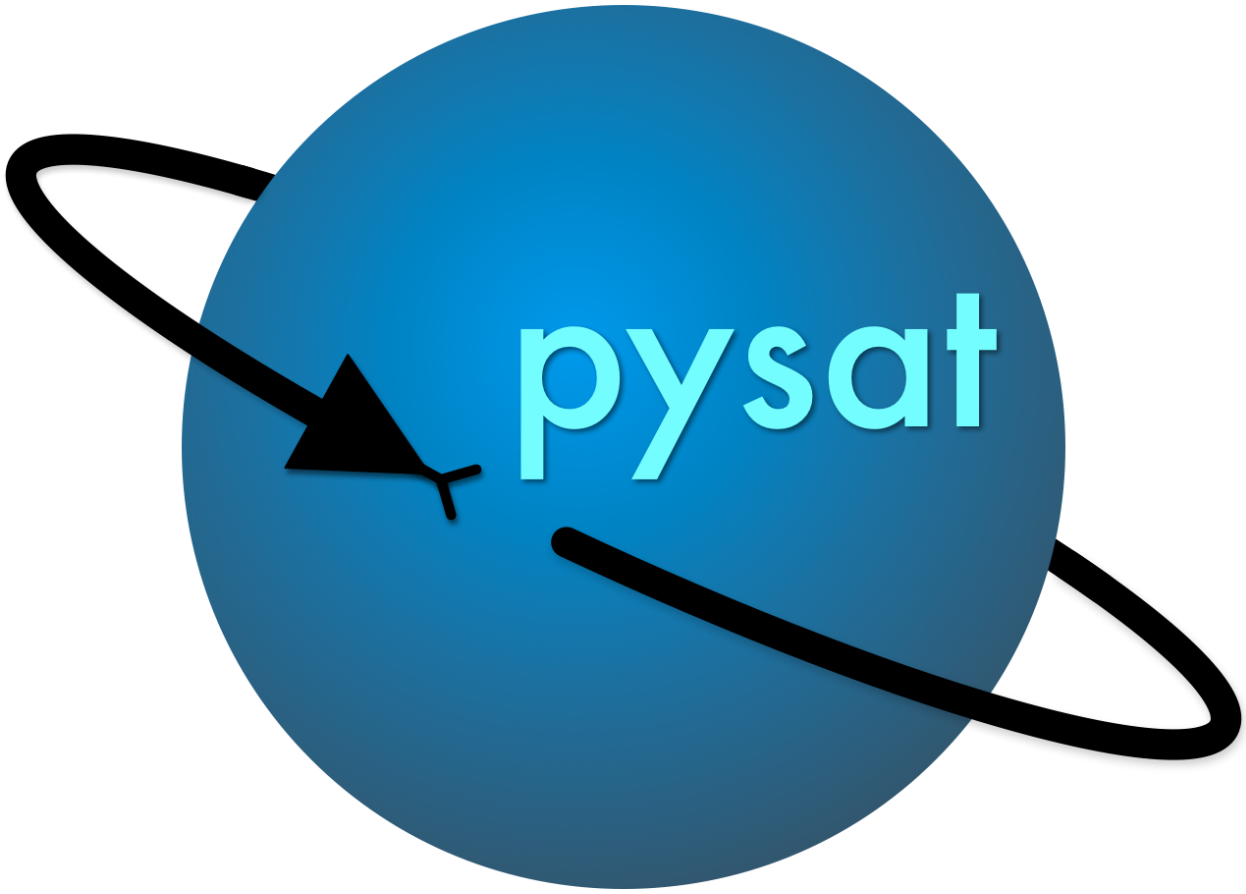


Python Satellite Data Analysis Toolkit (pysat) Documentation

Release 1.2.0

Russell Stoneback

Oct 09, 2018

Contents

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.

CHAPTER 2

Installation

Starting from scratch

Python and associated packages for science are freely available. Convenient science python package setups are available from Enthought and Continuum Analytics. Enthought also includes an IDE, though there are a number of choices. 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 educational users, an IDE from Jet Brains is available for free.

pysat

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

```
pip install pysat
```

Pysat requires some external non-python libraries for loading science data sets stored in netCDF and CDF formats.

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)
```

Common Data Format

The CDF library must be installed, along with python support, before pysat is able to load CDF files.

- pysatCDF contains everything needed by pysat to load CDF files, including the NASA CDF library. At the terminal command line:

```
pip install pysatCDF
```

netCDF

netCDF libraries must be installed, along with python support, before pysat is able to load netCDF files.

- netCDF C Library from Unidata (http://www.unidata.ucar.edu/downloads/netcdf/index.jsp)
- netCDF4-python

CHAPTER 3

Tutorial

3.1 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)
```

Instantiation

To create a pysat.Instrument object, select a platform, instrument name, and measurement type to be analyzed from the list of *Supported Instruments*. To work with Magnetometer data from the Vector Electric Field Instrument onboard the Communications/Navigation Outage Forecasting System (C/NOFS), use:

```
vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b')
```

Behind the scenes pysat uses a python module named cnofs_vefi that understands how to interact with 'dc_b' data. VEFI also measures electric fields in several modes that offer different data products. Though these measurements are not currently supported by the cnofs_vefi module, when they are, they can be selected via the tag string.

To load measurements from a different instrument on C/NOFS, the Ion Velocity Meter, which measures thermal plasma parameters, use:

```
ivm = pysat.Instrument(platform='cnofs', name='ivm')
```

In the background pysat uses the module cnofs_ivm to handle this data. There is only one measurement option from IVM, so no tag string is required.

Measurements from a constellation of COSMIC satellites are also available. These satellites measure GPS signals as they travel through the atmosphere. A number of different data sets are available from COSMIC, and are also supported by the relevant module.

```
# electron density profiles
cosmic = pysat.Instrument(platform='cosmic2013', name='gps', tag='ionprf')
# atmosphere profiles
cosmic = pysat.Instrument(platform='cosmic2013', name='gps', tag='atmprf')
```

Though the particulars of VEFI magnetometer data, IVM plasma parameters, and COSMIC atmospheric measurements are going to be quite different, the processes demonstrated below with VEFI also apply equally to IVM and COSMIC.

Download

Let's download some data. VEFI data is hosted by the NASA Coordinated Data Analysis Web (CDAWeb) at http://cdaweb.gsfc.nasa.gov. The proper process for downloading VEFI data is built into the cnofs_vefi module, which is handled by pysat. All we have to do is invoke the .download method attached to the VEFI object, or any other pysat Instrument.

```
# define date range to download data and download
start = pysat.datetime(2009,5,6)
stop = pysat.datetime(2009,5,9)
vefi.download(start, stop)

# download COSMIC data, which requires username and password
cosmic.download(start, stop, user=user, password=password)
```

The data is downloaded to pysat_data_dir/platform/name/tag/, in this case pysat_data_dir/cnofs/vefi/dc_b/. At the end of the download, pysat will update the list of files associated with VEFI.

