pysatMissions Documentation

Release 0.3.4-alpha

Klenzing, Jeff, Stoneback, Russell, Burrell, Angeline G., Depew, M

Jun 22, 2023

CONTENTS

This documentation describes the pysatMissions module, which contains routines to generate simualted instruments using Two-Line Elements (TLEs), as well as apply geomagnetic field models to existing pysat instruments.

CHAPTER

ONE

INTRODUCTION

pysatMissions allows users to run build simulated satellites for Two-Line Elements (TLE) and add empirical data. It includes the missions_ephem and missions_sgp4 instrument modules which can be imported into pysat.

1.1 Main Features

- Simulate satellite orbits from TLEs and add data from empirical models
- Import magnetic coordinates through apexpy and aacgmv2

This document covers installation, a tutorial on pysatMissions including demonstration code, and an API reference.

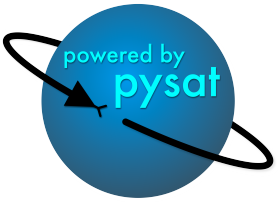
CHAPTER

TWO

INSTALLATION

The following instructions will allow you to install pysatMissions.

2.1 Prerequisites



pysatMissions uses common Python modules, as well as modules developed by and for the Space Physics community. This module officially supports Python 3.8+ and pysat 3.0.4+.

Common modules	Community modules	Optional Modules
numpy	pysat>=3.0.4	aacgmv2
pandas	pyEphem sgp4>=2.7 skyfield	apexpy OMMBV>=1.0

2.2 Installation Options

1. Clone the git repository

```
git clone https://github.com/pysat/pysatMissions.git
```

2. Install pysatMissions: Change directories into the repository folder and run the setup.py file. There are a few ways you can do this:

   A. Install on the system (root privileges required):

   ```
   sudo pip install .
   ```

   B. Install at the user level:

   ```
   pip install --user .
   ```

   C. Install with the intent to develop locally:

```
pip install -e --user .
```

2.3 Post Installation

After installation, you may register the `pysatMissions` `Instrument` sub-modules with pysat. If this is your first time using pysat, check out the quickstart guide for pysat. Once pysat is set up, you may choose to register the the `pysatMissions` `Instruments` sub-modules by:

```
import pysat
import pysatMissions

pysat.utils.registry.register_by_module(pysatMissions.instruments)
```

You may then use the pysat `platform` and `name` keywords to initialize the model `Instrument` instead of the `inst_module` keyword argument.

CHAPTER

THREE

CITATION GUIDELINES

When publishing work that uses pysatMissions, please cite the package and any package it depends on that plays an important role in your analysis. Specifying which version of pysatMissions used will also improve the reproducibility of your presented results.

3.1 pysatMissions

- Jeff Klenzing, Russell Stoneback, Angeline Burrell, Matthew Depew, & Carey Spence. (2020, July 29). pysat/pysatMissions: Version 0.2.1 (Version v0.2.1). Zenodo. https://doi.org/10.5281/zenodo.3966508

```
@software{pysatMissions,
author       = {Klenzing, Jeff and
                Stoneback, Russell and
                Burrell, Angeline and
                Depew, Matthew and
                Spence, Carey},
title        = {pysat/pysatMissions: Version 0.2.1},
month        = jul,
year         = 2020,
publisher    = {Zenodo},
version      = {v0.2.1},
doi          = {10.5281/zenodo.3966508},
url          = {https://doi.org/10.5281/zenodo.3966508}
}
```

CHAPTER

FOUR

SUPPORTED INSTRUMENTS

4.1 Missions ephem

Produce satellite orbit data.

Deprecated since version 0.3.0: pyephem is no longer updated, and the code maintainers suggest skyfield as a replacement. The functionality of the instrument will be replaced by the new *missions_sgp4* instrument. *missions_ephem* will be removed in versions 0.4.0+

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

4.1.1 Properties

platform
'missions'

name
'ephem'

tag
None supported

inst_id
None supported

pysatMissions.instruments.missions_ephem.**init**(*self*)

Add custom calculations to orbit simulation.

This routine is run once, and only once, upon instantiation. Adds custom routines for quasi-dipole coordinates, velocity calculation in ECEF coords, and attitude vectors of spacecraft (assuming x is ram pointing and z is generally nadir).

pysatMissions.instruments.missions_ephem.**load**(*fnames*, *tag=None*, *inst_id=None*, *obs_long=0.0*, *obs_lat=0.0*, *obs_alt=0.0*, *tle1=None*, *tle2=None*, *num_samples=None*, *cadence='1S'*)

Generate position of satellite in both geographic and ECEF co-ordinates.

