CLI Helpers Documentation

Release 2.2.1

dbcli

Jan 18, 2022

Contents

CLI Helpers is a Python package that makes it easy to perform common tasks when building command-line apps. It's a helper library for command-line interfaces.

Libraries like Click and Python Prompt Toolkit are amazing tools that help you create quality apps. CLI Helpers complements these libraries by wrapping up common tasks in simple interfaces.

CLI Helpers is not focused on your app's design pattern or framework – you can use it on its own or in combination with other libraries. It's lightweight and easy to extend.

What's included in CLI Helpers?

- Prettyprinting of tabular data with custom pre-processing
- Config file reading/writing

CHAPTER 1

Installation

You can get the library directly from PyPI:

```
$ pip install cli_helpers
```

CHAPTER 2

User Guide

2.1 Quickstart

2.1.1 Displaying Tabular Data

The Basics

CLI Helpers provides a simple way to display your tabular data (columns/rows) in a visually-appealing manner:

```
>>> from cli_helpers import tabular_output

>>> data = [[1, 'Asgard', True], [2, 'Camelot', False], [3, 'El Dorado', True]]
>>> headers = ['id', 'city', 'visited']

>>> print(tabular_output.format_output(data, headers, format_name='simple'))

  id  city       visited
----  ---------  ---------
   1  Asgard     True
   2  Camelot    False
   3  El Dorado  True
```

Let's take a look at what we did there.

1. We imported the *tabular_output* module. This module gives us access to the *format_output()* function.
2. Next we generate some data. Plus, we need a list of headers to give our data some context.
3. We format the output using the display format `simple`. That's a nice looking table!

Display Formats

To display your data, *tabular_output* uses tabulate, terminaltables, `csv`, and its own vertical table layout.

The best way to see the various display formats is to use the *TabularOutputFormatter* class. This is what the *format_output()* function in our first example uses behind the scenes.

Let's get a list of all the supported format names:

```
>>> from cli_helpers.tabular_output import TabularOutputFormatter
>>> formatter = TabularOutputFormatter()
>>> formatter.supported_formats
('vertical', 'csv', 'tsv', 'mediawiki', 'html', 'latex', 'latex_booktabs', 'textile',
↪'moinmoin', 'jira', 'plain', 'minimal', 'simple', 'grid', 'fancy_grid', 'pipe',
↪'orgtbl', 'psql', 'psql_unicode', 'rst', 'ascii', 'double', 'github')
```

You can format your data in any of those supported formats. Let's take the same data from our first example and put it in the `fancy_grid` format:

```
>>> data = [[1, 'Asgard', True], [2, 'Camelot', False], [3, 'El Dorado', True]]
>>> headers = ['id', 'city', 'visited']
>>> print(formatter.format_output(data, headers, format_name='fancy_grid'))
╒════╤═══════════╤═══════════╕
│ id │ city      │ visited   │
╞════╪═══════════╪═══════════╡
│  1 │ Asgard    │ True      │
├────┼───────────┼───────────┤
│  2 │ Camelot   │ False     │
├────┼───────────┼───────────┤
│  3 │ El Dorado │ True      │
╘════╧═══════════╧═══════════╛
```

That was easy! How about CLI Helper's vertical table layout?

```
>>> print(formatter.format_output(data, headers, format_name='vertical'))
***************************[ 1. row ]***************************
id      | 1
city    | Asgard
visited | True
***************************[ 2. row ]***************************
id      | 2
city    | Camelot
visited | False
***************************[ 3. row ]***************************
id      | 3
city    | El Dorado
visited | True
```

Default Format

When you create a *TabularOutputFormatter* object, you can specify a default formatter so you don't have to pass the format name each time you want to format your data:

```
>>> formatter = TabularOutputFormatter(format_name='plain')
>>> print(formatter.format_output(data, headers))
  id  city       visited
   1  Asgard     True
   2  Camelot    False
   3  El Dorado  True
```

Tip: You can get or set the default format whenever you'd like through *TabularOutputFormatter.format_name*.