Load Data

Data is loaded into vefi using the .load method using year, day of year; date; or filename.

```
vefi.load(2009,126)
vefi.load(date=start)
vefi.load(fname='cnofs_vefi_bfield_1sec_20090506_v05.cdf')
```

When the pysat load routine runs it stores the instrument data into vefi.data. The data structure is a pandas DataFrame, a highly capable structure with labeled rows and columns. Convenience access to the data is also available at the instrument level.

```
# all data
vefi.data
# particular magnetic component
vefi.data.dB_mer

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

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 pandas
import matplotlib.pyplot as plt
import numpy as np

# create empty series to hold result
mean_dB = pandas.Series()
# get list of dates between start and stop
date_array = pysat.utils.season_date_range(start, stop)
# iterate over season, calculate the mean absolute perturbation in
# meridional magnetic field
for date in date_array:
     vefi.load(date=date)
     if not vefi.data.empty:
          # isolate data to locations near geographic equator
          idx, = np.where((vefi['latitude'] < 5) & (vefi['latitude'] > -5))
          vefi.data = vefi.data.iloc[idx]
          # compute mean absolute db_Mer using pandas functions and store
          mean_dB[vefi.date] = vefi['dB_mer'].abs().mean(skipna=True)
# plot the result using pandas functionality
mean_dB.plot(title='Mean Absolute Perturbation in Meridional Magnetic Field')
plt.ylabel('Mean Absolute Perturbation ('+vefi.meta['dB_mer'].units+')')
```

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

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

is equivalent to

```
vefi.data = vefi[(vefi['latitude'] < 5) & (vefi['latitude'] > -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,

```
vefi = pysat.Instrument(platform='cnofs', name='vefi',
                        tag='dc_b', clean_level='none')
```

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

Metadata

Metadata is also stored along with the main science data.

```
# all metadata
vefi.meta.data
# dB_mer metadata
vefi.meta['dB_mer']
# units
vefi.meta['dB_mer'].units
# update units for dB_mer
vefi.meta['dB_mer'] = {'units':'new_units'}
# update display name, long_name
vefi.meta['dB_mer'] = {'long_name':'Fancy Name'}
# add new meta data
vefi.meta['new'] = {'units':'fake', 'long_name':'Display'}
```

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

```
vefi['new_data'] = new_data
```

The same activities may be performed for other instruments in the same manner. In particular, for measurements from the Ion Velocity Meter and profiles of electron density from COSMIC, use

```
# assignment with metadata
ivm = pysat.Instrument(platform='cnofs', name='ivm', tag='')
ivm.load(date=date)
ivm['double_mlt'] = {'data':2.*inst['mlt'], 'long_name':'Double MLT',
                     'units':'hours'}
```

```
cosmic = pysat.Instrument('cosmic2013','gps', tag='ionprf',  clean_level='clean')
start = pysat.datetime(2009,1,2)
stop = pysat.datetime(2009,1,3)
# requires CDAAC account
cosmic.download(start, stop, user='', password='')
cosmic.load(date=start)
# the profiles column has a DataFrame in each element which stores
# all relevant profile information indexed by altitude
# print part of the first profile, selection by integer location
print(cosmic[0,'profiles'].iloc[55:60, 0:3])
# print part of profile, selection by altitude value
print(cosmic[0,'profiles'].ix[196:207, 0:3])
```

Output for both print statements:

```
               ELEC_dens    GEO_lat    GEO_lon
MSL_alt
196.465454  81807.843750 -15.595786 -73.431015
198.882019  83305.007812 -15.585764 -73.430191
201.294342  84696.546875 -15.575747 -73.429382
203.702469  86303.039062 -15.565735 -73.428589
206.106354  87460.015625 -15.555729 -73.427803
```

3.2 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 .data.

Modify Functions

The instrument object is passed to function without copying, modify in place.

```
def custom_func_modify(inst, optional_param=False):
    inst['double_mlt'] = 2.*inst['mlt']
```

Add Functions

A copy of the instrument is passed to function, data to be added is returned.

```
def custom_func_add(inst, optional_param=False):
    return 2.*inst['mlt']
```

Add Function Including Metadata

```
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

```
ivm.custom.add(custom_func_modify, 'modify', optional_param2=True)
ivm.load(2009,1)
print (ivm['double_mlt'])
ivm.custom.add(custom_func_add, 'add', optional_param2=True)
ivm.bounds = (start,stop)
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 VEFI example, this time using nano-kernel functionality.

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

vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b')

def filter_vefi(inst):
    # select data near geographic equator
    idx, = np.where((vefi['latitude'] < 5) & (vefi['latitude'] > -5))
    vefi.data = vefi.data.iloc[idx]
    return
# attach filter to vefi object, function is run upon every load
vefi.custom.add(filter_ivm, 'modify')

# create empty series to hold result
mean_dB = pandas.Series()
# get list of dates between start and stop
date_array = pysat.utils.season_date_range(start, stop)
# iterate over season, calculate the mean absolute perturbation in
# meridional magnetic field
for date in date_array:
     vefi.load(date=date)
     if not vefi.data.empty:
         # compute mean absolute db_Mer using pandas functions and store
         mean_dB[vefi.date] = vefi['dB_mer'].abs().mean(skipna=True)
```

(continues on next page)

(continued from previous page)

```
# plot the result using pandas functionality
mean_dB.plot(title='Mean Absolute Perturbation in Meridional Magnetic Field')
plt.ylabel('Mean Absolute Perturbation ('+vefi.meta['dB_mer'].units+')')
```

Note the same result is obtained. The VEFI 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.

Adding Instrument Independence

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

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.season_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 db_Mer using pandas functions and store
            mean_val[inst.date] = inst[data_label].abs().mean(skipna=True)
   return mean_val

vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b')

def filter_vefi(inst):
    # select data near geographic equator
    idx, = np.where((vefi['latitude'] < 5) & (vefi['latitude'] > -5))
    vefi.data = vefi.data.iloc[idx]
    return
# attach filter to vefi object, function is run upon every load
vefi.custom.add(filter_ivm, 'modify')

# make a plot of daily 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.

```
cosmic = pysat.Instrument('cosmic2013','gps', tag='ionprf', clean_level='clean',␣
↪altitude_bin=3)

def filter_cosmic(inst):
    cosmic.data = cosmic[(cosmic['edmaxlat'] > -15) & (cosmic['edmaxlat'] < 15)]
```

(continues on next page)

(continued from previous page)

```
    return

cosmic.custom.add(filter_cosmic, 'modify')
data_label = 'edmax'
mean_max_dens = daily_mean(cosmic, start, stop, data_label)

# plot the result using pandas functionality
mean_max_dens.plot(title='Absolute Daily Mean of ' + cosmic.meta[data_label].long_
↪name)
plt.ylabel('Absolute Daily Mean ('+cosmic.meta[data_label].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

```
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.season_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
```

(continues on next page)

(continued from previous page)

```
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.season_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.utils.computational_form(data)
            mean_val[inst.date] = data.abs().mean(axis=0, skipna=True)

return mean_val
```

3.3 Time Series Analysis

Pending

3.4 Iteration

The seasonal analysis loop is repeated commonly:

```
date_array = pysat.utils.season_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 VEFI 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.utils.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
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 abstraction provided by the iteration support is also used for the next section on orbit data.

3.5 Orbit Support

Pysat has functionality to determine orbits on the fly from loaded data. These orbits will span day breaks as needed (generally). 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

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.

This section of pysat is still under development.

```
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 .orbit class. The data for the orbit is stored in .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:1
```

Note that getting the first orbit caused pysat to load the day previous, and then back to the current day. Orbits are one indexed though this will change. 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
```

3.6 Iteration and Instrument Independent Analysis

Now we can generalize daily_mean 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.

