proof Documentation

Release 0.3.0 (alpha)

Christopher Groskopf

September 25, 2015

Contents

CHAPTER 1

About

proof is a Python library for creating optimized, repeatable and self-documenting data analysis pipelines.

proof was designed to be used with the agate data analysis library, but can be used with numpy, pandas or any other method of processing data.

Important links:

- Documentation: http://proof.rtfd.org
- Repository: https://github.com/onyxfish/proof
- Issues: https://github.com/onyxfish/proof/issues

CHAPTER 2

Why use proof?

- Encourages self-documenting code patterns.
- Caches output of analyses for faster repeat execution.
- Plays well with agate or any other data analysis library.
- Pure Python. It runs everywhere.
- As simple as it gets. There is only one class.
- 100% test coverage.
- Thorough, accurate user documentation.

CHAPTER 3

Installation

3.1 Users

If you only want to use proof, install it this way:

```
pip install proof
```

3.2 Developers

If you are a developer that also wants to hack on proof, install it this way:

```
git clone git://github.com/onyxfish/proof.git
cd proof
mkvirtualenv proof
pip install -r requirements.txt
python setup.py develop
tox
```

Note: proof also supports running tests with coverage:

```
nosetests --with-coverage --cover-package=proof
```

3.3 Supported platforms

proof supports the following versions of Python:

- Python 2.6+
- Python 3.2+
- Latest PyPy

It is tested on OSX, but due to it's minimal dependencies should work fine on both Linux and Windows.

CHAPTER 4

Usage

proof creates data processing pipelines by defining "analyses", each of which is a stage in the process. These analyses naturally flow from one to another. For instance, the first analysis of a process might load a CSV of data. From there you might select a subset of the rows in the table, then group the results and finally take the median of each group. More complex pipelines might alse diverge at some point, having several analyses that use the same input, but each produce different outputs.

proof contains a single class, `Analysis`, which is used for creating processes like these. Each of the analyses is constructed by instantiating it with a function that does some work:

```
import proof

def load_data(data):
    # Load the data
    pass

data_loaded = proof.Analysis(load_data)
```

Analyses which depend on the result of this stage can then be created using the `Analysis.then()` method.

```
def select_rows(data):
    # Select relevant rows from the table
    pass

def calculate_average(data):
    # The average of a value in the rows is taken
    pass

data_loaded.then(select_rows)
data_loaded.then(calculate_average)
```

In the previous example, both `select_rows` and `calculate_average` depend on the result of `load_data`. If instead we wanted our average to be based on only the selected rows, we would instead do:

```
rows_selected = data_loaded.then(select_rows)
rows_selected.then(calculate_average)
```

Each analysis function must accept a `data` argument, which is a `dict` of data to be persisted between analyses. Modifications made to `data` in the scope of one analysis will be propagated to all dependent analyses. For example, the three functions we saw before might be implemented like this:

```
import csv

def load_data(data):
```

```
    # Load the data
    with open('example.csv') as f:
        reader = csv.DictReader(f, fieldnames=['name', 'salary'])
        reader.next()
        data['table'] = list(reader)

def select_rows(data):
    # Select relevant rows from the table
    data['low_income'] = filter(lambda r: int(r['salary']) < 25000, data['table'])

def calculate_average(data):
    # The average of a value in the rows is taken
    mean = sum([int(r['salary']) for r in data['low_income']]) / len(data['low_income'])
    print(mean)
```

We can see here how the `data` dictionary gets passed from function to function with it's state intact. You can also modify values that already exist in data and those changes will be propagated forward.

Finally, we run the analysis, starting at the beginning, by calling *`Analysis.run()`*:

```
data_loaded.run()
```

If we execute the script we've created, the output is:

```
Running: load_data
Refreshing: select_rows
Refreshing: calculate_average
13500
```

When `Analysis.run()` is invoked, the analysis function runs, followed by each of dependent analyses created with `Analysis.then()`. These in turn invoke their own dependent analyses, allowing a hierarchy to be created. Within each of those functions you can do whatever you want–print to the console, import other dependencies, save to disk–proof doesn't care *how* you analyze your data.

