mdacli

Release 0.1.30

Philip Loche, Joao MC Teixeira and Oliver Beckstein

Apr 03, 2024

CONTENTS

`mdacli` is a simple command line interface (CLI) to the analysis classes of MDAnalysis using argparse. This project is in an **early development stage** and work in progress. Contributions are welcome!

To install *mdacli* refer to the INSTALL file.

Run *mdacli*:

```
mda -h
```

For a help and an overview of the supported modules. A help message for each module is available using:

```
mda <module> -h
```

CHAPTER

ONE

AVAILABLE MODULES

Currently the following analysis modules are available

Module Name	Description
AlignTraj	RMS-align trajectory to a reference structure using a selection.
AverageStructure	RMS-align trajectory to a reference structure using a selection, and calculate the average coordinates of the trajectory.
Contacts	Calculate contacts based observables.
DensityAnalysis	Volumetric density analysis.
DistanceMatrix	Calculate the pairwise distance between each frame in a trajectory
Dihedral	Calculate dihedral angles for specified atomgroups.
Janin	Calculate _1 and _2 dihedral angles of selected group
Ramachandran	Calculate and dihedral angles of selected group
DielectricConstant	Computes the average dipole moment.
GNMAnalysis	Basic tool for GNM analysis.
closeContactGNMAnalysis	GNMAnalysis only using close contacts.
HELANAL	Perform HELANAL helix analysis on your trajectory.
HoleAnalysis	Run *hole* program on a trajectory.
LinearDensity	Linear density profile
EinsteinMSD	Class to calculate Mean Squared Displacement by the Einstein relation.
PCA	Principal component analysis on an MD trajectory.
InterRDF	Intermolecular pair distribution function
RMSD	Class to perform RMSD analysis on a trajectory.
RMSF	Calculate RMSF of given atoms across a trajectory.

More information about each module is available through the help page or at the MDAnalysis documentation.

1.1 Installation

`mdacli` is a *pluging* tool for MDAnalysis. So the installation of *mdacli* depends slightly on your installation of *MDAnalysis*. If you have *MDAnalysis* already installed, you should install *mdacli* on top of it in the same Python environment. So, activate that environment first.

1.1.1 pip

If you just want to use *mdacli*, you can install it with pip:

```
pip install mdacli --no-deps --upgrade
```

Or if you are a *Github* expert you can clone the repository and install it from the repository directly:

```
python setup.py develop --no-deps
```

If you don't have *MDAnalysis* installed and want to install everything together:

```
pip install mdacli --upgrade
```

1.1.2 conda

First installation with conda:

```
conda install -c conda-forge mdacli
```

which will automatically also install *MDAnalysis*.

To upgrade later:

```
conda update mdacli
```

Any doubts please *contact us <https://github.com/MDAnalysis/mdacli/issues>*.

1.2 Philosophy and approach

The mdacli project evolved from our experiences in taurenmd and maicos which both build a CLI interface on the fly.

They were developed since we were facing the same problem in the labs regularly. New students/scientists start writing their analysis scripts and face the same challenges and problems i.e.

- How to initialize the universe and loop through frames without copying many lines of code?
- How to write a CLI parser to analyze several of their simulations?
- How to process and save their trajectories in a clever way?
- …

Some of these problems can be solved by using the `MDAnalysis.analysis.base.AnalysisBase` class of MDA. However, this class is limited to python, and sometimes a direct CLI to these scripts is very helpful for the day-to-day analysis. A generic CLI wrapper for all classes based on the AnalysisBase could therefore help people to analyze their simulation data with the least effort. With this approach, it is easier to use for MDA-users since they just stay within their known universe with known selection commands and results structures. An existing framework makes it also more attractive for users and developers to write their analysis using the *base.AnalysisBase*.

Starting from *taurenmd* and *maicos* we developed a general CLI for any `MDAnalysis.analysis.base.AnalysisBase` class. mdacli detects all analysis classes located inside the MDA project and builds a CLI wrapper around them. The wrapper is generic so it also applies to any downstream project that uses the `MDAnalysis.analysis.base.AnalysisBase` as parent class. If new classes are added to the MDA codebase they are just they will show up without any adjustments to *mdacli* itself.

The core of the wrapper is a docstring parser in combination with an argument inspection using the *inspect* library. Based on a created dictionary containing each parameter of the class with its docstring and type, the actual command line interface is build using `argparse`. The syntax of the topology and the trajectory flags (*-f*, *-s*, ...) is inspired by the GROMACS CLI syntax.

The interface also provides a way to save the data using that all analysis results of an AnalysisBase class is stored inside results objects. The saving routines automatically detects the type of the results and saves them either as JSON dumps (for simple variables), CSV files (for 1D and 2D arrays), or zipped data dumps for high higher-dimensional arrays.