```
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
```

(continues on next page)

(continued from previous page)

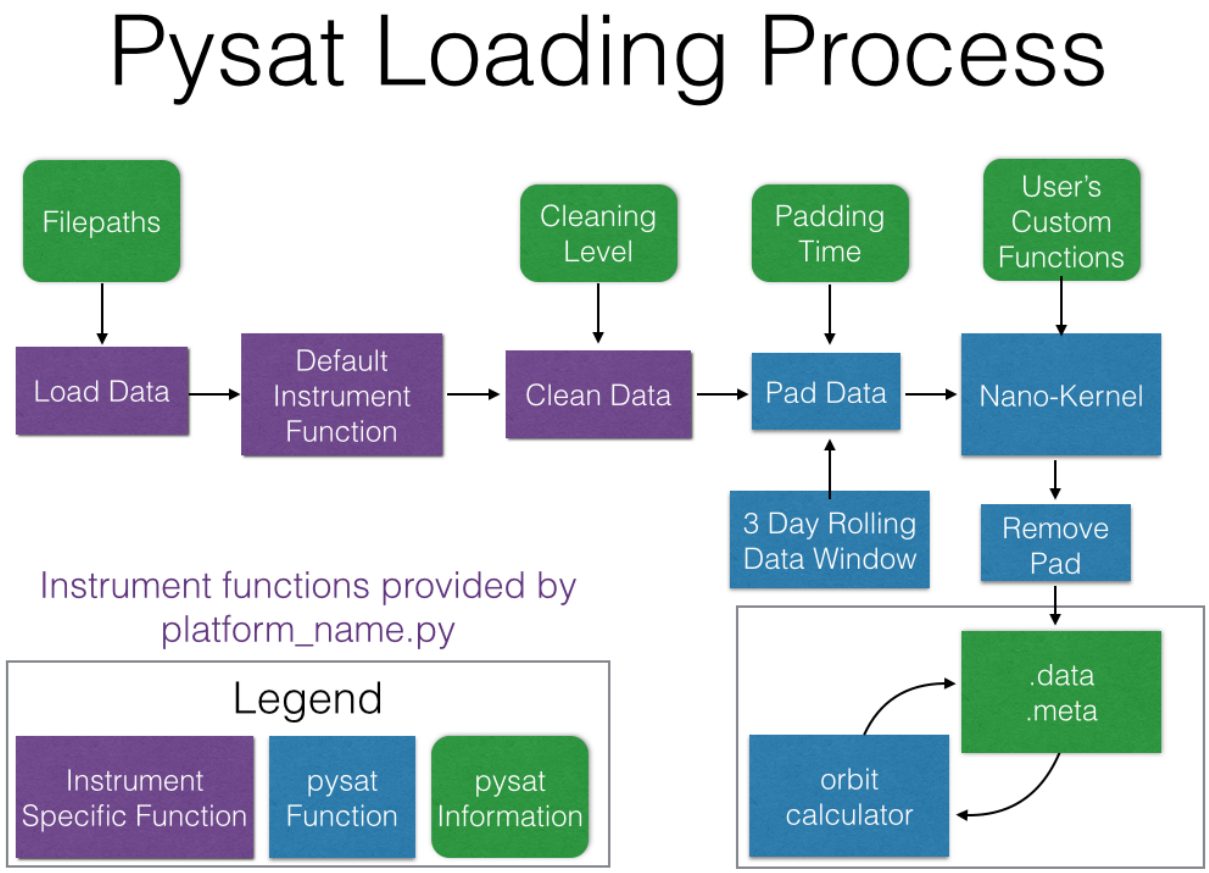
```
                data = inst[data_label]
                data.dropna(inplace=True)

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

                data = pysat.utils.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?

3.7 Summary Flow Charts



CHAPTER 4

Code Examples

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.

4.1 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.

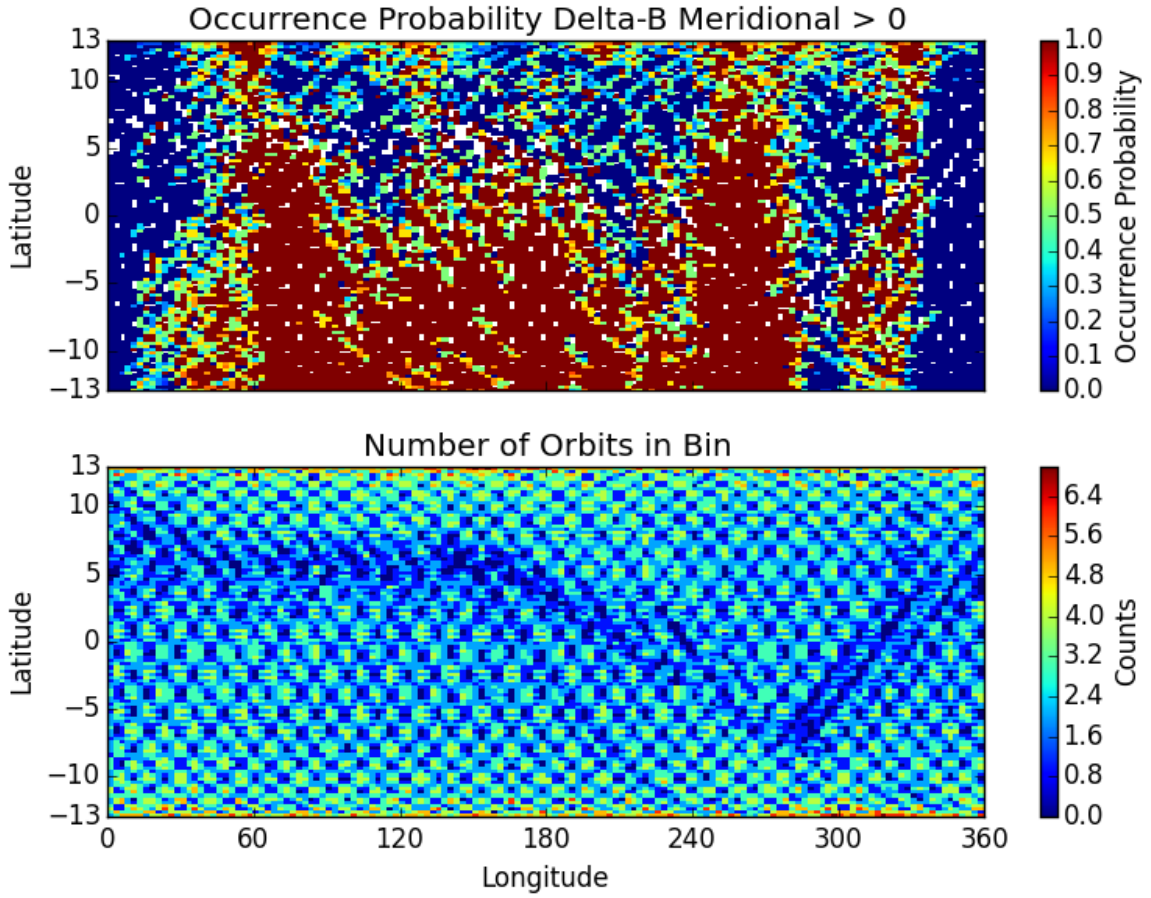
```
import os
import pysat
import matplotlib.pyplot as plt
import pandas as pds
import numpy as np

# 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(vefi['B_flag']==0)
    vefi.data = vefi.data.iloc[idx]
    return
# attach function to vefi
vefi.custom.add(filter_vefi,'modify')
# set limits on dates analysis will cover, inclusive
```

(continues on next page)

(continued from previous page)

```
start = pds.datetime(2010,5,9)
stop = pds.datetime(2010,5,15)

# if there is no vefi dc magnetometer data on your system
# 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)

# 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_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')
axarr[1].set_title('Number of Orbits in Bin')

plt.colorbar(im,ax=axarr[1], label='Counts')
f.tight_layout()
plt.show()
plt.savefig(os.path.join(results_dir,'ssnl_occurrence_by_orbit_demo') )
```

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, overrepresenting 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.

