flume Documentation

Release 0.1.0

iLoveTux

Jun 22, 2018

Contents:

CHAPTER 1

flume

Let the logs flow

Flume is a general-purpose stream processing framework. It includes a simple but powerful templating system and all the utilities to shape your data streams.

NOTE: This is new code. Master is in flux and docs are lacking, but it is in a point where it could be useful to someone. If it is useful to you, help us get to 1.0.0. You can start by reading the contributing guide at https://github.com/ilovetux/flume/CONTRIBUTING.rst.

- Free software: GNU General Public License v3
- Documentation: https://flume.readthedocs.io.

1.1 Features

- Simple, Powerful templating system
- Extensible input and output system
- Command line utilities
- TODO: Web GUI
- TODO: Many default use cases covered
- TODO: –no-overwrite option
- TODO: Improve test coverage

1.2 Installing

Currently the only way to install this package is to clone it which should look like the following:

```
$ git clone https://github.com/ilovetux/flume
$ cd flume
```

Then you can run the tests:: $ python setup.py test

And if they pass (fingers crossed!), go ahead with:

```
$ pip install .
```

1.3 Concepts

The built-in templating engine is very simple, it consists of a namespace and a template. The template is rendered within the context of the namespace.

Rendering involves two stages:

1. scanning the template for strings matching the pattern *{%<Expression>%}* where <*Expression*> is Python source code which is executed (*exec*) within the context of the namespace and removed from the output.
2. scanning the remaining output for strings matching the pattern *{{<Statement>}}* where <*Statement*> is a Python statement which is replaced (along with *{{* and *}}*) with the value to which it evaluates (*eval*)

This concept is applied to a variety of use cases and embodied in the form of command line utilities which cover a number of common use cases.

1.3.1 Usage

The command line utility, flume, can be run in two modes:

1. Streaming mode: Data is streamed through and used to populate templates
2. Document mode: Render files src and write the results to dst

1.4 Streaming mode

Streaming mode runs in the following manner:

1. reads data from *filenames*, which defaults to stdin
2. At this point any expressions passed to *–begins* are executed
3. The files specified are processed as follows in order
    1. Any expressions passed to *–begin-files* are executed
    2. The data from the current file is read line-by-line
        1. Any statements passed to *–tests* are evaluated
        2. Iff all tests pass, the following process is performed.
        1. Any expressions passed to *–begin-lines* are executed
        2. Any templates are rendered through the python logging system
        3. Any expressions passed to *–end-lines* are executed
    3. Any expressions passed to *–end-files* are executed

4. Any expressions passed to *–ends* are executed

Below are a few examples. See the documentation for more details:

```
$ # Like grep
$ flume stream --test "'error' in line.lower()" --template "{{line}}" *.log
$ # Like wc -l
$ flume stream --end-files "print(fnr, filename)" *.log
$ # Like wc -wl
$ flume stream --begins "words=0" --begin-lines "words += nf" --end-files
↪"print(words, fnr, filename)"
$ # Find the count of numbers "\d+" for each line
$ flume stream --begins "import re" --begin-lines "print(re.findall(r'\d+', line))" *.
↪log
```

Please see the documentation for more as well as trying:

```
$ flume stream --help
```

Important Note:

If anything passed to any of the hooks is determined to exist by *os.path.exists* then it will be read and executed as if that text was passed in on the CLI. This is useful for quickly solving character escaping issues.

1.5 Document mode

Document mode runs tries to render a group of files from one location to another. It is used like this:

```
$ flume doc <src> <dst>
```

There are options to control behavior, but the gist of it is:

1. if src is a file
1. if dst is a filename, src is rendered and written to dst
2. if dst is a directory, src is rendered and written to a file in dst with the same basename as src
2. if src is a directory
1. dst must be a directory and every file in src is rendered into a file in dst with the same basename as the file from src
2. If *–recursive* is specified, the subdirectories will be reproduced in dst

Some important notes:

- File and directory names can be templated
- If *–interval* is passed an integer value, the program will sleep for that many seconds and check for changes to your templates in which case they will be re-rendered

1.6 Use Cases

Streaming mode is great for processing incoming log files with *tail –follow=name* or for ad-hoc analysis of text files.

Document mode is incredibly useful for a powerful configuration templating system. The *–interval* option is incredibly useful as it will only re-render on a file change, so is great for developing your templates as you can view the results in real-time.

Document mode is also useful for near-real-time rendering of static web resources such as charts, tables, dashboards and more.

