pipettor Documentation

Release 1.0.0

Mark Diekhans

Jun 29, 2023

Contents

Robust, easy to use Python package for running Unix process pipelines

CHAPTER 1

Contents:

1.1 Pipettor Overview

pipettor - robust, easy to use Python package for running Unix process pipelines

1.1.1 Features

- Creating process pipelines in Python is either complex (e.g. `subprocess`), or not robust (e.g. `os.system()`). This package aims to address these shortcomings.
- Command pipelines are simply specified as a sequence of commands, with each command represented as a sequence of arguments.
- Failure of any process in the pipeline results in an exception, with `stderr` included in the exception.
- Pipeline `stdin/stdout/stderr` can be passed through from parent process, redirected to a file, or read/written by the parent process.
- Asynchronous reading and writing to and from the pipeline maybe done without risk of deadlock.
- Pipeline can run asynchronously or block until completion.
- Popen-style File-like objects for reading or writing a pipeline.
- Documentation: https://pipettor.readthedocs.org.

1.2 Installation

At the command line:

```
$ pip install pipettor
```

Source is available from: https://github.com/diekhans/pipettor

1.3 Usage

A single process in pipettor is specified as sequence (list or tuple) of the command and its arguments. A process pipeline is specified as a sequence of such commands (lists of lists, lists of tuples, etc). Functions to create processes check if a specified command is a sequence of commands or a single command based on the sequence structure.

Example commands are:

```
("date",)
("sort", "-u", "/etc/stuff")
[("sort", "-u", "/etc/stuff"), ("wc", "-l")]
```

Commands are *not* run through the UNIX shell to prevent security and robustness problems.

A non-zero exit or signal termination from any process in a pipe results in a `pipettor.ProcessException`, which contains the *stderr* of the failed process unless redirected.

The simplest way to execute a pipeline synchronously is to use the `pipettor.run()` or `pipettor.runout()` functions:

```
import pipettor
pipettor.run([("sort", "-u", "/etc/hosts"), ("wc", "-l")], stdout="hosts.linecnt")
out = pipettor.runout([("sort", "-u", "/etc/hosts"), ("wc", "-l")])
```

File-like objects to or from a pipeline maybe create using the `pipettor.Popen` class:

```
import pipettor
rfh = pipettor.Popen([("sort", "-u", "/etc/hosts"), ("wc", "-l")])
wfh = pipettor.Popen([("sort", "-u"), ("wc", "-l")], "w", stdout="uniq.linecnt")
```

In-memory data can be also be written to pipelines using `pipettor.DataWriter` objects:

```
import pipettor
dw = pipettor.DataWriter("line3\nline1\nline2\nline1\n")
pipettor.run([("sort", "-u",), ("wc", "-l")], stdin=dw, stdout="writer.linecnt")
```

Data can be read from pipelines into memory using `pipettor.DataReader` objects:

```
import pipettor
dr = pipettor.DataReader()
pipettor.run([("sort", "-u", "/etc/hosts"), ("wc", "-l")], stdout=dr)
print dr.data
```

The `pipettor.runlex()` or `pipettor.runlexout()` functions pass string arguments through *shlex.split* to split them into arguments:

```
import pipettor
out = pipettor.runlexout("sort -u /etc/hosts")
out = pipettor.runlexout(["sort -u /etc/hosts", ("wc", "-l")])
```

Full control of process pipelines can be achieved using `pipettor.Pipeline` class directly. The `pipettor.DataReader` and `pipettor.DataWriter` object create threads, allowing for both reading and writing to a process without risk of deadlocking.

1.4 Pipettor Library

1.4.1 Function Interface

pipettor.**run**(*cmds*, *stdin=None*, *stdout=None*, *stderr=<class 'pipettor.devices.DataReader'>*, *logger=None*, *logLevel=None*)

Construct and run an process pipeline. If any of the processes fail, a ProcessException is throw.

cmds is either a list of arguments for a single process, or a list of such lists for a pipeline. If the *stdin*, *stdout*, or *stderr* arguments are none, the open files are are inherited. Otherwise they can be string file names, file-like objects, file number, or `pipettor.Dev` object. *stdin* is input to the first process, *stdout* is output to the last process and *stderr* is attached to all processed. `pipettor.DataReader` and `pipettor.DataWriter` objects can be specified for *stdin*, *stdout*, or *stderr* asynchronously I/O with the pipeline without the danger of deadlock.

