Python data pipelines similar to R Documentation

Release 0.1.0

Jan Schulz

October 23, 2016

Contents

Contents:

CHAPTER 1

Python data pipelines

1.1 Features

This package implements the basics for building pipelines similar to magrittr in R. Pipelines are created using >>. Internally it uses singledispatch to provide a way for a unified API for different kinds of inputs (SQL databases, HDF, simple dicts, ...).

Basic example what can be build with this package:

```
>>> from my_library import append_col
>>> import pandas as pd

>>> pd.DataFrame({"a" : [1,2,3]}) >> append_col(x=3)
   a  X
0  1  3
1  2  3
2  3  3
```

In the future, this package might also implement the verbs from the R packages dplyr and tidyr for `pandas.DataFrame` and or I will fold this into one of the other available implementation of dplyr style pipelines/verbs for pandas.

1.2 Documentation

The documentaiton can be found on ReadTheDocs: https://pydatapipes.readthedocs.io

1.3 License

Free software: MIT license

1.4 Credits

- magrittr and it's usage in dplyr / tidyr for the idea of using pipelines in that ways
- lots of python implementations of dplyr style pipelines: dplython, pandas_ply, dfply

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install Python data pipelines similar to R, run this command in your terminal:

```
$ pip install pydatapipes
```

This is the preferred method to install Python data pipelines, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Python data pipelines similar to R can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/janschulz/pydatapipes
```

Or download the tarball:

```
$ curl  -OL https://github.com/janschulz/pydatapipes/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Background

Since a few years, pipelines (via `%>%` of the magrittr package) are quite popular in R and the grown ecosystem of the "tidyverse" is built around pipelines. Having tried both the pandas syntax (e.g. chaining like `df.groupby().mean()` or plain `function2(function1(input))`) and the R's pipeline syntax, I have to admit that I like the pipeline syntax a lot more.

In my opinion the strength of R's pipeline syntax is:

- The **same verbs can be used for different inputs** (there are SQL backends for dplyr), thanks to R's single-dispatch mechanism (called S3 objects).
- Thanks to **using function** instead of class methods, it's also more easily extendable (for a new method on `pandas.DataFrame` you have to add that to the pandas repository or you need to use monkey patching). Fortunatelly, both functions and singledispatch are also available in python :-)
- It **uses normal functions** as pipline parts: `input %>% function()` is equivalent to `function(input)`. Unfortunately, this isn't easily matched in python, as pythons evaluation rules would first evaluate `function()` (e.g. call functions without any input). So one has to make `function()` return a helper object which can then be used as a pipeline part.
- R's delayed evaluation rules make it easy to **evaluate arguments in the context of the pipeline**, e.g. `df %>% select(x)` would be converted to the equivalent of pandas `df[["x"]]`, e.g. the name of the variable will be used in the selection. In python it would either error (if `x` is not defined) or (if `x` was defined, e.g. `x = "column"`), would take the value of `x`, e.g. `df[["column"]]`. For this, some workarounds exist by using helper objects like `select(X.x)`, e.g. *pandas-ply and its ''Symbolic expression'* <https://github.com/coursera/pandas-ply>'__.

There exist a few implementation of dplyr like pipeline verbs for python (e.g. pandas itself, pandas-ply (uses method chaining instead of a pipe operator), dplython, and dfply), but they all focus on implementing dplyr style pipelines for `pandas.DataFrames` and I wanted to try out a simpler but more general approach to pipelines.

CHAPTER 4

Usage

4.1 Simple pipeline verbs

For end users wanting to build a new pipeline verb or add pipeline functionality to a new data source, there are two functions to build new pipeline parts:

```
from pydatapipes.pipes import singledispatch_pipeverb, make_pipesource
import pandas as pd
```

```
# generic version which defines the API and should raise NotImplementedError
@singledispatch_pipeverb
def append_col(input, x = 1):
    """Appends x to the data source"""
    raise NotImplementedError("append_col is not implemented for data of type %s" % type(input))

# concrete implementation for pandas.DataFrame
@append_col.register(pd.DataFrame)
def append_col_df(input, x = 1):
    # always ensure that you return new data!
    copy = input.copy()
    copy["X"] = x
    return copy

# ensure that pd.DataFrame is useable as a pipe source
make_pipesource(pd.DataFrame)
```

This can then be used in a pipeline:

```
import pandas as pd
print(pd.DataFrame({"a" : [1,2,3]}) >> append_col(x=3))
```

```
   a  X
0  1  3
1  2  3
2  3  3
```

The above example implements a pipeline verb for `pandas.DataFrame`, but due to the useage of `singledispatch`, this is generic. By implementing additional `append_col_<data_source_type>()` functions and registering it with the original `append_col` function, the `append_col` function can be used with other data sources, e.g. SQL databases, HDF5, or even builtin data types like `list` or `dict`:

```
@append_col.register(list)
def append_col_df(input, x = 1):
```

```
    return input + [x]

[1, 2] >> append_col()
```

```
[1, 2, 1]
```

If a verb has no actual implementation for a data source, it will simply raise an `NotImplementedError`:

```
try:
    1 >> append_col()
except NotImplementedError as e:
    print(e)
```

```
append_col is not implemented for data of type <class 'int'>
```