Passing Options to the Formatters

Many of the formatters have settings that can be tweaked by passing an optional argument when you format your data. For example, if we wanted to enable or disable number parsing on any of tabulate's formats, we could:

```
>>> data = [[1, 1.5], [2, 19.605], [3, 100.0]]
>>> headers = ['id', 'rating']
>>> print(format_output(data, headers, format_name='simple', disable_numparse=True))
id    rating
----  --------
1     1.5
2     19.605
3     100.0
>>> print(format_output(data, headers, format_name='simple', disable_numparse=False))
  id    rating
----  --------
   1     1.5
   2    19.605
   3   100
```

Lists and tuples and bytearrays. Oh my!

tabular_output supports any iterable, not just a `list` or `tuple`. You can use a `range`, `enumerate()`, a `str`, or even a `bytearray`! Here is a far-fetched example to prove the point:

```
>>> step = 3
>>> data = [range(n, n + step) for n in range(0, 9, step)]
>>> headers = 'abc'
>>> print(format_output(data, headers, format_name='simple'))
  a    b    c
---  ---  ---
  0    1    2
  3    4    5
  6    7    8
```

Real life examples include a PyMySQL `Cursor` with database results or NumPy `ndarray` with data points.

2.2 How to Contribute

CLI Helpers would love your help! We appreciate your time and always give credit.

2.2.1 Development Setup

Ready to contribute? Here's how to set up CLI Helpers for local development.

1. Fork the repository on GitHub.
2. Clone your fork locally:

```
$ git clone <url-for-your-fork>
```

3. Add the official repository (`upstream`) as a remote repository:

```
$ git remote add upstream git@github.com:dbcli/cli_helpers.git
```

4. Set up a virtual environment for development:

```
$ cd cli_helpers
$ pip install virtualenv
$ virtualenv cli_helpers_dev
```

We've just created a virtual environment called `cli_helpers_dev` that we'll use to install all the dependencies and tools we need to work on CLI Helpers. Whenever you want to work on CLI Helpers, you need to activate the virtual environment:

```
$ source cli_helpers_dev/bin/activate
```

When you're done working, you can deactivate the virtual environment:

```
$ deactivate
```

5. From within the virtual environment, install the dependencies and development tools:

```
$ pip install -r requirements-dev.txt
$ pip install --editable .
```

6. Create a branch for your bugfix or feature based off the `master` branch:

```
$ git checkout -b <name-of-bugfix-or-feature> master
```

7. While you work on your bugfix or feature, be sure to pull the latest changes from `upstream`. This ensures that your local codebase is up-to-date:

```
$ git pull upstream master
```

8. When your work is ready for the CLI Helpers team to review it, make sure to add an entry to CHANGELOG file, and add your name to the AUTHORS file. Then, push your branch to your fork:

```
$ git push origin <name-of-bugfix-or-feature>
```

9. Create a pull request on GitHub.

2.2.2 Running the Tests

While you work on CLI Helpers, it's important to run the tests to make sure your code hasn't broken any existing functionality. To run the tests, just type in:

```
$ pytest
```

CLI Helpers supports Python 3.6+. You can test against multiple versions of Python by running:

```
$ tox
```

You can also measure CLI Helper's test coverage by running:

```
$ pytest --cov-report= --cov=cli_helpers
$ coverage report
```

2.2.3 Coding Style

When you submit a PR, the changeset is checked for pep8 compliance using black. If you see a build failing because of these checks, install `black` and apply style fixes:

```
$ pip install black
$ black .
```

Then commit and push the fixes.

To enforce `black` applied on every commit, we also suggest installing `pre-commit` and using the `pre-commit` hooks available in this repo:

```
$ pip install pre-commit
$ pre-commit install
```

2.2.4 Git blame

Use `git blame my_file.py --ignore-revs-file .git-blame-ignore-revs` to exclude irrelevant commits (specifically Black) from `git blame`. For more information, see here.

2.2.5 Documentation

If your work in CLI Helpers requires a documentation change or addition, you can build the documentation by running:

```
$ make -C docs clean html
$ open docs/build/html/index.html
```

That will build the documentation and open it in your web browser.

