

---

# **Python Satellite Analysis Toolkit (pysat)**

## **Documentation**

***Release 0.3.3***

**Russell Stoneback**

January 14, 2016



## 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 . . . . .	8
3.3	Time Series Analysis . . . . .	11
3.4	Iteration . . . . .	11
3.5	Orbit Support . . . . .	13
3.6	Iteration and Instrument Independent Analysis . . . . .	14
3.7	Summary Flow Charts . . . . .	16
<b>4</b>	<b>Examples</b>	<b>17</b>
4.1	Seasonal Occurrence by Orbit . . . . .	17
4.2	Orbit-by-Orbit Plots . . . . .	19
4.3	Seasonal Averaging of Ion Drifts and Density Profiles . . . . .	21
<b>5</b>	<b>Supported Instruments</b>	<b>27</b>
5.1	C/NOFS VEFI . . . . .	27
5.2	C/NOFS IVM . . . . .	27
5.3	COSMIC 2013 GPS . . . . .	28
5.4	COSMIC GPS . . . . .	28
5.5	OMNI . . . . .	29
5.6	SuperDARN . . . . .	29
<b>6</b>	<b>Adding a New Instrument</b>	<b>31</b>
6.1	List Files . . . . .	31
6.2	Load Data . . . . .	31
6.3	Download Data . . . . .	32
6.4	Optional Routines . . . . .	32
<b>7</b>	<b>API</b>	<b>35</b>
7.1	Instrument . . . . .	35
7.2	Custom . . . . .	40
7.3	Files . . . . .	41
7.4	Meta . . . . .	43
7.5	Orbits . . . . .	44
7.6	Seasonal Analysis . . . . .	46
7.7	Utilities . . . . .	49

<b>Python Module Index</b>	<b>51</b>
<b>Index</b>	<b>53</b>

## **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 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 codes, an overview of adding new instruments to pysat, and an API reference.



## INSTALLATION

### Starting from scratch

---

Python and associated packages for science are freely available. Convenient science python package setups are available from [Enthought](#) and [Contiuum 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.

- **CDF Library from NASA (<http://cdf.gsfc.nasa.gov>)**
  - [Mac OS X Installer](#)
- SpacePy

```
pip install spacepy
```

### 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



## 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
print vefi.data
# particular magnetic component
print vefi.data.dB_mer

# Convenience access
print vefi['dB_mer']
# slicing
print vefi[0:10, 'dB_mer']
# slicing by date time
print 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
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, 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)
# 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)]
    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 (pending).

### 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]
            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
```



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.

Let’s go back an orbit.

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,

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,

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.

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

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

        if isinstance(data.iloc[0], pandas.DataFrame):
            data_panel = pandas.Panel.from_dict(dict([(i, data.iloc[i]) for i in xrange(len(data))]))
            mean_val[date] = data_panel.abs().mean(axis=0, skipna=True)
        elif isinstance(data.iloc[0], pandas.Series):
            data_frame = pandas.DataFrame(data.tolist())
            data_frame.index = data.index
            mean_val[date] = data_frame.abs().mean(axis=0, skipna=True)
        else:
            mean_val[date] = inst[data_label].abs().mean(axis=0, skipna=True)

del iterator
return mean_val