4.2 A more complex example: grouped and ungrouped aggregation on a pandas DataFrame

`singledispatch` also makes it easy to work with grouped and ungrouped `pd.DataFrames`:

```
@singledispatch_pipeverb
def groupby(input, columns):
    """Group the input by columns"""
    raise NotImplementedError("groupby is not implemented for data of type %s" % type(input))

@groupby.register(pd.DataFrame)
def groupby_DataFrame(input, columns):
    """Group a DataFrame"""
    return input.groupby(columns)

@singledispatch_pipeverb
def summarize_mean(input):
    """Summarize the input via mean aggregation"""
    raise NotImplementedError("summarize_mean is not implemented for data of type %s" % type(input))

@summarize_mean.register(pd.DataFrame)
def summarize_mean_DataFrame(input):
    """Summarize a DataFrame via mean aggregation"""
    return input.mean()

@summarize_mean.register(pd.core.groupby.GroupBy)
def summarize_mean_GroupBy(input):
    """Summarize a grouped DataFrame via mean aggregation"""
    return input.mean()
```

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

```
print(df >> summarize_mean())
```

```
a    2.5
b    1.5
dtype: float64
```

```
print(df >> groupby("b") >> summarize_mean())
```

```
     a
b
1  1.5
2  3.5
```

4.3 Limitiations

Compared to R's implementation in the magrittr package, `input >> verb(x)` can't be used as `verb(input, x)`.

The problem here is that `verb(x)` under the hood constructs a helper object (`PipeVerb`) which is used in the rshift operation. At the time of calling `verb(...)`, we can't always be sure whether we want an object which can be used in the pipeline or already compute the result. As an example consider a verb `merge(*additional_data)`. You could call that as `data >> merge(first, second)` to indicate that you want all three (`data`, `first`, and `second`) merged. On the other hand, `merge(first, second)` is also valid ("merge `first` and `second` together).

4.4 Usage as function and pipeline verb

To help work around this problem, the convenience decorator `singledispatch_pipeverb` is actually not the best option if you want to create reuseable pipline verbs. Instead, the `singledispatch_pipeverb` decorator is also available in two parts, so that one can both expose the original function (with `singledispatch` enabled) and the final pipeline verb version:

```
#from pydatapipes.pipes import pipeverb, singledispatch

# first use singledispatch on the original function, but define it with a trailing underscore
@singledispatch
def my_verb_(input, x=1, y=2):
    raise NotImplemented("my_verb is not implemented for data of type %s" % type(input))

# afterwards convert the original function to the pipeline verb:
my_verb = pipeverb(my_verb_)

# concrete implementations can be registered on both ``my_verb`` and ``my_verb_``
@my_verb_.register(list)
def my_verb_df(input, x=1, y=2):
    return input + [x, y]
```

A user can now use both versions:

```
[1] >> my_verb(x=2, y=3)
```

```
[1, 2, 3]
```

```
my_verb_([9], x=2, y=3)
```

```
[9, 2, 3]
```

4.5 Rules and conventions

To work as a pipline verb, functions **must** follow these rules:

- Pipelines assume that the verbs itself are side-effect free, i.e. they do not change the inputs of the data pipeline. This means that actual implementations of a verb for a specific data source must ensure that the input is not changed in any way, e.g. if you want to pass on a changed value of a `pd.DataFrame`, make a copy first.
- The initial function (not the actual implementations for a specific data source) should usually do nothing but simply raise `NotImplementedError`, as it is called for all other types of data sources.

The strength of the tidyverse is it's coherent API design. To ensure a coherent API for pipeline verbs, it would be nice if verbs would follow these conventions:

- Pipeline verbs should actually be named as verbs, e.g. use `input >> summarize()` instead of `input >> Summary()`
- If you expose both the pipeline verb and a normal function (which can be called directly), the pipeline verb should get the "normal" verb name and the function version should get an underscore _ appended: `x >> verb()` -> `verb_(x)`
- The actual implementation function of a `verb()` for a data source of class `Type` should be called `verb_Type(...)`, e.g. `select_DataFrame()`

4.6 Missing parts

So what is missing? Quite a lot :-)

- Symbolic expressions: e.g. `select(X.x)` instead of `select("x")`
- Helper for dplyr style column selection (e.g. `select(starts_with("y2016_"))` and `select(X[X.first_column:X.last_column])`)
- all the dplyr, tidyr, ... verbs which make the tidyverse so great

Some of this is already implemented in the other dplyr like python libs (pandas-ply, dplython, and dfply), so I'm not sure how to go on. I really like my versions of pipelines but duplicating the works of them feels like a waste of time. So my next step is seeing if it's possible to integrate this with one of these solutions, probably dfply as that looks the closest implementation.

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/janschulz/pydatapipes/issues.

If you are 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.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

Python data pipelines similar to R could always use more documentation, whether as part of the official Python data pipelines similar to R docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/janschulz/pydatapipes/issues.

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 contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *pydatapipes* for local development.

1. Fork the *pydatapipes* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pydatapipes.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pydatapipes
$ cd pydatapipes/
$ python setup.py develop
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 pydatapipes tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.

2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/janschulz/pydatapipes/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_pipes
```

CHAPTER 6

Credits

6.1 Development Lead

- Jan Schulz <jasc@gmx.net>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.1.0 (2016-10-22)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search