Note: Routine is directly called by pysat and not the user.

Parameters

- **fnames** (*list*) – List of filenames

- **tag** (*str or NoneType*) – Identifies a particular subset of satellite data (accepts '') (default=None)
- **inst_id** (*str or NoneType*) – Instrument satellite ID (accepts '') (default=None)
- **obs_long** (*float*) – Longitude of the observer on the Earth's surface (default=0.)
- **obs_lat** (*float*) – Latitude of the observer on the Earth's surface (default=0.)
- **obs_alt** (*float*) – Altitude of the observer on the Earth's surface (default=0.)
- **tle1** (*string or NoneType*) – First string for Two Line Element. Must be in TLE format (default=None)
- **tle2** (*string or NoneType*) – Second string for Two Line Element. Must be in TLE format (default=None)
- **num_samples** (*int or NoneType*) – Number of samples per day (default=None)
- **cadence** (*str*) – Uses pandas.frequency string formatting ('1S', etc) (default='1S')

Returns

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

Example

```
tle1='1 25544U 98067A   18135.61844383  .00002728  00000-0  48567-4 0  9998'
tle2='2 25544  51.6402 181.0633 0004018  88.8954  22.2246 15.54059185113452'
inst = pysat.Instrument('pysat', 'ephem', tle1=tle1, tle2=tle2)
inst.load(2018, 1)
```

pysatMissions.instruments.missions_ephem.**preprocess**(*self*)

Add modeled magnetic field values and attitude vectors to spacecraft.

Runs after load is invoked.

4.2 Missions SGP4

Simulate satellite orbit data using Two Line Elements (TLEs) and SGP4.

4.2.1 Properties

platform
'missions'

name
'sgp4'

tag
None supported

inst_id
None supported

pysatMissions.instruments.missions_sgp4.**init**(*self*)

Initialize the Instrument object with required values.

Runs once upon instantiation.

pysatMissions.instruments.missions_sgp4.**load**(*fnames*, *tag=None*, *inst_id=None*, *tle1=None*, *tle2=None*, *alt_periapsis=None*, *alt_apoapsis=None*, *inclination=None*, *raan=0.0*, *arg_periapsis=0.0*, *mean_anomaly=0.0*, *epoch=None*, *bstar=0.0*, *one_orbit=False*, *num_samples=None*, *cadence='1S'*)

Generate position of satellite in ECI co-ordinates.

Parameters

- **fnames** (*list-like collection*) – File name that contains date in its name.
- **tag** (*string*) – Identifies a particular subset of satellite data
- **inst_id** (*string*) – Instrument satellite ID (accepts '' or a number (i.e., '10'), which specifies the number of seconds to simulate the satellite) (default='')
- **tle1** (*str or NoneType*) – First string for Two Line Element. Must be in TLE format. tle1 and tle2 both required if instantiating instrument by TLEs. (defalt=None)
- **tle2** (*str or NoneType*) – Second string for Two Line Element. Must be in TLE format. tle1 and tle2 both required if instantiating instrument by TLEs. (default=None)
- **alt_periapsis** (*float or NoneType*) – The lowest altitude from the mean planet surface along the orbit (km). Required along with inclination if instantiating via orbital elements. (default=None)
- **alt_apoapsis** (*float or NoneType*) – The highest altitude from the mean planet surface along the orbit (km) If None, assumed to be equal to periapsis (ie, circular orbit). Optional when instantiating via orbital elements. (default=None)
- **inclination** (*float or NoneType*) – Orbital Inclination in degrees. Required along with alt_periapsis if instantiating via orbital elements. (default=None)
- **raan** (*float*) – Right Ascension of the Ascending Node (RAAN) in degrees. This defines the orientation of the orbital plane to the generalized reference frame. The Ascending Node is the point in the orbit where the spacecraft passes through the plane of reference moving northward. For Earth orbits, the location of the RAAN is defined as the angle eastward of the First Point of Aries. Optional when instantiating via orbital elements. (default=0.)
- **arg_periapsis** (*float*) – Argument of Periapsis in degrees. This defines the orientation of the ellipse in the orbital plane, as an angle measured from the ascending node to the periapsis. Optional when instantiating via orbital elements. (default=0.)
- **mean_anomaly** (*float*) – The fraction of an elliptical orbit's period that has elapsed since the orbiting body passed periapsis. Note that this is a "fictitious angle" (input in degrees) which defines the location of the spacecraft in the orbital plane based on the orbital period. Optional when instantiating via orbital elements. (default=0.)
- **epoch** (*dt.datetime or NoneType*) – The epoch used for calculating orbital propagation from Keplerians. If None, then use the first date in the file list for consistency across multiple days. Note that this will be set in *init*. (default=None)
- **bstar** (*float*) – Inverse of the ballistic coefficient. Used to model satellite drag. Measured in inverse distance (1 / earth radius). Optional when instantiating via orbital elements. (default=0.)

