pyjanitor: A Cleaner API for Cleaning Data

Eric J. Ma[‡*], Zachary Barry[‡], Sam Zuckerman, Zachary Sailer[§]

Abstract—The pandas library has become the de facto library for data wrangling in the Python programming language. However, inconsistencies in the pandas application programming interface (API), while idiomatic due to historical use, prevent use of expressive, fluent programming idioms that enable self-documenting pandas code. Here, we introduce pyjanitor, an open source Python package that extends the pandas API with such idioms. We describe its design and implementation of the package, provide usage examples from a variety of domains, and discuss the ways that the pyjanitor project has enabled the inclusion of first-time contributors to open source projects.

Index Terms—data engineering, data science, data cleaning

Introduction

Data preprocessing, or data wrangling, is an unavoidable task in data science. It is a common experience amongst data scientists that data wrangling can occupy up to 80% of their time [nyt] [Wic14]. Part of this time is spent defining modelling approaches, and part of this time is writing code that executes the sequence of transformations on raw data that wrangle it into the necessary shape for downstream modelling work.

In the Python ecosystem, pandas is the *de facto* tool for data manipulation. This is because it provided an API for manipulating tabular data when conducting data analysis. This API was noticeably missing from the Python standard library and NumPy, which, prior to pandas emergence, were the primary tools for data analysis in Python. Hence, through the DataFrame object and its interfaces, pandas provided a key API that enabled statisticians, data scientists, and machine learners to wrangle their data into a usable shape. That said, there are inconsistencies in the *pandas* API which, though now are idiomatic due to historical use, prevent the use of expressive, fluent [flu] programming idioms[1] that enable self-documenting data science code.

Idiomatic Inconsistencies of pandas

A case in point is the following elementary sequence of data preprocessing operations:

1) Standardizing column names to snake-case (spelled_like_this, rather than Spelled! Like! This?),
2) Removing unnecessary columns,
3) Adding a column of data,
4) Log-transforming a column,
5) Filtering the log-transformed column,
6) Dropping rows that have null values,
7) Adding a column that is the mean of each sample's group.

To do this with the pandas API, one might write the following code.

```
import pandas as pd
import numpy as np
import re

df = pd.DataFrame(...)

def clean_name(x):
    """Custom function to sanitize column name."""
    FIXES = [(r"[\* /:,?!()\.-]", "_"), (r"['’]", "")]
    for search, replace in FIXES:
        x = re.sub(search, replace, x)
    return x.lower().replace('__', '_')

df = (
    df
    # clean column names
    .rename(columns=clean_name)
    # remove column
    .drop('column_name_14', axis='columns')
    # log transform
    .assign(
        column_name_13=lambda x: np.log10(x['column_name_13'])
    )
    # drop null values
    .dropna()
    # filter based on column value
    .query("column_name_13 < 3")
)

# add a column that is the mean of each sample's group.
col13_means = df.groupby('group').mean()['column_name_13']
df = df.join(col13_means, rsuffix='_mean', on='group')
```

By using pyjanitor, end-users can instead write code that reads much closer to the plain English description.

```
import pandas as pd
import numpy as np
import janitor

df = (
    pd.DataFrame(...)
    .clean_names()
    .remove_column('column_name_14')
    .transform_column('column_name_13', np.log10)
    .query('column_name_13 < 3')
    .dropna()
    .groupby_agg(
        by="group",
        agg_column_name="column_name_13",
```

* Corresponding author: ericmajinglong@gmail.com
‡ Novartis Institutes for Biomedical Research
§ Jupyter Project

Copyright © 2019 Eric J. Ma et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

1. Fluent interfaces, as a term, were first coined in 2006, and describe a programming pattern allowing code to more closely resemble written prose. Method chaining is the most common way to achieve this.

```
        new_column_name="column_name_13_mean",
        agg="mean",
    )
)
```

This is the API design that `pyjanitor` aims to provide to `pandas` users: common data cleaning routines that can be mix-and-matched with existing `pandas` API calls. This is in keeping with Line 7 of the Zen of Python, which states that "Readability counts"; `pyjanitor` thus enables data scientists to construct their data processing code with an easily-readable sequence of meaningful verbs. By providing commonly-usable data processing routines, we also save time for data scientists and engineers, allowing them to accomplish their work more efficiently.

