drain Documentation

Release 0.0.6

Eric Potash

Mar 13, 2018

Contents

Drain is a lightweight framework for writing reproducible data science workflows in Python. The core features are:

- Turn a Python workflow (DAG) into steps that can be run by a tool like *make*.
- Transparently pass the results of one step as the input to another, handling any caching that the user requests using efficient tools like HDF and joblib.
- Enable easy *parallel* execution of workflows.
- Execute only those steps that are determined to be necessary based on timestamps (both source code and data) and dependencies, virtually guaranteeing *reproducibility* of results and efficient development.

Drain is designed around these principles:

- *Simplicity*: drain is very lightweight and easy to use. The core is just a few hundred lines of code. The steps you write in drain get executed with minimal overhead, making drain workflows easy to debug and manage.
- *Reusability*: Drain leverages mature tools drake to execute workflows. Drain provides a library of steps for data science workflows including feature generation and selection, model fitting and comparison.
- *Generality*: Virtually any workflow can be realized in drain. The core was written with extensibility in mind so new storage backends and job schedulers, for example, will be easy to incorporate.

Contents:

CHAPTER 1

Installation

1.1 Stable release

To install drain, run this command in your terminal:

```
$ pip install drain
```

This is the preferred method to install drain, 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.

1.2 From sources

The sources for drain can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/potash/drain
```

Or download the tarball:

```
$ curl  -OL https://github.com/potash/drain/tarball/master
```

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

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

CHAPTER 2

Usage

2.1 Arithmetic Example

This is a toy example, in which each `Step` produces a number.

1. We define a simple `Step` that wraps a numeric value:

```
class Scalar(Step):
    def __init__(self, value, **kwargs):

        # note how we do not need to say self.value=value; the parent constructor does that for us
        Step.__init__(self, value=value, **kwargs)

    def run(self):
        return self.value
```

2.

```
> s = Scalar(value=5)
```

Note that the result of a step's `run()` method is accessible via `get_result()`.

3. Steps can use the results of others steps, called `inputs`. For example we can define an `Add` step which adds the values of its inputs:

```
class Add(Step):
    def __init__(self, inputs):
        Step.__init__(self, inputs=inputs)

    def run(self, *values)
        return sum((i.get_result() for i in self.inputs))
```

In order to avoid calling `get_result()`, drain does so-called inputs mapping which is explained in the corresponding section below. In its most basic form, inputs mapping allows us to rewrite `Add.run` as follows:

```
def run(self, *values):
    return sum(values)
```

```
a = Add(inputs = [Scalar(value=v) for v in range(1,10)])
```

2.2 How does `drain` work?

`drain` is a pretty lightweight wrapper around `drake`; its core functionality is only a few hundred lines of code.

2.3 Steps

A workflow consists of steps, each of which is inherited from the drain.step.Step class. Each step must implement the `run()` method, whose return value is the `result` of the step. A step should be a deterministic function from its constructor arguments to its result.

Because a step is only a function of its arguments, serialization and hashing is easy. We use YAML for serialization, and hash the YAML for hashing. Thus all arguments to a step's constructor should be YAML serializable.

2.3.1 Design decisions

- `Step`'s constructor accepts any keyword argument, but does **not** accept positional arguments.
- A `Step` can decide to only accept certain keyword arguments by defining a custom `__init__()`.
- Reserved keyword arguments are `name`, `target`, `inputs`, `inputs_mapping`, and `resources`. These are handled specifically by `Step.__new__()`.
- When passing keyword arguments to a `Step` constructor, then all the arguments (except `name` and `target`) become part of the signature (i.e., they will be part of this `Step`'s serialization). Any instance of a `Step` automatically has an attribute `_kwargs` holding these arguments.
- When a `Step` does not override `__init__()` (i.e., when it uses the default `Step.__init__()`), then all the keyword arguments that are being passed become attributes of the new instance. This is a mere convenience functionality. It can be overriden simply by overriding `__init__()`, and it does not affect serialization.

Each `Step` has several reserved keyword arguments, namely `target`, `name`,inputs_mapping,resources, `and`inputs‘.

2.3.2 `name` and `target`

`name` defaults to None and `target` to `False`. `name` is a string and allows you to name your current `Step`; this is useful later, when handling the step graph. `target` decides if the `Step`'s output should be cached on disk or not. These two arguments are *not* serialized.

2.3.3 `inputs`

The step attribute `inputs` should be a list of input step objects. Steps appearing in other arguments will not be run correctly. Note that the `Step.__init__` superconstructor automatically assigns all keywords to object attributes.

Inputs can also be declared within a step's constructor by setting the `inputs` attribute.

2.3.4 inputs_mapping

The inputs_mapping argument to a step allows for convenience and flexibility in passing that step's inputs' results to the step's run() method.

Default behavior

By default, results are passed as positional arguments. So a step with inputs=[a, b] will have run called as

```
run(a.get_result(), b.get_result())
```

When a step produces multiple items as the result of run() it is often useful to name them and return them as a dictionary. Dictionary results are merged (with later inputs overriding earlier ones?) and passed to run as keyword arguments. So if inputs a and b had dictionary results with keys a_0, a_1 and b_0, b_1, respectively, then run will be called as

```
run(a_0=a.get_result()['a_0'], a_1=a.get_result()['a_1'],
    b_0=a.get_result()['b_0'], b_1=b.get_result()['b_1'])
```

Custom behavior

This mapping of input results to run arguments can be customized when constructing steps. For example if the results of a and b are objects then specifying

```
inputs_mapping = ['a', 'b']
```

will result in the call

```
run(a=a.get_result(), b=b.get_result()
```

If a and b return dicts then the mapping can be used to change their keywords or exclude the values:

```
inputs_mapping = [{'a_0':'alpha_0', 'a_1': None}, {'b_1':'beta_1'}]
```

will result in the call

```
run(alpha_0=a.get_result()['a_0'],
    b_0=a.get_result()['b_0'], beta_1=b.get_result()['beta_1'])
```

where: - a_0 and b_1 have been renamed to alpha_0 and alpha_1, respectively - a_1 has been excluded, and - b_0 has been preserved.

To ignore the inputs mapping simply define

```
def run(self, *args, **kwargs):
    results = [i.get_result() for i in self.inputs]
```

CHAPTER 3

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:

3.1 Types of Contributions

3.1.1 Report Bugs

Report bugs at https://github.com/potash/drain/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.

3.1.2 Fix Bugs

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

3.1.3 Implement Features

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

3.1.4 Write Documentation

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

3.1.5 Submit Feedback

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

3.2 Get Started!

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

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

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

3.3 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/potash/drain/pull_requests and make sure that the tests pass for all supported Python versions.

3.4 Tips

To run a subset of tests:

```
$ py.test tests.test_drain
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search