```

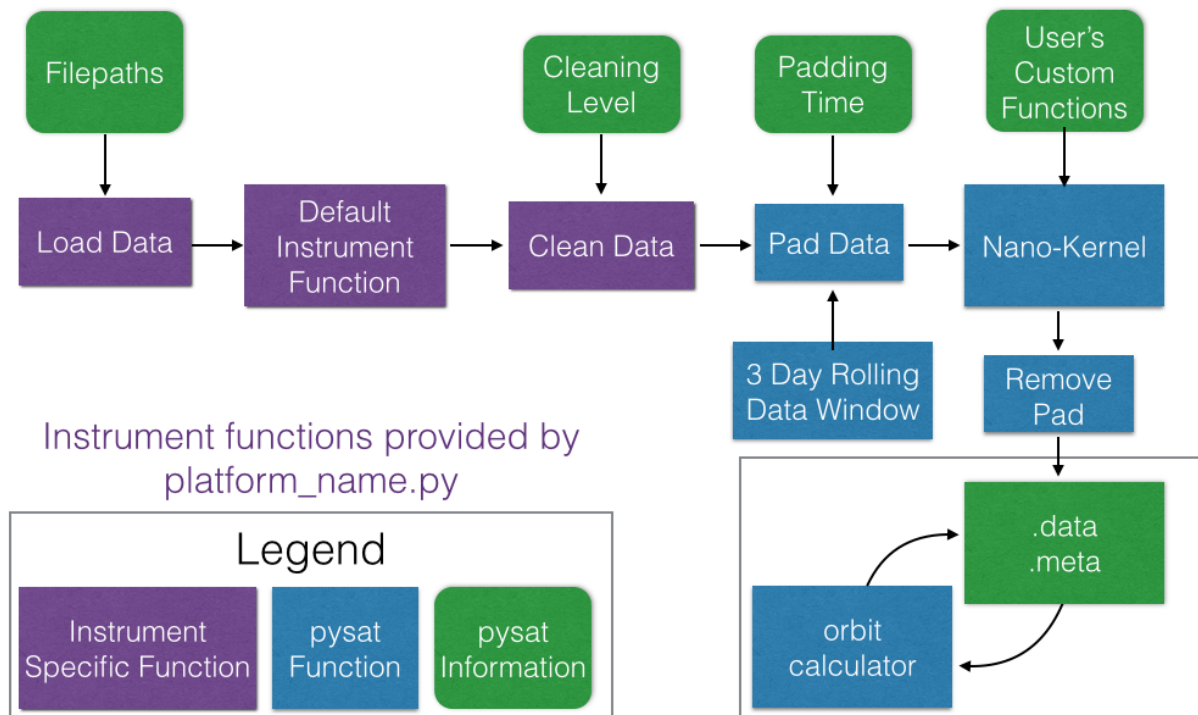
The addition of a few more lines to the `daily_mean` function adds support for averages by orbit, or by day, for any platform with data 3D or less. The date issue and the type of iteration are solved with simple if else checks. From a practical perspective, the code doesn't really deviate from the first 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

## Pysat Loading Process



## 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.

### 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
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)

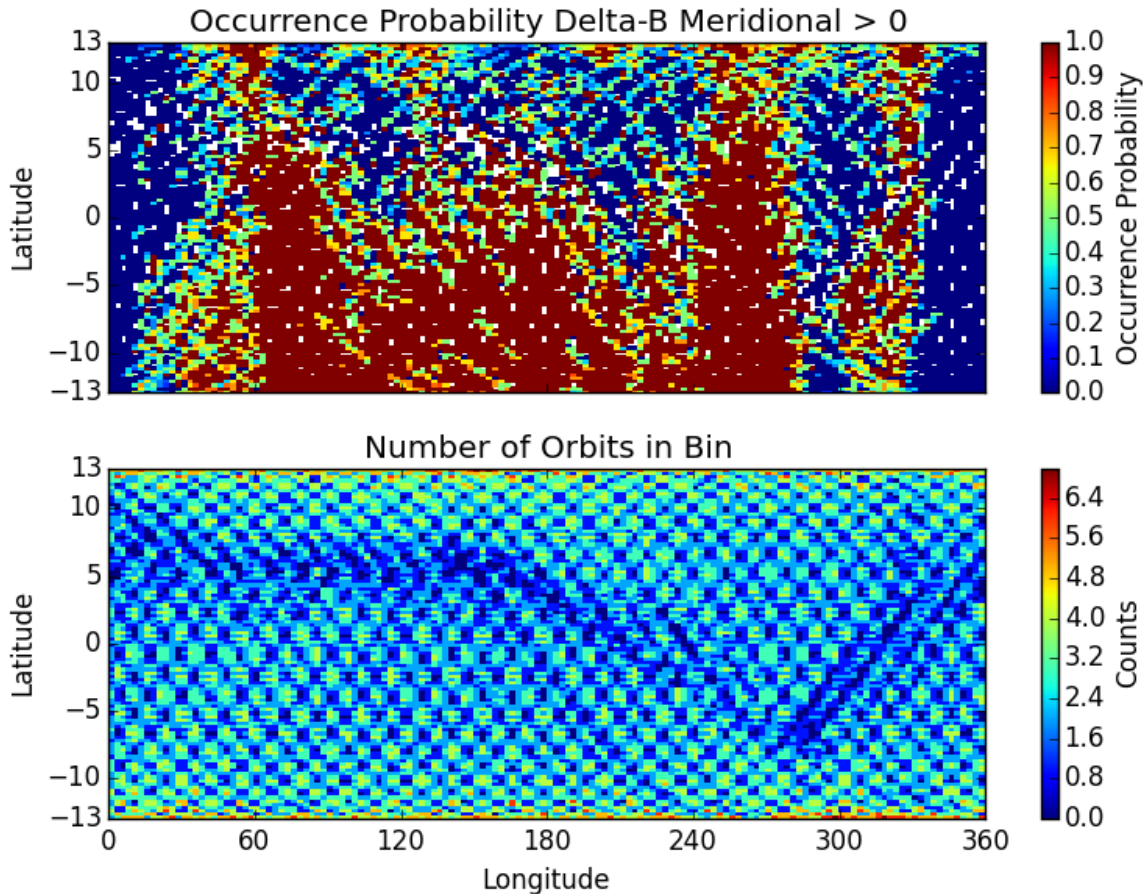
# perform occurrence probability calculation
# any data added by custom functions is available within routine below
ans = pysat.ssnl.occure_prob.by_orbit2D(vefi, [0,360,144], 'longitude',
    [-13,13,104], 'latitude', ['dB_mer'], [0.], returnBins=True)

