

---

# **pysat Documentation**

***Release 0.2.0***

**Russell Stoneback**

April 28, 2015



## CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
<b>3</b>	<b>Tutorial</b>	<b>5</b>
3.1	Basics . . . . .	5
3.2	Custom Functions . . . . .	6
3.3	Iteration . . . . .	7
3.4	Orbit Support . . . . .	8
<b>4</b>	<b>Adding a New Instrument</b>	<b>9</b>
4.1	List Files . . . . .	9
4.2	Load Data . . . . .	9
4.3	Download Data . . . . .	10
4.4	Optional Routines . . . . .	10
<b>5</b>	<b>Examples</b>	<b>13</b>
5.1	Seasonal Occurrence by Orbit . . . . .	13
5.2	Orbit-by-Orbit Plots . . . . .	15
<b>6</b>	<b>API</b>	<b>19</b>
6.1	Instrument . . . . .	19
6.2	Custom . . . . .	22
6.3	Files . . . . .	24
6.4	Meta . . . . .	26
6.5	Orbits . . . . .	27
6.6	Utilities . . . . .	29
<b>7</b>	<b>Supported Instruments</b>	<b>31</b>
7.1	C/NOFS VEFI . . . . .	31
7.2	C/NOFS IVM . . . . .	31
7.3	COSMIC 2013 GPS . . . . .	31
7.4	COSMIC GPS . . . . .	32
	<b>Python Module Index</b>	<b>33</b>
	<b>Index</b>	<b>35</b>



## **INTRODUCTION**

The Python Science Analysis Toolkit (pysat) is a package providing a simple and flexible interface for downloading, loading, cleaning, managing, processing, and analyzing scientific measurements. Though pysat was initially designed for in-situ satellite based measurements it aims to support all instruments in space science.

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. 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.

This instrument independence is achieved through a `pysat.Instrument` object which provides a layer of separation between the user and the particulars of any given science data set. Loading data for any instrument becomes the same process with the same result, `instrument.load()` produces data in `instrument.data`. Behind the scenes different load functions will actually be called for different instruments, as appropriate, but this is now a hidden implementation detail.

Frequently science data sets don't have all of the parameters needed for a given analysis. If an analysis routine is to be truly instrument independent then there needs to be a mechanism to get custom data into a routine without having to modify the routine itself. Thus, a nano-kernel is attached to the `pysat.Instrument` object. Upon each `instrument.load` call a queue of user selected custom functions are applied before the data is available in `instrument.data`. The instrument object is 'set and forget', regardless of location the data available in `instrument.data` will be properly processed.

The final component for instrument independence requires the Python Data Analysis Library (pandas), the underlying data object, capable of handling 1D through nD data in a single structure. Pandas data structures are also indexed, thus math operations between two arrays A and B are aligned before the operation. Measurements across platforms are rarely always at the same time, thus pandas also significantly reduces the overhead while increasing the rigor of inter-platform comparisons.

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



## INSTALLATION

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

- CDF: <http://cdf.gsfc.nasa.gov>
- netCDF: <http://www.unidata.ucar.edu/software/netcdf/>

**Pysat itself may be installed via** `pip install pysat`

**Pandas may be obtained similarly,** `pip install pandas`

**To get the forked pandas that accommodates pandas Series and DataFrames within each cell of a Series use**

`pip install git+https://github.com/rstoneback/pandas.git`

The forked pandas is required for higher dimensional data sets. A pull-request is planned.

**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)`





## TUTORIAL

### 3.1 Basics

The core functionality of pysat is exposed through the `pysat.Instrument` object. To work with Magnetometer data from the Vector Electric Field Instrument onboard the Communications/Navigation Outage Forecasting System (C/NOFS) begin with

```
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. Let’s download some data,

```
import pysat
# define date range to download data and download
start = pysat.datetime(2009,5,9)
stop = pysat.datetime(2009,5,12)
vefi.download(start, stop)
```

The data is downloaded to `pysat_data_dir/platform/name/tag/`, in this case `pysat_data_dir/cnofs/vefi/dc_b/`. `pysat_data_dir` should have previously been set using

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

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

```
vefi.load(2009,1)
vefi.load(date=start)
vefi.load(fname=fname)
```

The pandas DataFrame holding the data is available in `.data`. Convenience access is also available at the instrument level.

