Table-Cleaner Documentation

Release 0.1

Andreas Klostermann

August 22, 2015

Contents

CHAPTER 1

Introduction

Table-Cleaner is a validation framework for tabular data in the Pandas ecosystem. It can be used to validate data inside DataFrames and returns the cleaned data and information about errors as DataFrames for further processing.

Such data may come from CSV or Excel files. Using built-in and custom validators Table-Cleaner can be used to ensure data integrity.

DataFrames with information about the errors in such a table can then be used in documentation or web applications, or it can be further analyzed using the usual Pandas and scientific Python tools.

Table-Cleaner is largely inspired by the Django validators, which are used in forms, among other things.

Contents:

1.1 Tutorial

This tutorial will show you how to use the Table-Cleaner validation framework.

First, let's import the necessary modules. My personal style is to abbreviate the scientific python libraries with two letters. This avoids namespace cluttering on the one hand, and is still reasonably short.

```
import numpy as np
import pandas as pd
from IPython import display
import table_cleaner as tc
```

IPython.display provides us with the means to display Python objects in a "rich" way, especially useful for tables.

1.1.1 Introduction

Validating tabular data, especially from CSV or Excel files is a very common task in data science and even generic programming. Many times this data isn't "clean" enough for further processing. Writing custom code to transform or clean up this kind of data quickly gets out of hand.

Table-Cleaner is a framework to generalize this cleaning process.

1.1.2 Basic Example

First, let's create a DataFrame with messy data.

```
initial_df = pd.DataFrame(dict(name=["Alice", "Bob", "Wilhelm Alexander", 1, "Mary", "Andy"],
                               email=["alice@example.com", "bob@example.com", "blub", 4, "mary@example.c
                               "andy k@example .com"],
                               x=[0,3.2,"5","hello", -3,11,],
                               y=[0.2,3.2,1.3,"hello",-3.0,11.0],
                               active=["Y", None, "T", "false", "no", "T"]
                               ))

display.display(initial_df)
```

This dataframe contains several columns. Some of the cells don't look much like the other cells in the same column. For Example we have numbers in the email and name columns and strings in the number columns.

Looking at the dtypes assigned to the dataframe columns reveals a further issue with this mess:

```
initial_df.dtypes
```

```
active    object
email     object
name      object
x         object
y         object
dtype: object
```

All columns are referred to as "object", which means they are saved as individual Python objects, rather than strings, integers or floats. This can make further processing inefficient, but also error prone, because different Python objects may not work with certain dataframe functionality.

Let's define a cleaner:

```
class MyCleaner(tc.Cleaner):
    name = tc.String(min_length=2, max_length=10)
    email = tc.Email()
    x = tc.Int(min_value=0, max_value=10)
    y = tc.Float64(min_value=0, max_value=10)
    active = tc.Bool()
```

Cleaner classes contain fields with validators.

The tc.String validator validates every input to a string. Because most Python objects have some way of being represented as a string, this will more often work than not. Pretty much the only failure reason is if the string is encoded wrongly. Additionally, it can impose restrictions on minimum and maximum string length.

The tc.Int instance tries to turn the input into integer objects. This usually only works with numbers, or strings which look like integers. Here, also, minimum and maximum values can be optionally specified.

The cleaner object can now validate the input dataframe like this:

```
cleaner = MyCleaner(initial_df)
```

Instantiating the cleaner class with an input dataframe creates a cleaner instance with data and verdicts. The validation happens inside the constructor.

```
cleaner.cleaned
```

```
cleaner.cleaned.dtypes
```

```
active     object
email      object
name       object
x           int64
```

```
y         float64
dtype: object
```

The DataFrame only contains completely valid rows, because the default behavior is to delete any rows containing an error. See below on how to use missing values instead.

The datatypes for the "x" column is now int64 instead of object. "y" is now float64. Pandas uses the dtype system specified in numpy, and numpy references strings as "object". The main reason for this is that numeric data is usually stored in a contiguous way, meaning every value has the same "width" of bytes in memory. Strings, not so much. Their size varies. So arrays containing strings have to reference a string object with a pointer. Then the array of pointers is contiguous with a fixed number of bytes per pointer.