2.3 Changelog

2.3.1 Version 2.2.1

(released on 2022-01-17)

- Fix pygments tokens passed as strings

2.3.2 Version 2.2.0

(released on 2021-08-27)

- Remove dependency on terminaltables
- Add psql_unicode table format
- Add minimal table format

- Fix pip2 installing py3-only versions
- Format unprintable bytes (eg 0x00, 0x01) as hex

2.3.3 Version 2.1.0

(released on 2020-07-29)

- Speed up output styling of tables.

2.3.4 Version 2.0.1

(released on 2020-05-27)

- Fix newline escaping in plain-text formatters (ascii, double, github)
- Use built-in unittest.mock instead of mock.

2.3.5 Version 2.0.0

(released on 2020-05-26)

- Remove Python 2.7 and 3.5.
- Style config for missing value.

2.3.6 Version 1.2.1

(released on 2019-06-09)

- Pin Pygments to >= 2.4.0 for tests.
- Remove Python 3.4 from tests and Trove classifier.
- Add an option to skip truncating multi-line strings.
- When truncating long strings, add ellipsis.

2.3.7 Version 1.2.0

(released on 2019-04-05)

- Fix issue with writing non-ASCII characters to config files.
- Run tests on Python 3.7.
- Use twine check during packaging tests.
- Rename old tsv format to csv-tab (because it add quotes), introduce new tsv output adapter.
- Truncate long fields for tabular display.
- Return the supported table formats as unicode.
- Override tab with 4 spaces for terminal tables.

2.3.8 Version 1.1.0

(released on 2018-10-18)

- Adds config file reading/writing.
- Style formatted tables with Pygments (optional).

2.3.9 Version 1.0.2

(released on 2018-04-07)

- Copy unit test from pgcli
- Use safe float for unit test
- Move strip_ansi from tests.utils to cli_helpers.utils

2.3.10 Version 1.0.1

(released on 2017-11-27)

- Output all unicode for terminaltables, add unit test.

2.3.11 Version 1.0.0

(released on 2017-10-11)

- Output as generator
- Use backports.csv only for py2
- Require tabulate as a dependency instead of using vendored module.
- Drop support for Python 3.3.

2.3.12 Version 0.2.3

(released on 2017-08-01)

- Fix unicode error on Python 2 with newlines in output row.
- Fixes to accept iterator.

2.3.13 Version 0.2.2

(released on 2017-07-16)

- Fix IndexError from being raised with uneven rows.

2.3.14 Version 0.2.1

(released on 2017-07-11)

- Run tests on macOS via Travis.
- Fix unicode issues on Python 2 (csv and styling output).

2.3.15 Version 0.2.0

(released on 2017-06-23)

- Make vertical table separator more customizable.
- Add format numbers preprocessor.
- Add test coverage reports.
- Add ability to pass additional preprocessors when formatting output.
- Don't install tests.tabular_output.
- Add .gitignore
- Coverage for tox tests.
- Style formatted output with Pygments (optional).
- Fix issue where tabulate can't handle ANSI escape codes in default values.
- Run tests on Windows via Appveyor.

2.3.16 Version 0.1.0

(released on 2017-05-01)

- Pretty print tabular data using a variety of formatting libraries.

2.4 Authors

CLI Helpers is written and maintained by the following people:

- Amjith Ramanujam
- Dick Marinus
- Irina Truong
- Thomas Roten

2.4.1 Contributors

This project receives help from these awesome contributors:

- Terje Røsten
- Frederic Aoustin
- Zhaolong Zhu

- Karthikeyan Singaravelan
- laixintao
- Georgy Frolov
- Michał Górny
- Waldir Pimenta
- Mel Dafert
- Andrii Kohut

2.4.2 Thanks

This project exists because of the amazing contributors from pgcli and mycli.

2.5 License

CLI Helpers is licensed under the BSD 3-clause license. This basically means you can do what you'd like with the source code as long as you include a copy of the license, don't modify the conditions, and keep the disclaimer around. Plus, you can't use the authors' names to promote your software without their written consent.