```
# all data
print vefi.data
# particular magnetic component
print vefi.data.dB_mer
# alternative access
print vefi['dB_mer']
# slicing
print vefi[0:10, 'dB_mer']
# slicing by date time
print vefi[start:stop, 'dB_mer']
```

See `pysat.Instrument` for more.

Metadata is also stored along with the main science data.

```
# all metadata
print vefi.meta.data
# dB_mer metadata
print 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, measurements from the Ion Velocity Meter and profiles of electron density from COSMIC

```
# 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 custom functions may be attached to the Instrument object. Each function is run automatically when new data is loaded.

### 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.

## 3.3 Iteration

The whole VEFI data set may be iterated over on a daily basis

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

Each loop of the python for 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. It is equivalent to

```
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()
```

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
```

Bounds may be set to control the dates covered by the iteration,

```
# continuous season
vefi.bounds = (start, stop)
```

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

## 3.4 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.Orbit` for latest inputs.

```
info = {'index':'mlt', 'kind':'local time'}
ivm = pysat.Instrument(platform='cnofs', name='ivm', tag='',
                        clean_level='clean', orbit_info=info)
start = [pd.datetime(2009,1,1), pd.datetime(2010,1,1)]
stop = [pd.datetime(2009,4,1), pd.datetime(2010,4,1)]
ivm.bounds = (start, stop)
for ivm in ivm.orbits:
    print 'next available orbit ', ivm.data
```

## 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`.

Three functions are required:

### 4.1 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')
```

### 4.2 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 irregularly 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.

## 4.3 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.

## 4.4 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.





## 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.

### 5.1 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 (North/South) 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
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)

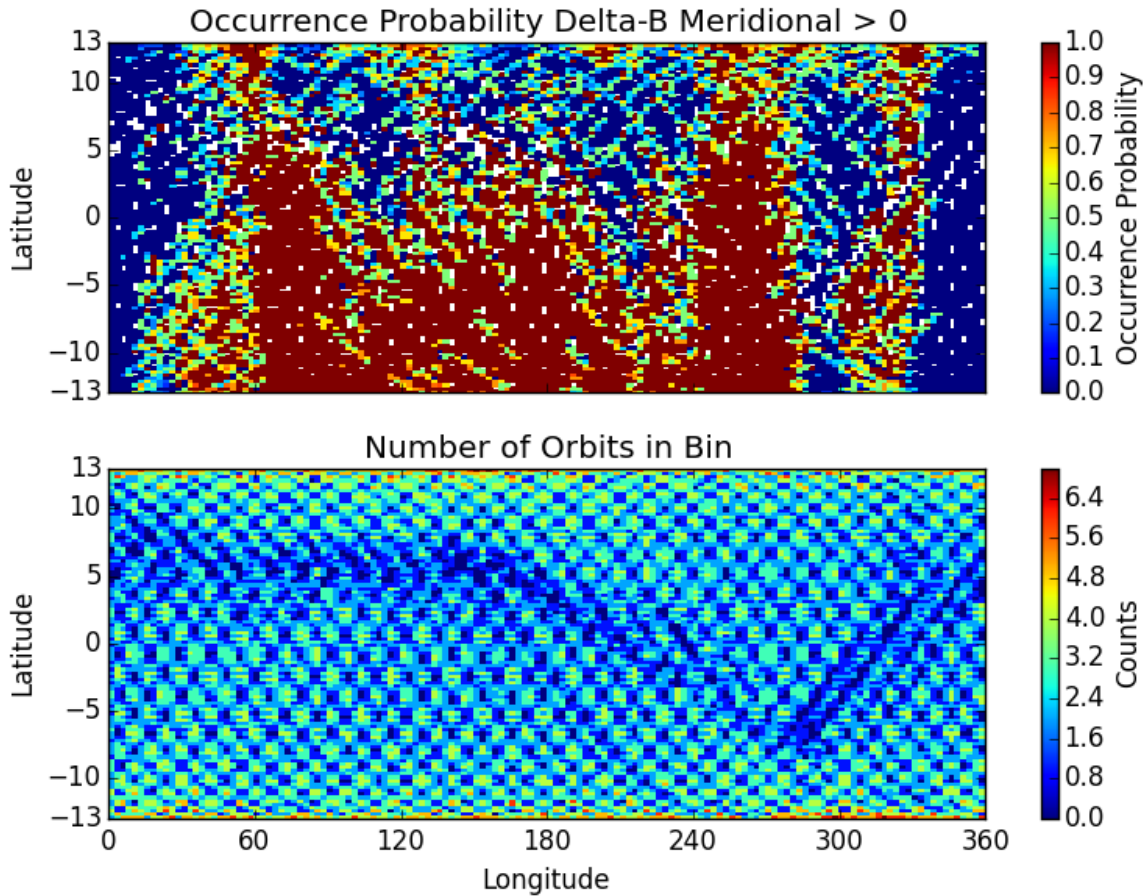
# perform occurrence probability calculation
# any data added by custom functions is available within routine below
ans = pysat.ssnl.occure_prob.by_orbit2D(vefi, [0,360,144], 'longitude',
    [-13,13,104], 'latitude', ['dB_mer'], [0.], returnBins=True)

# 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['binx'], ans['biny'], 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['biny'][0],ans['biny'][-1]))
plt.colorbar(im,ax=axarr[0], label='Occurrence Probability')

im=axarr[1].pcolor(ans['binx'], ans['biny'],ans['count'])
axarr[1].set_xlabel('Longitude')
axarr[1].set_xticks((0,60,120,180,240,300,360))
axarr[1].set_xlim((ans['binx'][0],ans['binx'][-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



## 5.2 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
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))

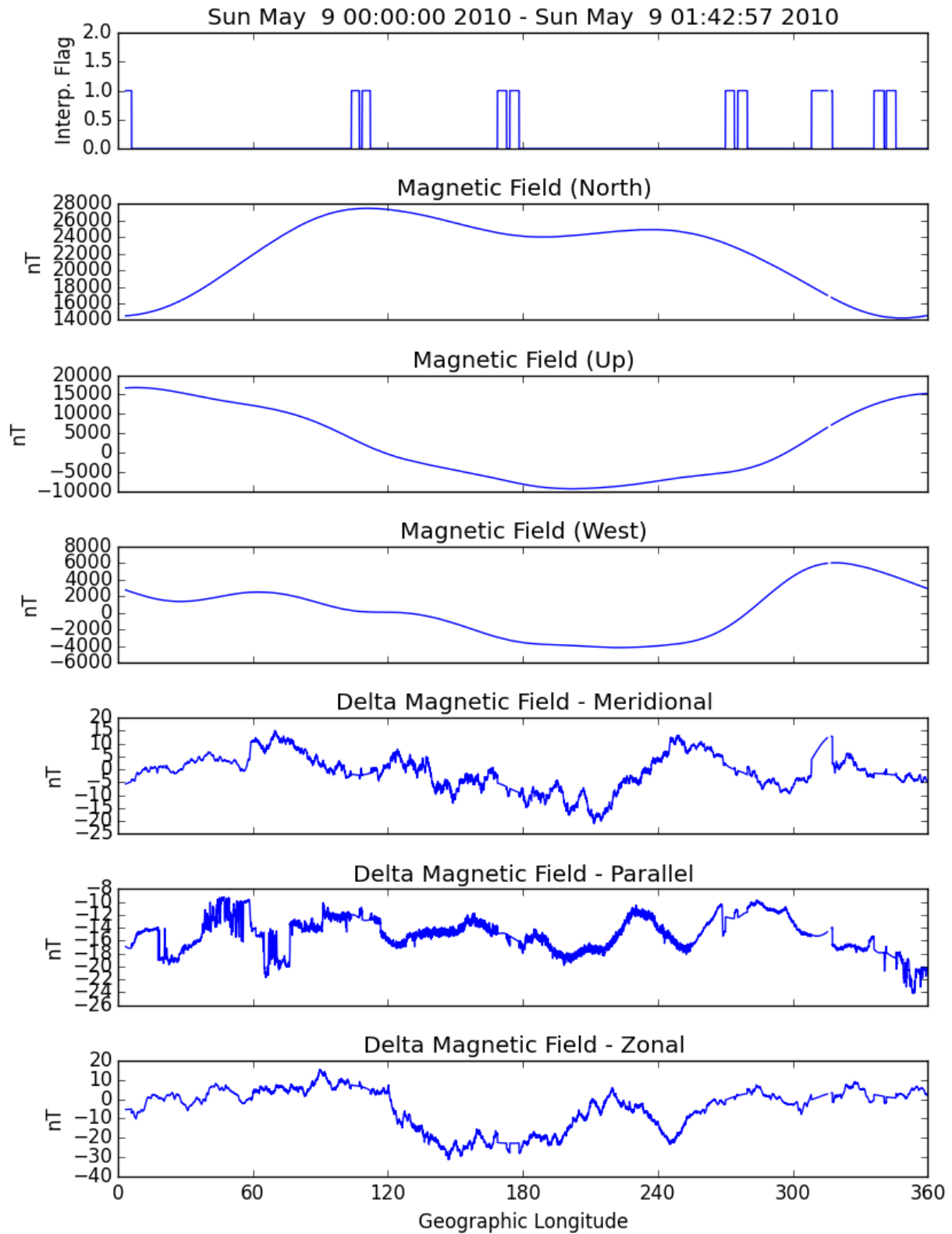
    ax[0].plot(vefi['longitude'], vefi['B_flag'])
    ax[0].set_title( vefi.data.index[0].ctime() + ' - ' +
                    vefi.data.index[-1].ctime() )
    ax[0].set_ylabel('Interp. Flag')
    ax[0].set_ylim((0,2))

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

Output





## 6.1 Instrument

`class pysat.Instrument` (*platform=None, name=None, tag=None, clean\_level='clean', update\_files=False, pad=None, orbit\_info=None, inst\_module=None, \*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.
- **inst\_module** (*module, optional*) – Provide instrument module directly (takes precedence over platform/name)
- **clean\_level** (*{'clean','dusty','dirty','none'}, optional*) – level of data quality
- **pad** (*pandas.DateOffset, or dictionary, optional*) – length of time to pad the beginning 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.
- **update\_files** (*boolean, optional*) – if True, query filesystem for instrument files and store. `files.get_new()` will return no files after this call until additional files are added.

### 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, `platform_name.load`**Notes**

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)
```