- **one_orbit** (*bool*) – Flag to override num_samples and only provide a single orbit. (default=False)
- **num_samples** (*int*) – Number of samples per day. (default=None)
- **cadence** (*str*) – Uses pandas.frequency string formatting ('1S', etc) (default='1S')

Returns

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

---

Note:

- Routine is directly called by pysat and not the user.
- Altitude accuracy expected to be on the order of 10 km in Low Earth Orbit. Efforts to improve accuracy documented under issue #79.

Example

```
tle1='1 25544U 98067A   18135.61844383  .00002728  00000-0  48567-4 0  9998'
tle2='2 25544  51.6402 181.0633 0004018  88.8954  22.2246 15.54059185113452'
inst = pysat.Instrument('pysat', 'sgp4', tle1=tle1, tle2=tle2)
inst.load(2018, 1)
```

4.3 Missions Skyfield

CHAPTER

FIVE

METHODS

5.1 magcoord

Routines for projecting aacgmv2 and apexpy model values onto pysat instruments.

pysatMissions.methods.magcoord.**add_aacgm_coordinates**(*inst*, *glat_label='glat'*, *glong_label='glong'*, *alt_label='alt'*)

Add AACGM coordinates to instrument object using AACGMV2 package.

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

Parameters

- **inst** (*pysat.Instrument*) – instrument object including lat, lon, and alt as timeseries
- **glat_label** (*string*) – label used in inst to identify WGS84 geodetic latitude (degrees N)
- **glong_label** (*string*) – label used in inst to identify WGS84 geodetic longitude (degrees E)
- **alt_label** (*string*) – label used in inst to identify WGS84 geodetic altitude (km, height above surface)

Returns

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

Return type

inst

Example

function added velow modifies the inst object upon every inst.load call inst.custom.attach(add_quasi_dipole_coordinates, kwargs={'glat_label': 'custom_label'})

pysatMissions.methods.magcoord.**add_quasi_dipole_coordinates**(*inst*, *glat_label='glat'*, *glong_label='glong'*, *alt_label='alt'*)

Add quasi-dipole coordinates to instrument object using Apexpy package.

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

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

Parameters

- **inst** (*pysat.Instrument*) – instrument object including lat, lon, and alt as timeseries
- **glat_label** (*string*) – label used in inst to identify WGS84 geodetic latitude (degrees)
- **glong_label** (*string*) – label used in inst to identify WGS84 geodetic longitude (degrees)
- **alt_label** (*string*) – label used in inst to identify WGS84 geodetic altitude (km, height above surface)

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

Return type
inst

Example

function added velow modifies the inst object upon every inst.load call inst.custom.attach(add_quasi_dipole_coordinates, kwargs={'glat_label': 'custom_label'})

5.2 spacecraft

Default routines for projecting values onto vectors for pysat instruments.

pysatMissions.methods.spacecraft.**add_ram_pointing_sc_attitude_vectors**(*inst*)

Add attitude vectors for spacecraft assuming ram pointing.

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

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

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

Return type
None

Notes

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

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

pysatMissions.methods.spacecraft.**calculate_ecef_velocity**(*inst*)

Calculate spacecraft velocity in ECEF frame.

Deprecated since version 0.4.0: This function is no longer needed with the deprecation of *missions_ephem*. Better calculations are available through geospacepy and skyfield. *calculate_ecef_velocity* will be removed in versions 0.5.0+

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

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

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

Return type
None

pysatMissions.methods.spacecraft.**normalize**(*vector*)

Normalize a time-series of vectors.

Parameters
vector (*pds.DataFrame*) – A time-series consisting of vector components at each time step.

