tableD Documentation

Release 0.1.0

Tommy Ip

Apr 03, 2017

Contents

v0.1.0 (Github)

tableD pretty prints your data in a tabular format. With its clean and simple API, you can generate fancy tables in no time! And did I mention it is extremely flexible and configurable?

Quick start:

```
>>> headings = ['x', 'f(x) = x^2', 'f(x) = x^x']
>>> data = [[x, x**2, x**x] for x in range(1, 11)]

# Generate table!
>>> tabled.new(headings, data, style='terminal').show()
âŢŤâŢŘâŢŘâŢŘâŢŘâŢęâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢęâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢ
âŢŚ x  âŢŚ f(x) = x^2 âŢŚ f(x) = x^x  âŢŚ
âŢăâŢŘâŢŘâŢŘâŢŘâŢňâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢňâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢ
âŢŚ 1  âŢŚ 1          âŢŚ 1           âŢŚ
âŢŚ 2  âŢŚ 4          âŢŚ 4           âŢŚ
âŢŚ 3  âŢŚ 9          âŢŚ 27          âŢŚ
âŢŚ 4  âŢŚ 16         âŢŚ 256         âŢŚ
âŢŚ 5  âŢŚ 25         âŢŚ 3125        âŢŚ
âŢŚ 6  âŢŚ 36         âŢŚ 46656       âŢŚ
âŢŚ 7  âŢŚ 49         âŢŚ 823543      âŢŚ
âŢŚ 8  âŢŚ 64         âŢŚ 16777216    âŢŚ
âŢŚ 9  âŢŚ 81         âŢŚ 387420489   âŢŚ
âŢŚ 10 âŢŚ 100        âŢŚ 10000000000 âŢŚ
âŢŽâŢŘâŢŘâŢŘâŢŘâŢl’âŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢl’âŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢŘâŢ
```

CHAPTER 1

Table Of Contents

Installation

You can only install the package from source code right now, but it will be available on PyPI as soon as the project reaches v1.0.0!

From source code

First clone the repository from Github:

```
$ git clone https://github.com/tommyip/tabled
```

Then you can install it into your site-packages easily:

```
$ cd tabled
$ python setup.py install
```

That's it! Now try to create your first pretty printed table.

User guide

If you haven't installed tableD already, please follow the instruction here.

Creating a new table

First, import the tabled package:

```
import tabled
```

The *TableD* object is the main interface for visualizing your data, you can create a new instance using the new constructor function:

```
>>> tabled.new(['Heading 1', 'Heading 2'],
...            [[1, 2], [3, 4]]).show()
+-----------+-----------+
| Heading 1 | Heading 2 |
+-----------+-----------+
| 1         | 2         |
| 3         | 4         |
+-----------+-----------+
```

Lets break this down. The new function creates and returns a TableD object, which accepts 4 optional arguments.

Headings

The heading argument should be a list of elements, the type of the cell would be converted to string automatically. Examples:

```
['Heading 1', 'Heading 2', some_variable, 10, True]

[x for x in range(10)]  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Data

The second argument, data, is a nested list of lists containing the table body. Same as the headings, each cell element could be in any Python types. Examples:

```
[[1, 2], [3, 4], ["Cell 5", 6], [True, False]]

[[x, x+1] for x in range(3)]  # [[0, 1], [1, 2], [2, 3]]
```

Style

The style of the table is configured through the style argument, which is *default* for default. There are only two styles available for now, but you are welcome to help create more, see *Contributors Guide* for more information.

```
headings = ['x', 'y1', 'y2']
data = [[x, x*x, x**3] for x in range(3)]
```

Default:

```
>>> tabled.new(headings, data).show()
+---+----+----+
| x | y1 | y2 |
+---+----+----+
| 0 | 0  | 0  |
| 1 | 1  | 1  |
| 2 | 4  | 8  |
+---+----+----+
```

Terminal:

```
>>> tabled.new(headings, data, style='terminal').show()
âŢŤâŢŘâŢŘâŢŘâŢęâŢŘâŢŘâŢŘâŢęâŢŘâŢŘâŢŘâŢŮ
âŢŚ x âŢŚ y1 âŢŚ y2 âŢŚ
âŢăâŢŘâŢŘâŢŘâŢňâŢŘâŢŘâŢŘâŢňâŢŘâŢŘâŢŘâŢč
âŢŚ 0 âŢŚ 0  âŢŚ 0  âŢŚ
âŢŚ 1 âŢŚ 1  âŢŚ 1  âŢŚ
âŢŚ 2 âŢŚ 4  âŢŚ 8  âŢŚ
âŢŽâŢŘâŢŘâŢŘâŢl’âŢŘâŢŘâŢŘâŢl’âŢŘâŢŘâŢŘâŢİ
```

Align

By default, the style template already provides an alignment setting, but it is possible to customize it using this argument. The available alignments are **left**, **center** and **right**:

```
>>> tabled.new(['Heading 1', 'Heading 2'], [[1, 2], [3, 4]],
...            align='right').show()
+-----------+-----------+
| Heading 1 | Heading 2 |
+-----------+-----------+
|         1 |         2 |
|         3 |         4 |
+-----------+-----------+
```

Device

The device argument controls where the output is shown. The default is stdout, which is your terminal or python shell.

More device options are coming soon...

Displaying the table

The `.show()` method displays your table to standard output by default. It supports caching, so it would rerender your data only if they were modified.

Modification after initialization

The arguments to `tabled.new()` are optional as mentioned above, they could be added or changed after the initialization of a new tabled instance. Here are the available setter methods, which are fairly self explanatory:

```
>>> t = tabled.new()

# Set the table headings
>>> t.set_headings(['Language', 'Typing', 'Runtime', 'Type'])

# Add a new row to the bottom of the table
>>> t.add_row(['Python', 'Dynamic', 'CPython', 'OOP'])

# Add multiple rows to the table (must be nested list)
>>> t.add_rows([
    ['Java', 'Static', 'JVM', 'OOP'],
    ['Elixir', 'Dynamic', 'BEAM', 'Functional']
])
```

Note: The number of columns of your table is determined by the headings. If any of your rows is shorter than the headings, blank cells would be appended to the end of the row.

If you now display the table, you will get:

```
>>> t.show()
+----------+---------+---------+------------+
| Language | Typing  | Runtime | Type       |
+----------+---------+---------+------------+
| Python   | Dynamic | CPython | OOP        |
| Java     | Static  | JVM     | OOP        |
```

```
| Elixir   | Dynamic | BEAM    | Functional |
+----------+---------+---------+------------+
```

API documentation

Public interface

All of TableD's functionality can be accessed through these 5 methods.

tabled.**new**(*headings: typing.Union[typing.List[typing.Any], NoneType] = None, data: typing.Union[typing.List[typing.List[typing.Any]], NoneType] = None, style: str = 'default', align: str = None, device: str = 'stdout', dataframe=None*) → tabled.tabled.TableD

Creates a new TableD object. This should be used instead of calling TableD's __init__() directly.

Parameters

- **headings** – A list of column headings.
- **data** – Nested list of lists of cell contents.
- **style** – Style of pretty printer.
- **align** – Align cell content to either left, center or right. Default to setting specified in style.
- **device** – Where to output pretty printed table.
- **dataframe** – existing pandas dataframe object.

Returns A TableD object.

Example

```
>>> new()
<tabled.tabled.TableD object at 0x...>
```

TableD.**add_row**(*row: typing.List[typing.Any]*) → None

Append a single row to table body.

Parameters **row** – A row of data to be appended to the table.

Example

```
>>> table = TableD()
>>> table.add_row(['x1', 'x2', 'x3'])
>>> table.data
[['x1', 'x2', 'x3']]
```

TableD.**add_rows**(*rows: typing.List[typing.List[typing.Any]]*) → None

Append multiple rows to table body.

Parameters **rows** – Multiple rows of data to be appended to table.

Example

```
>>> table = TableD()
>>> table.add_rows([['x1', 'x2', 'x3'],
...                 ['y1', 'y2', 'y3']])
>>> table.data
[['x1', 'x2', 'x3'], ['y1', 'y2', 'y3']]
```

TableD.**set_headings**(*headings: typing.List[typing.Any]*) → None
Overwrite or set the table headings.

Parameters **headings** – A list of column headings.

Example

```
>>> table = TableD()
>>> table.set_headings(['id', 2, 3])
>>> table.headings
['id', '2', '3']
```

TableD.**show**() → None
Generate, cache and display table to standard output. Use cached version if available.

Internals

tabled.tabled

synopsis Pretty print data in tabular format.

copyright

3. 2017, Tommy Ip.

license MIT

class tabled.tabled.**TableD**(*headings: typing.Union[typing.List[typing.Any], NoneType] = None, data: typing.Union[typing.List[typing.List[typing.Any]], NoneType] = None, style: str = 'default', align: str = None, device: str = 'stdout'*) → None
Public interface for the central table abstraction.

headings
A list of column headings.

data
Nested list of lists of cell contents.

style
Style of pretty printer.

align
Align cell content to either left, center or right. Default to setting specified in style.

device
Where to output pretty printed table.

_columns
The number of columns the table have.

_output
Cached table string.

_cache_valid
Validity of the cached table string.

Example