`__getitem__` (*key*)

Convenience notation for accessing data; obtain `inst.data.name` using `inst['name']`.

### Examples

**By position :** `inst[row index, 'name']`

**Slicing by row :** `inst[row1:row2, 'name']`

**By Date :** `inst[datetime, 'name']`

**Slicing by date [(inclusive)] :** `inst[datetime1:datetime2, 'name']`

**Slicing by name and row/date :** `inst[datetime1:datetime1, 'name1':'name2']`

`__iter__` ()

Iterates the instrument object by loading subsequent days or files as appropriate.

Limits of iteration, and iteration type (date/file) set by *bounds* attribute.

`__setitem__` (*key, new*)

Convenience method for adding data to instrument.

### Examples

**Simple Assignment :** `inst['name'] = newData`

**Assignment with Metadata :** `inst['name'] = {'data':new_data, 'long_name':long_name, 'units':units}`

---

**Note:** If no metadata provided and if metadata for 'name' not already stored then default meta information is also added, `long_name = 'name'`, and `units = ''`.

---

## 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

---

`copy` ()

Deep copy of the entire Instrument object.

`download` (*start, stop, user=None, password=None*)

Download data for given Instrument object.

### Parameters

- **start** (*pandas.datetime*) – start date to download data
- **stop** (*pandas.datetime*) – stop date to download data
- **user** (*string*) – username, if required by instrument data archive

- **password** (*string*) – password, if required by instrument data archive

**load** (*yr=None, doy=None, date=None, fname=None, fid=None, verifyPad=False*)

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`.

#### Keyword Arguments

- **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)

**next** ()

Manually iterate through the data loaded in satellite 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** ()

Manually iterate backwards through the data loaded in satellite 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\_netcdf3** (*fname=None*)

Stores loaded data into a netCDF3 64-bit file.

Stores 1-D data along dimension 'time' - the date time index. Stores object data (dataframes within dataframe) separately:

**The name of the object data is used to prepend extra variable** dimensions within netCDF, `key_2`, `key_3`, first dimension time

The index organizing the data stored as `key_sample_index` from `_netcdf3` uses this naming scheme to reconstruct data structure

**The datetime index is stored as 'UNIX time'. netCDF-3 doesn't support** 64-bit integers so it is stored as a 64-bit float. This results in a loss of datetime precision when converted back to datetime index up to hundreds of nanoseconds. Use netCDF4 if this is a problem.

All attributes attached to instrument meta are written to netCDF attrs.

## 6.2 Custom

**class** `pysat.Custom`

Applies a queue of functions when `instrument.load` called.