Returns
norm_vector – The normalized version of vector

Return type
pds.DataFrame

pysatMissions.methods.spacecraft.**project_ecef_vector_onto_sc**(*inst*, *x_label*, *y_label*, *z_label*, *new_x_label*, *new_y_label*, *new_z_label*, *meta=None*)

Express input vector using s/c attitude directions.

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

Parameters

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

CHAPTER

SIX

INSTRUMENT METHODS

6.1 Orbits

Methods to convert orbital information for propagators.

pysatMissions.instruments.methods.orbits.**convert_from_keplerian**(*eccentricity*, *mean_motion*, *planet='earth'*)

Calculate orbital eccentricity from periapsis and apoapsis.

Parameters

- **eccentricity** (*float*) – The eccentricty of the orbit (unitless)
- **mean_motion** (*float*) – The mean angular speed of the orbit (rad/minute)
- **planet** (*str*) – The name of the planet of interest. Used for radial calculations. (default='earth')

Returns

- **alt_periapsis** (*float*) – The lowest altitude from the mean planet surface along the orbit (km)
- **alt_apoapsis** (*float*) – The highest altitude from the mean planet surface along the orbit (km)

pysatMissions.instruments.methods.orbits.**convert_to_keplerian**(*alt_periapsis*, *alt_apoapsis=None*, *planet='earth'*)

Calculate orbital eccentricity from periapsis and apoapsis.

Parameters

- **alt_periapsis** (*float*) – The lowest altitude from the mean planet surface along the orbit (km)
- **alt_apoapsis** (*float or NoneType*) – The highest altitude from the mean planet surface along the orbit (km) If None, assumed to be equal to periapsis. (default=None)
- **planet** (*str*) – The name of the planet of interest. Used for radial calculations and mass. (default='earth')

Returns

- **eccentricity** (*float*) – The eccentricty of the orbit (unitless)
- **mean_motion** (*float*) – The mean angular speed of the orbit (rad/minute)

CHAPTER

SEVEN

TUTORIAL

7.1 Set up sgp4 with pysat

The *Missions SGP4* instrument within pysatMissions is designed to be run like any pysat instrument. To access, use

```
import pysat
from pysatMissions.instruments import missions_sgp4

sgp4 = pysat.Instrument(inst_module=missions_sgp4)
```

This can be permanently added via the instrument registry.

```
import pysat
import pysatMissions
pysat.utils.registry.register('pysatMissions.instruments.missions_sgp4')
```

or, to register all modules in pysat

```
import pysat
import pysatMissions
pysat.utils.registry.register_by_module(pysatMissions.instruments)
```

For other instruments, simply replace the module name (in this case, missions_sgp4) with the name of the desired instrument.

7.2 Orbital Propagators

Currently, two orbital propagators are included with pysatMissions. The *Missions SGP4* instrument uses the wgs72 gravity model to provide satellite position and velocity in ECI coordinates. The *Missions ephem* instrument uses the ephem pysat package to calculate an orbit in lat/lon/alt and ECEF coordinates. As an example, it also loads a series of empirical models to provide simulated magnetic data as an aid for mission planning.

The orbital propagators are activated by the load command, similar to any pysat instrument. To generate a simulated hour of orbital information with a one-second cadence, run

```
sgp4 = pysat.Instrument(inst_module=missions_sgp4, num_samples=3600)
sgp4.load(2019, 1)
```

7.3 Empirical Models

A number of methods are included to invoke several python wrappers for empirical models. This includes the aacgmv2, apexpy, and OMMBV models. These methods can be added to any pysat instrument using the *custom* functions in pysat. The example below adds the aacgmv2 coordinates to sgp4 instrument.

```
import pysat
from pysatMissions.methods import magcoord

sgp4 = pysat.Instrument(inst_module=missions_sgp4, num_samples=3600)
sgp4.custom_attach(magcoord.add_aacgm_coordinates,
                   kwargs={'glat_label': 'geod_latitude',
                           'glong_label': 'geod_longitude',
                           'alt_label': 'geod_altitude'})
sgp4.load(2019, 1)
```

Note that the latitude, longitude, and altitude variable names of the instrument should be specified since the dataset may use different variable names from those in the custom function. The method to add these empirical functions to a pysat instrument is identical across the pysat ecosystem.

CHAPTER

EIGHT

REFERENCES

8.1 aacgmv2

aacgmv2 is a python library for accessing the Altitude-Adjusted Corrected Geomagnetic (AACGM) coordinates.