History of `pyjanitor`

`pyjanitor` started as a Python port of the R package `janitor`, which provides the same functionality to R users. The initial goal was to explicitly copy the `janitor` function names while engineering it to be compatible with `pandas.DataFrames`, following Pythonic idioms, such as the method chaining provided by some `pandas` class methods. As the project evolved, the scope broadened, to provide a defined language for data processing as an extension on `pandas` DataFrames, including submodules with functions specific for bioinformatics, cheminformatics, and finance.

Architecture

`pyjanitor` relies completely on the `pandas` extension API (https://pandas.pydata.org/pandas-docs/stable/development/extending.html), which allows developers to create functions that behave as if they were native `pandas.DataFrame` class methods. The only requirement here for such functions is that the first argument to it be a `pandas.DataFrame` object:

```python
def data_cleaning_function(df, **kwargs):
    ...
    # data cleaning functions go here
    ...
    return df
```

In order to reduce the amount of boilerplate required, `pyjanitor` also makes heavy use of `pandas_flavor` [pf], which provides an easy-to-use function decorator that handles class method registration. As such, to extend the `pandas` API with more instance-method-like functions, we only have to decorate the custom function, as illustrated in the following code sample:

```python
import pandas_flavor as pf

@pf.register_dataframe_method
def data_cleaning_function(df, **kwargs):
    ...
    # data cleaning operations go here
    ...
    return df
```

`pandas-flavor` has functionality that warns, at runtime, whether a `DataFrame` attribute has been overwritten by a custom function. Our test suite allows us to catch this issue before committing contributed code to the library.

Underneath each data cleaning function, we are free to use both the imperative and functional APIs. What is exposed, then, is a functional and fluent API for the end-user.

Thanks to the `pandas.DataFrame.query()` API, symbolic evaluations are generally available in `pyjanitor` for filtering data. This enables us to write functions that do filtering of the DataFrame using a verb that might match end-users' intuitions better. One such example is the `.filter_on('criteria')` method, illustrated in the opening example.

Design

Inspired by the `dplyr` world, `pyjanitor` functions are named with verb expressions. This, as mentioned earlier, this helps with readability. Hence, if we want to "clean names", the end user can call on the `.clean_names()` function/class method. If the end user wants to "remove all empty rows and columns", they can call on `.remove_empty()`. As far as possible, function names are expressed using simple English verbs that are understandable cross-culturally and well-documented, to ensure that this API is inclusive and accessible to the widest subset of users possible.

Where domain-specific verbs are used, we strive to match the mental models and vocabulary of domain experts. One example comes from the `biology` submodule, where the `join_fasta` function allows a bioinformatics-oriented user to add in a column of sequences based on FASTA accession numbers that are keys for sequence values in a FASTA-formatted file [PL88].

Keyword arguments are also likewise named with verb expressions where relevant. For example, if one wants to preserve and record the original column names before cleaning, one can add the `preserve_original` keyword argument to the `.clean_names` method:

```python
(
    df
    .clean_names(
        preserve_original=True,
        remove_special=True,
        ...
    )
)
```

In order to adhere to a functional programming paradigm, no operations that change the original DataFrame are allowed. Hence, if the internal implementation of a function results in a mutation of the original DataFrame, we explicitly make a copy of the DataFrame first, though we also generally try to avoid double-copying as well. This decision, which was made after a fairly extensive discussion on our issue tracker, balances functional design principles and pragmatic considerations when dealing with potentially large dataframe objects.

A final design choice we made was to explicitly disallow overriding or duplicating existing DataFrame class methods. The goal here is to extend `pandas`, rather than replace its API, and we have turned down user requests to do so.

Documentation

Full API Documentation for *pyjanitor* is available on ReadTheDocs [doc].

An examples gallery, which contains Jupyter notebooks that showcase how to use `pyjanitor`, is also part of the documentation.