If stderr is the class DataReader, a new instance is created for each process in the pipeline. The contents of stderr will include an exception if an occurs in that process. If an instance of `pipettor.DataReader` is provided, the contents of stderr from all process will be included in the exception.

pipettor.**runout**(*cmds*, *stdin=None*, *stderr=<class 'pipettor.devices.DataReader'>*, *logger=None*, *logLevel=None*, *buffering=-1*, *encoding=None*, *errors=None*)

Construct and run an process pipeline, returning the output. If any of the processes fail, a ProcessException is throw.

See the `pipettor.run()` function for more details. Use *str.splitlines()* to split result into lines.

The logger argument can be the name of a logger or a logger object. If none, default is user.

Specifying binary access results in data of type bytes, otherwise str type is return. The buffering, encoding, and errors arguments are as used in the open() function.

pipettor.**runlex**(*cmds*, *stdin=None*, *stdout=None*, *stderr=<class 'pipettor.devices.DataReader'>*, *logger=None*, *logLevel=None*)

Call `pipettor.run()`, first splitting commands specified as strings are split into arguments using *shlex.split*.

If *cmds* is a string, it is split into arguments and run as as a single process. If *cmds* is a list, a multi-process pipeline is created. Elements that are strings are split into arguments to form commands. Elements that are lists are treated as commands without splitting.

pipettor.**runlexout**(*cmds*, *stdin=None*, *stderr=<class 'pipettor.devices.DataReader'>*, *logger=None*, *logLevel=None*, *buffering=-1*, *encoding=None*, *errors=None*)

Call `pipettor.runout()`, first splitting commands specified as strings are split into arguments using *shlex.split*.

If *cmds* is a string, it is split into arguments and run as as a single process. If *cmds* is a list, a multi-process pipeline is created. Elements that are strings are split into arguments to form commands. Elements that are lists are treated as commands without splitting.

The logger argument can be the name of a logger or a logger object. If none, default is user.

Specifying binary access results in data of type bytes, otherwise str type is returned. The buffering, encoding, and errors arguments are as used in the open() function.

1.4.2 Pipeline Classes

class pipettor.**Pipeline**(*cmds*, ***, *stdin=None*, *stdout=None*, *stderr=<class 'pipettor.devices.DataReader'>*, *logger=None*, *logLevel=None*)

A process pipeline. Once constructed, the pipeline is started with start(), poll(), or wait() functions.

The cmds argument is either a list of arguments for a single process, or a list of such lists for a pipeline. If the stdin/out/err arguments are none, the open files are are inherited. Otherwise they can be string file names, file-like objects, file number, or Dev object. Stdin is input to the first process, stdout is output to the last process and stderr is attached to all processed. DataReader and DataWriter objects can be specified for stdin/out/err asynchronously I/O with the pipeline without the danger of deadlock.

If stderr is the class DataReader, a new instance is created for each process in the pipeline. The contents of stderr will include an exception if an occurs in that process. If an instance of DataReader is provided, the contents of stderr from all process will be included in the exception.

Command arguments will be converted to strings.

The logger argument can be the name of a logger or a logger object. If none, default is user.

class pipettor.**Popen**(*cmds*, *mode='r'*, ***, *stdin=None*, *stdout=None*, *logger=None*, *logLevel=None*, *buffering=-1*, *encoding=None*, *errors=None*)

File-like object of processes to read from or write to a Pipeline.

The cmds argument is either a list of arguments for a single process, or a list of such lists for a pipeline. Mode is 'r' for a pipeline who's output will be read, or 'w' for a pipeline to that is to have data written to it. If stdin or stdout is specified, and is a string, it is a file to open as other file at the other end of the pipeline. If it's not a string, it is assumed to be a file object to use for input or output. For a read pipe, only stdin can be specified, for a write pipe, only stdout can be used.

read pipeline ('r'): stdin –> cmd[0] –> ... –> cmd[n] –> Popen

write pipeline ('w') Popen –> cmd[0] –> ... –> cmd[n] –> stdout

Command arguments will be converted to strings.