The "active" column is validated as a boolean field. There is a dtype called bool, but it only allows True and False. If there are missing values, the column reverts to "object". To force the bool dtype, read the section about booleans below.

So far, we have ensured only valid data is in the output table. But Table Cleaner can do more: The errors themselves can be treated as data:

```
cleaner.verdicts
```

In this case there is only one row per cell, or one per row and column. Except for the last row, where there are two warnings/errors for the Email column. In the current set of built-in validators this arises very rarely. Just keep in mind not to sum the errors up naively and call it the "number of invalid data points".

Let's filter the verdicts by validity:

```
errors = cleaner.verdicts[~cleaner.verdicts.valid]
display.display(errors)
```

As this is an ordinary DataFrame, we can do all the known shenanigans to it, for example:

```
errors.groupby(["column", "reason"])["counter",].count()
```

This functionality is the main reason why Table Cleaner was initially written. In reproducible data science, it is important not only to validate input data, but also be aware of, analyze and present the errors present in the data.

1.1.3 Markup Frames

Let's bring some color into our tables. First, define some CSS styles for the notebook, like so:

```
%%html
<style>
.tc-cell-invalid {
    background-color: #ff8080
}
.tc-highlight {
    color: red;
    font-weight: bold;
    margin: 3px solid black;
    background-color: #b0b0b0;
}

.tc-green {
    background-color: #80ff80
}
.tc-blue {
    background-color: #8080ff;
```

```
}
</style>
```

The MarkupFrame class is subclassed from Pandas' DataFrame class and is used to manipulate and render cell-specific markup. It behaves almost exactly the same as a DataFrame.

Caution: This functionality will soon be completely rewritten to have a simpler and cleaner API.

It can be created from a validation like this:

```
mdf = tc.MarkupFrame.from_validation(initial_df, cleaner.verdicts)
mdf
```

Note that we put in the initial_df table, because the verdicts always relate to the original dataframe, not the output, which has possibly been altered and shortened during the validation process.

Now watch this:

```
mdf.x[1] += "tc-highlight"
mdf.y += "tc-green"
mdf.ix[0, :] += "tc-blue"
mdf
```

1.1.4 Booleans

The trouble with Booleans

Boolean values are either True or False. In Pandas, and data science in general, things are a bit more tricky. There is a third state, which Pandas would refer to as a missing value. Numpy's Bool dtype does not support missing values though.

```
np.bool(None)
```

```
False
```

What's happening there is that many Python objects have a way of being interpreted as either True or False. An empty list, empty strings, and None, are all considered false, for example.

Now, let's try that in Pandas:

```
bools = pd.Series([True, False, None, np.NaN])
bools
```

```
0     True
1    False
2     None
3      NaN
dtype: object
```

The dtype is not "bool". Instead Pandas refers to the individual Python object, and thus dtype must be "object". We can make it bool, though:

```
bools.astype(bool)
```

```
0     True
1    False
2    False
3     True
dtype: bool
```

Notice how np.NaN, which is normally interpreted as a missing value, has been converted to True?

If you try to index something with this sequence, this is what happens:

```
original = pd.Series(range(3))
original[bools]
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-17-3d0782e602c7> in <module>()
      1 original = pd.Series(range(3))
----> 2 original[bools]


/usr/local/lib/python3.4/site-packages/pandas/core/series.py in __getitem__(self, key)
    544             key = list(key)
    545
--> 546         if _is_bool_indexer(key):
    547             key = _check_bool_indexer(self.index, key)
    548


/usr/local/lib/python3.4/site-packages/pandas/core/common.py in _is_bool_indexer(key)
   2058             if not lib.is_bool_array(key):
   2059                 if isnull(key).any():
-> 2060                     raise ValueError('cannot index with vector containing '
   2061                                      'NA / NaN values')
   2062                 return False


ValueError: cannot index with vector containing NA / NaN values
```