- Baker, K. B., & Wing, S. (1989). A new magnetic coordinate system for conjugate studies at high latitudes. Journal of Geophysical Research, 94, 9139–9143.
- Shepherd, S. G. (2014). Altitudeadjusted corrected geomagnetic coordinates: Definition and functional approximations. Journal of Geophysical Research: Space Physics, 119, 7501–7521. DOI: 10.1002/2014JA020264

8.2 apexpy

apexpy is a python library for calculating magnetic apex coordinates.

- Richmond, A. D. (1995). Ionospheric electrodynamics using magnetic apex coordinates. Journal of Geomagnetism and Geoelectricity, 47(2), 191–212.
- Emmert, J. T., Richmond, A. D., & Drob, D. P. (2010). A computationally compact representation of magnetic-apex and quasi-dipole coordinates with smooth base vectors. Journal of Geophysical Research, 115, A08322. DOI: 10.1029/2010JA015326
- Laundal, K. M., & Richmond, A. D. (2017). Magnetic coordinate systems. Space Science Reviews, 206, 27–59

8.3 OMMBV

OMMBV calculates Orthogonal Multipole Magnetic Basis Vectors for the earth's magnetic field.

CHAPTER

NINE

GUIDE FOR DEVELOPERS

9.1 Contributor Covenant Code of Conduct

9.1.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

9.1.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

9.1.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

9.1.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

9.1.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at pysat.developers@gmail.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

9.1.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

9.2 Contributing

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

Come join us on Slack! An invitation to the pysat workspace is available in the 'About' section of the pysat GitHub Repository. Development meetings are generally held fortnightly.

9.2.1 Short version

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

9.2.2 Bug reports

When reporting a bug please include:

- Your operating system name and version
- Any details about your local setup that might be helpful in troubleshooting
- Detailed steps to reproduce the bug

9.2.3 Feature requests and feedback

The best way to send feedback is to file an issue at GitHub.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

9.2.4 Development

To set up `pysatMissions` for local development:

1. Fork pysat on GitHub.
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/pysatMissions.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

   Tests for new instruments are performed automatically. Tests for custom functions should be added to the appropriately named file in `pysatMissions/tests`. If no test file exists, then you should create one. This testing uses pytest, which will run tests on any python file in the test directory that starts with `test_` as well.

4. When you're done making changes, run all the checks to ensure that nothing is broken on your local system:

```
pytest -vs
```

   1. You should also check for flake8 style compliance:

```
flake8 . --count --select=D,E,F,H,W --show-source --statistics
```

      Note that pysat uses the `flake-docstrings` and `hacking` packages to ensure standards in docstring formatting.

5. Update/add documentation (in `docs`), if relevant
6. Add your name to the .zenodo.json file as an author
7. Commit your changes:

```
git add .
git commit -m "AAA: Brief description of your changes"
```

Where AAA is a standard shorthand for the type of change (eg, BUG or DOC). `pysat` follows the numpy development workflow, see the discussion there for a full list of this shorthand notation.

8. Once you are happy with the local changes, push to GitHub:

```
git push origin name-of-your-bugfix-or-feature
```

Note that each push will trigger the Continuous Integration workflow.

9. Submit a pull request through the GitHub website. Pull requests should be made to the `develop` branch. Note that automated tests will be run on GitHub Actions, but these must be initialized by a member of the pysat team.

9.2.5 Pull Request Guidelines

If you need some code review or feedback while you're developing the code, just make a pull request. Pull requests should be made to the `develop` branch.

For merging, you should:

1. Include an example for use
2. Add a note to `CHANGELOG.md` about the changes
3. Update the author list in `zenodo.json` if applicable
4. Ensure that all checks passed (current checks include Github Actions and Coveralls)

If you don't have all the necessary Python versions available locally or have trouble building all the testing environments, you can rely on GitHub Actions to run the tests for each change you add in the pull request. Because testing here will delay tests by other developers, please ensure that the code passes all tests on your local system first.