Nano-kernel functionality enables instrument objects that are ‘fire 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

### Notes

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).

### Notes

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.

## 6.3 Files

**class** `pysat.Files` (*sat*)  
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/`

### Notes

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)
# 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, knowledge not written to disk.
vefi.files.refresh()

# search pysat appropriate directory for files and store new list
vefi.files.refresh(store=True)
# running get_new will now return an empty list until
# additional files are introduced
```

**classmethod from\_os** (*data\_path=None, format\_str=None, two\_digit\_year\_break=None*)

Produces a list of files and formats it for Files class.

#### 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\_string** (*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.
- **two\_digit\_year\_break** (*int*) – If filenames only store two digits for the year, then ‘1900’ will be added for years  $\geq$  `two_digit_year_break`, and ‘2000’ will be added for years  $<$  `two_digit_year_break`.

#### Notes

Does not produce a Files instance, but the proper output from `instrument_module.list_files` method.

**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

#### Notes

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 all new files since last time list was stored.

pysat stores filenames in the user\_home/.pysat directory. Returns a list of all new filenames since the last store. Filenames are stored if update\_files is True at instrument object level and if files.refresh(store=True) is called.

#### Returns

- *pandas Series of filenames*
- *False if no filenames*

**refresh** (*store=False*)

Refresh loaded instrument filelist by searching filesystem.

Searches pysat provided path, pysat\_data\_dir/platform/name/tag/, where pysat\_data\_dir is set by pysat.utils.set\_data\_dir(path=path).

**Parameters** *store* (*boolean*) – set True to store loaded file names into .pysat directory

## 6.4 Meta

**class** pysat.**Meta** (*metadata=None*)

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.

#### data

*pandas.DataFrame*

index is variable standard name, 'units' and 'long\_name' are also stored along with additional user provided labels.

**\_\_getitem\_\_** (*key*)

Convenience method for obtaining metadata.

Maps to pandas DataFrame.ix method.

#### Examples

```
print meta['name']
```

**\_\_setitem\_\_** (*name, value*)

Convenience method for adding metadata.

#### Examples

```
meta = pysat.Meta()
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}
```

**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 separator for supplied csv filename

---

**Note:** column names must include at least ['name', 'long\_name', 'units'], assumed if col\_names is None.

---

**classmethod from\_dict** ()

not implemented yet, load metadata from dict of items/list types

**classmethod from\_nc** ()

not implemented yet, load metadata from netCDF

**replace** (*metadata=None*)

Replace stored metadata with input data.

**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.

## 6.5 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 determining orbit breaks
- **kind** (*{'local time', 'longitude', 'polar'}*) – 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
- **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

#### Notes

class should not be called directly by the user, use the interface provided by inst.orbits where inst = pysat.Instrument()

## Examples

```
info = {'index':'longitude', 'kind':'longitude'}
vefi = pysat.Instrument(platform='cnofs', name='vefi', tag='dc_b',
                        clean_level=None, orbit_info=info)
start = pysat.datetime(2009,1,1)
stop = pysat.datetime(2009,1,10)
vefi.load(date=start)
vefi.bounds(start, stop)

# iterate over orbits
for vefi in vefi.orbits:
    print 'Next available orbit ', vefi['dB_mer']

# load fifth orbit of first day
vefi.load(date=start)
vefi.orbits[5]

# less convenient load
vefi.orbits.load(5)

# manually iterate orbit
vefi.orbits.next()
# backwards
vefi.orbits.prev()
```

**\_\_getitem\_\_** (*key*)

Enable convenience notation for loading orbit into parent object.

## Example

```
inst.load(date=date)
inst.orbits[4]
print 'Orbit data ', inst.data
```

---

**Note:** A day of data must already be loaded.

---

**\_\_iter\_\_** ()

Support iteration by orbit.

For each iteration the next available orbit is loaded into inst.data.

## Example

