

name 'ivm'

tag None supported

sat_id 'a' or 'b'

Warning:

- No download routine as ICON has not yet been launched
- Data not yet publicly available

Example

```
import pysat
ivm = pysat.Instrument(platform='icon', name='ivm', sat_id='a')
ivm.download(dt.datetime(2020, 1, 1), dt.datetime(2020, 1, 31))
ivm.load(2020, 1)
```

By default, pysat removes the ICON level tags from variable names, ie, ICON_L27_Ion_Density becomes Ion_Density. To retain the original names, use

```
ivm = pysat.Instrument(platform='icon', name='ivm', sat_id='a',
                       keep_original_names=True)
```

7.14.2 Author

R. A. Stoneback

7.15 ICON MIGHTI

Supports the Michelson Interferometer for Global High-resolution Thermospheric Imaging (MIGHTI) instrument on-board the Ionospheric CONnection Explorer (ICON) satellite. Accesses local data in netCDF format.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA). Note that the ICON files are retrieved from different servers here and in pysatNASA, resulting in a difference in local file names. Please see the migration guide there for more details.

7.15.1 Properties

platform 'icon'

name 'mighti'

tag Supports 'los_wind_green', 'los_wind_red', 'vector_wind_green', 'vector_wind_red', 'temperature'. Note that not every data product available for every sat_id

sat_id '', 'a', or 'b'

Warning:

- The cleaning parameters for the instrument are still under development.
- Only supports level-2 data.

Example

```
import pysat
mighti = pysat.Instrument('icon', 'mighti', 'vector_wind_green',
                          clean_level='clean')
mighti.download(dt.datetime(2020, 1, 30), dt.datetime(2020, 1, 31))
mighti.load(2020, 2)
```

By default, pysat removes the ICON level tags from variable names, ie, ICON_L27_Ion_Density becomes Ion_Density. To retain the original names, use

```
mighti = pysat.Instrument(platform='icon', name='mighti',
                          tag='vector_wind_green', clean_level='clean',
                          keep_original_names=True)
```

7.15.2 Authors

Originated from EUV support. Jeff Klenzing, Mar 17, 2018, Goddard Space Flight Center Russell Stoneback, Mar 23, 2018, University of Texas at Dallas Conversion to MIGHTI, Oct 8th, 2028, University of Texas at Dallas

7.16 ISS-FPMU

Supports the Floating Potential Measurement Unit (FPMU) instrument onboard the International Space Station (ISS). Downloads data from the NASA Coordinated Data Analysis Web (CDAWeb).

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.16.1 Properties

platform 'iss'

name 'fpmu'

tag None Supported

sat_id None supported

Warning:

- Currently clean only replaces fill values with Nans.
- Module not written by FPMU team.

7.17 JRO ISR

Supports the Incoherent Scatter Radar at the Jicamarca Radio Observatory

The Incoherent Scatter Radar (ISR) at the Jicamarca Radio Observatory (JRO) observes ion drifts, line-of-sight neutral winds, electron density and temperature, ion temperature, and ion composition through three overarching experiments.

Downloads data from the JRO Madrigal Database.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatMadrigal (https://github.com/pysat/pysatMadrigal)

7.17.1 Properties

platform 'jro'

name 'isr'

tag 'drifts', 'drifts_ave', 'oblique_stan', 'oblique_rand', 'oblique_long'

Examples

```
import pysat
jro = pysat.Instrument('jro', 'isr', 'drifts', clean_level='clean')
jro.download(pysat.datetime(2017, 12, 30), pysat.datetime(2017, 12, 31),
             user='Firstname+Lastname', password='email@address.com')
jro.load(2017, 363)
```

Note: Please provide name and email when downloading data with this routine.

7.18 OMNI_HRO

Supports OMNI Combined, Definitive, IMF and Plasma Data, and Energetic Proton Fluxes, Time-Shifted to the Nose of the Earth's Bow Shock, plus Solar and Magnetic Indices. Downloads data from the NASA Coordinated Data Analysis Web (CDAWeb). Supports both 5 and 1 minute files.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.18.1 Properties

platform 'omni'

name 'hro'

tag Select time between samples, one of {'1min', '5min'}

sat_id None supported

Note: Files are stored by the first day of each month. When downloading use omni.download(start, stop, freq='MS') to only download days that could possibly have data. 'MS' gives a monthly start frequency.

This material is based upon work supported by the National Science Foundation under Grant Number 1259508.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Warning:

- Currently no cleaning routine. Though the CDAWEB description indicates that these level-2 products are expected to be ok.
- Module not written by OMNI team.

7.18.2 Custom Functions

time_shift_to_magnetic_poles Shift time from bowshock to intersection with one of the magnetic poles

calculate_clock_angle Calculate the clock angle and IMF mag in the YZ plane

calculate_imf_steadiness Calculate the IMF steadiness using clock angle and magnitude in the YZ plane

calculate_dayside_reconnection Calculate the dayside reconnection rate

pysat.instruments.omni_hro.**calculate_clock_angle**(*inst*)
Calculate IMF clock angle and magnitude of IMF in GSM Y-Z plane

Deprecated since version 2.3.0: This function has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

Parameters **inst** (pysat.Instrument) – Instrument with OMNI HRO data

pysat.instruments.omni_hro.**calculate_imf_steadiness**(*inst*, *steady_window=15*, *min_window_frac=0.75*, *max_clock_angle_std=28.64788975654116*, *max_bmag_cv=0.5*)
Calculate IMF steadiness using clock angle standard deviation and the coefficient of variation of the IMF magnitude in the GSM Y-Z plane

Deprecated since version 2.3.0: This function has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

Parameters

- **inst** (pysat.Instrument) – Instrument with OMNI HRO data
- **steady_window** (*int*) – Window for calculating running statistical moments in min (default=15)
- **min_window_frac** (*float*) – Minimum fraction of points in a window for steadiness to be calculated (default=0.75)
- **max_clock_angle_std** (*float*) – Maximum standard deviation of the clock angle in degrees (default=22.5)
- **max_bmag_cv** (*float*) – Maximum coefficient of variation of the IMF magnitude in the GSM Y-Z plane (default=0.5)

pysat.instruments.omni_hro.**time_shift_to_magnetic_poles**(*inst*)
OMNI data is time-shifted to bow shock. Time shifted again to intersections with magnetic pole.

Deprecated since version 2.3.0: This function has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

Parameters **inst** (*Instrument class object*) – Instrument with OMNI HRO data

Notes

Time shift calculated using distance to bow shock nose (BSN) and velocity of solar wind along x-direction.

Warning: Use at own risk.

7.19 ROCSAT-1 IVM

Supports the Ion Velocity Meter (IVM) onboard the Republic of China Satellite (ROCSAT-1). Downloads data from the NASA Coordinated Data Analysis Web (CDAWeb).

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.19.1 Properties

platform 'rocsat1'

name 'ivm'

tag None

sat_id None supported

Note:

- no tag or sat_id required

Warning:

- Currently no cleaning routine.

7.20 SPORT IVM

Ion Velocity Meter (IVM) support for the NASA/INPE SPORT CubeSat.

This mission is still in development. This routine is here to help with the development of code associated with SPORT and the IVM.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.21 SuperDARN

SuperDARN data support for grdex files(Alpha Level!)

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatIncubator (https://github.com/pysat/pysatIncubator)

7.21.1 Properties

platform 'superdarn'

name 'grdex'

tag 'north' or 'south' for Northern/Southern hemisphere data

Note: Requires davitpy and davitpy to load SuperDARN files. Uses environment variables set by davitpy to download files from Virginia Tech SuperDARN data servers. davitpy routines are used to load SuperDARN data.

This material is based upon work supported by the National Science Foundation under Grant Number 1259508.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Warning: Cleaning only removes entries that have 0 vectors, grdex files are constituted from what it is thought to be good data.

7.22 SuperMAG

Supports SuperMAG ground magnetometer measurements and SML/SMU indices. Downloading is supported; please follow their rules of the road: http://supermag.jhuapl.edu/info/?page=rulesoftheroad

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatIncubator (https://github.com/pysat/pysatIncubator)

7.22.1 Properties

platform 'supermag'

name 'magnetometer'

tag Select {'indices', '', 'all', 'stations'}

Note: Files must be downloaded from the website, and is freely available after registration.

This material is based upon work supported by the National Science Foundation under Grant Number 1259508.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Warning:

- Currently no cleaning routine, though the SuperMAG description indicates that these products are expected to be good. More information about the processing is available
- Module not written by the SuperMAG team.

7.23 SW Dst

Supports Dst values. Downloads data from NGDC.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatSpaceWeather (https://github.com/pysat/pysatSpaceWeather)

7.23.1 Properties

platform 'sw'

name 'dst'

tag None supported

Note: Will only work until 2057.

Download method should be invoked on a yearly frequency, dst.download(date1, date2, freq='AS')

This material is based upon work supported by the National Science Foundation under Grant Number 1259508.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7.24 SW F107

Supports F10.7 index values. Downloads data from LASP and the SWPC.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatSpaceWeather (https://github.com/pysat/pysatSpaceWeather)

7.24.1 Properties

platform 'sw'

name 'f107'

tag

- '' : LASP F10.7 data (downloads by month, loads by day)
- 'all' : All LASP standard F10.7
- 'prelim' : Preliminary SWPC daily solar indices
- 'daily' : Daily SWPC solar indices (contains last 30 days)
- 'forecast' : Grab forecast data from SWPC (next 3 days)
- '45day' : 45-Day Forecast data from the Air Force

Note: The forecast data is stored by generation date, where each file contains the forecast for the next three days. Forecast data downloads are only supported for the current day. When loading forecast data, the date specified with the load command is the date the forecast was generated. The data loaded will span three days. To always ensure you are loading the most recent data, load the data with tomorrow's date.

```
f107 = pysat.Instrument('sw', 'f107', tag='forecast')
f107.download()
f107.load(date=f107.tomorrow())
```

The forecast or prelim data should not be used with the data padding option available from pysat.Instrument objects. The 'all' tag shouldn't be used either, no other data available to pad with.

Warning: The 'forecast' F10.7 data loads three days at a time. The data padding feature and multi_file_day feature available from the pyast.Instrument object is not appropriate for 'forecast' data.

7.25 SW Kp

Supports Kp index values. Downloads data from ftp.gfz-potsdam.de or SWPC.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatSpaceWeather (https://github.com/pysat/pysatSpaceWeather)

param platform 'sw'

param name 'kp'

param tag

- '' : Standard Kp data
- 'forecast' : Grab forecast data from SWPC (next 3 days)
- 'recent' : Grab last 30 days of Kp data from SWPC

Note: Standard Kp files are stored by the first day of each month. When downloading use kp.download(start, stop, freq='MS') to only download days that could possibly have data. 'MS' gives a monthly start frequency.

The forecast data is stored by generation date, where each file contains the forecast for the next three days. Forecast data downloads are only supported for the current day. When loading forecast data, the date specified with the load command is the date the forecast was generated. The data loaded will span three days. To always ensure you are loading the most recent data, load the data with tomorrow's date.

```
kp = pysat.Instrument('sw', 'kp', tag='recent')
kp.download()
kp.load(date=kp.tomorrow())
```

Recent data is also stored by the generation date from the SWPC. Each file contains 30 days of Kp measurements. The load date issued to pysat corresponds to the generation date.

The recent and forecast data should not be used with the data padding option available from pysat.Instrument objects.

Warning: The 'forecast' Kp data loads three days at a time. The data padding feature and multi_file_day feature available from the pyast.Instrument object is not appropriate for Kp 'forecast' data.

This material is based upon work supported by the National Science Foundation under Grant Number 1259508.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

7.25.1 Custom Functions

filter_geoquiet Filters pysat.Instrument data for given time after Kp drops below gate.

pysat.instruments.sw_kp.**filter_geoquiet**(*sat*, *maxKp=None*, *filterTime=None*, *kpData=None*, *kp_inst=None*)

Filters pysat.Instrument data for given time after Kp drops below gate.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and has been replaced with the more adaptable function, *pysatSpaceWeather.instruments.methods.kp_ap.filter_geomag*. Be sure to update to use the new kwargs.

Parameters

- **sat** (pysat.Instrument) – Instrument to be filtered
- **maxKp** (*float*) – Maximum Kp value allowed. Kp values above this trigger sat.data filtering.
- **filterTime** (*int*) – Number of hours to filter data after Kp drops below maxKp
- **kpData** (pysat.Instrument *(optional)*) – Kp pysat.Instrument object with data already loaded
- **kp_inst** (pysat.Instrument *(optional)*) – Kp pysat.Instrument object ready to load Kp data.Overrides kpData.

Notes

Loads Kp data for the same timeframe covered by sat and sets sat.data to NaN for times when Kp > maxKp and for filterTime after Kp drops below maxKp.

This routine is written for standard Kp data, not the forecast or recent data.

7.26 TIMED/SABER

Supports the Sounding of the Atmosphere using Broadband Emission Radiometry (SABER) instrument on the Thermosphere Ionosphere Mesosphere Energetics Dynamics (TIMED) satellite.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.26.1 Properties

platform 'timed'

name 'saber'

tag None supported

sat_id None supported

Note: SABER "Rules of the Road" for DATA USE Users of SABER data are asked to respect the following guidelines

- Mission scientific and model results are open to all.
- Guest investigators, and other members of the scientific community or general public should contact the PI or designated team member early in an analysis project to discuss the appropriate use of the data.
- Users that wish to publish the results derived from SABER data should normally offer co-authorship to the PI, Associate PI or designated team members. Co-authorship may be declined. Appropriate acknowledgement of institutions, personnel, and funding agencies should be given.
- Users should heed the caveats of SABER team members as to the interpretation and limitations of the data. SABER team members may insist that such caveats be published, even if co-authorship is declined. Data and model version numbers should also be specified.
- Pre-prints of publications and conference abstracts should be widely distributed to interested parties within the mission and related projects.

Warning:

- Note on Temperature Errors: http://saber.gats-inc.com/temp_errors.php

7.26.2 Authors

J. Klenzing, 4 March 2019

7.27 TIMED/SEE

Supports the SEE instrument on TIMED.

Downloads data from the NASA Coordinated Data Analysis Web (CDAWeb).

Supports two options for loading that may be specified at instantiation.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatNASA (https://github.com/pysat/pysatNASA)

7.27.1 Properties

platform 'timed'

name 'see'

tag None

sat_id None supported

flatten_twod If True, then two dimensional data is flattened across columns. Name mangling is used to group data, first column is 'name', last column is 'name_end'. In between numbers are appended 'name_1', 'name_2', etc. All data for a given 2D array may be accessed via, data.loc[:, 'item':'item_end'] If False, then 2D data is stored as a series of DataFrames, indexed by Epoch. data.loc[0, 'item'] (default=True)

Note:

- no tag required

Warning:

- Currently no cleaning routine.

7.28 UCAR TIEGCM

Supports loading data from files generated using TIEGCM (Thermosphere Ionosphere Electrodynamics General Circulation Model) model. TIEGCM file is a netCDF file with multiple dimensions for some variables.

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatModels (https://github.com/pysat/pysatModels)

7.28.1 Properties

platform 'ucar'

name 'tiegcm'

tag None supported

sat_id None supported

CHAPTER 8

Adding a New Instrument

pysat works by calling modules written for specific instruments that load and process the data consistent with the pysat standard. The name of the python file corresponds to the combination 'platform_name' provided when initializing a pysat.Instrument object. The module should be placed in the pysat instruments directory for native support. A compatible module may also be supplied directly using

```
pysat.Instrument(inst_module=python_module_object).
```

Some data repositories have pysat templates prepared to assist in integrating a new instrument. See the Supported Data Templates section or the template instrument module code under *pysat/instruments/templates/* for more. A general template has also been included to make starting any Instrument module easier.

8.1 Naming Conventions

pysat uses a hierarchy of named variables to define each specific data product. In order, this is

- platform
- name
- sat_id
- tag

The exact usage of these can be tailored to the nature of the mission and data products. In general, each combination should point to a unique data file. Not every data product will need all of these variable names. Both *sat_id* and *tag* can be instantiated as an empty string if unused or used to support a 'default' data set if desired. Examples are given below.

platform

In general, this is the name of the mission or observatory. Examples include ICON, JRO, COSMIC, and SuperDARN. Note that this may be a single satellite or ground-based observatory, a constellation of satellites, or a collaboration of ground-based observatories.

name

In general, this is the name of the instrument or high-level data product. When combined with the platform, this forms a unique file in the *instruments* directory. Examples include the EUV instrument on ICON (icon_euv) and the Incoherent Scatter Radar at JRO (jro_isr).

sat_id

In general, this is a unique identifier for a satellite in a constellation of identical or similar satellites, or multiple instruments on the same satellite with different look directions. For example, the DMSP satellites carry similar instrument suites across multiple spacecraft. These are labeled as f11-f18.

Note that sat_id will be updated to inst_id in pysat v3.0.

tag

In general, the tag points to a specific data product. This could be a specific processing level (such as L1, L2), or a product file (such as the different profile products for cosmic_gps data, 'ionprf', 'atmprf', ...).

Naming Requirements in Instrument Module

Each instrument file must include the platform and name as variables at the top-code-level of the file. Additionally, the tags and sat_ids supported by the module must be stored as dictionaries.

```
platform = 'your_platform_name'
name = 'name_of_instrument'
# dictionary keyed by tag with a string description of that dataset
tags = {'': 'The standard processing for the data.  Loaded by default',
        'fancy': 'A higher-level processing of the data.'}
# dictionary keyed by sat_id with a list of supported tags
# for each key
sat_ids = {'A': ['', 'fancy'], 'B': ['', 'fancy'], 'C': ['']}
```

Note that the possible tags that can be invoked are '' and 'fancy'. The tags dictionary includes a short description for each of these tags. A blank tag will be present by default if the user does not specify a tag.

The supported sat_ids should also stored in a dictionary. Each key name here points to a list of the possible tags that can be associated with that particular *sat_id*. Note that not all satellites in the example support every level of processing. In this case, the 'fancy' processing is available for satellites A and B, but not C.

For a dataset that does not need multiple levels of tags and sat_ids, an empty string can be used. The code below only supports loading a single data set.

```
platform = 'your_platform_name'
name = 'name_of_instrument'
tags = {'': ''}
sat_ids = {'': ['']}
```

The DMSP IVM (dmsp_ivm) instrument module is a practical example of a pysat instrument that uses all levels of variable names.

8.2 Required Routines

Three methods are required within a new instrument module to support pysat operations, with functionality to cover finding files, loading data from specified files, and downloading new files. While the methods below are sufficient to engage with pysat, additional optional methods are needed for full pysat support.

Note that these methods are not directly invoked by the user, but by pysat as needed in response to user inputs.

list_files

pysat maintains a list of files to enable data management functionality. To get this information, pysat expects a module method platform_name.list_files to return a pandas Series of filenames indexed by time with a method signature of:

```
def list_files(tag=None, sat_id=None, data_path=None, format_str=None):
    return pandas.Series(files, index=datetime_index)
```

sat_id and tag are passed in by pysat to select a specific subset of the available data. The location on the local filesystem to search for the files is passed in data_path. The list_files method must return a pandas Series of filenames indexed by datetime objects.

A user is also able to supply a file template string suitable for locating files on their system at pysat.Instrument instantiation, passed via format_str, that must be supported. Sometimes users obtain files from non-traditional sources and format_str makes it easier for those users to use an existing instrument module to work with those files.

pysat will by default store data in pysat_data_dir/platform/name/tag, helpfully provided in data_path, where pysat_data_dir is specified by using *pysat.utils.set_data_dir(pysat_data_dir)*. Note that an alternative directory structure may be specified using the pysat.Instrument keyword directory_format at instantiation. The default is recreated using

```
dformat = '{platform}/{name}/{tag}'
inst=pysat.Instrument(platform, name, directory_format=dformat)
```

Note that pysat handles the path information thus instrument module developers do not need to do anything to support the directory_format keyword.

Pre-Built list_files Methods and Support

Finding local files is generally similar across data sets thus pysat includes a variety of methods to make supporting this functionality easier. The simplest way to construct a valid list_files method is to use one of these included pysat methods.

A complete method is available in `pysat.instruments.methods.general.list_files` that may find broad use.

`pysat.Files.from_os` is a convenience constructor provided for filenames that include time information in the filename and utilize a constant field width or a consistent delimiter. The location and format of the time information is specified using standard python formatting and keywords year, month, day, hour, minute, second. Additionally, both version and revision keywords are supported. When present, the from_os constructor will filter down the file list to the latest version and revision combination.

