Bulwark Documentation

Release 0.5.0

Zax Rosenberg

Aug 24, 2019

Contents

Bulwark is a package for convenient property-based testing of pandas dataframes, supported for Python 3.5+.

Documentation: https://bulwark.readthedocs.io/en/latest/index.html

This project was heavily influenced by the no-longer-supported Engarde library by Tom Augspurger(thanks for the head start, Tom!), which itself was modeled after the R library assertr.

CHAPTER 1

Why?

Data are messy, and pandas is one of the go-to libraries for analyzing tabular data. In the real world, data analysts and scientists often feel like they don't have the time or energy to think of and write tests for their data. Bulwark's goal is to let you check that your data meets your assumptions of what it should look like at any(and every) step in your code, without making you work too hard.

CHAPTER 2

Installation

```
pip install bulwark
```

CHAPTER 3

Usage

Bulwark comes with checks for many of the common assumptions you might want to validate for the functions that make up your ETL pipeline, and lets you toss those checks as decorators on the functions you're already writing:

```
import bulwark.decorators as dc

@dc.IsShape((-1, 10))
@dc.IsMonotonic(strict=True)
@dc.HasNoNans()
def compute(df):
    # complex operations to determine result
    ...
return result_df
```

Still want to have more robust test files? Bulwark's got you covered there, too, with importable functions.

```
import bulwark.checks as ck

df.pipe(ck.has_no_nans())
```

Won't I have to go clean up all those decorators when I'm ready to go to production? Nope - just toggle the built-in "enabled" flag available for every decorator.

```
@dc.IsShape((3, 2), enabled=False)
def compute(df):
    # complex operations to determine result
    ...
return result_df
```

What if the test I want isn't part of the library? Use the built-in `CustomCheck` to use your own custom function!

```
def len_longer_than(df, l):
    if len(df) <= l:
        raise AssertionError("df is not as long as expected.")
    return df
```

(continues on next page)

(continued from previous page)

```
@dc.CustomCheck(len_longer_than, df=df, l=6)
def append_a_df(df, df2):
    return df.append(df2, ignore_index=True)

df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})

append_a_df(df, df2)
```

What if I want to run a lot of tests and want to see all the errors at once? You can use the built-in `MultiCheck`. It will collect all of the errors and either display a warning message of throw an exception based on the `warn` flag. You can even use custom functions with MultiCheck:

```
def len_longer_than(df, l):
    if len(df) <= l:
        raise AssertionError("df is not as long as expected.")
    return df

# `checks` takes a dict of function: dict of params for that function.
# Note that those function params EXCLUDE df.
# Also note that when you use MultiCheck, there's no need to use CustomCheck - just␣
↪feed in the function.
@dc.MultiCheck(checks={ck.has_no_nans: {"columns": None},
                       len_longer_than: {"l": 6}},
               warn=False)
def append_a_df(df, df2):
    return df.append(df2, ignore_index=True)

df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})

append_a_df(df, df2)
```

See examples to see more advanced usage.

3.1 Changelog

Added

- Add support for old Engarde function names with deprecation warnings for v0.7.0.
- Add ability to check bulwark version with `bulwark.__version__`
- Add status badges to README.md
- Add Sphinx markdown support and single-source readme, changelog.

Changed

- Upgrade Development Status to Beta (from Alpha)
- Update gitignore for venv
- Update contributing documentation
- Single-sourced project version

Changed

- Hotfix to allow import bulwark to work.

Changed

- Hotfix to allow import bulwark to work.

Added

- Add `has_no_x`, `has_no_nones`, and `has_set_within_vals`.

Changed

- `has_no_nans` now checks only for np.nans and not also None. Checking for None is available through has_no_nones.

Added

- Add `exact_order` param to `has_columns`

Changed

- Hotfix for reversed `has_columns` error messages for missing and unexpected columns
- Breaking change to `has_columns` parameter name `exact`, which is now `exact_cols`

Added

- Add `has_columns` check, which asserts that the given columns are contained within the df or exactly match the df's columns.
- Add changelog

Changed

- Breaking change to rename unique_index to has_unique_index for consistency

Changed

- Improve code base to automatically generate decorators for each check
- Hotfix multi_check and unit tests

Changed

- Hotfix to setup.py for the sphinx.setup_command.BuildDoc requirement.