After each analysis the value of `data` is cached to disk along with a "fingerprint" describing the source code of the analysis function at the time it was invoked. If you run the same analysis twice without modifying the code, the cached version out of the `data` will be used for its dependents. This allows you to experiment with a dependent analysis without constantly recomputing the results of its parent. For example, if I rerun the previous script, I will see:

```
Deferring to cache: load_data
Deferring to cache: select_rows
Deferring to cache: calculate_average
```

This indicates that the results of each analysis will be loaded from disk if they are needed. proof tries to be very smart about how much work it does. So, for instance, if you modify the middle analysis in this process, `select_rows`, only it and other analyses that depend on it will be rerun. Try modifying the threshold for `low_income` down to `20000` and rerun the script. You should see:

```
Deferring to cache: load_data
Stale cache: select_rows
Refreshing: calculate_average
10666
```

Warning: One very important caveat exists to this automated dependency resolution. The fingerprint which is generated for each analysis function is **not recursive**, which is to say, it does not include the source of any functions which are invoked by that function. If you modify the source of a function invoked by the analysis function, you will need to ensure that the analysis is manually refreshed by passing `refresh=True` to `Analysis.run()` or deleting the cache directory (`.proof` by default).

Sometimes there are analysis functions you always want to run, even if they are up to date. This is most commonly the case when you simply want to print your results. proof allows you to flag a that an analysis function should always run using the `never_cache()` decorator. Let's modify our previous example to move the `print` statement into a separate analysis:

```
def calculate_average(data):
    # The average of a value in the rows is taken
    data['mean'] = sum([int(r['salary']) for r in data['low_income']]) / len(data['low_income'])

@proof.never_cache
def print_results(data):
    print(data['mean'])

data_loaded = proof.Analysis(load_data)
rows_selected = data_loaded.then(select_rows)
average_calculated = rows_selected.then(calculate_average)
average_calculated.then(print_results)

data_loaded.run()
```

Now when you run the analysis the results will always be printed:

```
Deferring to cache: load_data
Deferring to cache: select_rows
Deferring to cache: calculate_average
Never cached: print_results
10666
```

CHAPTER 5

API

There is only one class!

class `proof.`**`Analysis`**(*func*, *cache_dir='.proof'*, *_trace=[]*)

An Analysis is a function whose source code fingerprint and output can be serialized to disk. When it is invoked again, if it's code has not changed the serialized output will be used rather than executing the code again.

Implements a callback-like API so that Analyses can depend on one another. If a parent analysis changes then it and all it's children will be refreshed.

Parameters

- **`func`** – A callable that implements the analysis. Must accept a *data* argument that is the state inherited from its ancestors analysis.
- **`cache_dir`** – Where to stored the cache files for this analysis.
- **`_trace`** – The ancestors this analysis, if any. For internal use only.

`then`(*child_func*)

Create a new analysis which will run after this one has completed with access to the data it generated.

Parameters **`func`** – A callable that implements the analysis. Must accept a *data* argument that is the state inherited from its ancestors analysis.

`run`(*refresh=False*, *_parent_cache=None*)

Execute this analysis and its descendents. There are four possible execution scenarios:

1. This analysis has never been run. Run it and cache the results.
2. This analysis is the child of a parent analysis which was run, so it must be run because its inputs may have changed. Cache the result.
3. This analysis has been run, its parents were loaded from cache and its fingerprints match. Load the cached result.
4. This analysis has been run and its parents were loaded from cache, but its fingerprints do not match. Run it and cache updated results.

On each run this analysis will clear any unused cache files from the cache directory. If you have multiple analyses running in the same location, specify separate cache directories for them using the `cache_dir` argument to the the `Analysis` constructor.

Parameters **`refresh`** – Flag indicating if this analysis must refresh because one of its ancestors did.

Parent _parent_cache Data cache for the parent analysis. For internal usage only.

(There is also one decorator.)

proof.**never_cache**(*func*)

Decorator to flag that a given analysis function should never be cached.

CHAPTER 6

Authors

The following individuals have contributed code to proof:

- Christopher Groskopf

CHAPTER 7

License

CHAPTER 8

Changelog

8.1 0.3.0

- Fix errors when caching unicode analysis function names.
- Fix errors when caching unicode source. (#10)
- Add six dependency.
- Add decorator and logic for analysis functions that are never cached. (#5)

8.2 0.2.0

- Don't load analysis cache unless needed. (#1)

8.3 0.1.0

- Initial code migration from agate.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

A

N

R

T