A complete list_files routine could be as simple as

```
def list_files(tag=None, sat_id=None, data_path=None, format_str=None):
    if format_str is None:
        # set default string template consistent with files from
        # the data provider that will be supported by the instrument
        # module download method
        # template string below works for CINDI IVM data that looks like
        # 'cindi-2009310-ivm-v02.hdf'
        # format_str supported keywords: year, month, day,
        # hour, minute, second, version, and revision
        format_str = 'cindi-{year:4d}{day:03d}-ivm-v{version:02d}.hdf'
    return pysat.Files.from_os(data_path=data_path, format_str=format_str)
```

The constructor presumes the template string is for a fixed width format unless a delimiter string is supplied. This constructor supports conversion of years with only 2 digits and expands them to 4 using the two_digit_year_break keyword. Note the support for format_str above.

If the constructor is not appropriate, then lower level methods within pysat._files may also be used to reduce the workload in adding a new instrument. Note in pysat 3.0 this module will be renamed pysat.files for greater visibility.

See pysat.utils.time.create_datetime_index for creating a datetime index for an array of irregularly sampled times.

pysat will invoke the list_files method the first time a particular instrument is instantiated. After the first instantiation, by default pysat will not search for instrument files as some missions can produce a large number of files which may take time to identify. The list of files associated with an Instrument may be updated by adding *update_files=True* at instantiation.

```
inst = pysat.Instrument(platform=platform, name=name, update_files=True)
```

The output provided by the list_files function that has been pulled into pysat the Instrument object above can be inspected from within Python by checking *inst.files.files*.

load

Loading data is a fundamental activity for data science and is required for all pysat instruments. The work invested by the instrument module author makes it possible for users to work with the data easily.

The load module method signature should appear as:

```
def load(fnames, tag=None, sat_id=None):
    return data, meta
```

- fnames contains a list of filenames with the complete data path that pysat expects the routine to load data for. For most data sets the method should return the exact data that is within the file. However, pysat is also currently optimized for working with data by day. This can present some issues for data sets that are stored by month or by year. See *instruments.methods.nasa_cdaweb.py* for an example of returning daily data when stored by month.
- Some instruments, notably space weather indices, return more than a day of data. More robust support for time spans that exceed a day is under evaluation.
- tag and sat_id specify the data set to be loaded.
- The load routine should return a tuple with (data, pysat metadata object).
- *data* is a pandas DataFrame, column names are the data labels, rows are indexed by datetime objects.
- For multi-dimensional data, an xarray can be used instead. When returning xarray data, a variable at the instrument module top-level must be set,

```
pandas_format = False
```

- The pandas DataFrame or xarray needs to be indexed with datetime objects. For xarray objects this index needs to be named 'Epoch' or 'time'. In a future version the supported names for the time index may be reduced. 'Epoch' should be used for pandas though wider compatibility is expected.
- *pysat.utils.create_datetime_index* provides for quick generation of an appropriate datetime index for irregularly sampled data set with gaps
- A pysat meta object may be obtained from *pysat.Meta()*. The Meta object uses a pandas DataFrame indexed by variable name with columns for metadata parameters associated with that variable, including items like 'units' and 'long_name'. A variety of parameters are included by default. Additional arbitrary columns allowed. See *pysat.Meta* for more information on creating the initial metadata.
- Note that users may opt for a different naming scheme for metadata parameters thus the most general code for working with metadata uses the attached labels,

```
# update units to meters, 'm' for variable
inst.meta[variable, inst.units_label] = 'm'
```

- If metadata is already stored with the file, creating the Meta object is generally trivial. If this isn't the case, it can be tedious to fill out all information if there are many data parameters. In this case it may be easier to fill out a text file. A basic convenience function is provided for this situation. See *pysat.Meta.from_csv* for more information.

download

Download support significantly lowers the hassle in dealing with any dataset. Fetch data from the internet.

```
def download(date_array, data_path=None, user=None, password=None):
    return
```

- date_array, a list of dates to download data for
- data_path, the full path to the directory to store data
- user, string for username
- password, string for password

Routine should download data and write it to disk.

8.3 Optional Routines and Support

Custom Keywords in load Method

pysat supports the definition and use of keywords for an instrument module so that users may trigger optional features, if provided. All custom keywords for an instrument module must be defined in the *load* method.

```
def load(fnames, tag=None, sat_id=None, custom1=default1, custom2=default2):
    return data, meta
```

pysat passes any supported custom keywords and values to *load* with every call. All custom keywords along with the assigned defaults are copied into the Instrument object itself under inst.kwargs for use in other areas.

```
inst = pysat.Instrument(platform, name, custom1=new_value)
# show user supplied value for custom1 keyword
print(inst.kwargs['custom1'])
# show default value applied for custom2 keyword
print(inst.kwargs['custom2'])
```

If a user supplies a keyword that is not supported by pysat or by the specific instrument module then an error is raised.

init

If present, the instrument init method runs once at instrument instantiation.

```
def init(inst):
    return None
```

inst is a pysat.Instrument() instance. init should modify inst in-place as needed; equivalent to a 'modify' custom routine.

keywords are not supported within the init module method signature, though custom keyword support for instruments is available via inst.kwargs.

default

First custom function applied, once per instrument load.

```
def default(inst):
    return None
```

inst is a pysat.Instrument() instance. default should modify inst in-place as needed; equivalent to a 'modify' custom routine.

clean

Cleans instrument for levels supplied in inst.clean_level.

- 'clean' : expectation of good data
- 'dusty' : probably good data, use with caution
- 'dirty' : minimal cleaning, only blatant instrument errors removed
- 'none' : no cleaning, routine not called

```
def clean(inst):
    return None
```

inst is a pysat.Instrument() instance. clean should modify inst in-place as needed; equivalent to a 'modify' custom routine.

list_remote_files

Returns a list of available files on the remote server. This method is required for the Instrument module to support the *download_updated_files* method, which makes it trivial for users to ensure they always have the most up to date data. pysat developers highly encourage the development of this method, when possible.

```
def list_remote_files(inst):
    return list_like
```

This method is called by several internal *pysat* functions, and can be directly called by the user through the *inst.remote_file_list* command. The user can search for subsets of files through optional keywords, such as

```
inst.remote_file_list(year=2019)
inst.remote_file_list(year=2019, month=1, day=1)
```

8.4 Logging

pysat is connected to the Python logging module. This allows users to set the desired level of direct feedback, as well as where feedback statements are delivered. The following line in each module is encouraged at the top-level so that the instrument module can provide feedback using the same mechanism

```
logger = logging.getLogger(__name__)
```

Within any instrument module,

```
logger.info(information_string)
logger.warning(warning_string)
logger.debug(debug_string)
```

will direct information, warnings, and debug statements appropriately.

8.5 Testing Support

All modules defined in the __init__.py for pysat/instruments are automatically tested when pysat code is tested. To support testing all of the required routines, additional information is required by pysat.

Example code from dmsp_ivm.py. The attributes are set at the top level simply by defining variable names with the proper info. The various satellites within DMSP, F11, F12, F13 are separated out using the sat_id parameter. 'utd' is used as a tag to delineate that the data contains the UTD developed quality flags.

```
platform = 'dmsp'
name = 'ivm'
tags = {'utd': 'UTDallas DMSP data processing',
        '': 'Level 1 data processing'}
sat_ids = {'f11': ['utd', ''], 'f12': ['utd', ''], 'f13': ['utd', ''],
           'f14': ['utd', ''], 'f15': ['utd', ''], 'f16': [''], 'f17': [''],
           'f18': ['']}
_test_dates = {'f11': {'utd': pysat.datetime(1998, 1, 2)},
               'f12': {'utd': pysat.datetime(1998, 1, 2)},
               'f13': {'utd': pysat.datetime(1998, 1, 2)},
               'f14': {'utd': pysat.datetime(1998, 1, 2)},
               'f15': {'utd': pysat.datetime(2017, 12, 30)}}

 # support load routine
 def load(fnames, tag=None, sat_id=None):
     # code normally follows, example terminates here
```

The rationale behind the variable names is explained above under Naming Conventions. What is important here are the _test_dates. Each of these points to a specific date for which the unit tests will attempt to download and load data as part of end-to-end testing. Make sure that the data exists for the given date. The tags without test dates will not be tested. The leading underscore in _test_dates ensures that this information is not added to the instrument's meta attributes, so it will not be present in IO operations.

8.6 Data Acknowledgements

Acknowledging the source of data is key for scientific collaboration. This can generally be put in the *init* function of each instrument.

```
def init(self):
    """Initializes the Instrument object with instrument specific values.

    Runs once upon instantiation.

    Parameters
    ----------
    inst : (pysat.Instrument)
        Instrument class object

    """

    self.meta.acknowledgements = acknowledgements_string
    self.meta.references = references_string

    return
```

8.7 Supported Instrument Templates

Instrument templates may be found at `pysat.instruments.templates` and supporting methods may be found at `pysat.instruments.methods`.

8.7.1 General

A general instrument template is included with pysat, `pysat.instruments.templates.template_instrument`, that has the full set of required and optional methods, and docstrings, that may be used as a starting point for adding a new instrument to pysat.

Note that there are general supporting methods for adding an Instrument. See *General* for more.

8.7.2 NASA CDAWeb

A template for NASA CDAWeb pysat support is provided. Several of the routines within are intended to be used with functools.partial in the new instrument support code. When writing custom routines with a new instrument file download support would normally be added via

```
def download(.....)
```

Using the CDAWeb template the equivalent action is

```
download = functools.partial(methods.nasa_cdaweb.download,
                             supported_tags)
```

where supported_tags is defined as dictated by the download function. See the routines for cnofs_vefi and cnofs_ivm for practical uses of the NASA CDAWeb support code.

See *NASA CDAWeb* for more.

8.7.3 Madrigal

A template for Madrigal pysat support is provided. Several of the routines within are intended to be used with functools.partial in the new instrument support code. When writing custom routines with a new instrument file download support would normally be added via

```
def download(.....)
```

Using the Madrigal template the equivalent action is

```
def download(date_array, tag='', sat_id='', data_path=None, user=None,
             password=None):
    methods.madrigal.download(date_array, inst_code=str(madrigal_inst_code),
                              kindat=str(madrigal_tag[sat_id][tag]),
                              data_path=data_path, user=user,
                              password=password)
```

See the routines for *dmsp_ivm* and *jro_isr* for practical uses of the Madrigal support code.

Additionally, use of the methods.madrigal class should acknowledge the CEDAR rules of the road. This can be done by Adding

```
def init(self):

    print(methods.madrigal.cedar_rules())
    return
```

to each routine that uses Madrigal data access.

See *NASA ICON* for more.

CHAPTER 9

API

9.1 Instrument

class pysat.**Instrument**(*platform=None*, *name=None*, *tag=None*, *inst_id=None*, *sat_id=None*, *clean_level='clean'*, *update_files=None*, *pad=None*, *orbit_info=None*, *inst_module=None*, *multi_file_day=None*, *manual_org=None*, *directory_format=None*, *file_format=None*, *temporary_file_list=False*, *strict_time_flag=False*, *ignore_empty_files=False*, *units_label='units'*, *name_label='long_name'*, *notes_label='notes'*, *desc_label='desc'*, *plot_label='label'*, *axis_label='axis'*, *scale_label='scale'*, *min_label='value_min'*, *max_label='value_max'*, *fill_label='fill'*, **arg*, ***kwargs*)

Download, load, manage, modify and analyze science data.

Deprecated since version 2.3.0: Several attributes and methods will be removed or replaced in pysat 3.0.0: sat_id, default, multi_file_day, manual_org, units_label, name_label, notes_label, desc_label, min_label, max_label, fill_label, plot_label, axis_label, scale_label, and _filter_datetime_input

Parameters

- **platform** (*string*) – name of platform/satellite.
- **name** (*string*) – name of instrument.
- **tag** (*string, optional*) – identifies particular subset of instrument data.
- **inst_id** (*string*) – Replaces *sat_id*
- **sat_id** (*string, optional*) – identity within constellation
- **clean_level** (*{'clean','dusty','dirty','none'}, optional*) – level of data quality
- **pad** (*pandas.DateOffset, or dictionary, optional*) – Length of time to pad the begining and end of loaded data for time-series processing. Extra data is removed after applying all custom functions. Dictionary, if supplied, is simply passed to pandas DateOffset.

- **orbit_info** (*dict*) – Orbit information, {'index':index, 'kind':kind, 'period':period}. See pysat.Orbits for more information.
- **inst_module** (*module, optional*) – Provide instrument module directly. Takes precedence over platform/name.
- **update_files** (*boolean, optional*) – If True, immediately query filesystem for instrument files and store.
- **temporary_file_list** (*boolean, optional*) – If true, the list of Instrument files will not be written to disk. Prevents a race condition when running multiple pysat processes.
- **strict_time_flag** (*boolean, option (False)*) – If true, pysat will check data to ensure times are unique and monotonic. In future versions, this will be fixed to True.
- **multi_file_day** (*boolean, optional*) – Set to True if Instrument data files for a day are spread across multiple files and data for day n could be found in a file with a timestamp of day n-1 or n+1. Deprecated at this level in pysat 3.0.0.
- **manual_org** (*bool*) – if True, then pysat will look directly in pysat data directory for data files and will not use default /platform/name/tag. Deprecated in pysat 3.0.0, as this flag is not needed to use *directory_format*.
- **directory_format** (*str*) – directory naming structure in string format. Variables such as platform, name, and tag will be filled in as needed using python string formatting. The default directory structure would be expressed as '{platform}/{name}/{tag}'
- **file_format** (*str or NoneType*) – File naming structure in string format. Variables such as year, month, and sat_id will be filled in as needed using python string formatting. The default file format structure is supplied in the instrument list_files routine.
- **ignore_empty_files** (*boolean*) – if True, the list of files found will be checked to ensure the filesizes are greater than zero. Empty files are removed from the stored list of files.
- **units_label** (*str*) – String used to label units in storage. Defaults to 'units'.
- **name_label** (*str*) – String used to label long_name in storage. Defaults to 'name'.
- **notes_label** (*str*) – label to use for notes in storage. Defaults to 'notes'
- **desc_label** (*str*) – label to use for variable descriptions in storage. Defaults to 'desc'
- **plot_label** (*str*) – label to use to label variables in plots. Defaults to 'label'
- **axis_label** (*str*) – label to use for axis on a plot. Defaults to 'axis'
- **scale_label** (*str*) – label to use for plot scaling type in storage. Defaults to 'scale'
- **min_label** (*str*) – label to use for typical variable value min limit in storage. Defaults to 'value_min'
- **max_label** (*str*) – label to use for typical variable value max limit in storage. Defaults to 'value_max'
- **fill_label** (*str*) – label to use for fill values. Defaults to 'fill' but some implementations will use 'FillVal'

data
loaded science data

Type pandas.DataFrame

date
date for loaded data

Type pandas.datetime

yr
year for loaded data

Type int

bounds
bounds for loading data, supply array_like for a season with gaps. Users may provide as a tuple or tuple of lists, but the attribute is stored as a tuple of lists for consistency

Type (datetime/filename/None, datetime/filename/None)

doy
day of year for loaded data

Type int

files
interface to instrument files

Type *pysat.Files*

meta
interface to instrument metadata, similar to netCDF 1.6

Type *pysat.Meta*

orbits
interface to extracting data orbit-by-orbit

Type *pysat.Orbits*

custom
interface to instrument nano-kernel

Type *pysat.Custom*

kwargs
keyword arguments passed to instrument loading routine

Type dictionary

Note: Pysat attempts to load the module platform_name.py located in the pysat/instruments directory. This module provides the underlying functionality to download, load, and clean instrument data. Alternatively, the module may be supplied directly using keyword inst_module.

Examples

```
# 1-second mag field data
vefi = pysat.Instrument(platform='cnofs',
                        name='vefi',
                        tag='dc_b',
                        clean_level='clean')
start = pysat.datetime(2009,1,1)
stop = pysat.datetime(2009,1,2)
vefi.download(start, stop)
vefi.load(date=start)
print(vefi['dB_mer'])
```

(continues on next page)

(continued from previous page)

```
print(vefi.meta['db_mer'])

# 1-second thermal plasma parameters
ivm = pysat.Instrument(platform='cnofs',
                       name='ivm',
                       tag='',
                       clean_level='clean')
ivm.download(start,stop)
ivm.load(2009,1)
print(ivm['ionVelmeridional'])

# Ionosphere profiles from GPS occultation
cosmic = pysat.Instrument('cosmic',
                          'gps',
                          'ionprf',
                          altitude_bin=3)
# bins profile using 3 km step
cosmic.download(start, stop, user=user, password=password)
cosmic.load(date=start)
```

bounds
Boundaries for iterating over instrument object by date or file.

Parameters

- **start** (*datetime object, filename, or None (default)*) – start of iteration, if None uses first data date. list-like collection also accepted
- **end** (*datetime object, filename, or None (default)*) – end of iteration, inclusive. If None uses last data date. list-like collection also accepted

---

Note: Both start and stop must be the same type (date, or filename) or None. Only the year, month, and day are used for date inputs.

Examples

```
inst = pysat.Instrument(platform=platform,
                        name=name,
                        tag=tag)
start = pysat.datetime(2009,1,1)
stop = pysat.datetime(2009,1,31)
inst.bounds = (start,stop)

start2 = pysat.datetetime(2010,1,1)
stop2 = pysat.datetime(2010,2,14)
inst.bounds = ([start, start2], [stop, stop2])
```

concat_data(*data*, **args*, ***kwargs*)
Concats data1 and data2 for xarray or pandas as needed

Parameters **data** (*pandas or xarray*) – Data to be appended to data already within the Instrument object

Returns Instrument.data modified in place.

Return type void

Notes

For pandas, sort=False is passed along to the underlying pandas.concat method. If sort is supplied as a keyword, the user provided value is used instead.

For xarray, dim='Epoch' is passed along to xarray.concat except if the user includes a value for dim as a keyword argument.

copy()
Deep copy of the entire Instrument object.

date
Date for loaded data.

download(*start=None*, *stop=None*, *freq='D'*, *user=None*, *password=None*, *date_array=None*, ***kwargs*)
Download data for given Instrument object from start to stop.

Parameters

- **start** (*pandas.datetime (yesterday)*) – start date to download data
- **stop** (*pandas.datetime (tomorrow)*) – stop date to download data
- **freq** (*string*) – Stepsize between dates for season, 'D' for daily, 'M' monthly (see pandas)
- **user** (*string*) – username, if required by instrument data archive
- **password** (*string*) – password, if required by instrument data archive
- **date_array** (*list-like*) – Sequence of dates to download date for. Takes precendence over start and stop inputs
- ****kwargs** (*dict*) – Dictionary of keywords that may be options for specific instruments

---

Note: Data will be downloaded to pysat_data_dir/patform/name/tag

If Instrument bounds are set to defaults they are updated after files are downloaded.

---

download_updated_files(*user=None*, *password=None*, ***kwargs*)
Grabs a list of remote files, compares to local, then downloads new files.

Parameters

- **user** (*string*) – username, if required by instrument data archive
- **password** (*string*) – password, if required by instrument data archive
- ****kwargs** (*dict*) – Dictionary of keywords that may be options for specific instruments

---

Note: Data will be downloaded to pysat_data_dir/patform/name/tag

If Instrument bounds are set to defaults they are updated after files are downloaded.

---

empty
Boolean flag reflecting lack of data.

True if there is no Instrument data.

generic_meta_translator(*meta_to_translate*)
Translates the metadate contained in an object into a dictionary suitable for export.

Parameters **meta_to_translate** (Meta) – The metadata object to translate

Returns A dictionary of the metadata for each variable of an output file e.g. netcdf4

Return type dict

index
Returns time index of loaded data.

load(*yr=None*, *doy=None*, *date=None*, *fname=None*, *fid=None*, *verifyPad=False*)
Load instrument data into Instrument object .data.

Parameters

- **yr** (*integer*) – year for desired data
- **doy** (*integer*) – day of year
- **date** (*datetime object*) – date to load
- **fname** ('*string*') – filename to be loaded
- **verifyPad** (*boolean*) – if True, padding data not removed (debug purposes)

Returns

Return type Void. Data is added to self.data

Note: Loads data for a chosen instrument into .data. Any functions chosen by the user and added to the custom processing queue (.custom.add) are automatically applied to the data before it is available to user in .data.

---

next(*verifyPad=False*)
Manually iterate through the data loaded in Instrument object.

Bounds of iteration and iteration type (day/file) are set by *bounds* attribute.