# 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['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
```

```
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)

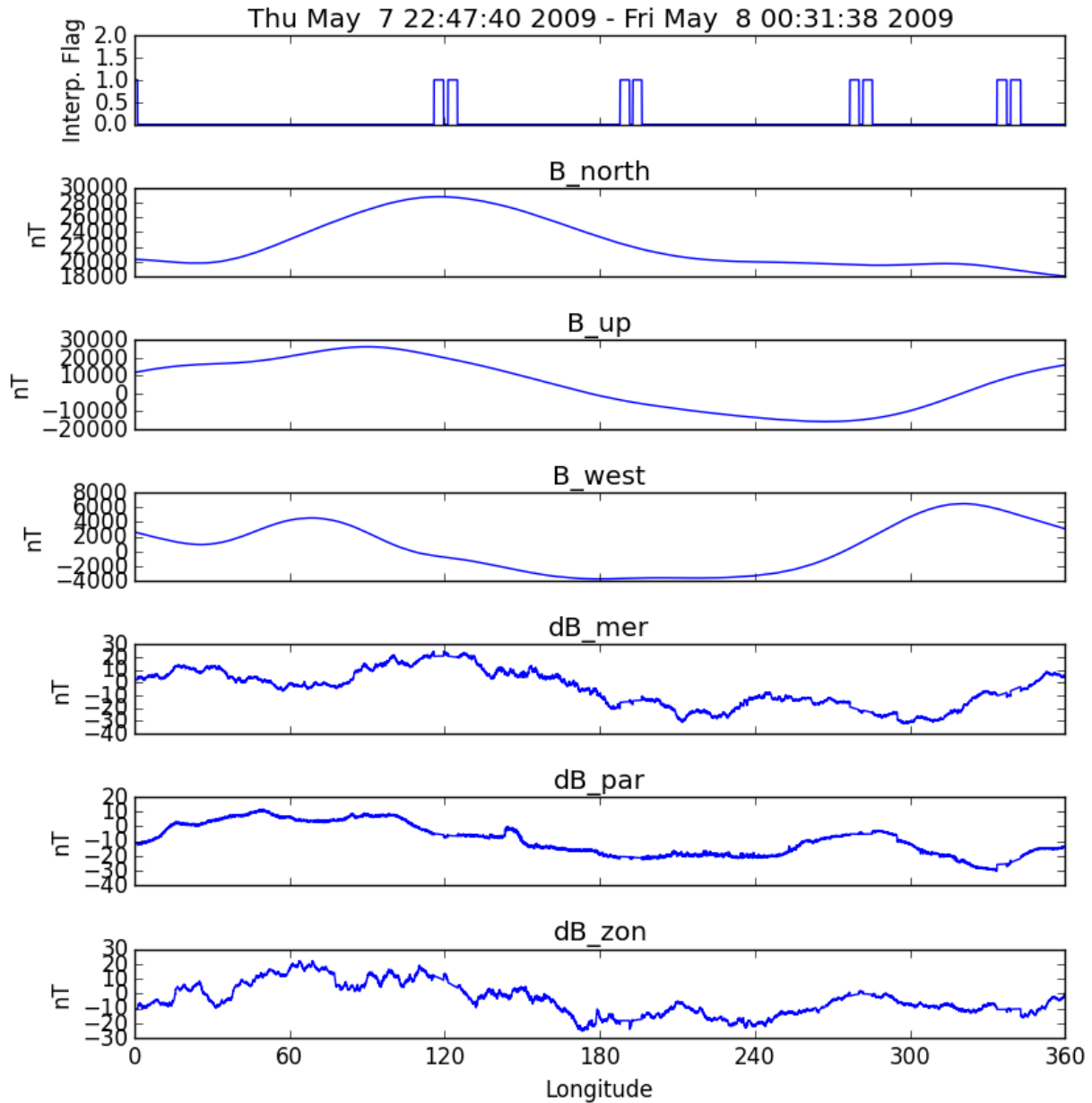
    ax[6].set_xlabel(vefi.meta['longitude'].long_name)
    ax[6].set_xticks([0,60,120,180,240,300,360])
    ax[6].set_xlim((0,360))