4.2 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
import pandas as pds

# set the directory to save plots to
```

(continues on next page)

(continued from previous page)

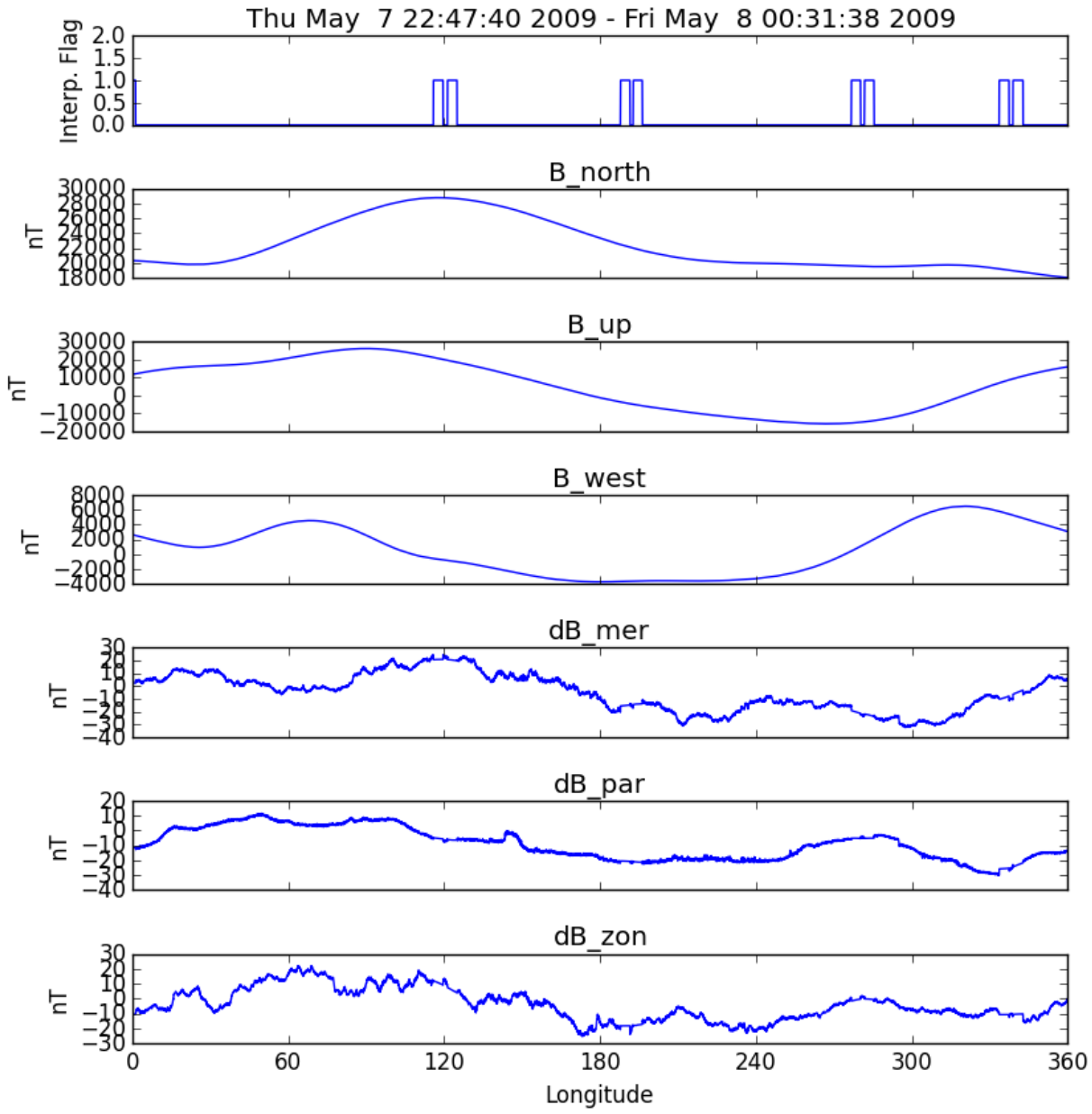
```
results_dir = ''

# 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)

# 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( 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()
```

(continues on next page)

(continued from previous page)

```
    fname = 'orbit_%05i.png' % orbit_count
    plt.savefig(os.path.join(results_dir, fname) )
    plt.close()
```

Output



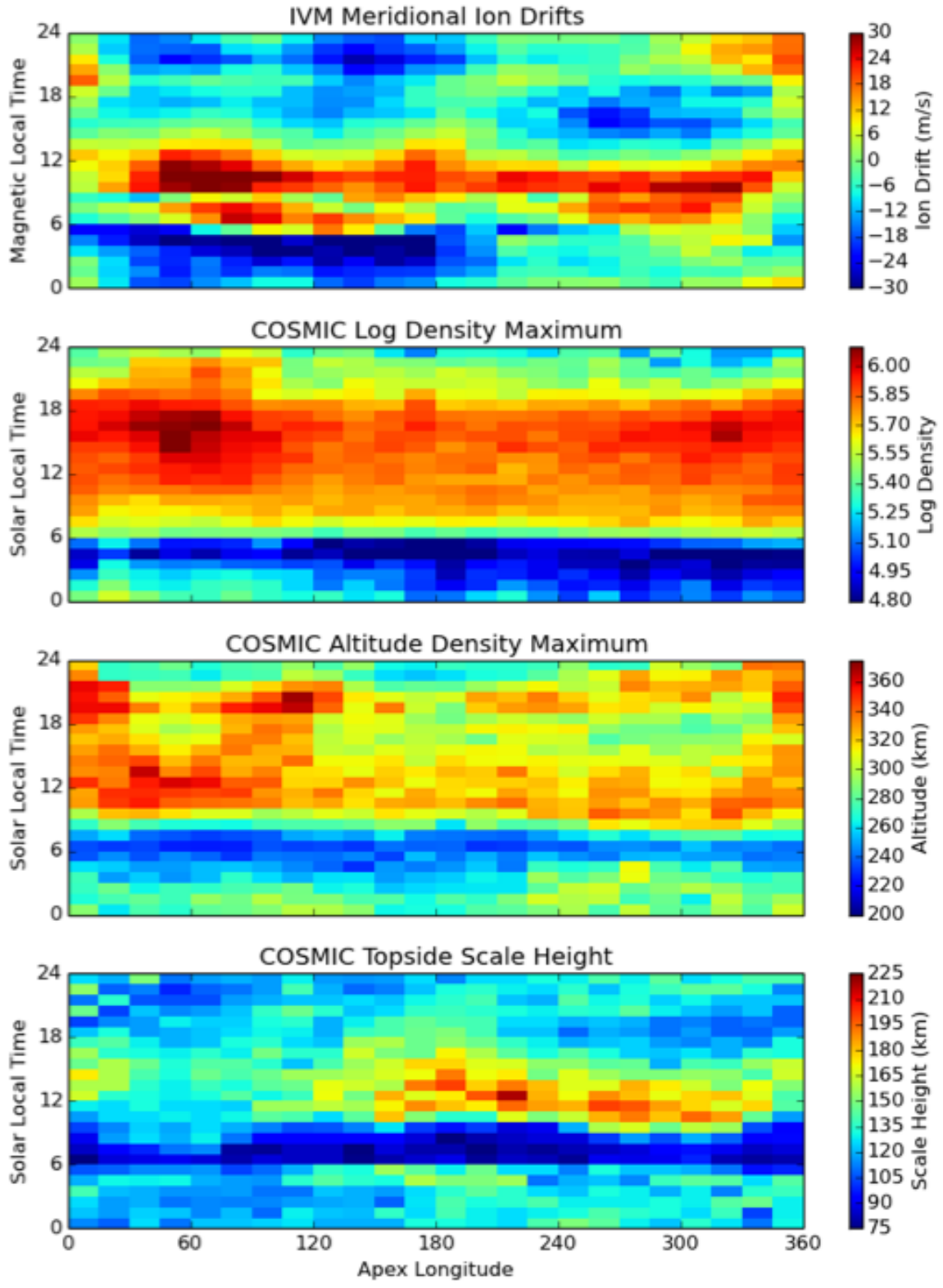
4.3 Seasonal Averaging of Ion Drifts and Density Profiles

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, use the code below:

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

