Good Tables Documentation

Release 0.1

Open Knowledge Foundation

February 18, 2016

Contents

Good Tables is a python library and command line tool for validating and transforming tabular data.

Tabular data in the form of CSV or Excel is passed through a pipeline of **validators**. These validators can **check structure**, for example are their blank rows or columns, do rows have the same length as the header etc, and they can also **validate against a schema**, for example does the data have the expected columns, is the data of the right type (are dates actually dates).

Optionally, the data source is **transformed** as it passes through the pipeline.

In return, the client receives a **report** on processing performed and, optionally, the output data.

CHAPTER 1

Get involved

You can contribute to the project with content, code, and ideas!

Start at one of the following channels:

Documentation: An overview of the features that are currently in place.

Issues: See current issues, the backlog, and/or file a new issue.

Code: Get the code here.

CHAPTER 2

Table of contents

2.1 Quickstart

Let's get started.

2.2 Installation

Good Tables runs on Python 2.7, 3.3 and 3.4.

PyPI:

:: pip install goodtables

Git:

:: git clone https://github.com/okfn/goodtables.git

2.3 Tutorials

Some tutorials for using and extending Good Tables.

2.3.1 1. Implementing a custom processor

TODO: This is unfinished.

Implementing a custom validator that can be invoked in a pipeline is easy.

Let's write one that checks that values are in a certain range.

For data, see the file *custom-range.csv* in the *examples* directory.

In our data, we have name, age and city data for a group of people.

We want to ensure that all the people in our data are in the 25-50 age range.

For demonstration, we'll write a pretty specific validator for this.

Of course the implementation could be made more generic for a range scenarios.

Our validator class:

```
class AgeRangeValidator(object):
    column_name = 'age'
    column_type = int
    column_range = (25, 50)
    report = {}

    def run_row(self, index, headers, row):
        valid = True

        return valid

    def run():
        valids = []
        return valid, report

    def generate_report():
        return report
```

As you can see, we hard coded *column_name*, *column_type* and *column_range*.

Also, we are running our validation through the *run_row* method, which is the most common method used for validations.

However, we could easily run the same validation in the *run_column* instead:

```
# stuff
def run_column():
valid = True
    return valid
```

So, let's see it in action. First, we'll run the validator in 'stand alone' via its run method:

```
validator = AgeRangeValidator()
filepath = 'examples/custom-range.csv'
valid, report = validator.run(filepath)
```

And the same, but part of a validation pipeline using the structure validator with our *AgeRangeValidator*:

```
validators = ('structure', 'my_module.AgeRangeValidator')
filepath = 'examples/custom-range.csv'
validation_pipeline = ValidationPipeline(filepath, validators)
valid, report = validation_pipeline.run()
```

2.4 Pipeline

Naturally, the *pipeline.Pipeline* class implements the processing pipeline.

2.4.1 Validator registration

Register by constructor

The *pipeline.Pipeline* constructor takes a *validators* keyword argument, which is a list of validators to run in the pipeline.

Each value in the *validators* list is expected to be a string describing the path to a validator class, for import via *importlib*.

Optionally, for builtin validators, the *validator.name* property can be used as a shorthand convenience.

Example

:: validators = ['structure', 'schema'] # short hand names for builtin validators validators = ['my_module.CustomValidatorOne', 'my_module.CustomValidatorTwo'] # import from string validators = ['structure', 'my_module.CustomValidatorTwo'] # both combined

Register by instance method

Once you have a *pipeline.Pipeline* instance, you can also register validators via the *register_validator* method.

Registering new validators this way will by default append the new validators to any existing pipeline.

You can define the position in the pipeline explicitly using the *position* argument.

Example

:: pipeline = Pipeline(args, kwargs) pipeline.register_validator('structure', structure_options) pipeline.register_validator('spec', spec_options, 0)

2.4.2 Validator options

Pipeline takes an *options* keyword argument to pass options into each validator in the pipeline.

options should be a dict, with each top-level key being the name of the validator.

Example

::

pipeline_options = {

'structure': { # keyword args for the StructureValidator

}, 'schema': {

keyword args for the SchemaValidator

}

}

2.4.3 Instantiating the pipeline

WIP

TODO: This is not complete

2.4.4 Running the pipeline

Run the pipeline with the *run* method.

run in turn calls the supported **validator methods** of each validator.

Once the data table has been run through all validators, *run* returns a tuple of *valid, report*, where:

- *valid* is a boolean, indicating if the data table is valid according to the pipeline validation
- *report* is *tellme.Report* instance, which can be used to generate a report in various formats

2.4.5 Validator arguments

Most validators will have custom keyword arguments for their configuration.

Additionally, all validators are expected to take the following keyword arguments, and exhibit certain behaviour based on their values.

The *base.Validator* signature implements these arguments.

fail_fast

fail_fast is a boolean that defaults to *False*.