    f.tight_layout()
    fname = 'orbit_%05i.png' % orbit_count
```



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

Output



### 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:

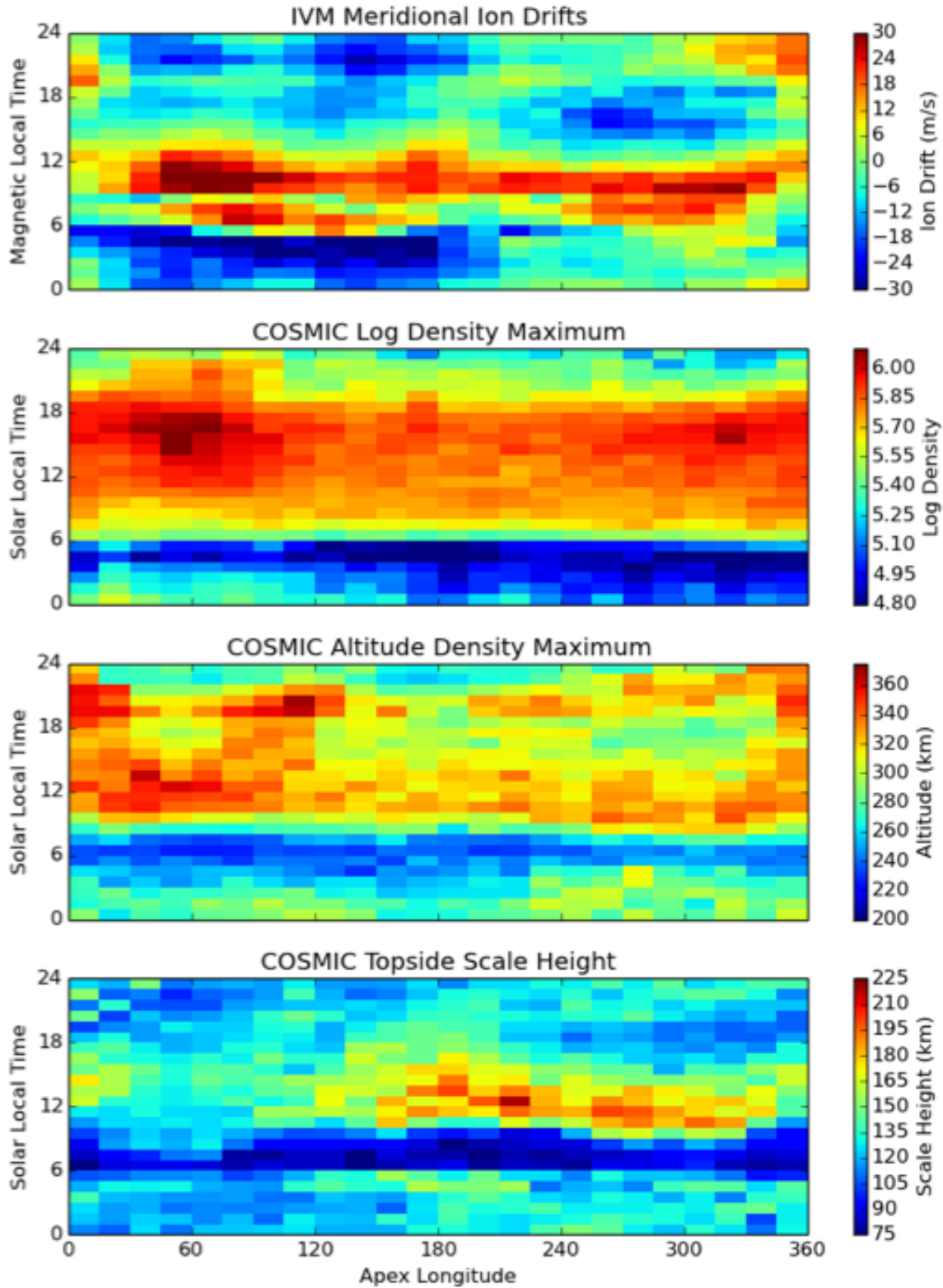
Note the same averaging routine is used for both COSMIC and IVM, and that both 1D and 2D data are handled correctly. The routine below has not yet been uploaded to the repo. (Pending)

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

# create COSMIC 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
```