```
# instantiate IVM Object
ivm = pysat.Instrument(platform='cnofs',name='ivm',clean_level='clean')
# restrict meausurements 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], 'apex_long',
                  [0,24,24], 'mlt', ['iv_mer'])

# create CODMIC instrument object
cosmic = pysat.Instrument(platform='cosmic2013', 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'])

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



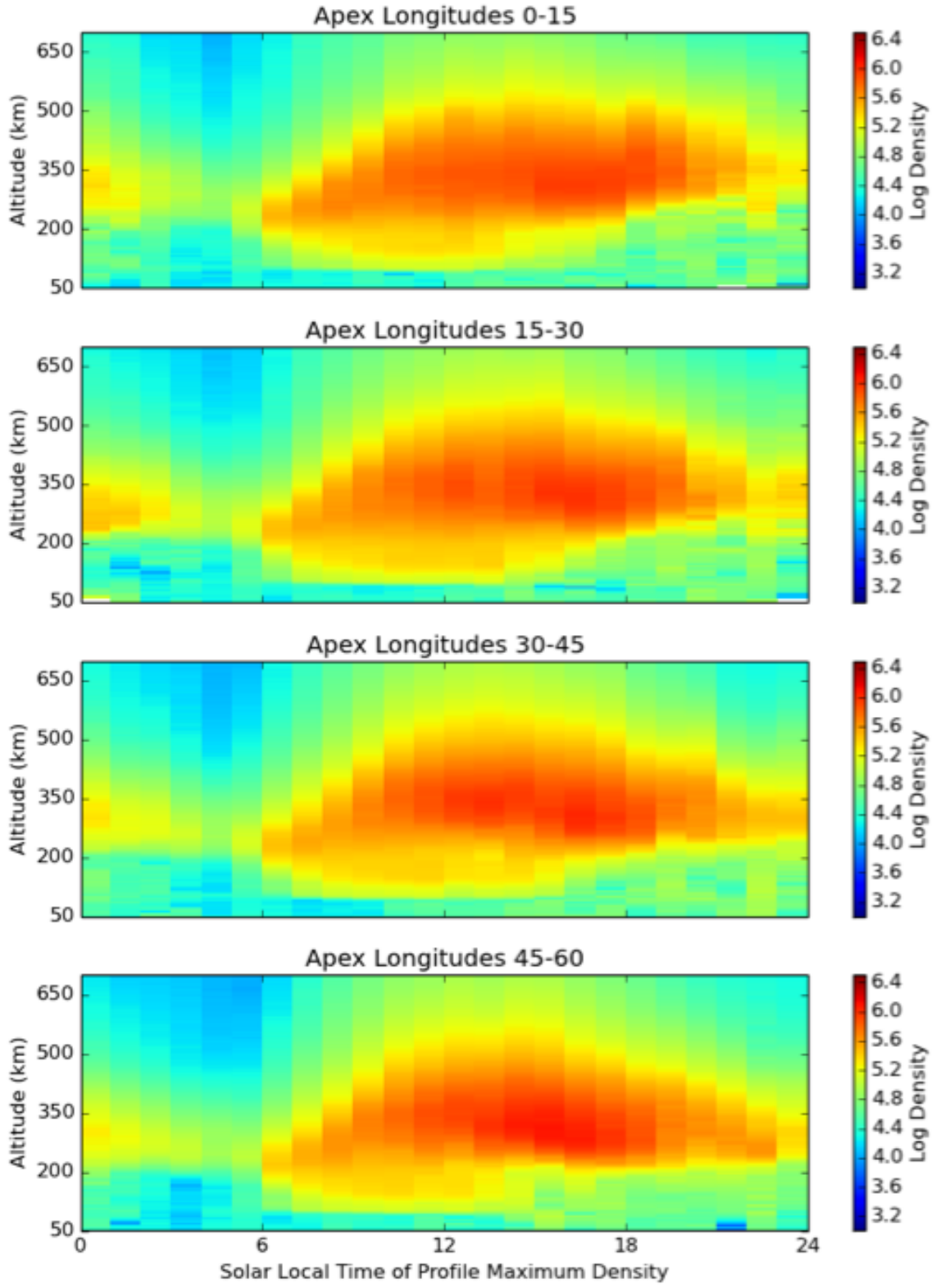
The top image is the median ion drift from the IVM, while the remaining plots are derived from the COSMIC density profiles. COSMIC data does not come with the location of the profiles in magnetic coordinates, so this information is added using the nano-kernel.

```
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 5

Supported Instruments

5.1 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.

param platform 'cnofs'

type platform string

param name 'ivm'

type name string

param tag None supported

type tag string

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.

5.2 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).

param platform 'cnofs'

type platform string

param name 'plp'

type name string

Warning:

- Currently no cleaning routine.
- Module not written by PLP team.

5.3 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).

param platform 'cnofs'

type platform string

param name 'vefi'

type name string

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

type tag string

Note:

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

Warning:

- Limited cleaning routine.
- Module not written by VEFI team.

5.4 CHAMP-STAR

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

param platform 'champ'

type platform string

param name 'star'

type name string

param tag None supported

type tag string

Warning:

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

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

5.5 COSMIC 2013 GPS

Loads data from the COSMIC satellite, 2013 reprocessing.

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.

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

type altitude_bin integer

param platform 'cosmic2013'

type platform string

param name 'gps' for Radio Occultation profiles

type name string

param tag Select profile type, one of {'ionprf', 'sonprf', 'wetprf', 'atmprf'}

type tag string

Note:

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

Warning:

- Routine was not produced by COSMIC team

5.6 COSMIC GPS

Loads and downloads 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.

param platform 'cosmic'

type platform string

param name 'gps' for Radio Occultation profiles

type name string

param tag Select profile type, one of {'ionprf', 'sonprf', 'wetprf', 'atmprf'}

type tag string

Note:

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

Warning:

- Routine was not produced by COSMIC team

5.7 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.

param platform 'dmsp'

type platform string

param name 'ivm'

type name string

param tag 'utd'

type tag string

Example

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

5.8 Dst

Supports Dst values. Downloads data from NGDC.

param platform 'sw'

type platform string

param name 'dst'

type name string

param tag None supported

type tag string

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.

5.9 Kp

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

param platform 'sw'

type platform string

param name 'kp'

type name string

param tag None supported

type tag string

Note: 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.

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.

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.

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.

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.

Returns **None** – sat Instrument object modified in place

Return type NoneType

5.10 ICON EUV

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

param platform 'icon'

type platform string

param name 'euv'

type name string

param tag None supported

type tag string

Warning:

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

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

5.11 ICON IVM

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

param platform 'icon'

type platform string

param name 'ivm'

type name string

param tag None supported

type tag string

param sat_id 'a' or 'b'

type sat_id string

Warning:

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

18. (a) Stoneback

pysat.instruments.icon_ivm.**remove_icon_names**(*inst*, *target=None*)

Removes leading text on ICON project variable names

Parameters

- **inst** (pysat.Instrument) – ICON associated pysat.Instrument object
- **target** (*str*) – Leading string to remove. If none supplied, ICON project standards are used to identify and remove leading text

Returns Modifies Instrument object in place

Return type None

5.12 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).

param platform 'iss'

type platform string

param name 'fpmu'

type name string

param tag None Supported

type tag string

Warning:

- Currently no cleaning routine.
- Module not written by FPMU team.

5.13 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/nasa_cdaweb_methods.py to make adding new CDAWeb instruments easy.

pysat.instruments.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.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
```