```
>>> table = TableD(
...     ['Repository', 'Author', 'Type'],
...     [['tableD', 'Tommy Ip', 'Python library'],
...      ['VueJS', 'Evan You', 'Frontend JS framework'],
...      ['flask', 'Armin Ronacher', 'Web framework']]
... )
>>> table.show()
+------------+----------------+-----------------------+
| Repository | Author         | Type                  |
+------------+----------------+-----------------------+
| tableD     | Tommy Ip       | Python library        |
| VueJS      | Evan You       | Frontend JS framework |
| flask      | Armin Ronacher | Web framework         |
+------------+----------------+-----------------------+
```

__init__(*headings: typing.Union[typing.List[typing.Any], NoneType] = None, data: typing.Union[typing.List[typing.List[typing.Any]], NoneType] = None, style: str = 'default', align: str = None, device: str = 'stdout'*) → None

Initialize data storage engine for TableD. You should use tabled.new() to construct a TableD object.

add_row(*row: typing.List[typing.Any]*) → None

Append a single row to table body.

Parameters **row** – A row of data to be appended to the table.

Example

```
>>> table = TableD()
>>> table.add_row(['x1', 'x2', 'x3'])
>>> table.data
[['x1', 'x2', 'x3']]
```

add_rows(*rows: typing.List[typing.List[typing.Any]]*) → None

Append multiple rows to table body.

Parameters **rows** – Multiple rows of data to be appended to table.

Example

```
>>> table = TableD()
>>> table.add_rows([['x1', 'x2', 'x3'],
...                 ['y1', 'y2', 'y3']])
>>> table.data
[['x1', 'x2', 'x3'], ['y1', 'y2', 'y3']]
```

set_headings(*headings: typing.List[typing.Any]*) → None

Overwrite or set the table headings.

Parameters **headings** – A list of column headings.

Example

```
>>> table = TableD()
>>> table.set_headings(['id', 2, 3])
>>> table.headings
['id', '2', '3']
```

show() → None
Generate, cache and display table to standard output. Use cached version if available.

__weakref__
list of weak references to the object (if defined)

tabled.pretty_print

synopsis Pretty printing engine for tableD.

copyright

3. 2017, Tommy Ip.

license MIT

tabled.pretty_print.**left_pad**(*string: str*, *width: int*) → str
Insert spaces to the left of a string to fit into a container.

Parameters

- **string** – A text value to be padded with spaces.
- **width** – The width of the string container.

Returns A string aligned right in a container.

Example

```
>>> left_pad('tableD', 10)
'    tableD'
```

tabled.pretty_print.**right_pad**(*string: str*, *width: int*) → str
Insert spaces to the right of a string to fit into a container.

Parameters

- **string** – A text value to be padded with spaces.
- **width** – The width of the string container.

Returns A string aligned left in a container.

Example

```
>>> right_pad('tableD', 10)
'tableD    '
```

tabled.pretty_print.**left_right_pad**(*string: str*, *width: int*) → str
Insert spaces to both sides of a string to fit into a container.

Note: The right side of the string would be allocated more spaces if the amount of blank spaces cannot be divided equally by 2.

Parameters

- **string** – A text value to be padded with spaces.
- **width** – The width of the string container.

Returns A string aligned center in a container.

Example

```
>>> left_right_pad('tableD', 11)
'  tableD   '
```

tabled.pretty_print.**pad**(*string: str*, *width: int*, *align: str*, *margin: int = 1*) → str

Pad and align a string in a container.

Parameters

- **string** – Input text to be padded and aligned.
- **width** – The width of the container.
- **align** – Left, center or right alignment.
- **margin** – Margin width between the string and the side wall.

Returns A string padded and aligned in a container.

Example

```
>>> pad('library', 13, 'left')
' library     '
```

tabled.pretty_print.**render_row**(*row: typing.List[str]*, *widths: typing.List[int]*, *delimiters: typing.Dict[str, str]*, *align: str = 'left'*, *margin: int = 1*) → str

Render a table row.

Parameters

- **row** – A row of string, each is a cell of their columns.
- **widths** – A list of column widths.
- **delimiter** – A dictionary of column dividers.
- **align** – Left, center or right alignment of each cell.
- **margin** – Margin width between the string and the side wall.

Returns A string containing a print ready row.

Example