If *fail_fast* is *True*, the validator is expected to stop processing as soon as an error occurs.

transform

transform is a boolean that defaults to *True*.

If *transform* is *True*, then the validator is "allowed" to return transformed data.

The caller (e.g., the pipeline class) is responsible for persisting transformed data.

report_limit

report_limit is an int that defaults to *1000*, and refers to the maximum amount of entries that this validator can write to a report.

If this number is reached, the validator should stop processing.

row_limit

row_limit is an int that defaults to *20000*, and refers to the maximum amount of rows that this validator will process.

report_stream

report_stream allows calling code to pass in a writable, seekable text stream to write report entries to.

2.4.6 Validator attributes

Validators are also expected to have the following attributes.

report

A *tellme.Report* instance. See TellMe

Validators are expected to write report entries to the report instance.

pipeline.Pipeline will call *validator.report.generate* for each validator to build the pipeline report.

name

A shorthand name for this validator. *name* should be unique when called in a pipeline.

Validators that inherit from *base.Validator* have a name that defaults to a lower-cased version of the class name.

2.5 Batch

pipeline.Batch Allows the configuration and running of pipelines on multiple sources.

Data sources can be extracted from either a CSV file, or a Data Package.

2.5.1 Arguments

- *source*: Filepath to the lust of data sources to run the batch against.
- *source_type*: 'csv' (CSV file) or 'dp' (Data Package file).
- *data_key*: If *source_type* is 'csv', then this is the name of the header that indicates the data URL.
- *schema_key*: If *source_type* is 'csv', then this is the name of the header that indicates the schema URL.
- *pipeline_options*: The *options* keyword argument for the *pipeline.Pipeline* constructor.
- *post_task*: Any callable that takes the batch instance as its only argument. Runs after the batch processing is complete.
- *pipeline_post_task*: Any callable that takes a pipeline instance as its only argument. Runs on completion of each pipeline.

For an example of the batch processor at work, including use of *post_task* and *pipeline_post_task*, see spd-admin.

2.6 Reports

The results of any run over data, either by a standalone processor or a pipeline, are written to a report.

Each report is an instance of a *tellme.Report*, which is a small library we also developed (See the TellMe library for more information on its API).

Reports can then be generated in a variety of output formats supported by TellMe.

2.6.1 Pipeline reports

In a pipeline, the *pipeline.Pipeline* each processor writes report results to the pipeline's report instance.

After processing of the data is complete, additional calculations are performed for a summary.

Finally, the report is generated to an output format (a Python dict in this case) and returned.

From a top-level view, a pipeline report will have the following structure:

```
{
    'success': True,
    'meta': {'name': 'Pipeline'},
    'summary': {#summary},
    'results': [...]
}
```

All the interesting stuff is happening in the results array and the sumamry object.

See below for a description of each object in the results array, and likewise a description of the summary object.

2.6.2 Standalone processor reports

Standalone processors (for example, the built-in *StructureProcessor*) have a report object almost identical to that of a pipeline report, except they do not have a summary object.

2.6.3 Report result schema

```
{
    'result_type': '# type of this result',
    'result_category': '# category of this result (row/header)',
    'result_level': '# level of this result (info/warning/error)',
    'result_message': '# message of this result',
    'result_context': [# a list of the values of the row that the result was generated from]
    'row_index': '# index of the row',
    'row_name': # 'headers' or valud of id or _id if present, or empty
    'column_index': '# index of the column (can be None)',
    'column_name': '# name of the column (can be '')',
}
```

2.6.4 Report summary schema

```
{
    'message': '# a summary message',
    'total_rows': # int,
    'total_columns': # int,
    'bad_rows': # int,
    'bad_columns': # int,
    'columns': [# list of dicts with position, name, type conformance (%) per column]
}
```

2.7 CLI

Good Tables includes a command line interface, *goodtables*.

2.7.1 Pipeline

Run a Good Tables pipeline.

Example

```
goodtables pipeline *data_source* --schema filepath_or_url --fail_fast --dry_run --row_limit 20000 --
```

2.7.2 StructureProcessor

Run the Good Tables StructureProcessor.

Example

```
goodtables structure *data_source* --fail_fast --row_limit 20000 --report_limit 1000
```

2.7.3 SchemaProcessor

Run the Good Tables SchemaProcessor.

Example

```
goodtables schema *data_source* --schema filepath_or_url --fail_fast --row_limit 20000 --report_limit
```

Design goals

High-level design goals for Good Tables:

- Process tabular data in CSV, Excel and JSON formats
- Provide a suite of small tools that each implement a type of processing to run
- Provide a pipeline API for registering built-in and custom processors
- Components should be easily usable in 3rd party (Python) code

CHAPTER 4

Indices and tables

- genindex
- modindex
- search