(continues on next page)

(continued from previous page)

```
# 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.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.
- **sat_id** (*string ('')*) – Satellite ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself.
- **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.
- **format_str** (*string (None)*) – 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.

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 nasa_cdaweb_methods 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 nasa_cdaweb_methods.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.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.
- **sat_id** (*string ('')*) – Satellite ID string identifier used for particular dataset. This input is provided by pysat.
- **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.

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

Return type (NoneType)

5.14 NASA CDAWeb

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

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

Load NASA CDAWeb CDF files.

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.

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.nasa_cdaweb_methods.**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.

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 tags supported by list_files routine. Values are the default format_str values for key. (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.

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_methods.list_files,
                               supported_tags=supported_tags)

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

pysat.instruments.nasa_cdaweb_methods.**download**(*supported_tags*, *date_array*, *tag*, *sat_id*, *ftp_site='cdaweb.gsfc.nasa.gov'*, *data_path=None*, *user=None*, *password=None*, *fake_daily_files_from_monthly=False*)

Routine to download NASA CDAWeb CDF data.

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)*) – tag or None (default=None)
- **sat_id** (*(str or NoneType)*) – satellite id or None (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)
- **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_methods.download,
                             supported_tags=supported_tags)
```

5.15 OMNI

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.

param platform 'omni'

type platform string

param name 'hro'

type name string

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

type tag string

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.

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

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

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.

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.

5.16 PYSAT SGP4

Produces satellite orbit data. Orbit is simulated using Two Line Elements (TLEs) and SGP4. Satellite position is coupled to several space science models to simulate the atmosphere the satellite is in.

pysat.instruments.pysat_sgp4.**load**(*fnames*, *tag=None*, *sat_id=None*, *obs_long=0.0*, *obs_lat=0.0*, *obs_alt=0.0*, *TLE1=None*, *TLE2=None*)
Returns data and metadata in the format required by pysat. Finds position of satellite in both ECI and ECEF co-ordinates.

Routine is directly called by pysat and not the user.

Parameters

- **fnames** (*list-like collection*) – File name that contains date in its name.
- **tag** (*string*) – Identifies a particular subset of satellite data
- **sat_id** (*string*) – Satellite ID
- **obs_long** (*float*) – Longitude of the observer on the Earth's surface
- **obs_lat** (*float*) – Latitude of the observer on the Earth's surface
- **obs_alt** (*float*) – Altitude of the observer on the Earth's surface
- **TLE1** (*string*) – First string for Two Line Element. Must be in TLE format
- **TLE2** (*string*) – Second string for Two Line Element. Must be in TLE format

Example

inst = pysat.Instrument('pysat', 'sgp4', TLE1='1 25544U 98067A 18135.61844383 .00002728 00000-0 48567-4 0 9998', TLE2='2 25544 51.6402 181.0633 0004018 88.8954 22.2246 15.54059185113452')

inst.load(2018, 1)

pysat.instruments.pysat_sgp4.**add_sc_attitude_vectors**(*inst*)
Add attitude vectors for spacecraft assuming ram pointing.

Presumes spacecraft is pointed along the velocity vector (x), z is generally nadir pointing (positive towards Earth), and y completes the right handed system (generally southward).

Notes

Expects velocity and position of spacecraft in Earth Centered Earth Fixed (ECEF) coordinates to be in the instrument object and named velocity_ecef_* (=*x,y,z) and position_ecef_* (*=x,y,z)

Adds attitude vectors for spacecraft in the ECEF basis by calculating the scalar product of each attitude vector with each component of ECEF.

Parameters **inst** (pysat.Instrument) – Instrument object

Returns Modifies pysat.Instrument object in place to include S/C attitude unit vectors, expressed in ECEF basis. Vectors are named sc_(x,y,z)hat_ecef_(x,y,z). sc_xhat_ecef_x is the spacecraft unit vector along x (positive along velocity vector) reported in ECEF, ECEF x-component.

Return type None

pysat.instruments.pysat_sgp4.**calculate_ecef_velocity**(*inst*)

Calculates spacecraft velocity in ECEF frame.

Presumes that the spacecraft velocity in ECEF is in the input instrument object as position_ecef_*. Uses a symmetric difference to calculate the velocity thus endpoints will be set to NaN. Routine should be run using pysat data padding feature to create valid end points.

Parameters **inst** (pysat.Instrument) – Instrument object

Returns Modifies pysat.Instrument object in place to include ECEF velocity using naming scheme velocity_ecef_* (*=x,y,z)

Return type None

pysat.instruments.pysat_sgp4.**add_quasi_dipole_coordinates**(*inst*, *glat_label='glat'*, *glong_label='glong'*, *alt_label='alt'*)

Uses Apexpy package to add quasi-dipole coordinates to instrument object.

The Quasi-Dipole coordinate system includes both the tilt and offset of the geomagnetic field to calculate the latitude, longitude, and local time of the spacecraft with respect to the geomagnetic field.

This system is preferred over AACGM near the equator for LEO satellites.

Example

function added velow modifies the inst object upon every inst.load call inst.custom.add(add_quasi_dipole_coordinates, 'modify', glat_label='custom_label')

Parameters

- **inst** (pysat.Instrument) – Designed with pysat_sgp4 in mind
- **glat_label** (*string*) – label used in inst to identify WGS84 geodetic latitude (degrees)
- **glong_label** (*string*) – label used in inst to identify WGS84 geodetic longitude (degrees)
- **alt_label** (*string*) – label used in inst to identify WGS84 geodetic altitude (km, height above surface)

Returns Input pysat.Instrument object modified to include quasi-dipole coordinates, 'qd_lat' for magnetic latitude, 'qd_long' for longitude, and 'mlt' for magnetic local time.

Return type inst

pysat.instruments.pysat_sgp4.**add_aacgm_coordinates**(*inst*, *glat_label='glat'*, *glong_label='glong'*, *alt_label='alt'*)

Uses AACGMV2 package to add AACGM coordinates to instrument object.

The Altitude Adjusted Corrected Geomagnetic Coordinates library is used to calculate the latitude, longitude, and local time of the spacecraft with respect to the geomagnetic field.

Example

function added velow modifies the inst object upon every inst.load call inst.custom.add(add_quasi_dipole_coordinates, 'modify', glat_label='custom_label')

Parameters

- **inst** (pysat.Instrument) – Designed with pysat_sgp4 in mind
- **glat_label** (*string*) – label used in inst to identify WGS84 geodetic latitude (degrees N)
- **glong_label** (*string*) – label used in inst to identify WGS84 geodetic longitude (degrees E)
- **alt_label** (*string*) – label used in inst to identify WGS84 geodetic altitude (km, height above surface)

Returns Input pysat.Instrument object modified to include quasi-dipole coordinates, 'aacgm_lat' for magnetic latitude, 'aacgm_long' for longitude, and 'aacgm_mlt' for magnetic local time.

Return type inst

pysat.instruments.pysat_sgp4.**add_iri_thermal_plasma**(*inst*, *glat_label='glat'*, *glong_label='glong'*, *alt_label='alt'*)

Uses IRI (International Reference Ionosphere) model to simulate an ionosphere.

Uses pyglow module to run IRI. Configured to use actual solar parameters to run model.