```
for inst in inst.orbits:
    print 'next available orbit ', inst.data
```

**load** (*orbit=None*)

Load a particular orbit into .data for loaded day.

**Parameters = orbit number, 1 indexed** (*orbit*) –

---

**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 next 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.

---

## 6.6 Utilities

`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.getyrday` (date)  
Return a tuple of year, day of year for a supplied datetime object.

`pysat.utils.load_netcdf3` (fnames=None, strict\_meta=False, index\_label=None, unix\_time=False, \*\*kwargs)  
Load netCDF-3 file produced by pysat.

### Parameters

- **fnames** (string or array\_like of strings) – filenames to load
- **strict\_meta** (boolean) – check if metadata across filenames is the same
- **index\_label** (string) – name of data to be used as DataFrame index
- **unix\_time** (boolean) – True if index\_label refers to UNIX time

`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)  
set the top level directory pysat uses to look for data.



## SUPPORTED INSTRUMENTS

### 7.1 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 tag**

**type tag** {'dc\_b'}

#### Notes

- tag = 'dc\_b': 1 second DC magnetometer data

**Warning:**

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

### 7.2 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 Natural Dynamics Investigation (CINDI). Downloads data from the NASA Coordinated Data Analysis Web (CDAWeb) in CDF format.

**param tag** No tags 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.

### 7.3 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 Analysis 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

#### Notes

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

#### Warning:

- Routine was not produced by COSMIC team

## 7.4 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 Analysis and Archival Center.

#### Notes

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

#### Warning:

- Routine was not produced by COSMIC team

**p**

`pysat.instruments.cnofs_ivm`, [31](#)  
`pysat.instruments.cnofs_vefi`, [31](#)  
`pysat.instruments.cosmic2013_gps`, [31](#)  
`pysat.instruments.cosmic_gps`, [32](#)  
`pysat.utils`, [29](#)



## Symbols

`__getitem__()` (pysat.Instrument method), 20  
`__getitem__()` (pysat.Meta method), 26  
`__getitem__()` (pysat.Orbits method), 28  
`__iter__()` (pysat.Instrument method), 21  
`__iter__()` (pysat.Orbits method), 28  
`__setitem__()` (pysat.Instrument method), 21  
`__setitem__()` (pysat.Meta method), 26

## A

`add()` (pysat.Custom method), 23

## B

`base_path` (Files attribute), 24  
`bounds` (Instrument attribute), 19  
`bounds` (pysat.Instrument attribute), 21

## C

`clear()` (pysat.Custom method), 23  
`copy()` (pysat.Instrument method), 21  
`create_datetime_index()` (in module pysat.utils), 29  
`Custom` (class in pysat), 22  
`custom` (Instrument attribute), 20

## D

`data` (Instrument attribute), 19  
`data` (Meta attribute), 26  
`data_path` (Files attribute), 24  
`date` (Instrument attribute), 19  
`download()` (pysat.Instrument method), 21  
`doy` (Instrument attribute), 19

## F

`Files` (class in pysat), 24  
`files` (Instrument attribute), 20  
`from_csv()` (pysat.Meta class method), 26  
`from_dict()` (pysat.Meta class method), 27  
`from_nc()` (pysat.Meta class method), 27  
`from_os()` (pysat.Files class method), 25

## G

`get_file_array()` (pysat.Files method), 25

`get_index()` (pysat.Files method), 25  
`get_new()` (pysat.Files method), 25  
`getyrday()` (in module pysat.utils), 29

## I

`Instrument` (class in pysat), 19

## K

`kwargs` (Instrument attribute), 20

## L

`load()` (pysat.Instrument method), 22  
`load()` (pysat.Orbits method), 28  
`load_netcdf3()` (in module pysat.utils), 29

## M

`Meta` (class in pysat), 26  
`meta` (Instrument attribute), 20

## N

`next()` (pysat.Instrument method), 22  
`next()` (pysat.Orbits method), 28

## O

`Orbits` (class in pysat), 27  
`orbits` (Instrument attribute), 20

## P

`prev()` (pysat.Instrument method), 22  
`prev()` (pysat.Orbits method), 29  
`pysat.instruments.cnofs_ivm` (module), 31  
`pysat.instruments.cnofs_vefi` (module), 31  
`pysat.instruments.cosmic2013_gps` (module), 31  
`pysat.instruments.cosmic_gps` (module), 32  
`pysat.utils` (module), 29

## R

`refresh()` (pysat.Files method), 26  
`replace()` (pysat.Meta method), 27

## S

`season_date_range()` (in module pysat.utils), 29

`set_data_dir()` (in module `pysat.utils`), [29](#)  
`start_date` (Files attribute), [24](#)  
`stop_date` (Files attribute), [24](#)

## T

`to_netcdf3()` (`pysat.Instrument` method), [22](#)

## Y

`yr` (Instrument attribute), [19](#)