2.5.1 License Text

CHAPTER 3

API

3.1 API

3.1.1 Tabular Output

CLI Helper's tabular output module makes it easy to format your data using various formatting libraries.

When formatting data, you'll primarily use the `format_output()` function and `TabularOutputFormatter` class.

cli_helpers.tabular_output.**format_output**(*data*, *headers*, *format_name*, ***kwargs*)
Format output using *format_name*.

This is a wrapper around the `TabularOutputFormatter` class.

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.
- **format_name** (*str*) – The display format to use.
- ****kwargs** – Optional arguments for the formatter.

Returns The formatted data.

Return type str

class cli_helpers.tabular_output.**TabularOutputFormatter**(*format_name=None*)
An interface to various tabular data formatting libraries.

The formatting libraries supported include:

- tabulate
- terminaltables
- a CLI Helper vertical table layout

- delimited formats (CSV and TSV)

Parameters **format_name** (*str*) – An optional, default format name.

Usage:

```
>>> from cli_helpers.tabular_output import TabularOutputFormatter
>>> formatter = TabularOutputFormatter(format_name='simple')
>>> data = ((1, 87), (2, 80), (3, 79))
>>> headers = ('day', 'temperature')
>>> print(formatter.format_output(data, headers))
  day    temperature
-----  -------------
    1             87
    2             80
    3             79
```

You can use any iterable for the data or headers:

```
>>> data = enumerate(('87', '80', '79'), 1)
>>> print(formatter.format_output(data, headers))
  day    temperature
-----  -------------
    1             87
    2             80
    3             79
```

format_name
The current format name.

This value must be in *supported_formats*.

format_output(*data*, *headers*, *format_name=None*, *preprocessors=()*, *column_types=None*, ***kwargs*)
Format the headers and data using a specific formatter.

format_name must be a supported formatter (see *supported_formats*).

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.
- **format_name** (*str*) – The display format to use (optional, if the *TabularOutputFormatter* object has a default format set).
- **preprocessors** (*tuple*) – Additional preprocessors to call before any formatter preprocessors.
- ****kwargs** – Optional arguments for the formatter.

Returns The formatted data.

Return type str

Raises **ValueError** – If the *format_name* is not recognized.

classmethod register_new_formatter(*format_name*, *handler*, *preprocessors=()*, *kwargs=None*)
Register a new output formatter.

Parameters

- **format_name** (*str*) – The name of the format.
- **handler** (*callable*) – The function that formats the data.
- **preprocessors** (*tuple*) – The preprocessors to call before formatting.
- **kwargs** (*dict*) – Keys/values for keyword argument defaults.

supported_formats
The names of the supported output formats in a `tuple`.

Preprocessors

These preprocessor functions are used to process data prior to output.

`cli_helpers.tabular_output.preprocessors.`**align_decimals**(*data*, *headers*, *column_types=()*, **_)
Align numbers in *data* on their decimal points.

Whitespace padding is added before a number so that all numbers in a column are aligned.

Outputting data before aligning the decimals:

```
1
2.1
10.59
```

Outputting data after aligning the decimals:

```
 1
 2.1
10.59
```

Parameters
- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.
- **column_types** (*iterable*) – The columns' type objects (e.g. int or float).

Returns The processed data and headers.

Return type tuple

`cli_helpers.tabular_output.preprocessors.`**bytes_to_string**(*data*, *headers*, **_)
Convert all *data* and *headers* bytes to strings.

Binary data that cannot be decoded is converted to a hexadecimal representation via `binascii.hexlify()`.

Parameters
- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.

Returns The processed data and headers.

Return type tuple

`cli_helpers.tabular_output.preprocessors.`**convert_to_string**(*data*, *headers*, **_)
Convert all *data* and *headers* to strings.

Binary data that cannot be decoded is converted to a hexadecimal representation via `binascii.hexlify()`.

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.

Returns The processed data and headers.