Development

The reception to `pyjanitor` has been encouraging thus far. Newcomer contributors to open source have made their first contributions to `pyjanitor`, and experienced software developers have also chipped in. Many contributors are data scientists themselves, who are also seeking cleaner APIs to help them get their work done. There is a salient lesson here: with open source tools, savvy users can help steer development in a direction that they need, and we would encourage other contributors to join in too.

As with most open source software development, maintenance and new feature development are entirely volunteer driven. Users are invited to post feature requests on the source repository issue tracker, but are more so invited to contribute an implementation themselves to share. To date, 31 contributors have made pull requests into `pyjanitor`, and we look forward to further contributions being made at the SciPy conference sprints.

In the spirit of being beginner-friendly, new contributions to the pyjanitor library are encouraged to solve one and only one specific problem first, before we figure out how to either (1) generalize the function use case, or (2) generalize the implementation.

As an example, the commit history for `clean_names()` follows this pattern. The initial implementation manually listed out every character to be replaced by an underscore, in a DataFrame with a single column level. A later pull request extended the implementation to multi-level columns, and the current improved version uses regex string replacement to concisely express the cleaning operation. Most notably, each of these contributions were made by first-time open source contributors.

For the long-term health of the package, we are on the lookout for underrepresented contributors who would like to help maintain the package long-term as well. A code of conduct document, and a community guidelines document, are also on our development roadmap.

Other Related Tools

When developing `pyjanitor`, we originally set out to port `janitor` (the R package) to Python, providing compatibility with `pandas` DataFrames in a style compatible with Pythonic idioms (e.g. method chaining). While development was under way, we also found the Python alternatives described below, and found them to either (a) be lacking active development, (b) inventing a new pipe-like operator, (c) be restricted to dplyr verbs, and/or (d) lacking a robust community of developers. Hence, the development of `pyjanitor` was, and still is, oriented towards solving these problems.

For the convenience of our readers, we list our assessment of related tools below.

janitor [jan]: This is the original source of inspiration for `pyjanitor`, and the original creator of `janitor` is aware of `pyjanitor`'s existence. A number of function names in `janitor` have been directly copied over to `pyjanitor` and re-implemented in a `pandas`-compatible syntax.

dplyr [dplb]: The `dplyr` R package can be considered as "the originator" for verb-based data processing syntax. `janitor` the R package extends `dplyr`. It is available for use by Python users through `rpy2`; however, its primary usage audience is R users.

pandas-ply [pan]: This is a tool developed by Coursera, and aims to provide the `dplyr` syntax to `pandas` users. One advantage that it has over `pyjanitor` is that symbolic expressions can be used inside functions, which automatically get parsed into an appropriate lambda function in Python. However, it is restricted to the `dplyr` verb set.

dplython [dpla]: Analogous to `pandas-ply`, `dplython` also aims to provide the `dplyr` syntax to *pandas* users, but just like `pandas-ply`, it is restricted to `dplyr` verbs.

dfply [dfp]: This is the most actively-developed, pandas-compatible `dplyr` port. Its emphasis is on porting over the piping syntax to the pandas world. From our study of its source code, in principle, every function there can be wrapped with `pandas-flavor`'s `.register_dataframe_method` decorator, thus bringing the most feature-complete implementation of `dplyr` verbs to the `pandas` world. It does, however, re-implement a number of `pandas` functions using `dplyr` names. This makes it distinct from the pyjanitor project, where extension, rather than replacement, of existing `pandas` functionality is generally encouraged. Whether the developers are interested in collaboration remains to be discussed.

plydata [ply]: Like the others mentioned before, `plydata` also aims to provide the `dplyr`-style data manipulation grammar to `pandas`. It also provides a *pipe*-like operator (>>), and features integration with `plotnine`, a grammar of graphics plotting library for the Python programming language.

kadro [kad]: Kadro uses a wrapper around `pandas.DataFrame` objects to provide `dplyr`-style syntax.

pdpipe [pdp]: `pdpipe` provides a language for creating data preprocessing pipelines that are turned into Python callables, through which a DataFrame can be passed. Its design choice is to create fluent pipelines as pre-declared functions that are chained, rather than as methods that are attached onto a DataFrame. This distinction separates `pyjanitor` and `pdpipe`.