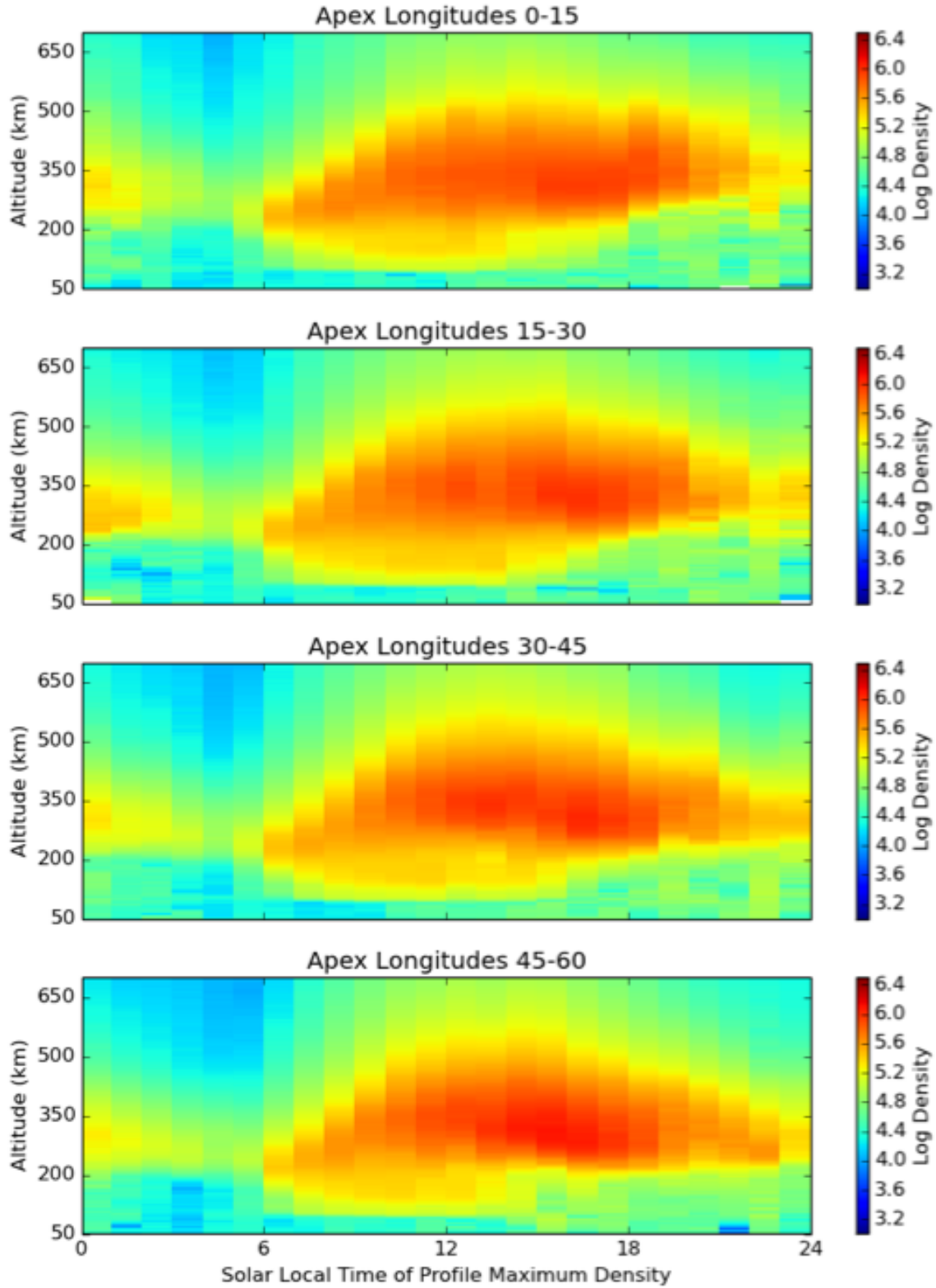
The top image is the median ion drift from the IVM, while the remaining plots are derived from the COSMIC density profiles. COSMIC data does not come with the location of the profiles in magnetic coordinates, so this information is added using the nano-kernel.

```
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.

## SUPPORTED INSTRUMENTS

### 5.1 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 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:**

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

### 5.2 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 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.

## 5.3 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

**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
- 

## 5.4 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.

**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
- 

## 5.5 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.
- 

## 5.6 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 `pydarn` to load SuperDARN files. Uses environment variables set by `davitpy` to download files from Virginia Tech SuperDARN data servers. `Pydarn` 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.

## 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:

### 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 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.

## 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.



## 7.1 Instrument