Return type tuple

cli_helpers.tabular_output.preprocessors.**escape_newlines**(*data*, *headers*, **_)

Escape newline characters (-> n, -> r)

param iterable data An iterable (e.g. list) of rows.

param iterable headers The column headers.

return The processed data and headers.

rtype tuple

cli_helpers.tabular_output.preprocessors.**format_numbers**(*data*, *headers*, *column_types=()*, *integer_format=None*, *float_format=None*, **_)

Format numbers according to a format specification.

This uses Python's format specification to format numbers of the following types: `int`, `long` (Python 2), `float`, and `Decimal`. See the Format Specification Mini-Language for more information about the format strings.

Note: A column is only formatted if all of its values are the same type (except for `None`).

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.
- **column_types** (*iterable*) – The columns' type objects (e.g. int or float).
- **integer_format** (*str*) – The format string to use for integer columns.
- **float_format** (*str*) – The format string to use for float columns.

Returns The processed data and headers.

Return type tuple

cli_helpers.tabular_output.preprocessors.**override_missing_value**(*data*, *headers*, *style=None*, *missing_value_token=Token.Output.Null*, *missing_value=''*, **_)

Override missing values in the *data* with *missing_value*.

A missing value is any value that is `None`.

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.

- **headers** (*iterable*) – The column headers.
- **style** – Style for missing_value.
- **missing_value_token** – The Pygments token used for missing data.
- **missing_value** – The default value to use for missing data.

Returns The processed data and headers.

Return type tuple

cli_helpers.tabular_output.preprocessors.**override_tab_value**(*data*, *headers*, *new_value=' '*, ***_*)

Override tab values in the *data* with *new_value*.

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.
- **new_value** – The new value to use for tab.

Returns The processed data and headers.

Return type tuple

cli_helpers.tabular_output.preprocessors.**quote_whitespaces**(*data*, *headers*, *quotestyle="'"*, ***_*)

Quote leading/trailing whitespace in *data*.

When outputing data with leading or trailing whitespace, it can be useful to put quotation marks around the value so the whitespace is more apparent. If one value in a column needs quoted, then all values in that column are quoted to keep things consistent.

Note: `string.whitespace` is used to determine which characters are whitespace.

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.
- **quotestyle** (*str*) – The quotation mark to use (defaults to `'`).

Returns The processed data and headers.

Return type tuple

cli_helpers.tabular_output.preprocessors.**style_output**(*data*, *headers*, *style=None*, *header_token=Token.Output.Header*, *odd_row_token=Token.Output.OddRow*, *even_row_token=Token.Output.EvenRow*, ***_*)

Style the *data* and *headers* (e.g. bold, italic, and colors)

Note: This requires the Pygments library to be installed. You can install it with CLI Helpers as an extra:

```
$ pip install cli_helpers[styles]
```

Example usage:

```
from cli_helpers.tabular_output.preprocessors import style_output
from pygments.style import Style
from pygments.token import Token

class YourStyle(Style):
    default_style = ""
    styles = {
        Token.Output.Header: 'bold ansibrightred',
        Token.Output.OddRow: 'bg:#eee #111',
        Token.Output.EvenRow: '#0f0'
    }

headers = ('First Name', 'Last Name')
data = [['Fred', 'Roberts'], ['George', 'Smith']]

data, headers = style_output(data, headers, style=YourStyle)
```

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.
- **style** (*str/pygments.style.Style*) – A Pygments style. You can create your own styles.
- **header_token** (*str*) – The token type to be used for the headers.
- **odd_row_token** (*str*) – The token type to be used for odd rows.
- **even_row_token** (*str*) – The token type to be used for even rows.

Returns The styled data and headers.

Return type tuple

cli_helpers.tabular_output.preprocessors.**truncate_string**(*data*, *headers*, *max_field_width=None*, *skip_multiline_string=True*, ***_*)

Truncate very long strings. Only needed for tabular representation, because trying to tabulate very long data is problematic in terms of performance, and does not make any sense visually.

Parameters

- **data** (*iterable*) – An iterable (e.g. list) of rows.
- **headers** (*iterable*) – The column headers.
- **max_field_width** (*int*) – Width to truncate field for display

Returns The processed data and headers.

