Tableview Documentation

Release 0.1.1

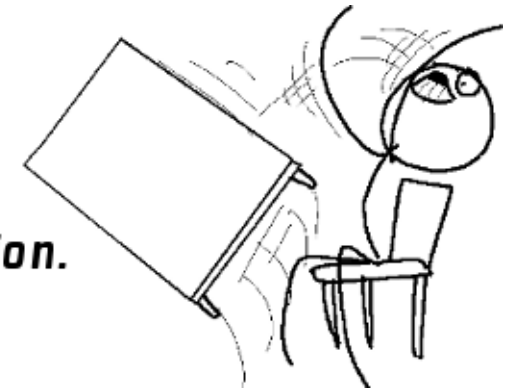
Ryan Sturmer

February 10, 2015

Contents

Tableview

Flip your data... *Into submission.*



Tableview is a pythonic library for manipulating tabular data. It is spiritually similar to Kenneth Reitz's Tablib, as well as the view functionality in numpy.

Visit the *Quickstart* guide if you'd like to get started.

CHAPTER 1

Quickstart

This page gives a good introduction to the basics of Tableview. This assumes that you have tableview installed and up to date.

1.1 Create a TableView

The heart of the Tableview library is the `TableView` object. It represents a shallow view of any tabular data. To create one, pass some tabular data to its constructor:

```
d = [['Name', 'Age', 'Drink', 'Color'], \
          ['Ryan', 30, 'Tea', 'Purple'], \
          ['Michael', 31, 'Coffee', 'Blue'], \
          ['Keith', 40, 'Tea', 'Maroon'], \
          ['Brent', 26, 'Coffee', 'Blue'], \
          ['Craig', '??', 'Bourbon', 'Turquoise']]

table = tableview.TableView(d)
```

Example Context

From here on, if you see `table`, assume that it is a fresh `TableView` object that represents the data above.

1.2 Pretty Printing

The `pretty()` method returns a pretty string for a `TableView`:

```
>>> print table.pretty()
Name    Age Drink     Color
Ryan    30  Tea       Purple
Michael 31  Coffee    Blue
Keith   40  Tea       Maroon
Brent   26  Coffee    Blue
Craig   ??  Bourbon    Turquoise
```

1.3 Rows and Columns

Views are indexed by by row, just like a 2D list

```
>>> print table[0].pretty()
Name Age Drink Color
```

But to make operations symmetrical between rows and columns, the `rows` and `cols` properties are provided. They work just like you think:

```
>>> print table.rows[0].pretty()
Name Age Drink Color
>>> print table.cols[0].pretty()
Name
Ryan
Michael
Keith
Brent
Craig
```

Row Notation

From here on, the `table.rows[]` notation will be used, but any such operation can be done using the `table[]` notation. They are equivalent.

1.4 Removing Data from a View

You can delete a row or column from a view using the standard `del` operator:

```
del table.rows[0]
```

The same operation works on columns:

```
del table.cols[1]
```

Looking at the data now, we see there are no column headers, and the Age column has been removed:

```
>>> print table.pretty()
Ryan    Tea       Purple
Michael Coffee    Blue
Keith   Tea       Maroon
Brent   Coffee    Blue
Craig   Bourbon   Turquoise
```

It is important to remember that we haven't modified the source data. Row and column operations only affect the view itself. We can create a new view from the data, and take a look at it:

```
>>> table2 = tableview.TableView(d)
>>> print table2.pretty()
Name    Age Drink     Color
Ryan    30  Tea       Purple
Michael 31  Coffee    Blue
Keith   40  Tea       Maroon
Brent   26  Coffee    Blue
Craig   ??  Bourbon   Turquoise
```

1.5 Selecting Rows and Columns

Frequently, we want to select data based on some criteria, rather than by index. `select_rows` and `select_cols` do just that. Let's say we are only interested in the coffee drinkers:

```
selection = table.select_rows(lambda row : row[2] == 'Coffee')
```

The select methods take a single callable that takes a single argument (a row or column). They return True if the row or column is to be returned in the selection. Let's inspect our selected data:

```
>>> print selection.pretty()
Michael 31  Coffee    Blue
Brent   26  Coffee    Blue
```

Like all operations, the same can be done with columns. Using a fresh `table`:

```
>>> print table.select_cols(lambda col : col[0] in ('Name', 'Drink')).pretty()
Name    Drink
Ryan    Tea
Michael Coffee
Keith   Tea
Brent   Coffee
Craig   Bourbon
```