The logger argument can be the name of a logger or a logger object. If none, default is user.

Specifying binary access results in data of type bytes, otherwise str type is returned. The buffering, encoding, and errors arguments are as used in the open() function.

1.4.3 Process I/O Classes

class pipettor.**DataReader**(***, *binary=False*, *buffering=-1*, *encoding=None*, *errors=None*)

Object to asynchronously read data from process into memory via a pipe. A thread is use to prevent deadlock when both reading and writing to a child pipeline.

Specifying binary access results in data of type bytes, otherwise str type is returned. The buffering, encoding, and errors arguments are as used in the open() function.

class pipettor.**DataWriter**(*data*, ***, *buffering=-1*, *encoding=None*, *errors=None*)

Object to asynchronously write data to process from memory via a pipe. A thread is use to prevent deadlock when both reading and writing to a child pipeline. Text or binary output is determined by the type of data.

The buffering, encoding, and errors arguments are as used in the open() function.

class pipettor.**File**(*path*, *mode='r'*)

A file path for input or output, used for specifying stdio associated with files. Mode is invalued on of standard r, w, or a

1.4.4 Logging Control

pipettor.**setDefaultLogger**(*logger*)

Set the default pipettor logger used in logging command and errors. If None, there is no default logging. The logger can be the name of a logger or the logger itself. Standard value is None

pipettor.**getDefaultLogger**()
    return the current value of the pipettor default logger

1.4.5 Exceptions

class pipettor.**PipettorException**
    Base class for Pipettor exceptions.

class pipettor.**ProcessException**(*procDesc*, *returncode=None*, *stderr=None*)
    Exception associated with running a process. A None returncode indicates a exec failure.

1.5 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:

1.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/diekhans/pipettor/issues.

If you are reporting a bug, please include:

1. Your operating system name and version.
2. Any details about your local setup that might be helpful in troubleshooting.
3. Detailed steps to reproduce the bug.

Fix Bugs

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

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

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

Submit Feedback

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

1.5.2 Get Started!

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

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

```
$ git clone git@github.com:diehkhans/pipettor.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 pipettor
$ cd pipettor/
$ 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:

```
$ make lint
$ make test
$ make test-all
```

   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.

1.5.3 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 3.6+, and for PyPy. Check https://travis-ci.org/diekhans/pipettor/pull_requests and make sure that the tests pass for all supported Python versions.

1.5.4 Tips

To run a subset of tests, supply the test class:

```
$ python -m unittest tests.pipettorTests.PipelineTests
```

To run a single of test, supply the test function:

```
$ python -m unittest tests.pipettorTests.PipelineTests.testTrivial
```

1.6 Credits

1.6.1 Development Lead

- Mark Diekhans <markd@ucsc.edu>

1.6.2 Contributors

1.7 History

1.7.1 1.0.0 (2023-06-29)

- Don't use a process group; as it caused signals to not get propagated. Processes are explicitly waited for by pid, so this will not consume the exit of other process not create by this module.

1.7.2 0.8.0 (2023-02-05)

- make most optional arguments require keyword form to help prevent errors, especially if open() options are assumed
- added more functions to make Popen objects file-like objects

1.7.3 0.7.0 (2023-01-06)

- don't fail if invalid UTF-8 characters are written to capture stderr

1.7.4 0.6.0 (2022-11-16)

- remove use of deprecated pipes module

1.7.5 0.5.0 (2020-12-25)

- Removed Python-2 support.
- Switch to using subprocess as a base rather interface directly with Unix system calls. This lets subprocess deal with various issues dealing with the Python interpreter environment.

1.7.6 0.4.0 (2018-04-21)

- Allow passing through universial newline mode for PY2.
- Fix bug with not using specified log level.

1.7.7 0.3.0 (2018-02-25)

- added open-stying buffering, encoding, and errors options
- source cleanup

1.7.8 0.2.0 (2017-09-19)

- Simplified and log of info and errors levels by removing logLevel options.
- Improvements to documentation.

1.7.9 0.1.3 (2017-06-13)

- Documentation fixes

1.7.10 0.1.2 (2017-06-11)

- First public release on PyPI.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search