Example

function added velow modifies the inst object upon every inst.load call inst.custom.add(add_iri_thermal_plasma, 'modify', glat_label='custom_label')

Parameters

- **inst** (pysat.Instrument) – Designed with pysat_sgp4 in mind
- **glat_label** (*string*) – label used in inst to identify WGS84 geodetic latitude (degrees)
- **glong_label** (*string*) – label used in inst to identify WGS84 geodetic longitude (degrees)
- **alt_label** (*string*) – label used in inst to identify WGS84 geodetic altitude (km, height above surface)

Returns Input pysat.Instrument object modified to include thermal plasma parameters. 'ion_temp' for ion temperature in Kelvin 'e_temp' for electron temperature in Kelvin 'ion_dens' for the total ion density (O+ and H+) 'frac_dens_o' for the fraction of total density that is O+ 'frac_dens_h' for the fraction of total density that is H+

Return type inst

pysat.instruments.pysat_sgp4.**add_hwm_winds_and_ecef_vectors**(*inst*, *glat_label='glat'*, *glong_label='glong'*, *alt_label='alt'*)

Uses HWM (Horizontal Wind Model) model to obtain neutral wind details.

Uses pyglow module to run HWM. Configured to use actual solar parameters to run model.

Example

function added velow modifies the inst object upon every inst.load call inst.custom.add(add_hwm_winds_and_ecef_vectors, 'modify', glat_label='custom_label')

Parameters

- **inst** (pysat.Instrument) – Designed with pysat_sgp4 in mind
- **glat_label** (*string*) – label used in inst to identify WGS84 geodetic latitude (degrees)
- **glong_label** (*string*) – label used in inst to identify WGS84 geodetic longitude (degrees)
- **alt_label** (*string*) – label used in inst to identify WGS84 geodetic altitude (km, height above surface)

Returns Input pysat.Instrument object modified to include HWM winds. 'zonal_wind' for the east/west winds (u in model) in m/s 'meiridional_wind' for the north/south winds (v in model) in m/s 'unit_zonal_wind_ecef_*' (=*x,y,z) is the zonal vector expressed in the ECEF basis 'unit_mer_wind_ecef_*' (=*x,y,z) is the meridional vector expressed in the ECEF basis 'sim_inst_wind_*' (*=x,y,z) is the projection of the total wind vector onto s/c basis

Return type inst

pysat.instruments.pysat_sgp4.**add_igrf**(*inst*, *glat_label='glat'*, *glong_label='glong'*, *alt_label='alt'*)

Uses International Geomagnetic Reference Field (IGRF) model to obtain geomagnetic field values.

Uses pyglow module to run IGRF. Configured to use actual solar parameters to run model.

Example

function added velow modifies the inst object upon every inst.load call inst.custom.add(add_igrf, 'modify', glat_label='custom_label')

Parameters

- **inst** (pysat.Instrument) – Designed with pysat_sgp4 in mind
- **glat_label** (*string*) – label used in inst to identify WGS84 geodetic latitude (degrees)
- **glong_label** (*string*) – label used in inst to identify WGS84 geodetic longitude (degrees)
- **alt_label** (*string*) – label used in inst to identify WGS84 geodetic altitude (km, height above surface)

Returns Input pysat.Instrument object modified to include HWM winds. 'B' total geomagnetic field 'B_east' Geomagnetic field component along east/west directions (+ east) 'B_north' Geomagnetic field component along north/south directions (+ north) 'B_up' Geomagnetic field component along up/down directions (+ up) 'B_ecef_x' Geomagnetic field component along

ECEF x 'B_ecef_y' Geomagnetic field component along ECEF y 'B_ecef_z' Geomagnetic field component along ECEF z

Return type inst

pysat.instruments.pysat_sgp4.**project_ecef_vector_onto_sc**(*inst*, *x_label*, *y_label*, *z_label*, *new_x_label*, *new_y_label*, *new_z_label*, *meta=None*)

Express input vector using s/c attitude directions

x - ram pointing y - generally southward z - generally nadir

Parameters

- **x_label** (*string*) – Label used to get ECEF-X component of vector to be projected
- **y_label** (*string*) – Label used to get ECEF-Y component of vector to be projected
- **z_label** (*string*) – Label used to get ECEF-Z component of vector to be projected
- **new_x_label** (*string*) – Label used to set X component of projected vector
- **new_y_label** (*string*) – Label used to set Y component of projected vector
- **new_z_label** (*string*) – Label used to set Z component of projected vector
- **meta** (*array_like of dicts (None)*) – Dicts contain metadata to be assigned.

5.17 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).

param platform 'rocsat1'

type platform string

param name 'ivm'

type name string

param tag None

type tag string

Note:

- no tag required

Warning:

- Currently no cleaning routine.

5.18 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.

5.19 SuperDARN

SuperDARN data support for grdex files(Alpha Level!)

param platform 'superdarn'

type platform string

param name 'grdex'

type name string

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

type tag string

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.

5.20 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

param platform 'supermag'

type platform string

param name 'magnetometer'

type name string

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

type tag string

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.

5.21 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.

param platform 'timed'

type platform string

param name 'see'

type name string

param tag None

type tag string

param 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.ix[:,'item':'item_end'] If False, then 2D data is stored as a series of DataFrames, indexed by Epoch. data.ix[0, 'item']

type flatten_twod bool (True)

Note:

- no tag required

Warning:

- Currently no cleaning routine.

CHAPTER 6

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 module corresponds to the combination 'platform_name' provided when initializing a pysat instrument object. The module should be placed in the pysat instruments directory or in the user specified location (via mechanism to be added) for automatic discovery. A compatible module may also be supplied directly to pysat.Instrument(inst_module=input module) if it also contains attributes platform and name.

Some data repositories have pysat templates prepared to assist in integrating a new instrument. See Supported Templates for more.

Three functions are required by pysat for operation, with supporting information for testing:

6.1 List Files

Pysat maintains a list of files to enable data management functionality. It needs a pandas Series of filenames indexed by time. Pysat expects the module method platform_name.list_files to be:

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

where tag indicates a specific subset of the available data from cnofs_vefi.

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

Pysat will store data in pysat_data_dir/platform/name/tag, helpfully provided in data_path, where pysat_data_dir is specified by user in pysat settings.

pysat.Files.from_os is a convenience constructor provided for filenames that include time information in the filename and utilize a constant field width. The location and format of the time information is specified using standard python formatting and keywords year, month, day, hour, minute, second. A complete list_files routine could be as simple as

```
def list_files(tag=None, data_path=None):
    return pysat.Files.from_os(data_path=data_path,
                   format_str='cindi-{year:4d}{day:03d}-ivm.hdf')
```

6.2 Load Data

Loading is a fundamental pysat activity, this routine enables the user to consider loading a hidden implementation 'detail'.

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

- 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.
- pysat.utils.create_datetime_index provides for quick generation of an appropriate datetime index for irregulary sampled data set with gaps
- pysat meta object obtained from pysat.Meta(). Use pandas DataFrame indexed by name with columns for 'units' and 'long_name'. Additional arbitrary columns allowed. See pysat.Meta for more information on creating the initial metadata.
- If metadata is already stored with the file, creating the Meta object is 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 is easier to fill out a text file. A convenience function is provided for this situation. See pysat.Meta.from_csv for more information.

6.3 Download Data

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.

6.4 Optional Routines

Initialize

Initialize any specific instrument info. Runs once.

```
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.

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 Data

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.

6.5 Testing Support

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

The following attributes must be defined.

- platform : platform name
- name : instrument name
- tags : dictionary of all tags supported by routine with a description
- sat_ids : dictionaary of sat_ids, with a list of tags supported by each id
- test_dates : dictionary of sat_ids, containing dictionary of tags, with date to download for testing

Note that platform and name must match those used to name the file, platform_name.py

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

CHAPTER 7

Supported Data Templates

7.1 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 be added via

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

Using the CDAWeb template the equivalent action is

```
download = functools.partial(nasa_cdaweb_methods.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.