Note: If there were no previous calls to load then the first day(default)/file will be loaded.

---

prev(*verifyPad=False*)
Manually iterate backwards through the data in Instrument object.

Bounds of iteration and iteration type (day/file) are set by *bounds* attribute.

Note: If there were no previous calls to load then the first day(default)/file will be loaded.

---

remote_date_range(*year=None*, *month=None*, *day=None*)
Returns fist and last date for remote data. Default behaviour is to search all files. User may additionally specify a given year, year/month, or year/month/day combination to return a subset of available files.

remote_file_list(*year=None*, *month=None*, *day=None*)
List remote files for chosen instrument. Default behaviour is to return all files. User may additionally specify a given year, year/month, or year/month/day combination to return a subset of available files.

to_netcdf4(*fname=None*, *base_instrument=None*, *epoch_name='Epoch'*, *zlib=False*, *complevel=4*, *shuffle=True*, *preserve_meta_case=False*, *export_nan=None*, *unlimited_time=True*)
Stores loaded data into a netCDF4 file.

Parameters

- **fname** (*string*) – full path to save instrument object to
- **base_instrument** (pysat.Instrument) – used as a comparison, only attributes that are present with self and not on base_instrument are written to netCDF
- **epoch_name** (*str*) – Label in file for datetime index of Instrument object
- **zlib** (*boolean*) – Flag for engaging zlib compression (True - compression on)
- **complevel** (*int*) – an integer between 1 and 9 describing the level of compression desired (default 4). Ignored if zlib=False
- **shuffle** (*boolean*) – the HDF5 shuffle filter will be applied before compressing the data (default True). This significantly improves compression. Default is True. Ignored if zlib=False.
- **preserve_meta_case** (*bool (False)*) – if True, then the variable strings within the MetaData object, which preserves case, are used to name variables in the written netCDF file. If False, then the variable strings used to access data from the Instrument object are used instead. By default, the variable strings on both the data and metadata side are the same, though this relationship may be altered by a user.
- **export_nan** (*list or None*) – By default, the metadata variables where a value of NaN is allowed and written to the netCDF4 file is maintained by the Meta object attached to the pysat.Instrument object. A list supplied here will override the settings provided by Meta, and all parameters included will be written to the file. If not listed and a value is NaN then that attribute simply won't be included in the netCDF4 file.
- **unlimited_time** (*bool*) – If True, then the main epoch dimension will be set to 'unlimited' within the netCDF4 file. (default=True)

Note: Stores 1-D data along dimension 'epoch' - the date time index.

Stores higher order data (e.g. dataframes within series) separately

- The name of the main variable column is used to prepend subvariable names within netCDF, var_subvar_sub
- A netCDF4 dimension is created for each main variable column with higher order data; first dimension Epoch
- The index organizing the data stored as a dimension variable
- from_netcdf4 uses the variable dimensions to reconstruct data structure

---

All attributes attached to instrument meta are written to netCDF attrs with the exception of 'Date_End', 'Date_Start', 'File', 'File_Date', 'Generation_Date', and 'Logical_File_ID'. These are defined within to_netCDF at the time the file is written, as per the adopted standard, SPDF ISTP/IACG Modified for NetCDF. Atrributes 'Conventions' and 'Text_Supplement' are given default values if not present.

today()

Returns today's date, with no hour, minute, second, etc.

Parameters **None** –

Returns Today's date

Return type datetime

tomorrow()

Returns tomorrow's date, with no hour, minute, second, etc.

Parameters **None** –

Returns Tomorrow's date

Return type datetime

variables
Returns list of variables within loaded data.

yesterday()
Returns yesterday's date, with no hour, minute, second, etc.

Parameters **None** –

Returns Yesterday's date

Return type datetime

9.2 Instrument Methods

The following methods support the variety of actions needed by underlying pysat.Instrument modules.

9.2.1 Demeter

Provides non-instrument routines for DEMETER microsatellite data

Deprecated since version 2.3.0: This module has been removed from pysat in the 3.0.0 release and can now be found in pysatIncubator (https://github.com/pysat/pysatIncubator)

pysat.instruments.methods.demeter.**download**(*date_array*, *tag*, *sat_id*, *data_path=None*, *user=None*, *password=None*)
Download

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatIncubator.instruments.methods.demeter*

pysat.instruments.methods.demeter.**bytes_to_float**(*chunk*)
Convert a chunk of bytes to a float

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatIncubator.instruments.methods.demeter*

Parameters **chunk** (*string or bytes*) – A chunk of bytes

Returns **value** – A 32 bit float

Return type float

pysat.instruments.methods.demeter.**load_general_header**(*fhandle*)
Load the general header block (block 1 for each time)

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatIncubator.instruments.methods.demeter*

Parameters **fhandle** (*(file handle)*) – File handle

Returns

- **data** (*list*) – List of data values containing: P field, Number of days from 01/01/1950, number of miliseconds in the day, UT as datetime, Orbit number, downward (False) upward (True) indicator

- **meta** (*dict*) – Dictionary with meta data for keys: 'telemetry station', 'software processing version', 'software processing subversion', 'calibration file version', and 'calibration file subversion', 'data names', 'data units'

pysat.instruments.methods.demeter.**load_location_parameters**(*fhandle*)
Load the orbital and geomagnetic parameter block (block 1 for each time)

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatIncubator.instruments.methods.demeter*

Parameters **fhandle** (*(file handle)*) – File handle

Returns

- **data** (*list*) – List of data values containing: geoc lat, geoc lon, alt, lt, geom lat, geom lon, mlt, inv lat, L-shell, geoc lat of conj point, geoc lon of conj point, geoc lat of N conj point at 110 km, geoc lon of N conj point at 110 km, geoc lat of S conj point at 110 km, geoc lon of S conj point at 110 km, components of magnetic field at sat point, proton gyrofreq at sat point, solar position in geog coords
- **meta** (*dict*) – Dictionary with meta data for keys: 'software processing version', 'software processing subversion', 'data names', 'data units'

pysat.instruments.methods.demeter.**load_attitude_parameters**(*fhandle*)
Load the attitude parameter block (block 1 for each time)

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatIncubator.instruments.methods.demeter*

Parameters **fhandle** (*(file handle)*) – File handle

Returns

- **data** (*list*) – list of data values containing: matrix elements from satellite coord system to geographic coordinate system, matrix elements from geographic coordinate system to local geomagnetic coordinate system, quality index of attitude parameters.
- **meta** (*dict*) – Dictionary with meta data for keys: 'software processing version', 'software processing subversion', 'data names', 'data units'

pysat.instruments.methods.demeter.**load_binary_file**(*fname*, *load_experiment_data*)
Load the binary data from a DEMETER file

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatIncubator.instruments.methods.demeter*

Parameters

- **fname** (*string*) – Filename
- **load_experiment_data** (*function*) – Function to load experiment data, taking the file handle as input

Returns

- **data** (*np.array*) – Data from file stored in a numpy array
- **meta** (*dict*) – Meta data for file, including data names and units

pysat.instruments.methods.demeter.**set_metadata**(*name*, *meta_dict*)
Set metadata for each DEMETER instrument, using dict containing metadata

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatIncubator.instruments.methods.demeter*

Parameters

- **name** (*string*) – DEMETER instrument name
- **meta_dict** (*dict*) – Dictionary containing metadata information and data attributes. Data attributes are available in the keys 'data names' and 'data units'

Returns **meta** – Meta class boject

Return type *pysat.Meta*

9.2.2 General

Provides generalized routines for integrating instruments into pysat.

pysat.instruments.methods.general.**convert_timestamp_to_datetime**(*inst*, *sec_mult=1.0*)

Use datetime instead of timestamp for Epoch

Parameters

- **inst** (pysat.Instrument) – associated pysat.Instrument object
- **sec_mult** (*float*) – Multiplier needed to convert epoch time to seconds (default=1.0)

pysat.instruments.methods.general.**list_files**(*tag=None*, *sat_id=None*, *data_path=None*, *format_str=None*, *supported_tags=None*, *fake_daily_files_from_monthly=False*, *two_digit_year_break=None*, *file_cadance=datetime.timedelta(days=1)*)

Return a Pandas Series of every file for chosen satellite data.

This routine provides a standard interfacefor pysat instrument modules.

Deprecated since version 2.3.0: The fake_daily_files_from_monthly kwarg has been deprecated and replaced with file_cadance in pysat 3.0.0.

Parameters

- **tag** (*string or NoneType*) – Denotes type of file to load. Accepted types are <tag strings>. (default=None)
- **sat_id** (*string or NoneType*) – Specifies the satellite ID for a constellation. Not used. (default=None)
- **data_path** (*string or NoneType*) – Path to data directory. If None is specified, the value previously set in Instrument.files.data_path is used. (default=None)
- **format_str** (*string or NoneType*) – User specified file format. If None is specified, the default formats associated with the supplied tags are used. (default=None)
- **supported_tags** (*dict or NoneType*) – keys are sat_id, each containing a dict keyed by tag where the values file format template strings. (default=None)
- **fake_daily_files_from_monthly** (*bool*) – Some CDAWeb instrument data files are stored by month, interfering with pysat's functionality of loading by day. This flag, when true, appends daily dates to monthly files internally. These dates are used by load routine in this module to provide data by day. This keyword arg has been deprecated. In pysat 2.3.0, setting file_cadance=dt.datetime(days=1) is equivalent to setting this to False, while using file_cadance=pds.DateOffset(months=1) is equivalent to setting this to True. (default=False)

- **two_digit_year_break** (*int*) – If filenames only store two digits for the year, then '1900' will be added for years >= two_digit_year_break and '2000' will be added for years < two_digit_year_break.
- **file_cadence** (*dt.timedelta or pds.DateOffset*) – pysat assumes a daily file cadence, but some instrument data file contain longer periods of time. This parameter allows the specification of regular file cadences greater than or equal to a day (e.g., weekly, monthly, or yearly). In pysat 2.3.0, only daily and monthly cadances are supported. (default=dt.timedelta(days=1))

Returns **pysat.Files.from_os** – A class containing the verified available files

Return type (pysat._files.Files)

Examples

```
fname = 'cnofs_vefi_bfield_1sec_{year:04d}{month:02d}{day:02d}_v05.cdf'
supported_tags = {'dc_b': fname}
list_files = functools.partial(nasa_cdaweb.list_files,
                               supported_tags=supported_tags)

fname = 'cnofs_cindi_ivm_500ms_{year:4d}{month:02d}{day:02d}_v01.cdf'
supported_tags = {'': fname}
list_files = functools.partial(mm_gen.list_files,
                               supported_tags=supported_tags)
```

pysat.instruments.methods.general.**remove_leading_text**(*inst*, *target=None*)

Removes leading text on variable names :param inst: associated pysat.Instrument object :type inst: pysat.Instrument :param target: Leading string to remove. If none supplied, returns unmodified :type target: str or list of strings

Returns Modifies Instrument object in place

Return type None

9.2.3 NASA CDAWeb

Provides default routines for integrating NASA CDAWeb instruments into pysat. Adding new CDAWeb datasets should only require mininal user intervention.

pysat.instruments.methods.nasa_cdaweb.**load**(*fnames*, *tag=None*, *sat_id=None*, *fake_daily_files_from_monthly=False*, *flatten_twod=True*)

Load NASA CDAWeb CDF files.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatNASA.instruments.methods.cdaweb*

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **fnames** (*(pandas.Series)*) – Series of filenames
- **tag** (*(str or NoneType)*) – tag or None (default=None)
- **sat_id** (*(str or NoneType)*) – satellite id or None (default=None)

- **fake_daily_files_from_monthly** (*bool*) – Some CDAWeb instrument data files are stored by month, interfering with pysat's functionality of loading by day. This flag, when true, parses of daily dates to monthly files that were added internally by the list_files routine, when flagged. These dates are used here to provide data by day.
- **flatted_twod** (*bool*) – Flattens 2D data into different columns of root DataFrame rather than produce a Series of DataFrames

Returns

- **data** (*(pandas.DataFrame)*) – Object containing satellite data
- **meta** (*(pysat.Meta)*) – Object containing metadata such as column names and units

Examples

```
# within the new instrument module, at the top level define
# a new variable named load, and set it equal to this load method
# code below taken from cnofs_ivm.py.

# support load routine
# use the default CDAWeb method
load = cdw.load
```

pysat.instruments.methods.nasa_cdaweb.**list_files**(*tag=None*, *sat_id=None*, *data_path=None*, *format_str=None*, *supported_tags=None*, *fake_daily_files_from_monthly=False*, *two_digit_year_break=None*)

Return a Pandas Series of every file for chosen satellite data.

Deprecated since version 2.2.0: *list_files* will be removed in pysat 3.0.0, it will be replaced by the copy in instruments.methods.general

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **tag** (*(string or NoneType)*) – Denotes type of file to load. Accepted types are <tag strings>. (default=None)
- **sat_id** (*(string or NoneType)*) – Specifies the satellite ID for a constellation. Not used. (default=None)
- **data_path** (*(string or NoneType)*) – Path to data directory. If None is specified, the value previously set in Instrument.files.data_path is used. (default=None)
- **format_str** (*(string or NoneType)*) – User specified file format. If None is specified, the default formats associated with the supplied tags are used. (default=None)
- **supported_tags** (*(dict or NoneType)*) – keys are sat_id, each containing a dict keyed by tag where the values file format template strings. (default=None)
- **fake_daily_files_from_monthly** (*(bool)*) – Some CDAWeb instrument data files are stored by month, interfering with pysat's functionality of loading by day. This flag, when true, appends daily dates to monthly files internally. These dates are used by load routine in this module to provide data by day.
- **two_digit_year_break** (*(int)*) – If filenames only store two digits for the year, then '1900' will be added for years >= two_digit_year_break and '2000' will be added for years < two_digit_year_break.

Returns **pysat.Files.from_os** – A class containing the verified available files

Return type (pysat._files.Files)

Examples

```
fname = 'cnofs_vefi_bfield_1sec_{year:04d}{month:02d}{day:02d}_v05.cdf'
supported_tags = {'dc_b': fname}
list_files = functools.partial(nasa_cdaweb.list_files,
                               supported_tags=supported_tags)

fname = 'cnofs_cindi_ivm_500ms_{year:4d}{month:02d}{day:02d}_v01.cdf'
supported_tags = {'': fname}
list_files = functools.partial(cdw.list_files,
                               supported_tags=supported_tags)
```

pysat.instruments.methods.nasa_cdaweb.**list_remote_files**(*tag*, *sat_id*, *remote_site='https://cdaweb.gsfc.nasa.gov'*, *supported_tags=None*, *user=None*, *password=None*, *fake_daily_files_from_monthly=False*, *two_digit_year_break=None*, *delimiter=None*, *year=None*, *month=None*, *day=None*)

Return a Pandas Series of every file for chosen remote data.

Deprecated since version 2.3.0: This routine will be removed in pysat 3.0.0, it will be moved to the pysatNASA repository. Also, as of 2.2.0 the *year/month/day* keywords will be removed in pysat 3.0.0, they will be replaced with a start/stop syntax consistent with the download routine

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **tag** (*string or NoneType*) – Denotes type of file to load. Accepted types are <tag strings>. (default=None)
- **sat_id** (*string or NoneType*) – Specifies the satellite ID for a constellation. (default=None)
- **remote_site** (*string or NoneType*) – Remote site to download data from (default='https://cdaweb.gsfc.nasa.gov')
- **supported_tags** (*dict*) – dict of dicts. Keys are supported tag names for download. Value is a dict with 'dir', 'remote_fname', 'local_fname'. Inteded to be pre-set with functools.partial then assigned to new instrument code.
- **user** (*string or NoneType*) – Username to be passed along to resource with relevant data. (default=None)
- **password** (*string or NoneType*) – User password to be passed along to resource with relevant data. (default=None)
- **fake_daily_files_from_monthly** (*bool*) – Some CDAWeb instrument data files are stored by month. This flag, when true, accomodates this reality with user feedback on a monthly time frame. (default=False)

- **two_digit_year_break** (*(int or NoneType)*) – If filenames only store two digits for the year, then '1900' will be added for years >= two_digit_year_break and '2000' will be added for years < two_digit_year_break. (default=None)
- **delimiter** (*(string or NoneType)*) – If filename is delimited, then provide delimiter alone e.g. '_' (default=None)
- **year** (*(int or NoneType)*) – Selects a given year to return remote files for. None returns all years. (default=None)
- **month** (*(int or NoneType)*) – Selects a given month to return remote files for. None returns all months. Requires year to be defined. (default=None)
- **day** (*(int or NoneType)*) – Selects a given day to return remote files for. None returns all days. Requires year and month to be defined. (default=None)

Returns **pysat.Files.from_os** – A class containing the verified available files

Return type (pysat._files.Files)

Examples

```
fname = 'cnofs_vefi_bfield_1sec_{year:04d}{month:02d}{day:02d}_v05.cdf'
supported_tags = {'dc_b': fname}
list_remote_files =             functools.partial(nasa_cdaweb.list_remote_files,
                      supported_tags=supported_tags)

fname = 'cnofs_cindi_ivm_500ms_{year:4d}{month:02d}{day:02d}_v01.cdf'
supported_tags = {'': fname}
list_remote_files =             functools.partial(cdw.list_remote_files,
                      supported_tags=supported_tags)
```

pysat.instruments.methods.nasa_cdaweb.**download**(*supported_tags*, *date_array*, *tag*, *sat_id*, *remote_site='https://cdaweb.gsfc.nasa.gov'*, *data_path=None*, *user=None*, *password=None*, *fake_daily_files_from_monthly=False*, *multi_file_day=False*)

Routine to download NASA CDAWeb CDF data.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatNASA.instruments.methods.cdaweb*

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **supported_tags** (*dict*) – dict of dicts. Keys are supported tag names for download. Value is a dict with 'dir', 'remote_fname', 'local_fname'. Inteded to be pre-set with functools.partial then assigned to new instrument code.
- **date_array** (*array_like*) – Array of datetimes to download data for. Provided by pysat.
- **tag** (*str or NoneType (None)*) – tag or None
- **sat_id** (*(str or NoneType)*) – satellite id or None (default=None)
- **remote_site** (*(string or NoneType)*) – Remote site to download data from (default='https://cdaweb.gsfc.nasa.gov')

- **data_path** (*(string or NoneType)*) – Path to data directory. If None is specified, the value previously set in Instrument.files.data_path is used. (default=None)
- **user** (*(string or NoneType)*) – Username to be passed along to resource with relevant data. (default=None)
- **password** (*(string or NoneType)*) – User password to be passed along to resource with relevant data. (default=None)
- **fake_daily_files_from_monthly** (*bool*) – Some CDAWeb instrument data files are stored by month. This flag, when true, accomodates this reality with user feedback on a monthly time frame.

Returns **Void** – Downloads data to disk.

Return type (NoneType)

Examples

```
# download support added to cnofs_vefi.py using code below
rn = '{year:4d}/cnofs_vefi_bfield_1sec_{year:4d}{month:02d}{day:02d}'+
    '_v05.cdf'
ln = 'cnofs_vefi_bfield_1sec_{year:4d}{month:02d}{day:02d}_v05.cdf'
dc_b_tag = {'dir':'/pub/data/cnofs/vefi/bfield_1sec',
            'remote_fname': rn,
            'local_fname': ln}
supported_tags = {'dc_b': dc_b_tag}

download = functools.partial(nasa_cdaweb.download,
                             supported_tags=supported_tags)
```

9.2.4 NASA ICON

Provides non-instrument specific routines for ICON data

Deprecated since version 2.3.0: This module has been removed from pysat in the 3.0.0 release and can now be found in pysatIncubator (https://github.com/pysat/pysatNASA)

pysat.instruments.methods.icon.**list_remote_files**(*tag*, *sat_id*, *user=None*, *password=None*, *supported_tags=None*, *year=None*, *month=None*, *day=None*, *start=None*, *stop=None*)

Return a Pandas Series of every file for chosen remote data.