1.7 Credits

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

CHAPTER 2

Installation

2.1 Stable release

NOTE: This will not work! This project is pending a rename because “flume” is not available on PyPI. Looking for suggestions at https://github.com/iLoveTux/flume/issues/4

To install flume, run this command in your terminal:

```
$ pip install flume
```

This is the preferred method to install flume, 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 flume can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/ilovetux/flume
```

Or download the tarball:

```
$ curl  -OL https://github.com/ilovetux/flume/tarball/master
```

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

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

CHAPTER 3

Usage

In streaming mode, your templates and hooks are ‘eval‘uated or ‘exec‘uted within an single, shared namespace. This namespace is injected with the following variables at various times throughout processing:

- *filename*: The filename currently being processed
- *line*: The text of the current line
- *fields*: The result of calling *line.split(field_sep)*
- *nr*: The number of the current record being processed
- *fnr*: The number of the current record within the current file
- *nf*: The result of *len(line.split(field_sep))*

to run in streaming mode, use the stream subcommand. Here is an example of a command which is similar to grep:

```
$ flume stream --test "'error' in line.lower()" --template {{line}} *.log
```

Here is a command which outputs the number of lines in each file:

```
$ flume stream --end-files "print(fnr, filename)" *.log
```

Here is an updated example which also prints out the word count:

```
$ flume stream --begins "words=0" --begin-lines "words += nf" --end-files
↪"print(words, fnr, filename)"
```

Everything that looks advanced is literally just Python, so it’s easy to pick up and batteries are included. Imports work just fine and there is no magic. Here is an example which uses the *re* module find the count of numbers:

```
$ flume stream --begins "import re" --begin-lines "print(re.findall(r'\d+', line))" *.
↪log
```

A list of the hooks you can tie into are, If more than any are provided, they are processed in the order given. If a filename is given and is said to exist by *os.path.exists*, then that file is read in and executed by `runpy.run_path`:

If Python source code is provided, then that is ‘exec‘uted.

- –begins: Executed once at startup.
- –begin-files: Executed once for each file processed.
- **–begin-lines: Executed once for each line processed. These hooks are only** executed if all tests specified by *–tests* evaluate to Truthy values.
- **–end-lines: Executed once after any processing of line is complete.** end-lines are rendered regardless of the results of *–tests*.
- –end-files: Executed once after processing is complete for each file.
- **–ends: Executed once after all lines are complete. This means that** either all files are exhausted or *Ctrl + C* has been pressed.

Other parameters:

- **–tests: Each of these are 'eval'uated when a new line is received.** if and only if all tests provided evaluate to Truthy values processing of the line will continue otherwise processing is continued with the next line.
- **–templates: Templates are treated differently. Templates are rendered** once per line according to the rules defined above in "Concepts". The result of each rendering is put out to a logger unique to that template. This allows the Python *logging.config* package to provide a very fine grain of control. The main use case for this is to extract information according to a variety of KPI and output to multiple destinations, while also maintaining a record of authority.

3.1 Document mode

In document mode, your templates reside in files and are read from *src* and written to *dst*.The behavior differs depending on the values provided for *src* and *dst*.

If *src* is a directory or multiple values are provided for *src* then *dst* must be a directory in which case all files in *src* will be rendered into *dst*. If *–recursive* is specified then files will be rendered recursively from subdirectories within *src*.

If *src* is a file then *dst* can be either a directory or a filename. If a filename is provided then *src* will be rendered into that file, otherwise if a directory is provided for *dst* then a file with the same name as *src* will be created.

If *–interval* is specified, then after all files are rendered the process will sleep for the specified interval. When the process awakens again all files in *src* will be examined and if any have changed then that file is re-rendered into *dst*. Said process will continue indefinately until the process is killed, ie by pressing *Ctrl + C*.

To use flume in a project:

```
from flume.render import render
```

CHAPTER 4

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:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/ilovetux/flume/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.

4.1.2 Fix Bugs

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

4.1.3 Implement Features

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

4.1.4 Write Documentation

flume could always use more documentation, whether as part of the official flume docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ilovetux/flume/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 :)

4.2 Get Started!

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

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

```
$ git clone git@github.com:your_name_here/flume.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 flume
$ cd flume/
$ 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 flume 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.

4.3 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 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/ilovetux/flume/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_flume
```

4.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 5

Credits

5.1 Development Lead

- iLoveTux <cliffbressette@gmail.com> (Original Author)

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.1.0 (2018-06-20)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search