Let's take a look at how to bring some sanity into this issue with Table Cleaner. First, define a messy DataFrame, with columns that are identical:

```
bools = [True, False, None, np.NaN]

bool_df = pd.DataFrame(dict(a=bools, b=bools, c=bools, d=bools))
bool_df
```

Now create a cleaner which validates each column differently:

```
class BoolCleaner(tc.Cleaner):
    a = tc.Bool()
    b = tc.Bool(true_values=[True], false_values=[False], allow_nan=False)
    c = tc.Bool(true_values=[True], false_values=[False, None], allow_nan=False)
    d = tc.Bool(true_values=[True], false_values=[False, np.nan], nan_values=[None], allow_nan=False)

bool_cleaner = BoolCleaner(bool_df)
tc.MarkupFrame.from_validation(bool_df, bool_cleaner.verdicts)
```

Note that I used "delete=False" to keep rows with invalid data, while still converting available values. Then this dataframe has the same shape as MarkupFrame.from_validation expects. "allow_nan" defaults to True and controls whether or not missing values are considered an error.

```
bool_cleaner.verdicts[~bool_cleaner.verdicts.valid]
```

Tables coming from external sources, especially spreadsheet data is notorious for having all sorts of ways to indicate

booleans or missing values. The Bool validator takes three arguments to handle these cases: true_values, false_values and nan_values.

```
messy_bools_column =["T","t","on","yes", "No", "F"]
messy_bools = pd.DataFrame(dict(a=messy_bools_column, b=messy_bools_column))
```

```
class BoolCleaner2(tc.Cleaner):
    a=tc.Bool()
    b=tc.Bool(true_values=["T"], false_values=["F"], allow_nan=False)

bool_cleaner2 = BoolCleaner2(messy_bools)
tc.MarkupFrame.from_validation(messy_bools, bool_cleaner2.verdicts)
```

1.2 Email validation

Email validation is a subject onto its own. Some frameworks offer validation by simple regular expressions, which sometimes isn't enough. Other libraries or programs go so far as to ask the corresponding mail server if it knows a particular address.

In almost all generic usecases, you expect email names to adhere to a very specific form, meaning a username "at" a particular globally identifiable domain name. It is assumed that every computer in the world can resolve this domain name to the same physical server. Email standards and most eail servers however, don't require "fully qualified domain names" or even globally resolvable domains. "root@localhost" is a perfectly valid email address, but completely useless in most circumstances where you want to collect or use email addresses.

TableCleaner's Email validator class is based on Django's validation method.

```
messy_emails =["alice@example.com", "bob@bob.com", "chris", "delta@localhost", "ernest@hemmingway@er
email_df = pd.DataFrame(dict(email=messy_emails))
class EmailCleaner(tc.Cleaner):
    email = tc.Email()

email_cleaner = EmailCleaner(email_df)
tc.MarkupFrame.from_validation(email_df, email_cleaner.verdicts)
```

1.3 Development

Currently, Table Cleaner is in a very early state of development. Python 3 support and documentation is marginal. Built-in validators are still very few.

For easier development a few Dockerfiles have been prepared and stored in the docker/ subdirectory. Build these first. Then you can use "docker_test_py2.sh" and "docker_test_py3.sh" in the top level directory to quickly and reproducibly run the test suite.

This documentation can be generated with the "docker_make_html_docs.sh" script.

The directories with the Dockefiles each contain a build.sh script. These scripts assign certain names to the containers, and these names are expected by the other docker_* scripts to point to the respecting containers. In the future this simple system may be refactored into a hierarchical container structure, but for now, this is works quite well.

Feel free to send pull requests or discuss missing features in the Github issue tracker for this project!

1.4 Future Plans

- Implement colored html table output to indicate errors
- Integrate Flanker email validation
- Add IP Address validators (as found in Django.core.validators)
- Elaborate Docker containers
- More Documentation

CHAPTER 2

Indices and tables

- genindex
- modindex
- search