```
class pysat.Instrument (platform=None, name=None, tag=None, clean_level='clean', up-  
date_files=False, pad=None, orbit_info=None, inst_module=None,  
multi_file_day=False, manual_org=False, *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.
- **multi\_file\_day** (*boolean, optional (False by default)*) – 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 /platform/name/tag

### 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)

```

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

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

### Examples

```

# By name
inst['name']
# 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 instrument object by loading subsequent days or files.

---

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

Default bounds are the first and last dates from files on local system.

---

### 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)
for inst in inst:
    print('Another day loaded', inst.date)

```

**\_\_setitem\_\_** (*key, new*)

Convenience method for adding data to instrument.

## Examples

```
# Simple Assignment, default metadata assigned
# 'long_name' = 'name'
# 'units' = ''
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

---

## 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.datetime(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)

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

---

**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.

---

**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)

---

**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** ()

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** ()

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\_netcdf3** (*fname=None*)

Stores loaded data into a netCDF3 64-bit file.

**Parameters** **fname** (*string*) – full path to save instrument object to

---

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

Stores object data (e.g. dataframes within series) separately

- The name of the series 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.

---

## 7.2 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.

## 7.3 Files

**class** `pysat.Files` (*sat*, *manual\_org=False*)

Maintains collection of files for instrument object.

Uses the `list_files` functions for each specific instrument to create an ordered collection of files in time. Used by instrument object to load the correct files. Files also contains helper methods for determining the presence of new files and creating an ordered list of files.

**base\_path**

*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`

---

**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)
# 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 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`.

---

**Note:** 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

---

**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 filenames 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)`.