This routine is intended to be used by pysat instrument modules supporting a particular UC-Berkeley SSL dataset related to ICON.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatNASA.instruments.methods.icon*

Parameters

- **tag** (*string or NoneType*) – Denotes type of file to load. Accepted types are <tag strings>. (default=None)
- **sat_id** (*string or NoneType*) – Specifies the satellite ID for a constellation. Not used. (default=None)

- **user** (*string or NoneType*) – Username to be passed along to resource with relevant data. (default=None)
- **password** (*string or NoneType*) – User password to be passed along to resource with relevant data. (default=None)
- **start** (*dt.datetime or NoneType*) – Starting time for file list. A None value will start with the first file found. (default=None)
- **stop** (*dt.datetime or NoneType*) – Ending time for the file list. A None value will stop with the last file found. (default=None)

Returns A Series formatted for the Files class (pysat._files.Files) containing filenames and indexed by date and time

Return type pandas.Series

pysat.instruments.methods.icon.**ssl_download**(*date_array*, *tag*, *sat_id*, *data_path=None*, *user=None*, *password=None*, *supported_tags=None*)

Download ICON data from public area of SSL ftp server

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0. It is replaced by the pysatNASA.instruments.methods.cdaweb.download method.

Parameters

- **date_array** (*array-like*) – list of datetimes to download data for. The sequence of dates need not be contiguous.
- **tag** (*string*) – Tag identifier used for particular dataset. This input is provided by pysat. (default='')
- **sat_id** (*string*) – Satellite ID string identifier used for particular dataset. This input is provided by pysat. (default='')
- **data_path** (*string*) – Path to directory to download data to. (default=None)
- **user** (*string*) – User string input used for download. Provided by user and passed via pysat. If an account is required for downloads this routine here must error if user not supplied. (default=None)
- **password** (*string*) – Password for data download. (default=None)
- ****kwargs** (*dict*) – Additional keywords supplied by user when invoking the download routine attached to a pysat.Instrument object are passed to this routine via kwargs.

9.2.5 Madrigal

Provides default routines for integrating CEDAR Madrigal instruments into pysat, reducing the amount of user intervention.

Deprecated since version 2.3.0: This module has been removed from pysat in the 3.0.0 release and can now be found in pysatMadrigal (https://github.com/pysat/pysatMadrigal)

pysat.instruments.methods.madrigal.**cedar_rules**()

General acknowledgement statement for Madrigal data.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatMadrigal.instruments.methods.madrigal*

Returns **ackn** – String with general acknowledgement for all CEDAR Madrigal data

Return type string

pysat.instruments.methods.madrigal.**load**(*fnames*, *tag=None*, *sat_id=None*, *xarray_coords=[]*)

Loads data from Madrigal into Pandas.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatMadrigal.instruments.methods.madrigal*

This routine is called as needed by pysat. It is not intended for direct user interaction.

Parameters

- **fnames** (*array-like*) – iterable of filename strings, full path, to data files to be loaded. This input is nominally provided by pysat itself.
- **tag** (*string ('')*) – tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself. While tag defaults to None here, pysat provides '' as the default tag unless specified by user at Instrument instantiation.
- **sat_id** (*string ('')*) – Satellite ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself.
- **xarray_coords** (*list*) – List of keywords to use as coordinates if xarray output is desired instead of a Pandas DataFrame (default=[])

Returns

- **data** (*pds.DataFrame or xr.DataSet*) – A pandas DataFrame or xarray DataSet holding the data from the HDF5 file
- **metadata** (*pysat.Meta*) – Metadata from the HDF5 file, as well as default values from pysat

Examples

:: inst = pysat.Instrument('jro', 'isr', 'drifts') inst.load(2010,18)

pysat.instruments.methods.madrigal.**download**(*date_array*, *inst_code=None*, *kindat=None*, *data_path=None*, *user=None*, *password=None*, *url='http://cedar.openmadrigal.org'*, *file_format='hdf5'*)

Downloads data from Madrigal.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatMadrigal.instruments.methods.madrigal*

Parameters

- **date_array** (*array-like*) – list of datetimes to download data for. The sequence of dates need not be contiguous.
- **inst_code** (*string (None)*) – Madrigal instrument code(s), cast as a string. If multiple are used, separate them with commas.
- **kindat** (*string (None)*) – Experiment instrument code(s), cast as a string. If multiple are used, separate them with commas.
- **data_path** (*string (None)*) – Path to directory to download data to.
- **user** (*string (None)*) – User string input used for download. Provided by user and passed via pysat. If an account is required for dowloads this routine here must error if user not supplied.

- **password** (*string (None)*) – Password for data download.
- **url** (string ('http://cedar.openmadrigal.org')) – URL for Madrigal site
- **file_format** (*string ('hdf5')*) – File format for Madrigal data. Load routines currently only accept 'hdf5', but any of the Madrigal options may be used here.

Returns **Void** – Downloads data to disk.

Return type (NoneType)

Notes

The user's names should be provided in field user. Ruby Payne-Scott should be entered as Ruby+Payne-Scott

The password field should be the user's email address. These parameters are passed to Madrigal when downloading.

The affiliation field is set to pysat to enable tracking of pysat downloads.

pysat.instruments.methods.madrigal.**filter_data_single_date**(*self*)
Filters data to a single date.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatMadrigal.instruments.methods.madrigal*

Parameters **self** (pysat.Instrument) – This object

Note: Madrigal serves multiple days within a single JRO file to counter this, we will filter each loaded day so that it only contains the relevant day of data. This is only applied if loading by date. It is not applied when supplying pysat with a specific filename to load, nor when data padding is enabled. Note that when data padding is enabled the final data available within the instrument will be downselected by pysat to only include the date specified.

This routine is intended to be added to the Instrument nanokernel processing queue via

```
inst = pysat.Instrument()
inst.custom.add(filter_data_single_date, 'modify')
```

This function will then be automatically applied to the Instrument object data on every load by the pysat nanokernel.

Warning: For the best performance, this function should be added first in the queue. This may be ensured by setting the default function in a pysat instrument file to this one.

within platform_name.py set

```
default = pysat.instruments.methods.madrigal.filter_data_single_date
```

at the top level

9.2.6 Space Weather

Provides default routines for solar wind and geospace indices

Deprecated since version 2.3.0: This Instrument module has been removed from pysat in the 3.0.0 release and can now be found in pysatSpaceWeather (https://github.com/pysat/pysatSpaceWeather)

pysat.instruments.methods.sw.**calc_daily_Ap**(*ap_inst*, *ap_name='3hr_ap'*, *daily_name='Ap'*, *running_name=None*)

Calculate the daily Ap index from the 3hr ap index

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and has been replaced with *pysatSpaceWeather.instruments.methods.kp_ap.calc_daily_Ap*

Parameters

- **ap_inst** (*(pysat.Instrument)*) – pysat instrument containing 3-hourly ap data
- **ap_name** (*(str)*) – Column name for 3-hourly ap data (default='3hr_ap')
- **daily_name** (*(str)*) – Column name for daily Ap data (default='Ap')
- **running_name** (*(str or NoneType)*) – Column name for daily running average of ap, not output if None (default=None)

Returns **Void**

Return type updates intrument to include daily Ap index under daily_name

Notes

Ap is the mean of the 3hr ap indices measured for a given day

Option for running average is included since this information is used by MSIS when running with sub-daily geophysical inputs

pysat.instruments.methods.sw.**combine_f107**(*standard_inst*, *forecast_inst*, *start=None*, *stop=None*)

Combine the output from the measured and forecasted F10.7 sources

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and has been replaced with *pysatSpaceWeather.instruments.methods.f107.combine_f107*

Parameters

- **standard_inst** (*(pysat.Instrument or NoneType)*) – Instrument object containing data for the 'sw' platform, 'f107' name, and '', 'all', 'prelim', or 'daily' tag
- **forecast_inst** (*(pysat.Instrument or NoneType)*) – Instrument object containing data for the 'sw' platform, 'f107' name, and 'prelim', '45day' or 'forecast' tag
- **start** (*(dt.datetime or NoneType)*) – Starting time for combining data, or None to use earliest loaded date from the pysat Instruments (default=None)
- **stop** (*(dt.datetime)*) – Ending time for combining data, or None to use the latest loaded date from the pysat Instruments (default=None)

Returns **f107_inst** – Instrument object containing F10.7 observations for the desired period of time, merging the standard, 45day, and forecasted values based on their reliability

Return type (*pysat.Instrument*)

Notes

Merging prioritizes the standard data, then the 45day data, and finally the forecast data

Will not attempt to download any missing data, but will load data

pysat.instruments.methods.sw.**combine_kp**(*standard_inst=None*, *recent_inst=None*, *forecast_inst=None*, *start=None*, *stop=None*, *fill_val=nan*)

Combine the output from the different Kp sources for a range of dates

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and has been replaced with *pysatSpaceWeather.instruments.methods.kp_ap.combine_kp*

Parameters

- **standard_inst** (*(pysat.Instrument or NoneType)*) – Instrument object containing data for the 'sw' platform, 'kp' name, and '' tag or None to exclude (default=None)
- **recent_inst** (*(pysat.Instrument or NoneType)*) – Instrument object containing data for the 'sw' platform, 'kp' name, and 'recent' tag or None to exclude (default=None)
- **forecast_inst** (*(pysat.Instrument or NoneType)*) – Instrument object containing data for the 'sw' platform, 'kp' name, and 'forecast' tag or None to exclude (default=None)
- **start** (*(dt.datetime or NoneType)*) – Starting time for combining data, or None to use earliest loaded date from the pysat Instruments (default=None)
- **stop** (*(dt.datetime)*) – Ending time for combining data, or None to use the latest loaded date from the pysat Instruments (default=None)
- **fill_val** (*(int or float)*) – Desired fill value (since the standard instrument fill value differs from the other sources) (default=np.nan)

Returns **kp_inst** – Instrument object containing Kp observations for the desired period of time, merging the standard, recent, and forecasted values based on their reliability

Return type (*pysat.Instrument*)

Notes

Merging prioritizes the standard data, then the recent data, and finally the forecast data

Will not attempt to download any missing data, but will load data

pysat.instruments.methods.sw.**convert_ap_to_kp**(*ap_data*, *fill_val=-1*, *ap_name='ap'*)

Convert Ap into Kp

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and has been replaced with *pysatSpaceWeather.instruments.methods.kp_ap.convert_ap_to_kp*

Parameters

- **ap_data** (*array-like*) – Array-like object containing Ap data
- **fill_val** (*int, float, NoneType*) – Fill value for the data set (default=-1)
- **ap_name** (*str*) – Name of the input ap

Returns

- **kp_data** (*array-like*) – Array-like object containing Kp data
- **meta** (*Metadata*) – Metadata object containing information about transformed data

9.3 Instrument Templates

9.3.1 General Instrument

This is a template for a pysat.Instrument support file. Modify this file as needed when adding a new Instrument to pysat.

This is a good area to introduce the instrument, provide background on the mission, operations, instrumentation, and measurements.

Also a good place to provide contact information. This text will be included in the pysat API documentation.

Properties

platform *List platform string here*

name *List name string here*

sat_id *List supported sat_ids here*

tag *List supported tag strings here*

Note:

- Optional section, remove if no notes

Warning:

- Optional section, remove if no warnings
- Two blank lines needed afterward for proper formatting

Examples

```
Example code can go here
```

Authors

Author name and institution

pysat.instruments.templates.template_instrument.**init**(*self*)
Initializes the Instrument object with instrument specific values.

Runs once upon instantiation. Object modified in place. Optional.

Parameters **self** (pysat.Instrument) – This object

pysat.instruments.templates.template_instrument.**default**(*self*)
Default customization function.

This routine is automatically applied to the Instrument object on every load by the pysat nanokernel (first in queue). Object modified in place.

Parameters **self** (pysat.Instrument) – This object

pysat.instruments.templates.template_instrument.**load**(*fnames*, *tag=None*, *sat_id=None*, *custom_keyword=None*)

Loads PLATFORM data into (PANDAS/XARRAY).

This routine is called as needed by pysat. It is not intended for direct user interaction.

Parameters

- **fnames** (*array-like*) – iterable of filename strings, full path, to data files to be loaded. This input is nominally provided by pysat itself.
- **tag** (*string*) – tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself. While tag defaults to None here, pysat provides '' as the default tag unless specified by user at Instrument instantiation. (default='')
- **sat_id** (*string*) – Satellite ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **custom_keyword** (*type to be set*) – Developers may include any custom keywords, with default values defined in the method signature. This is included here as a place holder and should be removed.

Returns Data and Metadata are formatted for pysat. Data is a pandas DataFrame or xarray DataSet while metadata is a pysat.Meta instance.

Return type data, metadata

Note: Any additional keyword arguments passed to pysat.Instrument upon instantiation are passed along to this routine.

Examples

```
inst = pysat.Instrument('ucar', 'tiegcm')
inst.load(2019, 1)
```

pysat.instruments.templates.template_instrument.**list_files**(*tag=None*, *sat_id=None*, *data_path=None*, *format_str=None*)

Produce a list of files corresponding to PLATFORM/NAME.

This routine is invoked by pysat and is not intended for direct use by the end user. Arguments are provided by pysat.

Parameters

- **tag** (*string*) – tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **sat_id** (*string*) – Satellite ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **data_path** (*string*) – Full path to directory containing files to be loaded. This is provided by pysat. The user may specify their own data path at Instrument instantiation and it will appear here. (default=None)

- **format_str** (*string*) – String template used to parse the datasets filenames. If a user supplies a template string at Instrument instantiation then it will appear here, otherwise defaults to None. (default=None)

Returns Series of filename strings, including the path, indexed by datetime.

Return type pandas.Series

Examples

```
If a filename is SPORT_L2_IVM_2019-01-01_v01r0000.NC then the template
is 'SPORT_L2_IVM_{year:04d}-{month:02d}-{day:02d}_' +
'v{version:02d}r{revision:04d}.NC'
```

Note: The returned Series should not have any duplicate datetimes. If there are multiple versions of a file the most recent version should be kept and the rest discarded. This routine uses the pysat.Files.from_os constructor, thus the returned files are up to pysat specifications.

Multiple data levels may be supported via the 'tag' input string. Multiple instruments via the sat_id string.

---

pysat.instruments.templates.template_instrument.**list_remote_files**(*tag*, *sat_id*, *user=None*, *password=None*)

Return a Pandas Series of every file for chosen remote data.

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **tag** (*string or NoneType*) – Denotes type of file to load. Accepted types are <tag strings>. (default=None)
- **sat_id** (*string or NoneType*) – Specifies the satellite ID for a constellation. Not used. (default=None)
- **user** (*string or NoneType*) – Username to be passed along to resource with relevant data. (default=None)
- **password** (*string or NoneType*) – User password to be passed along to resource with relevant data. (default=None)

Returns A Series formatted for the Files class (pysat._files.Files) containing filenames and indexed by date and time

Return type pandas.Series

pysat.instruments.templates.template_instrument.**download**(*date_array*, *tag*, *sat_id*, *data_path=None*, *user=None*, *password=None*, *custom_keywords=None*)

Placeholder for PLATFORM/NAME downloads.

This routine is invoked by pysat and is not intended for direct use by the end user.

Parameters

- **date_array** (*array-like*) – list of datetimes to download data for. The sequence of dates need not be contiguous.

- **tag** (*string*) – Tag identifier used for particular dataset. This input is provided by pysat. (default='')
- **sat_id** (*string*) – Satellite ID string identifier used for particular dataset. This input is provided by pysat. (default='')
- **data_path** (*string*) – Path to directory to download data to. (default=None)
- **user** (*string*) – User string input used for download. Provided by user and passed via pysat. If an account is required for dowloads this routine here must error if user not supplied. (default=None)
- **password** (*string*) – Password for data download. (default=None)
- **custom_keywords** (*placeholder*) – Additional keywords supplied by user when invoking the download routine attached to a pysat.Instrument object are passed to this routine. Use of custom keywords here is discouraged.

pysat.instruments.templates.template_instrument.**clean**(*inst*)

Routine to return PLATFORM/NAME data cleaned to the specified level

Cleaning level is specified in inst.clean_level and pysat will accept user input for several strings. The clean_level is specified at instantiation of the Instrument object.

'clean' : All parameters should be good, suitable for statistical and case studies 'dusty' : All paramers should generally be good though same may not be great 'dirty' : There are data areas that have issues, data should be used with caution 'none' : No cleaning applied, routine not called in this case.

Parameters **inst** (pysat.Instrument) – Instrument class object, whose attribute clean_level is used to return the desired level of data selectivity.

9.3.2 Madrigal Pandas

Generic module for loading netCDF4 files into the pandas format within pysat.

This file may be used as a template for adding pysat support for a new dataset based upon netCDF4 files, or other file types (with modification).

This routine may also be used to add quick local support for a netCDF4 based dataset without having to define an instrument module for pysat. Relevant parameters may be specified when instantiating this Instrument object to support the relevant file location and naming schemes. This presumes the pysat developed utils.load_netCDF4 routine is able to load the file. See the load routine docstring in this module for more.

The routines defined within may also be used when adding a new instrument to pysat by importing this module and using the functools.partial methods to attach these functions to the new instrument model. See pysat/instruments/cnofs_ivm.py for more. NASA CDAWeb datasets, such as C/NOFS IVM, use the methods within pysat/instruments/methods/nasa_cdaweb.py to make adding new CDAWeb instruments easy.

pysat.instruments.templates.netcdf_pandas.**init**(*self*)

Initializes the Instrument object with instrument specific values.

Runs once upon instantiation. This routine provides a convenient location to print Acknowledgements or restrictions from the mission.

pysat.instruments.templates.netcdf_pandas.**load**(*fnames*, *tag=None*, *sat_id=None*, ***kwargs*)

Loads data using pysat.utils.load_netcdf4 .

This routine is called as needed by pysat. It is not intended for direct user interaction.

Parameters

- **fnames** (*array-like*) – iterable of filename strings, full path, to data files to be loaded. This input is nominally provided by pysat itself.
- **tag** (*string*) – tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself.
- **sat_id** (*string*) – Satellite ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself.
- ****kwargs** (*extra keywords*) – Passthrough for additional keyword arguments specified when instantiating an Instrument object. These additional keywords are passed through to this routine by pysat.

Returns Data and Metadata are formatted for pysat. Data is a pandas DataFrame while metadata is a pysat.Meta instance.

Return type data, metadata

Note: Any additional keyword arguments passed to pysat.Instrument upon instantiation are passed along to this routine and through to the load_netcdf4 call.

Examples

```
inst = pysat.Instrument('sport', 'ivm')
inst.load(2019,1)

# create quick Instrument object for a new, random netCDF4 file
# define filename template string to identify files
# this is normally done by instrument code, but in this case
# there is no built in pysat instrument support
# presumes files are named default_2019-01-01.NC
format_str = 'default_{year:04d}-{month:02d}-{day:02d}.NC'
inst = pysat.Instrument('netcdf', 'pandas',
                        custom_kwarg='test'
                        data_path='./',
                        format_str=format_str)
inst.load(2019,1)
```

pysat.instruments.templates.netcdf_pandas.**list_files**(*tag=None*, *sat_id=None*, *data_path=None*, *format_str=None*)

Produce a list of files corresponding to format_str located at data_path.

This routine is invoked by pysat and is not intended for direct use by the end user.

Multiple data levels may be supported via the 'tag' and 'sat_id' input strings.

Parameters

- **tag** (*string*) – tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **sat_id** (*string*) – Satellite ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **data_path** (*string*) – Full path to directory containing files to be loaded. This is provided by pysat. The user may specify their own data path at Instrument instantiation and it will appear here. (default=None)

- **format_str** (*string*) – String template used to parse the datasets filenames. If a user supplies a template string at Instrument instantiation then it will appear here, otherwise defaults to None. (default=None)

Returns Series of filename strings, including the path, indexed by datetime.

Return type pandas.Series

Examples

```
If a filename is SPORT_L2_IVM_2019-01-01_v01r0000.NC then the template
is 'SPORT_L2_IVM_{year:04d}-{month:02d}-{day:02d}_' +
'v{version:02d}r{revision:04d}.NC'
```

Note: The returned Series should not have any duplicate datetimes. If there are multiple versions of a file the most recent version should be kept and the rest discarded. This routine uses the pysat.Files.from_os constructor, thus the returned files are up to pysat specifications.

Normally the format_str for each supported tag and sat_id is defined within this routine. However, as this is a generic routine, those definitions can't be made here. This method could be used in an instrument specific module where the list_files routine in the new package defines the format_str based upon inputs, then calls this routine passing both data_path and format_str.

Alternately, the list_files routine in methods.nasa_cdaweb may also be used and has more built in functionality. Supported tages and format strings may be defined within the new instrument module and passed as arguments to methods.nasa_cdaweb.list_files . For an example on using this routine, see pysat/instrument/cnofs_ivm.py or cnofs_vefi, cnofs_plp, omni_hro, timed_see, etc.

---

pysat.instruments.templates.netcdf_pandas.**download**(*date_array*, *tag*, *sat_id*, *data_path=None*, *user=None*, *password=None*)

Downloads data for supported instruments, however this is a template call.