Changed

- Breaking change to rename unique_index to has_unique_index for consistency

3.2 Installation

```
pip install bulwark
```

3.3 Quickstart

Bulwark is designed to be easy to use and easy to add checks to code while you're writing it.

First, install Bulwark:

```
pip install bulwark
```

Next, import bulwark. You can either use function versions of the checks or decorator versions. By convension, import either/both of these as follow:

```
import bulwark.checks as ck
import bulwark.decorators as dc
```

If you've chosen to use decorators to interact with the checks (the recommended method for checks to be run on each function call), you can write a function for your project like normal, but with your chosen decorators on top:

```
import bulwark.decorators as dc
import pandas as pd


@dc.HasNoNans()
def add_five(df):
    return df + 5


df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
add_five(df)
```

You can stack multiple decorators on top of each other to have the first failed decorator check result in an assertion error or use the built-in MultiCheck to collect all of the errors are raise them at once.

See examples to see more advanced usage.

3.4 Design

It's important that `Bulwark` not get in your way. Your task is hard enough without a bunch of assertions cluttering up the logic of the code. And yet, it does help to explicitly state the assumptions fundamental to your analysis. Decorators provide a nice compromise.

3.4.1 Checks

Each check:

- takes a pd.DataFrame as its first argument, with optional additional arguments,
- make an assert about the pd.DataFrame, and
- return the original, unaltered pd.DataFrame

If the assertion fails, an `AssertionError` is raised and `Bulwark` tries to print out some informative information about where the failure occurred.

3.4.2 Decorators

Each `check` has an auto-magically-generated associated decorator. The decorator simply marshals arguments, allowing you to make your assertions *outside* the actual logic of your code. Besides making it quick and easy to add checks to a function, decorators also come with bonus capabilities, including the ability to enable/disable the check as well as switch from raising an error to logging a warning.

3.5 Examples

Coming soon!

3.6 API

bulwark.checks	Each function in this module should:
bulwark.decorators	

3.6.1 bulwark.checks

Each function in this module should:

- take a pd.DataFrame as its first argument, with optional additional arguments,
- make an assert about the pd.DataFrame, and
- return the original, unaltered pd.DataFrame

Functions

`custom_check`(check_func, df, *args, **kwargs)	Assert that *check(df, *args, **kwargs)* is true.
`has_columns`(df, columns[, exact_cols, ...])	Asserts that *df* has `columns`
`has_dtypes`(df, items)	Asserts that *df* has `dtypes`
`has_no_infs`(df[, columns])	Asserts that there are no np.infs in *df*.
`has_no_nans`(df[, columns])	Asserts that there are no np.nans in *df*.
`has_no_neg_infs`(df[, columns])	Asserts that there are no np.infs in *df*.
`has_no_nones`(df[, columns])	Asserts that there are no Nones in *df*.
`has_no_x`(df[, values, columns])	Asserts that there are no user-specified *values* in *df*'s *columns*.
`has_set_within_vals`(df, items)	Asserts that all given values are found in columns' values.
`has_unique_index`(df)	Asserts that *df*'s index is unique.
`is_monotonic`(df[, items, increasing, strict])	Asserts that the *df* is monotonic.
`is_same_as`(df, df_to_compare, **kwargs)	Asserts that two pd.DataFrames are equal.
`is_shape`(df, shape)	Asserts that *df* is of a known row x column *shape*.
`multi_check`(df, checks[, warn])	Asserts that all checks pass.
`one_to_many`(df, unitcol, manycol)	Asserts that a many-to-one relationship is preserved between two columns.
`unique`(df[, columns])	Asserts that columns in *df* only have unique values.
`has_vals_within_n_std`(df[, n])	Asserts that every value is within `n` standard deviations of its column's mean.
`has_vals_within_range`(df[, items])	Asserts that *df* is within a range.
`has_vals_within_set`(df[, items])	Asserts that *df* is a subset of items.