9.2.6 Project Style Guidelines

In general, pysat follows PEP8 and numpydoc guidelines. Pytest runs the unit and integration tests, flake8 checks for style, and sphinx-build performs documentation tests. However, there are certain additional style elements that have been settled on to ensure the project maintains a consistent coding style. These include:

- Line breaks should occur before a binary operator (ignoring flake8 W503)
- Combine long strings using `join`
- Preferably break long lines on open parentheses rather than using \
- Use no more than 80 characters per line
- Avoid using Instrument class key attribute names as unrelated variable names: `platform`, `name`, `tag`, and `inst_id`
- The pysat logger is imported into each sub-module and provides status updates at the info and warning levels (as appropriate)
- Several dependent packages have common nicknames, including:
  - `import datetime as dt`
  - `import numpy as np`
  - `import pandas as pds`

  - `import xarray as xr`

- All classes should have `__repr__` and `__str__` functions
- Docstrings use `Note` instead of `Notes`
- Try to avoid creating a try/except statement where except passes
- Use setup and teardown in test classes
- Use pytest parametrize in test classes when appropriate
- Provide testing class methods with informative failure statements and descriptive, one-line docstrings
- Block and inline comments should use proper English grammar and punctuation with the exception of single sentences in a block, which may then omit the final period
- When casting is necessary, use `np.int64` and `np.float64` to ensure operating system agnosticism

CHAPTER

TEN

CHANGE LOG

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning.

10.1 [0.3.4] - 2023-06-22

- Add support for skyfield propagation
- Maintenance
  - Update pytest syntax
  - Update Github Actions versions
  - Add manual GitHub Actions tests for pysat RC
  - Add manual GitHub Actions tests for optional dependencies
  - Add manual GitHub Actions tests for pysatMissions RC
  - Remove optional dependencies in readthedocs requirements
  - Add tests for NEP 29 testing
  - Add tests for Mac OS
  - Deprecate `calculate_ecef_velocity` method
  - Use pyproject.toml to handle metadata / installation
  - Update GitHub Actions workflow standards
- Testing
  - Include checks on sc coordinate transformation calculations

10.2 [0.3.3] - 2022-09-06

- Documentation Updates

10.3 [0.3.2] - 2022-09-06

- Make fortran dependencies optional installs
  - aacgmv2
  - apexpy
  - OMMBV
- Access logger directly from pysat
- Use pysat deprecation tests
- Incorporate Windows tests into Github Actions
- Bug Fix
  - Ensure default num_samples consistent for one day regardless of cadence
- Maintenance
  - Update instrument test standards
  - Added `utils.package_check`, a standard decorator to bypass functions using packages that are optional installs if the package is not installed
  - Use local vector functions rather than import from OMMBV

10.4 [0.3.1] - 2022-05-18

- Include license in package manifest

10.5 [0.3.0] - 2022-05-13

- Add Keplerian orbital inputs into missions_sgp4
- Update sgp4 interface to use new syntax for initialization from TLEs
- Include conversions to geodetic latitude / longitude / altitude for sgp4
- Improve metadata generation in missions_sgp4
- Update syntax to be compliant with OMMBV 1.0
- Documentation
  - Improve docstrings throughout
  - Added bypass for apexpy for readthedocs build
- Deprecations
  - Deprecated missions_ephem, as pyephem will no longer be updated
- Testing
  - Add style check for docstrings
  - Added checks for deprecation warnings
  - Improve checks in codeclimate

10.6 [0.2.2] - 2021-06-18

- Migrate pyglow interface to pysatIncubator
- Style updates for consistency with pysat 3.0
  - Use `inst_id` instead of `sat_id`
  - Use `cadence` instead of `freq`
  - Use 'missions' as the platform name ('pysat' now reserved for core code)
- Migrate CI testing to Github Actions
- Use OMMBV instead of pysatMagVect

10.7 [0.2.1] - 2020-07-29

- Use conda to manage Travis CI environment
- Updated style to be compliant with pandas 2.0 and pysat 3.0
  - Import datetime from datetime
  - import DataFrame and Series from pandas rather than pysat
- Rename default branch as `main`

10.8 [0.2.0] - 2020-03-07

- Renamed as pysatMissions
- Added method files to access aacgmv2, apexpy, pyglow for any pysat instrument
- Added method for spacecraft to handle attitude and coordinates
- Added method for plotting simulated data
- Added support for readthedocs
- Updates to testing environment
- Split pysat_sgp4 into pysat_sgp4 and pysat_ephem to allow different propagators
- Removed basemap
- Added numeric strings as options for sat_id
- Added `_get_times` to streamline time steps for simulated instruments
- Bugs
  - Fixed wrong metadata name for mlt in apexpy

10.9 [0.1.1] - 2019-10-22

- pypi compatibility
- Add DOI badge

10.10 [0.1.0] - 2019-10-07

- Initial release

PYTHON MODULE INDEX

p

INDEX