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

pysat.instruments.nasa_cdaweb_methods.**download**(*supported_tags*, *date_array*, *tag*, *sat_id*, *ftp_site='cdaweb.gsfc.nasa.gov'*, *data_path=None*, *user=None*, *password=None*, *fake_daily_files_from_monthly=False*)

Routine to download NASA CDAWeb CDF data.

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)*) – tag or None (default=None)
- **sat_id** (*(str or NoneType)*) – satellite id or None (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)
- **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_methods.download,
                             supported_tags=supported_tags)
```

pysat.instruments.nasa_cdaweb_methods.**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.

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 tags supported by list_files routine. Values are the default format_str values for key. (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.

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_methods.list_files,
                               supported_tags=supported_tags)

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

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

Load NASA CDAWeb CDF files.

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.

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
```

CHAPTER 8

API

8.1 Instrument

class pysat.**Instrument**(*platform=None*, *name=None*, *tag=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*, *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.

Parameters

- **platform** (*string*) – name of platform/satellite.
- **name** (*string*) – name of instrument.
- **tag** (*string, optional*) – identifies particular subset of instrument data.
- **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.
- **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.
- **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
- **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.
- **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
pandas.DataFrame – loaded science data

date
pandas.datetime – date for loaded data

yr
int – year for loaded data

bounds
(datetime/filename/None, datetime/filename/None) – bounds for loading data, supply array_like for a season with gaps

doy
int – day of year for loaded data

files
pysat.Files – interface to instrument files

meta
pysat.Meta – interface to instrument metadata, similar to netCDF 1.6

orbits
pysat.Orbits – interface to extracting data orbit-by-orbit

custom
pysat.Custom – interface to instrument nano-kernel

kwargs
dictionary – keyword arguments passed to instrument loading routine

---

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'])
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('cosmic2013',
                          '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

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])
```

copy()
Deep copy of the entire Instrument object.

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

Parameters

- **start** (*pandas.datetime*) – start date to download data
- **stop** (*pandas.datetime*) – 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
- ****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 metadata 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

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.

---

to_netcdf4 (*fname=None*, *base_instrument=None*, *epoch_name='Epoch'*, *zlib=False*, *complevel=4*, *shuffle=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.

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.

8.2 Constellation

class pysat.**Constellation**(*instruments=None*, *name=None*)

Manage and analyze data from multiple pysat Instruments.

Created as part of a Spring 2018 UTDesign project.

Constructs a Constellation given a list of instruments or the name of a file with a pre-defined constellation.

Parameters

- **instruments** (*list*) – a list of pysat Instruments
- **name** (*string*) – Name of a file in pysat/constellations containing a list of instruments.

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.

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.

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

8.3 Custom

class pysat.**Custom**
Applies a queue of functions when instrument.load called.

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.add(custom_func, 'modify', opt_param1=True)

def custom_func2(inst, opt_param1=False, opt_param2=False):
    return data_to_be_added
instrument.custom.add(custom_func2, 'add', opt_param2=True)
instrument.load(date=date)
print(instrument['data_to_be_added'])
```

See also:

Custom.add

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.

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)

---

clear()
Clear custom function list.

8.4 Files

class pysat.**Files**(*sat*, *manual_org=False*, *directory_format=None*, *update_files=False*, *file_format=None*, *write_to_disk=True*)
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
string – path to .pysat directory in user home

start_date
datetime – date of first file, used as default start bound for instrument object

stop_date
datetime – date of last file, used as default stop bound for instrument object

data_path
string – path to the directory containing instrument files, top_dir/platform/name/tag/

manual_org
bool – if True, then Files will look directly in pysat data directory for data files and will not use /platform/name/tag

update_files
bool – updates files on instantiation if True

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()
```

classmethod from_os(*data_path=None*, *format_str=None*, *two_digit_year_break=None*)
Produces a list of files and and formats it for Files class.

Requires fixed_width 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', 'min', 'sec', '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.

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).

8.5 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'*)
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
pandas.DataFrame – index is variable standard name, 'units', 'long_name', and other defaults are also stored along with additional user provided labels.

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

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’} 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

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

8.6 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)
```

(continues on next page)

(continued from previous page)

```
# 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.

8.7 Seasonal Analysis

8.7.1 Occurrence Probability

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.

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.

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.

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.

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.

8.7.2 Average

pysat.ssnl.avg.**mean_by_day**(*inst*, *data_label*)
Mean of data_label by day over Instrument.bounds

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

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

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.**median2D**(*const*, *bin1*, *label1*, *bin2*, *label2*, *data_label*, *returnData=False*)
Return a 2D average of data_label over a season and label1, label2.

Parameters

- **const** (*Constellation or Instrument*) –
- **bin#** (*[min, max, number of bins]*) –
- **label#** (*string*) – identifies data product for bin#
- **data_label** (*list-like*) – contains strings identifying data product(s) to be averaged

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

8.7.3 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.

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.*

8.8 Utilities

pysat.utils.**computational_form**(*data*)
Input Series of numbers, Series, or DataFrames repackaged for calculation.

Parameters **data** (*pandas.Series*) – Series of numbers, Series, DataFrames

Returns repacked data, aligned by indices, ready for calculation

Return type pandas.Series, DataFrame, or Panel

pysat.utils.**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.**getyrdoy**(*date*)
Return a tuple of year, day of year for a supplied datetime object.

pysat.utils.**load_netcdf4**(*fnames=None*, *strict_meta=False*, *file_format=None*, *epoch_name='Epoch'*, *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'*)
Load netCDF-3/4 file produced by pysat.

Parameters

- **fnames** (*string or array_like of strings*) – filenames to load
- **strict_meta** (*boolean*) – check if metadata across fnames is the same
- **file_format** (*string*) – file_format keyword passed to netCDF4 routine NETCDF3_CLASSIC, NETCDF3_64BIT, NETCDF4_CLASSIC, and NETCDF4

Returns

- **out** (*pandas.core.frame.DataFrame*) – DataFrame output
- **mdata** (*pysat._meta.Meta*) – Meta data

pysat.utils.**nan_circmean**(*samples*, *high=6.283185307179586*, *low=0.0*, *axis=None*)
NaN insensitive version of scipy's circular mean routine

Parameters

- **samples** (*array_like*) – Input array
- **low** (*float or int*) – Lower boundary for circular standard deviation range (default=0)
- **high** (*float or int*) – Upper boundary for circular standard deviation range (default=2 pi)
- **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.**nan_circstd**(*samples*, *high=6.283185307179586*, *low=0.0*, *axis=None*)
NaN insensitive version of scipy's circular standard deviation routine

Parameters

- **samples** (*array_like*) – Input array
- **low** (*float or int*) – Lower boundary for circular standard deviation range (default=0)
- **high** (*float or int*) – Upper boundary for circular standard deviation range (default=2 pi)
- **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

pysat.utils.**season_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.**set_data_dir**(*path=None*, *store=None*)

Set the top level directory pysat uses to look for data and reload.

Parameters

- **path** (*string*) – valid path to directory pysat uses to look for data
- **store** (*bool*) – if True, store data directory for future runs

CHAPTER 9

Contributing

Bug reports, feature suggestions and other contributions are greatly appreciated! Pysat is a community-driven project and welcomes both feedback and contributions.

CHAPTER 10

Short version

- Submit bug reports and feature requests at GitHub
- Make pull requests to the `develop` branch

CHAPTER 11

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

CHAPTER 12

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 :)

CHAPTER 13

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.

13.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code, just make a pull request.

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.

Python Module Index

p

Index

U

V

Y