Selection operations return *new* tableview objects. Our original `TableView` is untouched by calls to `select_rows` and `select_cols`

1.6 Stripping Rows and Columns

Stripping works just like selecting, except that the matching rows/columns are removed from the output, rather than included. Back to our coffee drinkers:

```
>>> print table.strip_rows(lambda row : row[2] == 'Coffee').pretty()
Name    Age Drink     Color
Ryan    30  Tea       Purple
Keith   40  Tea       Maroon
Craig   ??  Bourbon   Turquoise
```

Tea is better for you, anyway.

1.7 Splitting Tables

`TableViews` can be split into smaller views by rows or by columns. This is handy when you have a table that is composed of multiple subtables, separated by empty rows or a divider:

```
separated_data = [[1,2,3], \
                  [], \
                  [4,5,6], \
                  [7,8,9], \
                  ['','','']
                  [10,11,12], \
                  [13,14,15], \
                  [16,17,18]]
 table = tableview.TableView(separated_data)
```

```
one,two,three = table.split_rows()

>>> print one.pretty()
1 2 3

>>> print two.pretty()
4 5 6
7 8 9

>>> print three.pretty()
10 11 12
13 14 15
16 17 18
```

The default behavior of `split_rows` and `split_cols` is to split on empty rows/columns. The example above splits the table view into 3 parts, using the empty rows as delimiters. An 'empty' row is one whose elements all evaluate to `False` This is why the row of empty strings worked as a delimiter in the example above. This also means that a row of zeroes is a valid delimiter, so care must be taken when working with tables of numeric data.

The default delimiter behavior can be modified with an optional criteria argument to `split_rows` and `split_cols` It is a function that accepts a row or column as an argument, and should return `True` if that row or column is a delimiter:

```
mixed_data = [['Letters:'], \
              ['a','b','c'], \
              ['d','e','f'], \
              ['Numbers:'], \
              [1,2,3], \
              [4,5,6], \
              [7,8,9]], \
              ['Symbols:'], \
              ['*','%','!'], \
              ['$','#','@']]
table = tableview.TableView(mixed_data)
letters,numbers,symbols = table.split_rows(lambda row : row[0].endswith(':'))

>>> print letters.pretty()
a b c
d e f

>>> print numbers.pretty()
1 2 3
4 5 6

>>> print symbols.pretty()
* % !
$ # @
```

The example above performs a similar split, but instead of looking for empty rows, it uses rows whose first cells end with a colon (:), assuming these rows to be section headings.

1.8 Manipulating Data

While row and column operations don't affect a `TableView's` source data, assignments to its members do. Once you have the view configured to show the data you want, it can be modified. This is the real power of Tableview:

```
selection = table.select_rows(lambda row : row[2] == 'Coffee')
for row in selection:
   row[-1] = 'Yellow'
```

We've singled out all the coffee drinkers, and changed their favorite color to yellow. Remember that when we select rows, we're getting a new view of the table. Our original `table` object is still a view of all the source data. Let's inspect:

```
>>> print table.pretty()
Name    Age Drink     Color
Ryan    30  Tea       Purple
Michael 31  Coffee    Yellow
Keith   40  Tea       Maroon
Brent   26  Coffee    Yellow
Craig   ??  Bourbon   Turquoise
```

1.9 Raw Data

A copy of the contents of a view can be retrieved using its `raw` property. This returns *a copy* of the view's data as a list of lists:

```
>>> table.raw
[['Ryan', 30, 'Tea', 'Purple'], ['Michael', 31, 'Coffee', 'Blue'], ['Keith', 40, 'Diet Coke', 'Maroon
```

1.10 Loading Data from Disk

If you are working with CSV or text files, data can be easily loaded from disk:

```
table = tableview.load('data.csv')
```

In this case, the `TableView` object wasn't invoked directly. Like any `TableView`, we can access its source data using the `data` property:

```
>>> print table.data
[['Name', 'Age', 'Drink', 'Color'], ['Ryan', 30, 'Tea', 'Purple'], ['Michael', 31, 'Coffee', 'Blue'],
```

The `tableview.load` function uses the file extension to determine how to parse the file. A .csv extension indicates comma-separated-values, and any other extension is assumed to be tab-separated text.