This routine is invoked by pysat and is not intended for direct use by the end user.

Parameters

- **date_array** (*array-like*) – list of datetimes to download data for. The sequence of dates need not be contiguous.
- **tag** (*string*) – Tag identifier used for particular dataset. This input is provided by pysat. (default='')
- **sat_id** (*string*) – Satellite ID string identifier used for particular dataset. This input is provided by pysat. (default='')
- **data_path** (*string (None)*) – Path to directory to download data to. (default=None)
- **user** (*string*) – User string input used for download. Provided by user and passed via pysat. If an account is required for dowloads this routine here must error if user not supplied. (default=None)
- **password** (*string*) – Password for data download. (default=None)

9.3.3 NASA CDAWeb Instrument

This is a template for a pysat.Instrument support file that utilizes CDAWeb methods. Copy and modify this file as needed when adding a new Instrument to pysat.

This is a good area to introduce the instrument, provide background on the mission, operations, instrumenation, and measurements.

Also a good place to provide contact information. This text will be included in the pysat API documentation.

Properties

platform *List platform string here*

name *List name string here*

sat_id *List supported sat_ids here*

tag *List supported tag strings here*

Note:

- Optional section, remove if no notes

Warning:

- Optional section, remove if no warnings
- Two blank lines needed afterward for proper formatting

Examples

```
Example code can go here
```

Authors

Author name and institution

pysat.instruments.templates.template_cdaweb_instrument.**default**(*self*)

Default customization function.

This routine is automatically applied to the Instrument object on every load by the pysat nanokernel (first in queue).

Parameters **self** (pysat.Instrument) – This object

pysat.instruments.templates.template_cdaweb_instrument.**load**(*fnames*, *tag=None*, *sat_id=None*, *fake_daily_files_from_monthly=False*, *flatten_twod=True*)

Load NASA CDAWeb CDF files.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatNASA.instruments.methods.cdaweb*

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **fnames** (*(pandas.Series)*) – Series of filenames
- **tag** (*(str or NoneType)*) – tag or None (default=None)
- **sat_id** (*(str or NoneType)*) – satellite id or None (default=None)
- **fake_daily_files_from_monthly** (*bool*) – Some CDAWeb instrument data files are stored by month, interfering with pysat's functionality of loading by day. This flag, when true, parses of daily dates to monthly files that were added internally by the list_files routine, when flagged. These dates are used here to provide data by day.
- **flatted_twod** (*bool*) – Flattens 2D data into different columns of root DataFrame rather than produce a Series of DataFrames

Returns

- **data** (*(pandas.DataFrame)*) – Object containing satellite data
- **meta** (*(pysat.Meta)*) – Object containing metadata such as column names and units

Examples

```
# within the new instrument module, at the top level define
# a new variable named load, and set it equal to this load method
# code below taken from cnofs_ivm.py.

# support load routine
# use the default CDAWeb method
load = cdw.load
```

pysat.instruments.templates.template_cdaweb_instrument.**list_files**(*tag=None*, *sat_id=None*, *data_path=None*, *format_str=None*, ***, *supported_tags={'': {'': 'cnofs_vefi_bfield_1sec_{year:04d}{*
fake_daily_files_from_monthly=Fals
two_digit_year_break=None)

Return a Pandas Series of every file for chosen satellite data.

Deprecated since version 2.2.0: *list_files* will be removed in pysat 3.0.0, it will be replaced by the copy in instruments.methods.general

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **tag** (*(string or NoneType)*) – Denotes type of file to load. Accepted types are <tag strings>. (default=None)
- **sat_id** (*(string or NoneType)*) – Specifies the satellite ID for a constellation. Not used. (default=None)

- **data_path** (*(string or NoneType)*) – Path to data directory. If None is specified, the value previously set in Instrument.files.data_path is used. (default=None)
- **format_str** (*(string or NoneType)*) – User specified file format. If None is specified, the default formats associated with the supplied tags are used. (default=None)
- **supported_tags** (*(dict or NoneType)*) – keys are sat_id, each containing a dict keyed by tag where the values file format template strings. (default=None)
- **fake_daily_files_from_monthly** (*(bool)*) – Some CDAWeb instrument data files are stored by month, interfering with pysat's functionality of loading by day. This flag, when true, appends daily dates to monthly files internally. These dates are used by load routine in this module to provide data by day.
- **two_digit_year_break** (*(int)*) – If filenames only store two digits for the year, then '1900' will be added for years >= two_digit_year_break and '2000' will be added for years < two_digit_year_break.

Returns **pysat.Files.from_os** – A class containing the verified available files

Return type (pysat._files.Files)

Examples

```
fname = 'cnofs_vefi_bfield_1sec_{year:04d}{month:02d}{day:02d}_v05.cdf'
supported_tags = {'dc_b': fname}
list_files = functools.partial(nasa_cdaweb.list_files,
                               supported_tags=supported_tags)

fname = 'cnofs_cindi_ivm_500ms_{year:4d}{month:02d}{day:02d}_v01.cdf'
supported_tags = {'': fname}
list_files = functools.partial(cdw.list_files,
                               supported_tags=supported_tags)
```

pysat.instruments.templates.template_cdaweb_instrument.**list_remote_files**(*tag*, *sat_id*, *remote_site='https://cdawel*, ***, *supported_tags={'': {'': {'dir': '/pub/data/cnofs/vefi/bfield*, *'local_fname': 'cnofs_vefi_bfield_1sec_{y*, *'remote_fname': '{year:4d}/cnofs_vefi_bfie*, *user=None*, *password=None*, *fake_daily_files_from_mo*, *two_digit_year_break=No*, *delimiter=None*, *year=None*, *month=None*, *day=None*)

Return a Pandas Series of every file for chosen remote data.

Deprecated since version 2.3.0: This routine will be removed in pysat 3.0.0, it will be moved to the pysatNASA repository. Also, as of 2.2.0 the *year/month/day* keywords will be removed in pysat 3.0.0, they will be replaced with a start/stop syntax consistent with the download routine

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **tag** (*(string or NoneType)*) – Denotes type of file to load. Accepted types are <tag strings>. (default=None)
- **sat_id** (*(string or NoneType)*) – Specifies the satellite ID for a constellation. (default=None)
- **remote_site** (*(string or NoneType)*) – Remote site to download data from (default='https://cdaweb.gsfc.nasa.gov')
- **supported_tags** (*dict*) – dict of dicts. Keys are supported tag names for download. Value is a dict with 'dir', 'remote_fname', 'local_fname'. Inteded to be pre-set with functools.partial then assigned to new instrument code.
- **user** (*(string or NoneType)*) – Username to be passed along to resource with relevant data. (default=None)
- **password** (*(string or NoneType)*) – User password to be passed along to resource with relevant data. (default=None)
- **fake_daily_files_from_monthly** (*bool*) – Some CDAWeb instrument data files are stored by month. This flag, when true, accomodates this reality with user feedback on a monthly time frame. (default=False)

- **two_digit_year_break** (*(int or NoneType)*) – If filenames only store two digits for the year, then '1900' will be added for years >= two_digit_year_break and '2000' will be added for years < two_digit_year_break. (default=None)
- **delimiter** (*(string or NoneType)*) – If filename is delimited, then provide delimiter alone e.g. '_' (default=None)
- **year** (*(int or NoneType)*) – Selects a given year to return remote files for. None returns all years. (default=None)
- **month** (*(int or NoneType)*) – Selects a given month to return remote files for. None returns all months. Requires year to be defined. (default=None)
- **day** (*(int or NoneType)*) – Selects a given day to return remote files for. None returns all days. Requires year and month to be defined. (default=None)

Returns **pysat.Files.from_os** – A class containing the verified available files

Return type (pysat._files.Files)

Examples

```
fname = 'cnofs_vefi_bfield_1sec_{year:04d}{month:02d}{day:02d}_v05.cdf'
supported_tags = {'dc_b': fname}
list_remote_files =             functools.partial(nasa_cdaweb.list_remote_files,
                        supported_tags=supported_tags)

fname = 'cnofs_cindi_ivm_500ms_{year:4d}{month:02d}{day:02d}_v01.cdf'
supported_tags = {'': fname}
list_remote_files =             functools.partial(cdw.list_remote_files,
                        supported_tags=supported_tags)
```

pysat.instruments.templates.template_cdaweb_instrument.**download**(*date_array*, *tag*, *sat_id*, *remote_site='https://cdaweb.gsfc.nasa.go*, *data_path=None*, *user=None*, *password=None*, *fake_daily_files_from_monthly=False*, *multi_file_day=False*)

Routine to download NASA CDAWeb CDF data.

Deprecated since version 2.3.0: This routine has been deprecated in pysat 3.0.0, and will be accessible in *pysatNASA.instruments.methods.cdaweb*

This routine is intended to be used by pysat instrument modules supporting a particular NASA CDAWeb dataset.

Parameters

- **supported_tags** (*dict*) – dict of dicts. Keys are supported tag names for download. Value is a dict with 'dir', 'remote_fname', 'local_fname'. Inteded to be pre-set with functools.partial then assigned to new instrument code.
- **date_array** (*array_like*) – Array of datetimes to download data for. Provided by pysat.
- **tag** (*str or NoneType (None)*) – tag or None
- **sat_id** (*(str or NoneType)*) – satellite id or None (default=None)

- **remote_site** (*(string or NoneType)*) – Remote site to download data from (default='https://cdaweb.gsfc.nasa.gov')
- **data_path** (*(string or NoneType)*) – Path to data directory. If None is specified, the value previously set in Instrument.files.data_path is used. (default=None)
- **user** (*(string or NoneType)*) – Username to be passed along to resource with relevant data. (default=None)
- **password** (*(string or NoneType)*) – User password to be passed along to resource with relevant data. (default=None)
- **fake_daily_files_from_monthly** (*bool*) – Some CDAWeb instrument data files are stored by month. This flag, when true, accomodates this reality with user feedback on a monthly time frame.

Returns **Void** – Downloads data to disk.

Return type (NoneType)

Examples

```
# download support added to cnofs_vefi.py using code below
rn = '{year:4d}/cnofs_vefi_bfield_1sec_{year:4d}{month:02d}{day:02d}'+
    '_v05.cdf'
ln = 'cnofs_vefi_bfield_1sec_{year:4d}{month:02d}{day:02d}_v05.cdf'
dc_b_tag = {'dir':'/pub/data/cnofs/vefi/bfield_1sec',
            'remote_fname': rn,
            'local_fname': ln}
supported_tags = {'dc_b': dc_b_tag}

download = functools.partial(nasa_cdaweb.download,
                             supported_tags=supported_tags)
```

pysat.instruments.templates.template_cdaweb_instrument.**clean**(*inst*)

Routine to return PLATFORM/NAME data cleaned to the specified level

Cleaning level is specified in inst.clean_level and pysat will accept user input for several strings. The clean_level is specified at instantiation of the Instrument object.

'clean' : All parameters should be good, suitable for statistical and case studies 'dusty' : All paramers should generally be good though same may not be great 'dirty' : There are data areas that have issues, data should be used with caution 'none' : No cleaning applied, routine not called in this case.

Parameters **inst** (pysat.Instrument) – Instrument class object, whose attribute clean_level is used to return the desired level of data selectivity.

9.3.4 netCDF Pandas

Generic module for loading netCDF4 files into the pandas format within pysat.

This file may be used as a template for adding pysat support for a new dataset based upon netCDF4 files, or other file types (with modification).

This routine may also be used to add quick local support for a netCDF4 based dataset without having to define an instrument module for pysat. Relevant parameters may be specified when instantiating this Instrument object to support the relevant file location and naming schemes. This presumes the pysat developed utils.load_netCDF4 routine is able to load the file. See the load routine docstring in this module for more.

The routines defined within may also be used when adding a new instrument to pysat by importing this module and using the functools.partial methods to attach these functions to the new instrument model. See pysat/instruments/cnofs_ivm.py for more. NASA CDAWeb datasets, such as C/NOFS IVM, use the methods within pysat/instruments/methods/nasa_cdaweb.py to make adding new CDAWeb instruments easy.

pysat.instruments.templates.netcdf_pandas.**init**(*self*)

Initializes the Instrument object with instrument specific values.

Runs once upon instantiation. This routine provides a convenient location to print Acknowledgements or restrictions from the mission.

pysat.instruments.templates.netcdf_pandas.**load**(*fnames*, *tag=None*, *sat_id=None*, ***kwargs*)

Loads data using pysat.utils.load_netcdf4 .

This routine is called as needed by pysat. It is not intended for direct user interaction.

Parameters

- **fnames** (*array-like*) – iterable of filename strings, full path, to data files to be loaded. This input is nominally provided by pysat itself.
- **tag** (*string*) – tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself.
- **sat_id** (*string*) – Satellite ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself.
- ****kwargs** (*extra keywords*) – Passthrough for additional keyword arguments specified when instantiating an Instrument object. These additional keywords are passed through to this routine by pysat.

Returns Data and Metadata are formatted for pysat. Data is a pandas DataFrame while metadata is a pysat.Meta instance.

Return type data, metadata

Note: Any additional keyword arguments passed to pysat.Instrument upon instantiation are passed along to this routine and through to the load_netcdf4 call.

Examples

```
inst = pysat.Instrument('sport', 'ivm')
inst.load(2019,1)

# create quick Instrument object for a new, random netCDF4 file
# define filename template string to identify files
# this is normally done by instrument code, but in this case
# there is no built in pysat instrument support
# presumes files are named default_2019-01-01.NC
format_str = 'default_{year:04d}-{month:02d}-{day:02d}.NC'
inst = pysat.Instrument('netcdf', 'pandas',
                        custom_kwarg='test'
                        data_path='./',
                        format_str=format_str)
inst.load(2019,1)
```

pysat.instruments.templates.netcdf_pandas.**list_files**(*tag=None*, *sat_id=None*, *data_path=None*, *format_str=None*)

Produce a list of files corresponding to format_str located at data_path.

This routine is invoked by pysat and is not intended for direct use by the end user.

Multiple data levels may be supported via the 'tag' and 'sat_id' input strings.

Parameters

- **tag** (*string*) – tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **sat_id** (*string*) – Satellite ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **data_path** (*string*) – Full path to directory containing files to be loaded. This is provided by pysat. The user may specify their own data path at Instrument instantiation and it will appear here. (default=None)
- **format_str** (*string*) – String template used to parse the datasets filenames. If a user supplies a template string at Instrument instantiation then it will appear here, otherwise defaults to None. (default=None)

Returns Series of filename strings, including the path, indexed by datetime.

Return type pandas.Series

Examples

```
If a filename is SPORT_L2_IVM_2019-01-01_v01r0000.NC then the template
is 'SPORT_L2_IVM_{year:04d}-{month:02d}-{day:02d}_' +
'v{version:02d}r{revision:04d}.NC'
```

Note: The returned Series should not have any duplicate datetimes. If there are multiple versions of a file the most recent version should be kept and the rest discarded. This routine uses the pysat.Files.from_os constructor, thus the returned files are up to pysat specifications.

Normally the format_str for each supported tag and sat_id is defined within this routine. However, as this is a generic routine, those definitions can't be made here. This method could be used in an instrument specific module where the list_files routine in the new package defines the format_str based upon inputs, then calls this routine passing both data_path and format_str.

Alternately, the list_files routine in methods.nasa_cdaweb may also be used and has more built in functionality. Supported tages and format strings may be defined within the new instrument module and passed as arguments to methods.nasa_cdaweb.list_files . For an example on using this routine, see pysat/instrument/cnofs_ivm.py or cnofs_vefi, cnofs_plp, omni_hro, timed_see, etc.

---

pysat.instruments.templates.netcdf_pandas.**download**(*date_array*, *tag*, *sat_id*, *data_path=None*, *user=None*, *password=None*)

Downloads data for supported instruments, however this is a template call.

This routine is invoked by pysat and is not intended for direct use by the end user.

Parameters

- **date_array** (*array-like*) – list of datetimes to download data for. The sequence of dates need not be contiguous.
- **tag** (*string*) – Tag identifier used for particular dataset. This input is provided by pysat. (default='')
- **sat_id** (*string*) – Satellite ID string identifier used for particular dataset. This input is provided by pysat. (default='')
- **data_path** (*string (None)*) – Path to directory to download data to. (default=None)
- **user** (*string*) – User string input used for download. Provided by user and passed via pysat. If an account is required for dowloads this routine here must error if user not supplied. (default=None)
- **password** (*string*) – Password for data download. (default=None)

9.4 Constellation

class pysat.**Constellation**(*instruments=None*, *name=None*, *const_module=None*)

Manage and analyze data from multiple pysat Instruments.

Created as part of a Spring 2018 UTDesign project.

Deprecated since version 2.3.0: The *name* kwarg was changed to *const_module* in pysat 3.0.0

Constructs a Constellation given a list of instruments or the name of a file with a pre-defined constellation.

Deprecated since version 2.3.0: The *name* kwarg was changed to *const_module* in pysat 3.0.0

Parameters

- **instruments** (*list*) – a list of pysat Instruments
- **name** (*string*) – Name of a file in pysat/constellations containing a list of instruments.
- **const_module** (*string or NoneType*) – Name of a pysat constellation module (default=None)

Note: The name and instruments parameters should not both be set. If neither is given, an empty constellation will be created.

---

add(*bounds1*, *label1*, *bounds2*, *label2*, *bin3*, *label3*, *data_label*)

Combines signals from multiple instruments within given bounds.

Deprecated since version 2.2.0: *add* will be removed in pysat 3.0.0, it will be added to pysatSeasons

Parameters

- **bounds1** (*(min, max)*) – Bounds for selecting data on the axis of label1 Data points with label1 in [min, max) will be considered.
- **label1** (*string*) – Data label for bounds1 to act on.
- **bounds2** (*(min, max)*) – Bounds for selecting data on the axis of label2 Data points with label1 in [min, max) will be considered.
- **label2** (*string*) – Data label for bounds2 to act on.
- **bin3** (*(min, max, #bins)*) – Min and max bounds and number of bins for third axis.

- **label3** (*string*) – Data label for third axis.
- **data_label** (*array of strings*) – Data label(s) for data product(s) to be averaged.

Returns **median** – Dictionary indexed by data label, each value of which is a dictionary with keys 'median', 'count', 'avg_abs_dev', and 'bin' (the values of the bin edges.)

Return type dictionary

data_mod(**args*, ***kwargs*)

Register a function to modify data of member Instruments.

The function is not partially applied to modify member data.

When the Constellation receives a function call to register a function for data modification, it passes the call to each instrument and registers it in the instrument's pysat.Custom queue.

(Wraps pysat.Custom.add; documentation of that function is reproduced here.)

Parameters

- **function** (*string or function object*) – name of function or function object to be added to queue
- **kind** (*{'add, 'modify', 'pass'}*) –

  add Adds data returned from fuction to instrument object.

  modify pysat instrument object supplied to routine. Any and all changes to object are retained.

  pass A copy of pysat object is passed to function. No data is accepted from return.
- **at_pos** (*string or int*) – insert at position. (default, insert at end).
- **args** (*extra arguments*) –

Note: Allowed *add* function returns:

- {'data' : pandas Series/DataFrame/array_like, 'units' : string/array_like of strings, 'long_name' : string/array_like of strings, 'name' : string/array_like of strings (iff data array_like)}
- pandas DataFrame, names of columns are used
- pandas Series, .name required
- (string/list of strings, numpy array/list of arrays)

---

difference(*instrument1*, *instrument2*, *bounds*, *data_labels*, *cost_function*)

Calculates the difference in signals from multiple instruments within the given bounds.

Deprecated since version 2.2.0: *difference* will be removed in pysat 3.0.0, it will be added to pysatSeasons

Parameters

- **instrument1** (Instrument) – Information must already be loaded into the instrument.
- **instrument2** (Instrument) – Information must already be loaded into the instrument.
- **bounds** (*list of tuples in the form (inst1_label, inst2_label,)*) – min, max, max_difference) inst1_label are inst2_label are labels for the data in instrument1 and instrument2 min and max are bounds on the data

considered max_difference is the maximum difference between two points for the difference to be calculated

- **data_labels** (*list of tuples of data labels*) – The first key is used to access data in s1 and the second data in s2.
- **cost_function** (*function*) – function that operates on two rows of the instrument data. used to determine the distance between two points for finding closest points

Returns

- **data_df** (*pandas DataFrame*) – Each row has a point from instrument1, with the keys preceded by *1_*, and a point within bounds on that point from instrument2 with the keys preceded by *2_*, and the difference between the instruments' data for all the labels in data_labels
- *Created as part of a Spring 2018 UTDesign project.*

load(**args*, ***kwargs*)

Load instrument data into instrument object.data

(Wraps pysat.Instrument.load; documentation of that function is reproduced here.)