3.6.2 bulwark.decorators

Functions

`CustomCheck`(check_func, *args, **kwargs)	Assert that *func(df, *args, **kwargs)* is true.
`decorator_factory`(decorator_name, func)	Takes in a function and outputs a class that can be used as a decorator.

Classes

`BaseDecorator`(*args, **kwargs)	
`HasColumns`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`HasDtypes`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`HasNoInfs`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`HasNoNans`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`HasNoNegInfs`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`HasNoNones`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`HasNoX`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`HasSetWithinVals`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`HasUniqueIndex`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`IsMonotonic`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`IsSameAs`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`IsShape`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`MultiCheck`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`
`OneToMany`	alias of `bulwark.decorators.decorator_factory.<locals>.decorator_name`

Continued on next page

Table 4 – continued from previous page

Unique	alias of bulwark.decorators.decorator_factory.<locals>.decorator_name
WithinNStd	alias of bulwark.decorators.decorator_factory.<locals>.decorator_name
WithinRange	alias of bulwark.decorators.decorator_factory.<locals>.decorator_name
WithinSet	alias of bulwark.decorators.decorator_factory.<locals>.decorator_name

3.7 Contributing

3.7.1 Set up Git and a GitHub Account

- If you don't already have a GitHub account, you can register for free.
- If you don't already have Git installed, you can follow these git installation instructions.

3.7.2 Fork and Clone Bulwark

1. You will need your own fork to work on the code. Go to the Bulwark project page and hit the Fork button.
2. Next, you'll want to clone your fork to your machine:

```
git clone https://github.com/your-user-name/bulwark.git bulwark-dev
cd bulwark-dev
git remote add upstream https://github.com/ZaxR/bulwark.git
```

3.7.3 Set up a Development Environment

Bulwark supports Python 3.5+. It's recommended to use version 3.5 for development to ensure newer features aren't accidentally used, though CI tools will check all versions on the creation of a PR.

3.7.4 Create a Feature Branch

Bulwark loosely follows the gitflow workflow. To add a new feature, you will create every feature branch off of the develop branch:

```
git checkout develop
git checkout -b feature/<feature_name_in_snake_case>
```

3.7.5 Development Practices and Standards

- Unit tests covering added/changed code are required for a PR to be merged. There is currently no CI check for coverage, but this will be manually enforced. Test-Driven Development (TDD) is encouraged.

- Any new module, class, or function requires a docstring, in the Google docstring format.
- Please follow PEP-8

3.7.6 Create a Pull Request to the develop branch

Create a pull request to the develop branch of Bulwark. Tests will be be triggered to run via Travis CI. Check that your PR doesn't fail any tests, since it won't be reviewed for inclusion until it passes all tests.

3.7.7 For Maintainers

When it's time to create a release candidate, a new branch should be created from develop:

```
git checkout develop
git checkout -b release/x.x.x
```

However, several additional steps must also be taken:

- Update version in project_info.py, which updates three spots: setup.py, bulwark/__init__.py, and docs/conf.py.
- Update the CHANGELOG.md and the main README.md (as appropriate).
- Rebuild the docs in your local version using:

```
pip install -e ".[dev]"
sphinx-apidoc -o ./docs/_source ./bulwark -f
cd docs
make html
```

- Test distribution using TestPyPI with Twine:

```
# Installation
python3 -m pip install --user --upgrade setuptools wheel
python3 -m pip install --user --upgrade twine

# Build/Upload dist and install library
python3 setup.py sdist bdist_wheel
python3 -m twine upload --repository-url https://test.pypi.org/legacy/ dist/*
pip install --index-url https://test.pypi.org/simple/bulwark
```

- Merge the release candidate into both master (which will trigger updates for PyPi and readthedocs) and develop.
- Tag the release locally and push it to remote:

```
git tag -a v<#.#.#> <SHA-goes-here> -m "bulwark version <#.#.#>"
git push origin --tags
```

Python Module Index

b

Index

B