Return type tuple

3.1.2 Config

Read and write an application's config files.

class cli_helpers.config.**Config**(*app_name*, *app_author*, *filename*, *default=None*, *validate=False*, *write_default=False*, *additional_dirs=()*)

Config reader/writer class.

Parameters

- **app_name** (*str*) – The application's name.
- **app_author** (*str*) – The application author/organization.
- **filename** (*str*) – The config filename to look for (e.g. `config`).
- **default** (*dict/str*) – The default config values or absolute path to config file.
- **validate** (*bool*) – Whether or not to validate the config file.
- **write_default** (*bool*) – Whether or not to write the default config file to the user config directory if it doesn't already exist.
- **additional_dirs** (*tuple*) – Additional directories to check for a config file.

additional_files()

Get a list of absolute paths to the additional config files.

all_config_files()

Get a list of absolute paths to all the config files.

data = None

The `ConfigObj` instance.

read()

Read the default, additional, system, and user config files.

Raises ***DefaultConfigValidationError*** – There was a validation error with the *default* file.

read_config_file(*f*)

Read a config file *f*.

Parameters **f** (*str*) – The path to a file to read.

read_config_files(*files*)

Read a list of config files.

Parameters **files** (*iterable*) – An iterable (e.g. list) of files to read.

read_default_config()

Read the default config file.

Raises ***DefaultConfigValidationError*** – There was a validation error with the *default* file.

system_config_files()

Get a list of absolute paths to the system config files.

user_config_file()

Get the absolute path to the user config file.

write(*outfile=None*, *section=None*)

Write the current config to a file (defaults to user config).

Parameters

- **outfile** (*str*) – The path to the file to write to.
- **section** (*None/str*) – The config section to write, or `None` to write the entire config.

write_default_config(*overwrite=False*)
Write the default config to the user's config file.

Parameters **overwrite** (*bool*) – Write over an existing config if it exists.

exception cli_helpers.config.**ConfigError**
Base class for exceptions in this module.

exception cli_helpers.config.**DefaultConfigValidationError**
Indicates the default config file did not validate correctly.

cli_helpers.config.**get_system_config_dirs**(*app_name*, *app_author*, *force_xdg=True*)
Returns a list of system-wide config folders for the application.

For an example application called `"My App"` by `"Acme"`, something like the following folders could be returned:

macOS (non-XDG): `['/Library/Application Support/My App']`

Mac OS X (XDG): `['/etc/xdg/my-app']`

Unix: `['/etc/xdg/my-app']`

Windows 7: `['C:\ProgramData\Acme\My App']`

Parameters

- **app_name** – the application name. This should be properly capitalized and can contain whitespace.
- **app_author** – The app author's name (or company). This should be properly capitalized and can contain whitespace.
- **force_xdg** – if this is set to *True*, then on macOS the XDG Base Directory Specification will be followed. Has no effect on non-macOS systems.

cli_helpers.config.**get_user_config_dir**(*app_name*, *app_author*, *roaming=True*, *force_xdg=True*)
Returns the config folder for the application. The default behavior is to return whatever is most appropriate for the operating system.

For an example application called `"My App"` by `"Acme"`, something like the following folders could be returned:

macOS (non-XDG): `~/Library/Application Support/My App`

Mac OS X (XDG): `~/.config/my-app`

Unix: `~/.config/my-app`

Windows 7 (roaming): `C:\Users\<user>\AppData\Roaming\Acme\My App`

Windows 7 (not roaming): `C:\Users\<user>\AppData\Local\Acme\My App`

Parameters

- **app_name** – the application name. This should be properly capitalized and can contain whitespace.
- **app_author** – The app author's name (or company). This should be properly capitalized and can contain whitespace.
- **roaming** – controls if the folder should be roaming or not on Windows. Has no effect on non-Windows systems.

- **force_xdg** – if this is set to *True*, then on macOS the XDG Base Directory Specification will be followed. Has no effect on non-macOS systems.

Python Module Index

C

Index

S

T

U

W