Parameters

- **yr** (*integer*) – Year for desired data
- **doy** (*integer*) – day of year
- **data** (*datetime object*) – date to load
- **fname** (*'string'*) – filename to be loaded
- **verifyPad** (*boolean*) – if true, padding data not removed (debug purposes)

set_bounds(*start*, *stop*)

Sets boundaries for all instruments in constellation

9.5 Custom

class pysat.**Custom**

Applies a queue of functions when instrument.load called.

Deprecated since version 2.3.0: *Custom* will be removed in pysat 3.0.0, it is incorporated into *Instrument*

Nano-kernel functionality enables instrument objects that are 'set and forget'. The functions are always run whenever the instrument load routine is called so instrument objects may be passed safely to other routines and the data will always be processed appropriately.

Examples

```
def custom_func(inst, opt_param1=False, opt_param2=False):
    return None
instrument.custom.attach(custom_func, 'modify', opt_param1=True)

def custom_func2(inst, opt_param1=False, opt_param2=False):
    return data_to_be_added
instrument.custom.attach(custom_func2, 'add', opt_param2=True)
instrument.load(date=date)
print(instrument['data_to_be_added'])
```

See also:

Custom.attach

Note: User should interact with Custom through pysat.Instrument instance's attribute, instrument.custom

---

add (*function*, *kind='add'*, *at_pos='end'*, **args*, ***kwargs*)
Add a function to custom processing queue.

Deprecated since version 2.2.0: *Custom.add* will be removed in pysat 3.0.0, it is replaced by *Instrument.custom_attach* to clarify the syntax

Custom functions are applied automatically to associated pysat instrument whenever instrument.load command called.

Parameters

- **function** (*string or function object*) – name of function or function object to be added to queue
- **kind** (*{'add', 'modify', 'pass}*) –

  add Adds data returned from function to instrument object. A copy of pysat instrument object supplied to routine.

  modify pysat instrument object supplied to routine. Any and all changes to object are retained.

  pass A copy of pysat object is passed to function. No data is accepted from return.
- **at_pos** (*string or int*) – insert at position. (default, insert at end).
- **args** (*extra arguments*) – extra arguments are passed to the custom function (once)
- **kwargs** (*extra keyword arguments*) – extra keyword args are passed to the custom function (once)

Note: Allowed *add* function returns:

- {'data' : pandas Series/DataFrame/array_like, 'units' : string/array_like of strings, 'long_name' : string/array_like of strings, 'name' : string/array_like of strings (iff data array_like)}
- pandas DataFrame, names of columns are used
- pandas Series, .name required
- (string/list of strings, numpy array/list of arrays)

---

attach (*function*, *kind='add'*, *at_pos='end'*, **args*, ***kwargs*)
Attach a function to custom processing queue.

Deprecated since version 2.3.0: *Custom.attach* will be removed in pysat 3.0.0, it is replaced by *Instrument.custom_attach*

Custom functions are applied automatically to associated pysat instrument whenever instrument.load command called.

Parameters

- **function** (*string or function object*) – name of function or function object to be added to queue

- **kind** (*{'add', 'modify', 'pass}*) –

  add Adds data returned from function to instrument object. A copy of pysat instrument object supplied to routine.

  modify pysat instrument object supplied to routine. Any and all changes to object are retained.

  pass A copy of pysat object is passed to function. No data is accepted from return.

- **at_pos** (*string or int*) – insert at position. (default, insert at end).
- **args** (*extra arguments*) – extra arguments are passed to the custom function (once)
- **kwargs** (*extra keyword arguments*) – extra keyword args are passed to the custom function (once)

Note: Allowed *attach* function returns:

- {'data' : pandas Series/DataFrame/array_like, 'units' : string/array_like of strings, 'long_name' : string/array_like of strings, 'name' : string/array_like of strings (iff data array_like)}
- pandas DataFrame, names of columns are used
- pandas Series, .name required
- (string/list of strings, numpy array/list of arrays)

---

clear()
Clear custom function list.

Deprecated since version 2.3.0: *Custom.clear* will be removed in pysat 3.0.0, it is replaced by *Instrument.custom_clear*

9.6 Files

class pysat.**Files**(*sat*, *manual_org=False*, *directory_format=None*, *update_files=False*, *file_format=None*, *write_to_disk=True*, *ignore_empty_files=False*)
Maintains collection of files for instrument object.

Uses the list_files functions for each specific instrument to create an ordered collection of files in time. Used by instrument object to load the correct files. Files also contains helper methods for determining the presence of new files and creating an ordered list of files.

base_path
path to .pysat directory in user home

Type string

start_date
date of first file, used as default start bound for instrument object

Type datetime

stop_date
date of last file, used as default stop bound for instrument object

Type datetime

data_path
path to the directory containing instrument files, top_dir/platform/name/tag/

Type string

manual_org
if True, then Files will look directly in pysat data directory for data files and will not use /platform/name/tag

Type bool

update_files
updates files on instantiation if True

Type bool

Note: User should generally use the interface provided by a pysat.Instrument instance. Exceptions are the classmethod from_os, provided to assist in generating the appropriate output for an instrument routine.

Examples

```
# convenient file access
inst = pysat.Instrument(platform=platform, name=name, tag=tag,
                        sat_id=sat_id)
# first file
inst.files[0]

# files from start up to stop (exclusive on stop)
start = pysat.datetime(2009,1,1)
stop = pysat.datetime(2009,1,3)
print(vefi.files[start:stop])

# files for date
print(vefi.files[start])

# files by slicing
print(vefi.files[0:4])

# get a list of new files
# new files are those that weren't present the last time
# a given instrument's file list was stored
new_files = vefi.files.get_new()

# search pysat appropriate directory for instrument files and
# update Files instance.
vefi.files.refresh()
```

Initialization for Files class object

Parameters

- **sat** (*pysat._instrument.Instrument*) – Instrument object
- **manual_org** (*boolian*) – If True, then pysat will look directly in pysat data directory for data files and will not use default /platform/name/tag (default=False)
- **directory_format** (*string or NoneType*) – directory naming structure in string format. Variables such as platform, name, and tag will be filled in as needed using python string formatting. The default directory structure would be expressed as '{platform}/{name}/{tag}' (default=None)

- **update_files** (*boolean*) – If True, immediately query filesystem for instrument files and store (default=False)
- **file_format** (*str or NoneType*) – File naming structure in string format. Variables such as year, month, and sat_id will be filled in as needed using python string formatting. The default file format structure is supplied in the instrument list_files routine. (default=None)
- **write_to_disk** (*boolean*) – If true, the list of Instrument files will be written to disk. Setting this to False prevents a rare condition when running multiple pysat processes.
- **ignore_empty_files** (*boolean*) – if True, the list of files found will be checked to ensure the filesiizes are greater than zero. Empty files are removed from the stored list of files.

classmethod from_os(*data_path=None*, *format_str=None*, *two_digit_year_break=None*, *delimiter=None*)

Produces a list of files and and formats it for Files class.

Requires fixed_width or delimited filename

Parameters

- **data_path** (*string*) – Top level directory to search files for. This directory is provided by pysat to the instrument_module.list_files functions as data_path.
- **format_str** (*string with python format codes*) – Provides the naming pattern of the instrument files and the locations of date information so an ordered list may be produced. Supports 'year', 'month', 'day', 'hour', 'minute', 'second', 'version', and 'revision' Ex: 'cnofs_cindi_ivm_500ms_{year:4d}{month:02d}{day:02d}_v01.cdf'
- **two_digit_year_break** (*int*) – If filenames only store two digits for the year, then '1900' will be added for years >= two_digit_year_break and '2000' will be added for years < two_digit_year_break.
- **delimiter** (*string (None)*) – If set, then filename will be processed using delimiter rather than assuming a fixed width

---

Note: Does not produce a Files instance, but the proper output from instrument_module.list_files method.

The '?' may be used to indicate a set number of spaces for a variable part of the name that need not be extracted. 'cnofs_cindi_ivm_500ms_{year:4d}{month:02d}{day:02d}_v??.cdf'

---

get_file_array(*start*, *end*)

Return a list of filenames between and including start and end.

Parameters

- **start** (*array_like or single string*) – filenames for start of returned filelist
- **stop** (*array_like or single string*) – filenames inclusive end of list

Returns

- *list of filenames between and including start and end over all*
- *intervals.*

get_index(*fname*)

Return index for a given filename.

Parameters **fname** (*string*) – filename

Note: If fname not found in the file information already attached to the instrument.files instance, then a files.refresh() call is made.

get_new()
List new files since last recorded file state.

pysat stores filenames in the user_home/.pysat directory. Returns a list of all new fileanmes since the last known change to files. Filenames are stored if there is a change and either update_files is True at instrument object level or files.refresh() is called.

Returns files are indexed by datetime

Return type pandas.Series

refresh()
Update list of files, if there are changes.

Calls underlying list_rtn for the particular science instrument. Typically, these routines search in the pysat provided path, pysat_data_dir/platform/name/tag/, where pysat_data_dir is set by pysat.utils.set_data_dir(path=path).

9.7 Meta

class pysat.**Meta**(*metadata=None*, *units_label='units'*, *name_label='long_name'*, *notes_label='notes'*, *desc_label='desc'*, *plot_label='label'*, *axis_label='axis'*, *scale_label='scale'*, *min_label='value_min'*, *max_label='value_max'*, *fill_label='fill'*, *export_nan=[]*)
Stores metadata for Instrument instance, similar to CF-1.6 netCDFdata standard.

Parameters

- **metadata** (*pandas.DataFrame*) – DataFrame should be indexed by variable name that contains at minimum the standard_name (name), units, and long_name for the data stored in the associated pysat Instrument object.
- **units_label** (*str*) – String used to label units in storage. Defaults to 'units'.
- **name_label** (*str*) – String used to label long_name in storage. Defaults to 'long_name'.
- **notes_label** (*str*) – String used to label 'notes' in storage. Defaults to 'notes'
- **desc_label** (*str*) – String used to label variable descriptions in storage. Defaults to 'desc'
- **plot_label** (*str*) – String used to label variables in plots. Defaults to 'label'
- **axis_label** (*str*) – Label used for axis on a plot. Defaults to 'axis'
- **scale_label** (*str*) – string used to label plot scaling type in storage. Defaults to 'scale'
- **min_label** (*str*) – String used to label typical variable value min limit in storage. Defaults to 'value_min'
- **max_label** (*str*) – String used to label typical variable value max limit in storage. Defaults to 'value_max'
- **fill_label** (*str*) – String used to label fill value in storage. Defaults to 'fill' per netCDF4 standard

data
index is variable standard name, 'units', 'long_name', and other defaults are also stored along with additional user provided labels.

Type pandas.DataFrame

units_label
String used to label units in storage. Defaults to 'units'.

Type str

name_label
String used to label long_name in storage. Defaults to 'long_name'.

Type str

notes_label
String used to label 'notes' in storage. Defaults to 'notes'

Type str

desc_label
String used to label variable descriptions in storage. Defaults to 'desc'

Type str

plot_label
String used to label variables in plots. Defaults to 'label'

Type str

axis_label
Label used for axis on a plot. Defaults to 'axis'

Type str

scale_label
string used to label plot scaling type in storage. Defaults to 'scale'

Type str

min_label
String used to label typical variable value min limit in storage. Defaults to 'value_min'

Type str

max_label
String used to label typical variable value max limit in storage. Defaults to 'value_max'

Type str

fill_label
String used to label fill value in storage. Defaults to 'fill' per netCDF4 standard

Type str

export_nan
List of labels that should be exported even if their value is nan. By default, metadata with a value of nan will be exluded from export.

Type list

Notes

Meta object preserves the case of variables and attributes as it first receives the data. Subsequent calls to set new metadata with the same variable or attribute will use case of first call. Accessing or setting data thereafter is case insensitive. In practice, use is case insensitive but the original case is preserved. Case preseveration is built in to support writing files with a desired case to meet standards.

Metadata for higher order data objects, those that have multiple products under a single variable name in a pysat.Instrument object, are stored by providing a Meta object under the single name.

Supports any custom metadata values in addition to the expected metadata attributes (units, name, notes, desc, plot_label, axis, scale, value_min, value_max, and fill). These base attributes may be used to programatically access and set types of metadata regardless of the string values used for the attribute. String values for attributes may need to be changed depending upon the standards of code or files interacting with pysat.

Meta objects returned as part of pysat loading routines are automatically updated to use the same values of plot_label, units_label, etc. as found on the pysat.Instrument object.

Examples

```
# instantiate Meta object, default values for attribute labels are used
meta = pysat.Meta()
# set a couple base units
# note that other base parameters not set below will
# be assigned a default value
meta['name'] = {'long_name':string, 'units':string}
# update 'units' to new value
meta['name'] = {'units':string}
# update 'long_name' to new value
meta['name'] = {'long_name':string}
# attach new info with partial information, 'long_name' set to 'name2'
meta['name2'] = {'units':string}
# units are set to '' by default
meta['name3'] = {'long_name':string}

# assigning custom meta parameters
meta['name4'] = {'units':string, 'long_name':string
                 'custom1':string, 'custom2':value}
meta['name5'] = {'custom1':string, 'custom3':value}

# assign multiple variables at once
meta[['name1', 'name2']] = {'long_name':[string1, string2],
                            'units':[string1, string2],
                            'custom10':[string1, string2]}

# assiging metadata for n-Dimensional variables
meta2 = pysat.Meta()
meta2['name41'] = {'long_name':string, 'units':string}
meta2['name42'] = {'long_name':string, 'units':string}
meta['name4'] = {'meta':meta2}
# or
meta['name4'] = meta2
meta['name4'].children['name41']

# mixture of 1D and higher dimensional data
meta = pysat.Meta()
meta['dm'] = {'units':'hey', 'long_name':'boo'}
```

(continues on next page)

(continued from previous page)

```
meta['rpa'] = {'units':'crazy', 'long_name':'boo_whoo'}
meta2 = pysat.Meta()
meta2[['higher', 'lower']] = {'meta':[meta, None],
                              'units':[None, 'boo'],
                              'long_name':[None, 'boohoo']}

# assign from another Meta object
meta[key1] = meta2[key2]

# access fill info for a variable, presuming default label
meta[key1, 'fill']
# access same info, even if 'fill' not used to label fill values
meta[key1, meta.fill_label]

# change a label used by Meta object
# note that all instances of fill_label
# within the meta object are updated
meta.fill_label = '_FillValue'
meta.plot_label = 'Special Plot Variable'
# this feature is useful when converting metadata within pysat
# so that it is consistent with externally imposed file standards
```

accept_default_labels(*other*)
Applies labels for default meta labels from other onto self.

Parameters **other** (Meta) – Meta object to take default labels from

Returns

Return type *Meta*

apply_default_labels(*other*)
Applies labels for default meta labels from self onto other.

Parameters **other** (Meta) – Meta object to have default labels applied

Returns

Return type *Meta*

attr_case_name(*name*)
Returns preserved case name for case insensitive value of name.

Checks first within standard attributes. If not found there, checks attributes for higher order data structures. If not found, returns supplied name as it is available for use. Intended to be used to help ensure that the same case is applied to all repetitions of a given variable name.

Parameters **name** (*str*) – name of variable to get stored case form

Returns name in proper case

Return type str

attrs()
Yields metadata products stored for each variable name

concat(*other*, *strict=False*)
Concats two metadata objects together.

Parameters

- **other** (Meta) – Meta object to be concatenated

- **strict** (*bool*) – if True, ensure there are no duplicate variable names

Notes

Uses units and name label of self if other is different

Returns Concatenated object

Return type *Meta*

drop(*names*)
Drops variables (names) from metadata.

empty
Return boolean True if there is no metadata

classmethod from_csv(*name=None*, *col_names=None*, *sep=None*, ***kwargs*)
Create instrument metadata object from csv.

Parameters

- **name** (*string*) – absolute filename for csv file or name of file stored in pandas instruments location
- **col_names** (*list-like collection of strings*) – column names in csv and resultant meta object
- **sep** (*string*) – column seperator for supplied csv filename

---

Note: column names must include at least ['name', 'long_name', 'units'], assumed if col_names is None.

---

has_attr(*name*)
Returns boolean indicating presence of given attribute name

Case-insensitive check

Notes

Does not check higher order meta objects

Parameters **name** (*str*) – name of variable to get stored case form

Returns True if case-insesitive check for attribute name is True

Return type bool

keep(*keep_names*)
Keeps variables (keep_names) while dropping other parameters

Parameters **keep_names** (*list-like*) – variables to keep

keys()
Yields variable names stored for 1D variables

keys_nD()
Yields keys for higher order metadata

merge(*other*)
Adds metadata variables to self that are in other but not in self.

Parameters **other** (pysat.Meta) –

pop(*name*)
Remove and return metadata about variable

Parameters **name** (*str*) – variable name

Returns Series of metadata for variable

Return type pandas.Series

transfer_attributes_to_instrument(*inst*, *strict_names=False*)
Transfer non-standard attributes in Meta to Instrument object.

Pysat's load_netCDF and similar routines are only able to attach netCDF4 attributes to a Meta object. This routine identifies these attributes and removes them from the Meta object. Intent is to support simple transfers to the pysat.Instrument object.

Will not transfer names that conflict with pysat default attributes.

Parameters

- **inst** (pysat.Instrument) – Instrument object to transfer attributes to
- **strict_names** (*boolean (False)*) – If True, produces an error if the Instrument object already has an attribute with the same name to be copied.

Returns pysat.Instrument object modified in place with new attributes

Return type None

var_case_name(*name*)
Provides stored name (case preserved) for case insensitive input

If name is not found (case-insensitive check) then name is returned, as input. This function is intended to be used to help ensure the case of a given variable name is the same across the Meta object.

Parameters **name** (*str*) – variable name in any case

Returns string with case preserved as in metaobject

Return type str

9.8 Orbits

class pysat.**Orbits**(*sat=None*, *index=None*, *kind=None*, *period=None*)
Determines orbits on the fly and provides orbital data in .data.

Determines the locations of orbit breaks in the loaded data in inst.data and provides iteration tools and convenient orbit selection via inst.orbit[orbit num].

Parameters

- **sat** (*pysat.Instrument instance*) – instrument object to determine orbits for
- **index** (*string*) – name of the data series to use for determing orbit breaks
- **kind** (*{'local time', 'longitude', 'polar', 'orbit'}*) – kind of orbit, determines how orbital breaks are determined
  - local time: negative gradients in lt or breaks in inst.data.index
  - longitude: negative gradients or breaks in inst.data.index
  - polar: zero crossings in latitude or breaks in inst.data.index
  - orbit: uses unique values of orbit number

- **period** (*np.timedelta64*) – length of time for orbital period, used to gauge when a break in the datetime index (inst.data.index) is large enough to consider it a new orbit

Note: class should not be called directly by the user, use the interface provided by inst.orbits where inst = pysat.Instrument()

Warning: This class is still under development.

Examples

```
info = {'index':'longitude', 'kind':'longitude'}
vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b',
                        clean_level=None, orbit_info=info)
start = pysat.datetime(2009,1,1)
stop = pysat.datetime(2009,1,10)
vefi.load(date=start)
vefi.bounds(start, stop)

# iterate over orbits
for vefi in vefi.orbits:
    print('Next available orbit ', vefi['dB_mer'])

# load fifth orbit of first day
vefi.load(date=start)
vefi.orbits[5]

# less convenient load
vefi.orbits.load(5)

# manually iterate orbit
vefi.orbits.next()
# backwards
vefi.orbits.prev()
```

current
Current orbit number.

Returns None if no orbit data. Otherwise, returns orbit number, begining with zero. The first and last orbit of a day is somewhat ambiguous. The first orbit for day n is generally also the last orbit on day n - 1. When iterating forward, the orbit will be labeled as first (0). When iterating backward, orbit labeled as the last.

Return type int or None

load(*orbit=None*)
Load a particular orbit into .data for loaded day.

Parameters **orbit** (*int*) – orbit number, 1 indexed

Note: A day of data must be loaded before this routine functions properly. If the last orbit of the day is requested, it will automatically be padded with data from the next day. The orbit counter will be reset to 1.

next(**arg*, ***kwarg*)
Load the next orbit into .data.

Note: Forms complete orbits across day boundaries. If no data loaded then the first orbit from the first date of data is returned.