Limitations of `pyjanitor`

A current technical limitation of `pyjanitor` is the inability to symbolically parse expression strings to perform column-wise transformations. An example of a desired API might be:

```
df = (
    pd.DataFrame(...)
    .mutate(
        expression="column_name_12 + column_name_13",
        new_column_name="summation"
    )
)
```

As of now, because symbolic parsing is unavailable, this fluent and declarative syntax that is available to `dplyr` users is unavailable to `pyjanitor` users. We would welcome a contribution that enables this, perhaps using the `patsy` package.

Extensions beyond `pyjanitor`

`pyjanitor` does not aim to be the all-purpose data cleaning tool for all subject domains. Apart from providing a library of generally useful data manipulation and cleaning routines, one can also think of the project as a catalyst project for other specific domain applications. Following the verb-based grammar, one may imagine even more specific domain terms. Hence we have developed domain-specific submodules with a view towards encouraging their further development as independent packages.

For example, in our `chemistry` submodule, we have the following functions implemented that aid in cheminformatics-oriented data science tasks:

- `smiles2mol(df, col_name)`: to convert a column of smiles into RDKit [rdk] mol objects.
- `mol2graph(df, col_name)`: to convert a column of mol objects into NetworkX [HSS08] graph objects.

In our biology submodule, convenience functions exist to accomplish the following tasks:

- `join_fasta(df, file_name, id_col, col_name)`: create a column that contains the string representation of a biological sequence, by "joining" in a FASTA file, mapping the string to a particular column that already has the sequence identifiers in it.

The dependencies required for their usage are optional at install-time, and we provide instructions for end-users to install the relevant packages if they are not already installed locally.

Acknowledgments

We would like to thank the users who have made contributions to `pyjanitor`. These contributions have included documentation enhancements, bug fixes, development of tests, new functions, and new keyword arguments for functions. The list of contributors, which we anticipate will grow over time, can be found under `AUTHORS.rst` in the development repository.

We would also like to acknowledge the tremendous convenience provided by `pandas-flavor`, which was developed by one of our co-authors, Dr. Zachary Sailer.

References

[dfp] dplyr-style piping operations for pandas dataframes. https://github.com/kieferk/dfply. Accessed: 24 November 2019.
[doc] pyjanitor documentation. https://pyjanitor.readthedocs.io. Accessed: 22 May 2019.
[dpla] Dplython: Dplyr for python. https://github.com/dodger487/dplython. Accessed: 22 May 2019.
[dplb] A grammar of data manipulation: dplyr. https://dplyr.tidyverse.org. Accessed: 22 May 2019.
[flu] Fluentinterface. https://martinfowler.com/bliki/FluentInterface.html. Accessed: 22 May 2019.
[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.
[jan] janitor: Simple tools for examining and cleaning dirty data. https://github.com/sfirke/janitor. Accessed: 22 May 2019.
[kad] A friendly pandas wrapper with a more composable grammar support. https://github.com/koaning/kadro. Accessed: 22 May 2019.
[nyt] For big-data scientists, 'janitor work' is key hurdle to insights. https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html. Accessed: 22 May 2019.
[pan] pandas-ply: functional data manipulation for pandas. https://github.com/coursera/pandas-ply. Accessed: 24 November 2019.
[pdp] https://github.com/shaypal5/pdpipe. Accessed: 22 May 2019.
[pf] Pandas flavor: The easy way to write your own flavor of pandas. https://github.com/Zsailer/pandas_flavor. Accessed: 22 May 2019.
[PL88] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. U.S.A.*, 85(8):2444–2448, Apr 1988.
[ply] A grammar for data manipulation in python. https://github.com/has2k1/plydata. Accessed: 22 May 2019.
[rdk] Rdkit: Open-source cheminformatics. http://www.rdkit.org. Accessed: 22 May 2019.
[Wic14] Hadley Wickham. Tidy data. *Journal of Statistical Software, Articles*, 59(10):1–23, 2014. URL: https://www.jstatsoft.org/v059/i10, doi:10.18637/jss.v059.i10.