```
>>> render_row(['Some cell content', 'word', '1'], [22, 6, 7],
...                {'left': '|', 'right': '|', 'connector': '|'})
'| Some cell content    | word | 1      |'
```

tabled.pretty_print.**render_table**(*headings: typing.List[str]*, *table: typing.List[typing.List[str]]*, *style: str = 'default'*, *align: typing.Union[str, NoneType] = None*) → str

This is where the magic happens!

Parameters

- **headings** – A list of text containing the headings.
- **table** – Cells data in a nested list of lists structure.
- **style** – Style of formatting in the table.
- **align** – Override settings in style if specified.

Returns A string with formatting ready for output.

Example

```
>>> table = [['1', '1'],
...          ['2', '4'],
...          ['3', '9']]
>>> print(render_table(['x', 'f(x) = x^2'], table))
+---+------------+
| x | f(x) = x^2 |
+---+------------+
| 1 | 1          |
| 2 | 4          |
| 3 | 9          |
+---+------------+
```

tabled.utils

synopsis Utility functions.

copyright

3. 2017, Tommy Ip.

license MIT

tabled.utils.**max_width**(*column: typing.List[str]*) → int
Finds the longest string in a column and returns its length.

Parameters **column** – A list of strings represented as a column in a table.

Returns The length of the longest string.

Example

```
>>> max_width(['example text', 'Some long lines.', 'short'])
16
```

tabled.utils.**rotate_table**(*table: typing.List[typing.List[str]]*) → typing.List[typing.List[str]]
Transform rows to columns and columns to rows.

Parameters **table** – Nested list of lists that is represented as a table structure.

Returns A rotated table with columns as rows.

Note: This operation is non directional, so applying this function twice would return the orginal input.

Example

```
>>> rotate_table([['x1', 'x2', 'x3'],
...               ['y1', 'y2', 'y3'],
...               ['z1', 'z2', 'z3']])
...
[['x1', 'y1', 'z1'],
 ['x2', 'y2', 'z2'],
 ['x3', 'y3', 'z3']]
```

tabled.utils.**columns_width**(*table: typing.List[typing.List[str]]*) → typing.List[int]
Finds the width for each column in a table.

Parameters **table** – Nested list of lists that is represented as a table structure.

Returns A list of integers showing the width for each columns. The width is the determined by the longest text in a column.

Example

```
>>> columns_width([['Example', 'This is very long'],
...                ['Short', 'word'],
...                ['Another long example', 'var']])
[20, 17]
```

tabled.utils.**str_list**(*raw_list: typing.List[typing.Any]*) → typing.List[str]
Cast all elements in a list to Text type.

tabled.utils.**str_nested_list**(*nested_raw_list: typing.List[typing.List[typing.Any]]*) → typing.List[typing.List[str]]
Cast all elements in a nested list to Text type.

tabled.utils.**normalize_list**(*row: typing.List[str], size: int*) → typing.List[str]
Make a row the same length as other rows by filling in blank spaces.

Parameters

- **row** – A list of string to be normalized
- **size** – The length of the final list

Returns A list with of length *size* with blank element filled with "". Trim any extra elements if the list is longer than *size*.

tabled.style_templates

synopsis Contains table styles.

copyright

3. 2017, Tommy Ip.

license MIT

tabled.style_templates.**get_style**(*style: str = 'default'*) → typing.Dict[str, typing.Dict[str, str]]
Construct and return a table style.

Parameters **style** – Style name.

Returns A dictionary of style separated by categories.

Contributors guide

Contributions are welcome! Before you submit a pull requests, please read the following guide to make sure your changes are ready to be merged into the code base.

Version control

Quick start guide

We use **Git** as our version control system, you should have the following in mind when working on TableD:

1. We use a **rebase** workflow in order have to clean Git history. To update your local branch to the latest source tree, execute the following in your terminal:

```
// Set tommyip/tabled remote (FIRST TIME ONLY)
$ git remote add upstream https://github.com/tommyip/tabled

$ git pull --rebase upstream master
```

2. To have your pull requests merged as quickly as possible, please resolve all merge conflict and make sure your changes passed the test suite before submitting them.

Using Git & Github in tableD

Commits

Commits should have a short title clearly stating the purpose of the changes. You should also prepend the title with one of the following section headings to make future maintance easier:

- `code`: New feature
- `refactor`: Code refactoring
- `docs`: Documentation
- `deps`: Dependency
- `setup`: Project setup

Each commit should have a single purpose. Commits with modification to multiple areas of the codebase are not encouraged and will not be merged. If in doubt, just have a look at our Git history.

Testing

You can invoke the test suite with `make test` in the root directory. The following would be executed in the process:

1. pytest - Unit testing framework
2. mypy - Static type checking
3. flake8 - Source code linter

A coverage report would also be produced by pytest. We have a 100% test coverage and we would like to maintain that; when you make changes to the source code, please check if modifying or adding new tests are necessary.

Python Module Index

t

Index