## 7.4 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.

## 7.5 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

---

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

---

<p><b>Warning:</b> This class is still under development.</p>
---



## 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.

## Examples

```

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.

## Examples

```

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

```

**Note:** Limits of iteration set by setting inst.bounds.

**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 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.

---

## 7.6 Seasonal Analysis

### 7.6.1 Occurrence Probability

`pysat.ssnl.occure_prob.by_orbit2D` (*inst*, *bin1*, *label1*, *bin2*, *label2*, *data\_label*, *gate*, *return-*  
*Bins=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.occure_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` at least 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.ssn1.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.ssn1.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.  $\text{Prob} = (\# \text{ of times with at least one hit}) / (\# \text{ of times in bin})$

#### 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.

---

## 7.6.2 Average

`pysat.ssn1.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.ssn1.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.ssn1.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.ssn1.avg.median2D(inst, bin1, label1, bin2, label2, data_label, returnData=False)`

Return a 2D average of `data_label` over a season and `label1`, `label2`.

#### Parameters

- **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

### 7.6.3 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** (*numpy 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.*

## 7.7 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*)

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 fnames is the same

`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.

**p**

`pysat.instruments.cnofs_ivm`, [27](#)  
`pysat.instruments.cnofs_vefi`, [27](#)  
`pysat.instruments.cosmic2013_gps`, [28](#)  
`pysat.instruments.cosmic_gps`, [28](#)  
`pysat.instruments.omni_hro`, [29](#)  
`pysat.instruments.superdarn_grdex`, [29](#)  
`pysat.ssnl.avg`, [48](#)  
`pysat.ssnl.occure_prob`, [46](#)  
`pysat.ssnl.plot`, [49](#)  
`pysat.utils`, [49](#)





## Symbols

`__getitem__()` (pysat.Instrument method), 37  
`__getitem__()` (pysat.Meta method), 43  
`__getitem__()` (pysat.Orbits method), 45  
`__iter__()` (pysat.Instrument method), 37  
`__iter__()` (pysat.Orbits method), 45  
`__setitem__()` (pysat.Instrument method), 37  
`__setitem__()` (pysat.Meta method), 43

## A

`add()` (pysat.Custom method), 40

## B

`base_path` (Files attribute), 41  
`bounds` (Instrument attribute), 36  
`bounds` (pysat.Instrument attribute), 38  
`by_orbit2D()` (in module `pysat.ssnl.occure_prob`), 46  
`by_orbit3D()` (in module `pysat.ssnl.occure_prob`), 46

## C

`clear()` (pysat.Custom method), 41  
`copy()` (pysat.Instrument method), 38  
`create_datetime_index()` (in module `pysat.utils`), 49  
`Custom` (class in `pysat`), 40  
`custom` (Instrument attribute), 36

## D

`daily2D()` (in module `pysat.ssnl.occure_prob`), 47  
`daily3D()` (in module `pysat.ssnl.occure_prob`), 47  
`data` (Instrument attribute), 35  
`data` (Meta attribute), 43  
`data_path` (Files attribute), 41  
`date` (Instrument attribute), 35  
`download()` (pysat.Instrument method), 38  
`doy` (Instrument attribute), 36

## F

`Files` (class in `pysat`), 41  
`files` (Instrument attribute), 36  
`from_csv()` (pysat.Meta class method), 43  
`from_dict()` (pysat.Meta class method), 44  
`from_nc()` (pysat.Meta class method), 44

`from_os()` (pysat.Files class method), 42

## G

`get_file_array()` (pysat.Files method), 42  
`get_index()` (pysat.Files method), 42  
`get_new()` (pysat.Files method), 42  
`getyrday()` (in module `pysat.utils`), 49

## I

`Instrument` (class in `pysat`), 35

## K

`kwargs` (Instrument attribute), 36

## L

`load()` (pysat.Instrument method), 39  
`load()` (pysat.Orbits method), 45  
`load_netcdf3()` (in module `pysat.utils`), 49

## M

`manual_org` (Files attribute), 41  
`mean_by_day()` (in module `pysat.ssnl.avg`), 48  
`mean_by_file()` (in module `pysat.ssnl.avg`), 48  
`mean_by_orbit()` (in module `pysat.ssnl.avg`), 48  
`median2D()` (in module `pysat.ssnl.avg`), 48  
`Meta` (class in `pysat`), 43  
`meta` (Instrument attribute), 36

## N

`next()` (pysat.Instrument method), 39  
`next()` (pysat.Orbits method), 46

## O

`Orbits` (class in `pysat`), 44  
`orbits` (Instrument attribute), 36

## P

`prev()` (pysat.Instrument method), 39  
`prev()` (pysat.Orbits method), 46  
`pysat.instruments.cnofs_ivm` (module), 27  
`pysat.instruments.cnofs_vefi` (module), 27

[pysat.instruments.cosmic2013\\_gps \(module\)](#), 28  
[pysat.instruments.cosmic\\_gps \(module\)](#), 28  
[pysat.instruments.omni\\_hro \(module\)](#), 29  
[pysat.instruments.superdarn\\_grdex \(module\)](#), 29  
[pysat.ssnl.avg \(module\)](#), 48  
[pysat.ssnl.occure\\_prob \(module\)](#), 46  
[pysat.ssnl.plot \(module\)](#), 49  
[pysat.utils \(module\)](#), 49

## R

[refresh\(\)](#) ([pysat.Files](#) method), 43  
[replace\(\)](#) ([pysat.Meta](#) method), 44

## S

[scatterplot\(\)](#) (in module [pysat.ssnl.plot](#)), 49  
[season\\_date\\_range\(\)](#) (in module [pysat.utils](#)), 50  
[set\\_data\\_dir\(\)](#) (in module [pysat.utils](#)), 50  
[start\\_date](#) ([Files](#) attribute), 41  
[stop\\_date](#) ([Files](#) attribute), 41

## T

[to\\_netcdf3\(\)](#) ([pysat.Instrument](#) method), 39

## Y

[yr](#) ([Instrument](#) attribute), 35