---

prev(**arg*, ***kwarg*)
Load the previous orbit into .data.

Note: Forms complete orbits across day boundaries. If no data loaded then the last orbit of data from the last day is loaded into .data.

9.9 Seasonal Analysis

9.9.1 Occurrence Probability

Occurrence probability routines, daily or by orbit.

Routines calculate the occurrence of an event greater than a supplied gate occuring at least once per day, or once per orbit. The probability is calculated as the (number of times with at least one hit in bin)/(number of times in the bin).The data used to determine the occurrence must be 1D. If a property of a 2D or higher dataset is needed attach a custom function that performs the check and returns a 1D Series.

Deprecated since version 2.2.0: *ssnl.occur_prob* will be removed in pysat 3.0.0, it will be added to pysatSeasons: https://github.com/pysat/pysatSeasons

---

Note: The included routines use the bounds attached to the supplied instrument object as the season of interest.

---

pysat.ssnl.occur_prob.**by_orbit2D**(*inst*, *bin1*, *label1*, *bin2*, *label2*, *data_label*, *gate*, *returnBins=False*)
2D Occurrence Probability of data_label orbit-by-orbit over a season.

Deprecated since version 2.2.0: *by_orbit2D* will be removed in pysat 3.0.0, it will be added to pysatSeasons

If data_label is greater than gate atleast once per orbit, then a 100% occurrence probability results. Season delineated by the bounds attached to Instrument object. Prob = (# of times with at least one hit)/(# of times in bin)

Parameters

- **inst** (*pysat.Instrument()*) – Instrument to use for calculating occurrence probability
- **binx** (*list*) – [min value, max value, number of bins]
- **labelx** (*string*) – identifies data product for binx
- **data_label** (*list of strings*) – identifies data product(s) to calculate occurrence probability
- **gate** (*list of values*) – values that data_label must achieve to be counted as an occurrence

- **returnBins** (*Boolean*) – if True, return arrays with values of bin edges, useful for pcolor

Returns **occur_prob** – A dict of dicts indexed by data_label. Each entry is dict with entries 'prob' for the probability and 'count' for the number of orbits with any data; 'bin_x' and 'bin_y' are also returned if requested. Note that arrays are organized for direct plotting, y values along rows, x along columns.

Return type dictionary

Note: Season delineated by the bounds attached to Instrument object.

---

pysat.ssnl.occur_prob.**by_orbit3D**(*inst*, *bin1*, *label1*, *bin2*, *label2*, *bin3*, *label3*, *data_label*, *gate*, *returnBins=False*)

3D Occurrence Probability of data_label orbit-by-orbit over a season.

Deprecated since version 2.2.0: *by_orbit3D* will be removed in pysat 3.0.0, it will be added to pysatSeasons

If data_label is greater than gate atleast once per orbit, then a 100% occurrence probability results. Season delineated by the bounds attached to Instrument object. Prob = (# of times with at least one hit)/(# of times in bin)

Parameters

- **inst** (pysat.Instrument *()*) – Instrument to use for calculating occurrence probability
- **binx** (*list*) – [min value, max value, number of bins]
- **labelx** (*string*) – identifies data product for binx
- **data_label** (*list of strings*) – identifies data product(s) to calculate occurrence probability
- **gate** (*list of values*) – values that data_label must achieve to be counted as an occurrence
- **returnBins** (*Boolean*) – if True, return arrays with values of bin edges, useful for pcolor

Returns **occur_prob** – A dict of dicts indexed by data_label. Each entry is dict with entries 'prob' for the probability and 'count' for the number of orbits with any data; 'bin_x', 'bin_y', and 'bin_z' are also returned if requested. Note that arrays are organized for direct plotting, z,y,x.

Return type dictionary

Note: Season delineated by the bounds attached to Instrument object.

---

pysat.ssnl.occur_prob.**daily2D**(*inst*, *bin1*, *label1*, *bin2*, *label2*, *data_label*, *gate*, *returnBins=False*)

2D Daily Occurrence Probability of data_label > gate over a season.

Deprecated since version 2.2.0: *daily2D* will be removed in pysat 3.0.0, it will be added to pysatSeasons

If data_label is greater than gate at least once per day, then a 100% occurrence probability results.Season delineated by the bounds attached to Instrument object. Prob = (# of times with at least one hit)/(# of times in bin)

Parameters

- **inst** (*pysat.Instrument()*) – Instrument to use for calculating occurrence probability
- **binx** (*list*) – [min, max, number of bins]
- **labelx** (*string*) – name for data product for binx
- **data_label** (*list of strings*) – identifies data product(s) to calculate occurrence probability e.g. inst[data_label]
- **gate** (*list of values*) – values that data_label must achieve to be counted as an occurrence
- **returnBins** (*Boolean*) – if True, return arrays with values of bin edges, useful for pcolor

Returns **occur_prob** – A dict of dicts indexed by data_label. Each entry is dict with entries 'prob' for the probability and 'count' for the number of days with any data; 'bin_x' and 'bin_y' are also returned if requested. Note that arrays are organized for direct plotting, y values along rows, x along columns.

Return type dictionary

Note: Season delineated by the bounds attached to Instrument object.

---

pysat.ssnl.occur_prob.**daily3D**(*inst*, *bin1*, *label1*, *bin2*, *label2*, *bin3*, *label3*, *data_label*, *gate*, *returnBins=False*)

3D Daily Occurrence Probability of data_label > gate over a season.

Deprecated since version 2.2.0: *daily3D* will be removed in pysat 3.0.0, it will be added to pysatSeasons

If data_label is greater than gate atleast once per day, then a 100% occurrence probability results. Season delineated by the bounds attached to Instrument object. Prob = (# of times with at least one hit)/(# of times in bin)

Parameters

- **inst** (*pysat.Instrument()*) – Instrument to use for calculating occurrence probability
- **binx** (*list*) – [min, max, number of bins]
- **labelx** (*string*) – name for data product for binx
- **data_label** (*list of strings*) – identifies data product(s) to calculate occurrence probability
- **gate** (*list of values*) – values that data_label must achieve to be counted as an occurrence
- **returnBins** (*Boolean*) – if True, return arrays with values of bin edges, useful for pcolor

Returns **occur_prob** – A dict of dicts indexed by data_label. Each entry is dict with entries 'prob' for the probability and 'count' for the number of days with any data; 'bin_x', 'bin_y', and 'bin_z' are also returned if requested. Note that arrays are organized for direct plotting, z,y,x.

Return type dictionary

Note: Season delineated by the bounds attached to Instrument object.

9.9.2 Average

Instrument independent seasonal averaging routine. Supports averaging 1D and 2D data.

Deprecated since version 2.2.0: *ssnl.avg* will be removed in pysat 3.0.0, it will be added to pysatSeasons: https://github.com/pysat/pysatSeasons

pysat.ssnl.avg.**mean_by_day**(*inst*, *data_label*)
Mean of data_label by day over Instrument.bounds

Deprecated since version 2.2.0: *mean_by_day* will be removed in pysat 3.0.0, it will be added to pysatSeasons

Parameters **data_label** (*string*) – string identifying data product to be averaged

Returns **mean** – simple mean of data_label indexed by day

Return type pandas Series

pysat.ssnl.avg.**mean_by_file**(*inst*, *data_label*)
Mean of data_label by orbit over Instrument.bounds

Deprecated since version 2.2.0: *mean_by_file* will be removed in pysat 3.0.0, it will be added to pysatSeasons

Parameters **data_label** (*string*) – string identifying data product to be averaged

Returns **mean** – simple mean of data_label indexed by start of each file

Return type pandas Series

pysat.ssnl.avg.**mean_by_orbit**(*inst*, *data_label*)
Mean of data_label by orbit over Instrument.bounds

Deprecated since version 2.2.0: *mean_by_orbit* will be removed in pysat 3.0.0, it will be added to pysatSeasons

Parameters **data_label** (*string*) – string identifying data product to be averaged

Returns **mean** – simple mean of data_label indexed by start of each orbit

Return type pandas Series

pysat.ssnl.avg.**median1D**(*const*, *bin1*, *label1*, *data_label*, *auto_bin=True*, *returnData=False*)
Return a 1D median of data_label over a season and label1

Deprecated since version 2.2.0: *median1D* will be removed in pysat 3.0.0, it will be added to pysatSeasons

Parameters

- **const** (*Constellation or Instrument*) – Constellation or Instrument object
- **bin1** (*(array-like)*) – List holding [min, max, number of bins] or array-like containing bin edges
- **label1** (*(string)*) – data column name that the binning will be performed over (i.e., lat)
- **data_label** (*(list-like)*) – contains strings identifying data product(s) to be averaged
- **auto_bin** (*if True, function will create bins from the min, max and*) – number of bins. If false, bin edges must be manually entered
- **returnData** (*(boolean)*) – Return data in output dictionary as well as statistics

Returns **median** – 1D median accessed by data_label as a function of label1 over the season delineated by bounds of passed instrument objects. Also includes 'count' and 'avg_abs_dev' as well as the values of the bin edges in 'bin_x'

Return type dictionary

pysat.ssnl.avg.**median2D**(*const*, *bin1*, *label1*, *bin2*, *label2*, *data_label*, *returnData=False*, *auto_bin=True*)

Return a 2D average of data_label over a season and label1, label2.

Deprecated since version 2.2.0: *median2D* will be removed in pysat 3.0.0, it will be added to pysatSeasons

Parameters

- **const** (*Constellation or Instrument*) –
- **bin#** (*[min, max, number of bins], or array-like containing bin edges*) –
- **label#** (*string*) – identifies data product for bin#
- **data_label** (*list-like*) – contains strings identifying data product(s) to be averaged
- **auto_bin** (*if True, function will create bins from the min, max and*) – number of bins. If false, bin edges must be manually entered

Returns **median** – 2D median accessed by data_label as a function of label1 and label2 over the season delineated by bounds of passed instrument objects. Also includes 'count' and 'avg_abs_dev' as well as the values of the bin edges in 'bin_x' and 'bin_y'.

Return type dictionary

9.9.3 Plot

pysat.ssnl.plot.**scatterplot**(*inst*, *labelx*, *labely*, *data_label*, *datalim*, *xlim=None*, *ylim=None*)

Return scatterplot of data_label(s) as functions of labelx,y over a season.

Deprecated since version 2.2.0: *scatterplot* will be removed in pysat 3.0.0, it will be added to pysatSeasons

Parameters

- **labelx** (*string*) – data product for x-axis
- **labely** (*string*) – data product for y-axis
- **data_label** (*string, array-like of strings*) – data product(s) to be scatter plotted
- **datalim** (*numyp array*) – plot limits for data_label

Returns

- *Returns a list of scatter plots of data_label as a function*
- *of labelx and labely over the season delineated by start and*
- *stop datetime objects.*

9.10 Utilities

9.10.1 pysat.utils - utilities for running pysat

pysat.utils contains a number of functions used throughout the pysat package. This includes conversion of formats, loading of files, and user-supplied info for the pysat data directory structure.

9.10.2 Coordinates

pysat.utils.coords - coordinate transformations for pysat

pysat.utils.coords contains a number of coordinate-transformation functions used throughout the pysat package.

pysat.utils.coords.**adjust_cyclic_data**(*samples*, *high=6.283185307179586*, *low=0.0*)
Adjust cyclic values such as longitude to a different scale

Parameters

- **samples** (*array_like*) – Input array
- **high** (*float or int*) – Upper boundary for circular standard deviation range (default=2 pi)
- **low** (*float or int*) – Lower boundary for circular standard deviation range (default=0)
- **axis** (*int or NoneType*) – Axis along which standard deviations are computed. The default is to compute the standard deviation of the flattened array

Returns **out_samples** – Circular standard deviation

Return type float

pysat.utils.coords.**calc_solar_local_time**(*inst*, *lon_name=None*, *slt_name='slt'*)
Append solar local time to an instrument object

Parameters

- **inst** (*pysat.Instrument instance*) – instrument object to be updated
- **lon_name** (*string*) – name of the longtiude data key (assumes data are in degrees)
- **slt_name** (*string*) – name of the output solar local time data key (default='slt')

Returns

Return type updates instrument data in column specified by slt_name

pysat.utils.coords.**geodetic_to_geocentric**(*lat_in*, *lon_in=None*, *inverse=False*)
Converts position from geodetic to geocentric or vice-versa.

Deprecated since version 2.2.0: *geodetic_to_geocentric* will be removed in pysat 3.0.0, it will be added to pysatMadrigal

Parameters

- **lat_in** (*float*) – latitude in degrees.
- **lon_in** (*float or NoneType*) – longitude in degrees. Remains unchanged, so does not need to be included. (default=None)
- **inverse** (*bool*) – False for geodetic to geocentric, True for geocentric to geodetic. (default=False)

Returns

- **lat_out** (*float*) – latitude [degree] (geocentric/detic if inverse=False/True)
- **lon_out** (*float or NoneType*) – longitude [degree] (geocentric/detic if inverse=False/True)
- **rad_earth** (*float*) – Earth radius [km] (geocentric/detic if inverse=False/True)

Notes

Uses WGS-84 values

References

Based on J.M. Ruohoniemi's geopack and R.J. Barnes radar.pro

pysat.utils.coords.**geodetic_to_geocentric_horizontal**(*lat_in*, *lon_in*, *az_in*, *el_in*, *inverse=False*)

Converts from local horizontal coordinates in a geodetic system to local horizontal coordinates in a geocentric system

Deprecated since version 2.2.0: *geodetic_to_geocentric_horizontal* will be removed in pysat 3.0.0, it will be added to pysatMadrigal

Parameters

- **lat_in** (*float*) – latitude in degrees of the local horizontal coordinate system center
- **lon_in** (*float*) – longitude in degrees of the local horizontal coordinate system center
- **az_in** (*float*) – azimuth in degrees within the local horizontal coordinate system
- **el_in** (*float*) – elevation in degrees within the local horizontal coordinate system
- **inverse** (*bool*) – False for geodetic to geocentric, True for inverse (default=False)

Returns

- **lat_out** (*float*) – latitude in degrees of the converted horizontal coordinate system center
- **lon_out** (*float*) – longitude in degrees of the converted horizontal coordinate system center
- **rad_earth** (*float*) – Earth radius in km at the geocentric/detic (False/True) location
- **az_out** (*float*) – azimuth in degrees of the converted horizontal coordinate system
- **el_out** (*float*) – elevation in degrees of the converted horizontal coordinate system

References

Based on J.M. Ruohoniemi's geopack and R.J. Barnes radar.pro

pysat.utils.coords.**global_to_local_cartesian**(*x_in*, *y_in*, *z_in*, *lat_cent*, *lon_cent*, *rad_cent*, *inverse=False*)

Converts a position from global to local cartesian or vice-versa

Deprecated since version 2.2.0: *global_to_local_cartesian* will be removed in pysat 3.0.0, it will be added to pysatMadrigal

Parameters

- **x_in** (*float*) – global or local cartesian x in km (inverse=False/True)
- **y_in** (*float*) – global or local cartesian y in km (inverse=False/True)
- **z_in** (*float*) – global or local cartesian z in km (inverse=False/True)
- **lat_cent** (*float*) – geocentric latitude in degrees of local cartesian system origin
- **lon_cent** (*float*) – geocentric longitude in degrees of local cartesian system origin

- **rad_cent** (*float*) – distance from center of the Earth in km of local cartesian system origin
- **inverse** (*bool*) – False to convert from global to local cartesian coodiantes, and True for the inverse (default=False)

Returns

- **x_out** (*float*) – local or global cartesian x in km (inverse=False/True)
- **y_out** (*float*) – local or global cartesian y in km (inverse=False/True)
- **z_out** (*float*) – local or global cartesian z in km (inverse=False/True)

Notes

The global cartesian coordinate system has its origin at the center of the Earth, while the local system has its origin specified by the input latitude, longitude, and radius. The global system has x intersecting the equatorial plane and the prime meridian, z pointing North along the rotational axis, and y completing the right-handed coodinate system. The local system has z pointing up, y pointing North, and x pointing East.

pysat.utils.coords.**local_horizontal_to_global_geo**(*az*, *el*, *dist*, *lat_orig*, *lon_orig*, *alt_orig*, *geodetic=True*)

Convert from local horizontal coordinates to geodetic or geocentric coordinates

Deprecated since version 2.2.0: *local_horizontal_to_global_geo* will be removed in pysat 3.0.0, it will be added to pysatMadrigal

Parameters

- **az** (*float*) – Azimuth (angle from North) of point in degrees
- **el** (*float*) – Elevation (angle from ground) of point in degrees
- **dist** (*float*) – Distance from origin to point in km
- **lat_orig** (*float*) – Latitude of origin in degrees
- **lon_orig** (*float*) – Longitude of origin in degrees
- **alt_orig** (*float*) – Altitude of origin in km from the surface of the Earth
- **geodetic** (*bool*) – True if origin coordinates are geodetic, False if they are geocentric. Will return coordinates in the same system as the origin input. (default=True)

Returns

- **lat_pnt** (*float*) – Latitude of point in degrees
- **lon_pnt** (*float*) – Longitude of point in degrees
- **rad_pnt** (*float*) – Distance to the point from the centre of the Earth in km

References

Based on J.M. Ruohoniemi's geopack and R.J. Barnes radar.pro

pysat.utils.coords.**scale_units**(*out_unit*, *in_unit*)

Determine the scaling factor between two units

Deprecated since version 2.2.0: *utils.coords.scale_units* will be removed in pysat 3.0.0, it will be moved to *utils.scale_units*

Parameters

- **out_unit** (*str*) – Desired unit after scaling
- **in_unit** (*str*) – Unit to be scaled

Returns **unit_scale** – Scaling factor that will convert from in_units to out_units

Return type float

pysat.utils.coords.**spherical_to_cartesian**(*az_in*, *el_in*, *r_in*, *inverse=False*)
Convert a position from spherical to cartesian, or vice-versa

Deprecated since version 2.2.0: *spherical_to_cartesian* will be removed in pysat 3.0.0, it will be added to pysatMadrigal

Parameters

- **az_in** (*float*) – azimuth/longitude in degrees or cartesian x in km (inverse=False/True)
- **el_in** (*float*) – elevation/latitude in degrees or cartesian y in km (inverse=False/True)
- **r_in** (*float*) – distance from origin in km or cartesian z in km (inverse=False/True)
- **inverse** (*boolian*) – False to go from spherical to cartesian and True for the inverse

Returns

- **x_out** (*float*) – cartesian x in km or azimuth/longitude in degrees (inverse=False/True)
- **y_out** (*float*) – cartesian y in km or elevation/latitude in degrees (inverse=False/True)
- **z_out** (*float*) – cartesian z in km or distance from origin in km (inverse=False/True)

Notes

This transform is the same for local or global spherical/cartesian transformations.

Returns elevation angle (angle from the xy plane) rather than zenith angle (angle from the z-axis)

pysat.utils.coords.**update_longitude**(*inst*, *lon_name=None*, *high=180.0*, *low=-180.0*)
Update longitude to the desired range

Parameters

- **inst** (*pysat.Instrument instance*) – instrument object to be updated
- **lon_name** (*string*) – name of the longtiude data
- **high** (*float*) – Highest allowed longitude value (default=180.0)
- **low** (*float*) – Lowest allowed longitude value (default=-180.0)

Returns

Return type updates instrument data in column 'lon_name'

9.10.3 Statistics

pysat.utils.stats - statistical operations in pysat

pysat.coords contains a number of coordinate-transformation functions used throughout the pysat package.

pysat.utils.stats.**median1D**(*self*, *bin_params*, *bin_label*, *data_label*)
Calculates the median for a series of binned data.

Deprecated since version 2.2.0: *median1D* will be removed in pysat 3.0.0, a similar function will be added to pysatSeasons

Parameters

- **bin_params** (*array_like*) – Input array defining the bins in which the median is calculated
- **bin_label** (*string*) – Name of data parameter which the bins cover
- **data_level** (*string*) – Name of data parameter to take the median of in each bin

Returns **medians** – The median data value in each bin

Return type array_like

pysat.utils.stats.**nan_circmean**(*samples*, *high=6.283185307179586*, *low=0.0*, *axis=None*)
NaN insensitive version of scipy's circular mean routine

Deprecated since version 2.1.0: *nan_circmean* will be removed in pysat 3.0.0, this functionality has been added to scipy 1.4

Parameters

- **samples** (*array_like*) – Input array
- **high** (*float or int*) – Upper boundary for circular standard deviation range (default=2 pi)
- **low** (*float or int*) – Lower boundary for circular standard deviation range (default=0)
- **axis** (*int or NoneType*) – Axis along which standard deviations are computed. The default is to compute the standard deviation of the flattened array

Returns **circmean** – Circular mean

Return type float

pysat.utils.stats.**nan_circstd**(*samples*, *high=6.283185307179586*, *low=0.0*, *axis=None*)
NaN insensitive version of scipy's circular standard deviation routine

Deprecated since version 2.1.0: *nan_circstd* will be removed in pysat 3.0.0, this functionality has been added to scipy 1.4

Parameters

- **samples** (*array_like*) – Input array
- **high** (*float or int*) – Upper boundary for circular standard deviation range (default=2 pi)
- **low** (*float or int*) – Lower boundary for circular standard deviation range (default=0)
- **axis** (*int or NoneType*) – Axis along which standard deviations are computed. The default is to compute the standard deviation of the flattened array

Returns **circstd** – Circular standard deviation

Return type float

9.10.4 Time

pysat.utils.time - date and time operations in pysat

pysat.utils.time contains a number of functions used throughout the pysat package, including interactions with datetime objects, seasons, and calculation of solar local time

pysat.utils.time.**calc_freq**(*index*)
Determine the frequency for a time index

Parameters **index** (*(array-like)*) – Datetime list, array, or Index

Returns **freq** – Frequency string as described in Pandas Offset Aliases

Return type (str)

Notes

Calculates the minimum time difference and sets that as the frequency.

To reduce the amount of calculations done, the returned frequency is either in seconds (if no sub-second resolution is found) or nanoseconds.

pysat.utils.time.**create_date_range**(*start*, *stop*, *freq='D'*)
Return array of datetime objects using input frequency from start to stop

Supports single datetime object or list, tuple, ndarray of start and stop dates.

freq codes correspond to pandas date_range codes, D daily, M monthly, S secondly

pysat.utils.time.**create_datetime_index**(*year=None*, *month=None*, *day=None*, *uts=None*)
Create a timeseries index using supplied year, month, day, and ut in seconds.

Parameters

- **year** (*array_like of ints*) –
- **month** (*array_like of ints or None*) –
- **day** (*array_like of ints*) – for day (default) or day of year (use month=None)
- **uts** (*array_like of floats*) –

Returns

Return type Pandas timeseries index.

Note: Leap seconds have no meaning here.

---

pysat.utils.time.**getyrdoy**(*date*)
Return a tuple of year, day of year for a supplied datetime object.

Parameters **date** (*datetime.datetime*) – Datetime object

Returns

- **year** (*int*) – Integer year
- **doy** (*int*) – Integer day of year

pysat.utils.time.**parse_date**(*str_yr*, *str_mo*, *str_day*, *str_hr='0'*, *str_min='0'*, *str_sec='0'*, *century=2000*)
Basic date parser for file reading

Parameters

- **str_yr** (*string*) – String containing the year (2 or 4 digits)
- **str_mo** (*string*) – String containing month digits
- **str_day** (*string*) – String containing day of month digits
- **str_hr** (*string ('0')*) – String containing the hour of day
- **str_min** (*string ('0')*) – String containing the minutes of hour
- **str_sec** (*string ('0')*) – String containing the seconds of minute
- **century** (*int (2000)*) – Century, only used if str_yr is a 2-digit year

Returns **out_date** – Pandas datetime object

Return type pds.datetime

pysat.utils.time.**season_date_range**(*start*, *stop*, *freq='D'*)

Deprecated Function, will be removed in future version.

Deprecated since version 2.1.0: *season_date_range* will be removed in pysat 3.0.0, this will be replaced by create_date_range

CHAPTER 10

Contributing

Bug reports, feature suggestions and other contributions are greatly appreciated! Pysat is a community-driven project and welcomes both feedback and contributions.

10.1 Short version

- Submit bug reports and feature requests at GitHub
- Make pull requests to the `develop` branch

10.2 Bug reports

When reporting a bug please include:

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

10.3 Feature requests and feedback

The best way to send feedback is to file an issue at GitHub.

If you are proposing a feature:

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

10.4 Development

To set up *pysat* for local development:

1. Fork pysat on GitHub.
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/pysat.git
```

3. Create a branch for local development:

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

   Now you can make your changes locally. Tests for new instruments are performed automatically. Tests for custom functions should be added to the appropriately named file in `pysat/tests`. For example, custom functions for the OMNI HRO data are tested in `pysat/tests/test_omni_hro.py`. If no test file exists, then you should create one. This testing uses nose, which will run tests on any python file in the test directory that starts with `test_`.

4. When you're done making changes, run all the checks to ensure that nothing is broken on your local system:

```
nosetests -vs pysat
```

5. Update/add documentation (in `docs`), if relevant
5. Commit your changes and push your branch to GitHub:

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

6. Submit a pull request through the GitHub website. Pull requests should be made to the `develop` branch.

10.5 Pull Request Guidelines

If you need some code review or feedback while you're developing the code, just make a pull request. Pull requests should be made to the `develop` branch.

For merging, you should:

1. Include an example for use
2. Add a note to `CHANGELOG.md` about the changes
3. Ensure that all checks passed (current checks include Scrutinizer, Travis-CI, and Coveralls)[1]

[1] If you don't have all the necessary Python versions available locally or have trouble building all the testing environments, you can rely on Travis to run the tests for each change you add in the pull request. Because testing here will delay tests by other developers, please ensure that the code passes all tests on your local system first.

CHAPTER 11

Frequently Asked Questions

Q. Does pysat support data from . . . mission?

A. Possibly! A full list of supported instruments / models / scientific indices can be found here: https://pysat.readthedocs.io/en/latest/supported_instruments.html Things that are currently in development can be found here: https://github.com/pysat/pysat/blob/develop/docs/supported_instruments.rst

Q. Yeah, but what about . . . ?

A. If a dataset is not in the current list of instruments or in the list on the develop branch, feel free to add it as a pull request. For the most part, pysat makes it simple to add instruments. Templates are included for the NASA CDAWeb and Madrigal databases. Check out info about adding an nstrument here: https://pysat.readthedocs.io/en/latest/new_instrument.html# and make sure to read the guidelines for pull requests here: https://github.com/rstoneback/pysat/blob/main/CONTRIBUTING.md

Got a question? Add it to the wiki FAQ: https://github.com/rstoneback/pysat/wiki/FAQ

Python Module Index

p

pysat.instruments.champ_star, 44
pysat.instruments.cnofs_ivm, 41
pysat.instruments.cnofs_plp, 42
pysat.instruments.cnofs_vefi, 43
pysat.instruments.cosmic_gps, 44
pysat.instruments.de2_lang, 45
pysat.instruments.de2_nacs, 46
pysat.instruments.de2_rpa, 47
pysat.instruments.de2_wats, 48
pysat.instruments.demeter_iap, 49
pysat.instruments.dmsp_ivm, 50
pysat.instruments.icon_euv, 53
pysat.instruments.icon_fuv, 53
pysat.instruments.icon_ivm, 54
pysat.instruments.icon_mighti, 55
pysat.instruments.iss_fpmu, 56
pysat.instruments.jro_isr, 56
pysat.instruments.methods.demeter, 84
pysat.instruments.methods.general, 86
pysat.instruments.methods.icon, 91
pysat.instruments.methods.madrigal, 92
pysat.instruments.methods.nasa_cdaweb, 87
pysat.instruments.methods.sw, 94
pysat.instruments.omni_hro, 57
pysat.instruments.rocsat1_ivm, 59
pysat.instruments.sport_ivm, 59
pysat.instruments.superdarn_grdex, 59
pysat.instruments.supermag_magnetometer, 60
pysat.instruments.sw_dst, 61
pysat.instruments.sw_f107, 61
pysat.instruments.sw_kp, 62
pysat.instruments.templates.netcdf_pandas, 108
pysat.instruments.templates.template_cdaweb_instrument, 103
pysat.instruments.templates.template_instrument, 97
pysat.instruments.timed_saber, 63
pysat.instruments.timed_see, 64
pysat.instruments.ucar_tiegcm, 65
pysat.ssnl.avg, 128
pysat.ssnl.occur_prob, 125
pysat.ssnl.plot, 129
pysat.utils, 129
pysat.utils.coords, 130
pysat.utils.stats, 133
pysat.utils.time, 135

Index

A

accept_default_labels() (*pysat.Meta method*), 121
add() (*pysat.Constellation method*), 111
add() (*pysat.Custom method*), 114
add_drift_geo_coord() (*in module pysat.instruments.demeter_iap*), 50
add_drift_lgm_coord() (*in module pysat.instruments.demeter_iap*), 50
add_drift_sat_coord() (*in module pysat.instruments.demeter_iap*), 50
add_drift_unit_vectors() (*in module pysat.instruments.dmsp_ivm*), 52
add_drifts_polar_cap_x_y() (*in module pysat.instruments.dmsp_ivm*), 52
adjust_cyclic_data() (*in module pysat.utils.coords*), 130
apply_default_labels() (*pysat.Meta method*), 121
attach() (*pysat.Custom method*), 114
attr_case_name() (*pysat.Meta method*), 121
attrs() (*pysat.Meta method*), 121
axis_label (*pysat.Meta attribute*), 119

B

base_path (*pysat.Files attribute*), 115
bounds (*pysat.Instrument attribute*), 79, 80
by_orbit2D() (*in module pysat.ssnl.occur_prob*), 125
by_orbit3D() (*in module pysat.ssnl.occur_prob*), 126
bytes_to_float() (*in module pysat.instruments.methods.demeter*), 84

C

calc_daily_Ap() (*in module pysat.instruments.methods.sw*), 94
calc_freq() (*in module pysat.utils.time*), 135
calc_solar_local_time() (*in module pysat.utils.coords*), 130
calculate_clock_angle() (*in module pysat.instruments.omni_hro*), 58
calculate_imf_steadiness() (*in module pysat.instruments.omni_hro*), 58
cedar_rules() (*in module pysat.instruments.methods.madrigal*), 92
clean() (*in module pysat.instruments.templates.template_cdaweb_instrument*), 108
clean() (*in module pysat.instruments.templates.template_instrument*), 100
clear() (*pysat.Custom method*), 115
combine_f107() (*in module pysat.instruments.methods.sw*), 95
combine_kp() (*in module pysat.instruments.methods.sw*), 95
concat() (*pysat.Meta method*), 121
concat_data() (*pysat.Instrument method*), 80
Constellation (*class in pysat*), 111
convert_ap_to_kp() (*in module pysat.instruments.methods.sw*), 96
convert_timestamp_to_datetime() (*in module pysat.instruments.methods.general*), 86
copy() (*pysat.Instrument method*), 81
create_date_range() (*in module pysat.utils.time*), 135
create_datetime_index() (*in module pysat.utils.time*), 135
current (*pysat.Orbits attribute*), 124
Custom (*class in pysat*), 113
custom (*pysat.Instrument attribute*), 79

D

daily2D() (*in module pysat.ssnl.occur_prob*), 126
daily3D() (*in module pysat.ssnl.occur_prob*), 127
data (*pysat.Instrument attribute*), 78
data (*pysat.Meta attribute*), 118
data_mod() (*pysat.Constellation method*), 112
data_path (*pysat.Files attribute*), 115

date (*pysat.Instrument attribute*), 78, 81
default() (*in module pysat.instruments.templates.template_cdaweb_instrument*), 103
default() (*in module pysat.instruments.templates.template_instrument*), 97
desc_label (*pysat.Meta attribute*), 119
difference() (*pysat.Constellation method*), 112
download() (*in module pysat.instruments.methods.demeter*), 84
download() (*in module pysat.instruments.methods.madrigal*), 93
download() (*in module pysat.instruments.methods.nasa_cdaweb*), 90
download() (*in module pysat.instruments.templates.netcdf_pandas*), 102, 110
download() (*in module pysat.instruments.templates.template_cdaweb_instrument*), 107
download() (*in module pysat.instruments.templates.template_instrument*), 99
download() (*pysat.Instrument method*), 81
download_updated_files() (*pysat.Instrument method*), 81
doy (*pysat.Instrument attribute*), 79
drop() (*pysat.Meta method*), 122

E

empty (*pysat.Instrument attribute*), 81
empty (*pysat.Meta attribute*), 122
export_nan (*pysat.Meta attribute*), 119

F

Files (*class in pysat*), 115
files (*pysat.Instrument attribute*), 79
fill_label (*pysat.Meta attribute*), 119
filter_data_single_date() (*in module pysat.instruments.methods.madrigal*), 94
filter_geoquiet() (*in module pysat.instruments.sw_kp*), 63
from_csv() (*pysat.Meta class method*), 122
from_os() (*pysat.Files class method*), 117

G

generic_meta_translator() (*pysat.Instrument method*), 81
geodetic_to_geocentric() (*in module pysat.utils.coords*), 130
geodetic_to_geocentric_horizontal() (*in module pysat.utils.coords*), 131
get_file_array() (*pysat.Files method*), 117
get_index() (*pysat.Files method*), 117
get_new() (*pysat.Files method*), 118
getyrdoy() (*in module pysat.utils.time*), 135
global_to_local_cartesian() (*in module pysat.utils.coords*), 131

H

has_attr() (*pysat.Meta method*), 122

I

index (*pysat.Instrument attribute*), 82
init() (*in module pysat.instruments.templates.netcdf_pandas*), 100, 109
init() (*in module pysat.instruments.templates.template_instrument*), 97
Instrument (*class in pysat*), 77

K

keep() (*pysat.Meta method*), 122
keys() (*pysat.Meta method*), 122
keys_nD() (*pysat.Meta method*), 122
kwargs (*pysat.Instrument attribute*), 79

L

list_files() (*in module pysat.instruments.methods.general*), 86
list_files() (*in module pysat.instruments.methods.nasa_cdaweb*), 88
list_files() (*in module pysat.instruments.templates.netcdf_pandas*), 101, 109
list_files() (*in module pysat.instruments.templates.template_cdaweb_instrument*), 104
list_files() (*in module pysat.instruments.templates.template_instrument*), 98
list_remote_files() (*in module pysat.instruments.methods.icon*), 91
list_remote_files() (*in module pysat.instruments.methods.nasa_cdaweb*), 89
list_remote_files() (*in module pysat.instruments.templates.template_cdaweb_instrument*), 105
list_remote_files() (*in module pysat.instruments.templates.template_instrument*), 99
load() (*in module pysat.instruments.methods.madrigal*), 93
load() (*in module pysat.instruments.methods.nasa_cdaweb*), 87

load() (*in module pysat.instruments.templates.netcdf_pandas*), 100, 109
load() (*in module pysat.instruments.templates.template_cdaweb_instrument*), 103
load() (*in module pysat.instruments.templates.template_instrument*), 97
load() (*pysat.Constellation method*), 113
load() (*pysat.Instrument method*), 82
load() (*pysat.Orbits method*), 124
load_attitude_parameters() (*in module pysat.instruments.methods.demeter*), 85
load_binary_file() (*in module pysat.instruments.methods.demeter*), 85
load_general_header() (*in module pysat.instruments.methods.demeter*), 84
load_location_parameters() (*in module pysat.instruments.methods.demeter*), 85
local_horizontal_to_global_geo() (*in module pysat.utils.coords*), 132

M

manual_org (*pysat.Files attribute*), 116
max_label (*pysat.Meta attribute*), 119
mean_by_day() (*in module pysat.ssnl.avg*), 128
mean_by_file() (*in module pysat.ssnl.avg*), 128
mean_by_orbit() (*in module pysat.ssnl.avg*), 128
median1D() (*in module pysat.ssnl.avg*), 128
median1D() (*in module pysat.utils.stats*), 133
median2D() (*in module pysat.ssnl.avg*), 129
merge() (*pysat.Meta method*), 122
Meta (*class in pysat*), 118
meta (*pysat.Instrument attribute*), 79
min_label (*pysat.Meta attribute*), 119

N

name_label (*pysat.Meta attribute*), 119
nan_circmean() (*in module pysat.utils.stats*), 134
nan_circstd() (*in module pysat.utils.stats*), 134
next() (*pysat.Instrument method*), 82
next() (*pysat.Orbits method*), 124
notes_label (*pysat.Meta attribute*), 119

O

Orbits (*class in pysat*), 123
orbits (*pysat.Instrument attribute*), 79

P

parse_date() (*in module pysat.utils.time*), 135
plot_label (*pysat.Meta attribute*), 119
pop() (*pysat.Meta method*), 122
prev() (*pysat.Instrument method*), 82
prev() (*pysat.Orbits method*), 125
pysat.instruments.champ_star (*module*), 44
pysat.instruments.cnofs_ivm (*module*), 41
pysat.instruments.cnofs_plp (*module*), 42
pysat.instruments.cnofs_vefi (*module*), 43
pysat.instruments.cosmic_gps (*module*), 44
pysat.instruments.de2_lang (*module*), 45
pysat.instruments.de2_nacs (*module*), 46
pysat.instruments.de2_rpa (*module*), 47
pysat.instruments.de2_wats (*module*), 48
pysat.instruments.demeter_iap (*module*), 49
pysat.instruments.dmsp_ivm (*module*), 50
pysat.instruments.icon_euv (*module*), 53
pysat.instruments.icon_fuv (*module*), 53
pysat.instruments.icon_ivm (*module*), 54
pysat.instruments.icon_mighti (*module*), 55
pysat.instruments.iss_fpmu (*module*), 56
pysat.instruments.jro_isr (*module*), 56
pysat.instruments.methods.demeter (*module*), 84
pysat.instruments.methods.general (*module*), 86
pysat.instruments.methods.icon (*module*), 91
pysat.instruments.methods.madrigal (*module*), 92
pysat.instruments.methods.nasa_cdaweb (*module*), 87
pysat.instruments.methods.sw (*module*), 94
pysat.instruments.omni_hro (*module*), 57
pysat.instruments.rocsat1_ivm (*module*), 59
pysat.instruments.sport_ivm (*module*), 59
pysat.instruments.superdarn_grdex (*module*), 59
pysat.instruments.supermag_magnetometer (*module*), 60
pysat.instruments.sw_dst (*module*), 61
pysat.instruments.sw_f107 (*module*), 61
pysat.instruments.sw_kp (*module*), 62
pysat.instruments.templates.netcdf_pandas (*module*), 100, 108
pysat.instruments.templates.template_cdaweb_instrum (*module*), 103
pysat.instruments.templates.template_instrument (*module*), 97
pysat.instruments.timed_saber (*module*), 63
pysat.instruments.timed_see (*module*), 64
pysat.instruments.ucar_tiegcm (*module*), 65
pysat.ssnl.avg (*module*), 128
pysat.ssnl.occur_prob (*module*), 125
pysat.ssnl.plot (*module*), 129
pysat.utils (*module*), 129
pysat.utils.coords (*module*), 130
pysat.utils.stats (*module*), 133
pysat.utils.time (*module*), 135

R

refresh() (*pysat.Files method*), 118
remote_date_range() (*pysat.Instrument method*), 82
remote_file_list() (*pysat.Instrument method*), 82
remove_leading_text() (*in module pysat.instruments.methods.general*), 87

S

scale_label (*pysat.Meta attribute*), 119
scale_units() (*in module pysat.utils.coords*), 132
scatterplot() (*in module pysat.ssnl.plot*), 129
season_date_range() (*in module pysat.utils.time*), 136
set_bounds() (*pysat.Constellation method*), 113
set_metadata() (*in module pysat.instruments.methods.demeter*), 85
smooth_ram_drifts() (*in module pysat.instruments.dmsp_ivm*), 51
spherical_to_cartesian() (*in module pysat.utils.coords*), 133
ssl_download() (*in module pysat.instruments.methods.icon*), 92
start_date (*pysat.Files attribute*), 115
stop_date (*pysat.Files attribute*), 115

T

time_shift_to_magnetic_poles() (*in module pysat.instruments.omni_hro*), 58
to_netcdf4() (*pysat.Instrument method*), 82
today() (*pysat.Instrument method*), 83
tomorrow() (*pysat.Instrument method*), 83
transfer_attributes_to_instrument() (*pysat.Meta method*), 123

U

units_label (*pysat.Meta attribute*), 119
update_DMSP_ephemeris() (*in module pysat.instruments.dmsp_ivm*), 51
update_files (*pysat.Files attribute*), 116
update_longitude() (*in module pysat.utils.coords*), 133

V

var_case_name() (*pysat.Meta method*), 123
variables (*pysat.Instrument attribute*), 84

Y

yesterday() (*pysat.Instrument method*), 84
yr (*pysat.Instrument attribute*), 79