1.3 Usage

To use `mdacli`, after installation open your terminal and run:

```
mda -h
```

This command will provide a list of all available modules. A list of the Available modules is also available on the page of the documentation. Ask help *-h* in each module available for detailed instruction on how to use each module command-line client.

mdacli modules' parameters emulate the parameters of the *Analysis* classes from *MDAnalysis*. So, each module will have its own requirements. Some will require two trajectories, others *AtomGroup* selections, etc. You will see that all is explained in each client *-h* option.

For example, to calculate a radial distribution function (RDF) between two groups use for example:

```
mda interrdf -h
```

A sample water trajectory is provided of rigid SPC/E water is provided online. *topol.tpr* contains a GROMACS topolgy file and *traj.trr* is the corresponding trajectory. The oxygen-oxygen rdf can be calculated using:

```
mda interrdf -s topol.tpr -f traj.trr -g1 "name OW" -g2 "name OW"
```

The oxygen atoms are selected with the *-g1* and *-g2* flags.

A more verbose output is achieved by using the *-v* flag. Even more information is provided with the *–debug* flag. All warnings of the run can also directly stored to a log file using the *–logfile* flag:

```
mda --debug --logfile rdf.log interrdf -v -s topol.tpr -f traj.trr -g1 "name OW" -g2
↪"name OW"
```

The results of each analysis are stored by default in the current directory. The output directory can be changed with the *-o* flag and an additional prefix can be set using the *-pre* flag.

The results of the RDF calculations are two *.csv* files. The actual RDF is saved in the 2nd and 3rd columns of *Inter-RDF_count_bins_rdf.csv*. The header rows of each file provide information about the stored data. Simple results such as bare numbers or strings are stored as *JSON* dumps. More complex data such as 4 or higher-dimensional arrays are saved as a bunch of CSV files zipped together. A similar procedure will happen for each module.

1.4 Contributing

Contributing to this project is easy because we have set up a powerful CI environment. :relaxed: This document will guide you step-by-step.

Contributions are always welcome. Not only in the form of code contributions via pull requests but also issues, especially since the project is in an early development phase.

The way to interact with us is to *fork* the *mdacli* repository to your account, create a development branch in your fork, and finally pull request your changes to the main repository for review. Next, we present you guidelines for this process. If you are a *git* pro you may wish to apply your own *git* method; if not, you are safe following ours.

1.4.1 Fork this repository

Fork this repository before contributing.

1.4.2 Clone the main project to your computer

Next, clone the main repository to your local machine.:

```
git clone https://github.com/MDAnalysis/mdacli.git
cd mdacli
```

1.4.3 Add your fork as remote

You can't create a branch in the main repository :no_entry_sign:, you need to make it in your fork :smile: , first you need to add your fork as a *remote* in the cloned repository (following the previous commands)

```
git checkout main
git remote add myfork git://github.com/<your-username>/mdacli.git
git fetch myfork
```

1.4.4 Create a new branch and start developing

Now create a new branch and start developing:

```
git checkout -b <my-new-branch-with-a-nice-name>
```

Develop your code. You should commit your changes to your fork in encapsulated steps. That is, when you finish doing some task, you should commit it.

> git add <the new files> git commit -m "<your good commit message>" git push myfork <my-new-branch-with-a-nice-name>

You will see that your changes are now in your fork and branch.

1.4.5 Pull Request your changes

Once you finish your changes create a Pull Request to the main *mdacli* repository so we can review your contribution, give you feedback, and hopefully accept it :relaxed:. However, before PR, continue reading this guideline!

1.4.6 Install for developers

If you are contributing to `madcli`, most likely you are already a contributor of `MDAnalysis v2`. If you already have *MDAnalysis v2* installed, install *mdacli* in the same Python environment. For that, inside *mdacli*'s repository:

```
python setup.py develop --no-deps
```

It is very important to use the *develop* flag, so that the changes you make in the code are always reflected (real time) in your installation. The *–no-deps* avoids installing *MDAnalysis* twice.

If you don't have *MDAnalysis v2* installed how did you run *mdacli* in the first place? Please refer to *Installation* before continuing.

The whole *mdacli* Continuous Integration pipeline is based on *Tox*_. So you need to install *tox* to be our friend :smile_cat:

```
pip install tox tox-conda
# or
conda install tox tox-conda -c conda-forge
```

1.4.7 Running tests with tox

Before creating a Pull Request from your branch, certify that all the tests pass correctly by running:

```
tox
```

These are exactly the same tests that will be performed online in our Github Actions workflows. Possibly, some tests referring to specific Python versions may fail because the interpreter is not installed; ignored these tests.

Also, you can run individual environments if you wish to test only specific functionalities, for example:

```
tox -e lint  # code style
tox -e build  # packaging
tox -e docs  # only builds the documentation
tox -e test  # testing
```

1.4.8 Update CHANGELOG

Update the changelog file under `docs/CHANGELOG.rst` with an explanatory bullet list of your contribution bellow the *CHANGELOG* title. Add that list right after the main title and before the last version subtitle:

```
Changelog
=========

* here goes my new additions
* explain them shortly and well

```

(continues on next page)

(continued from previous page)

```
vX.X.X (1900-01-01)
-------------------
```

Also add your name to the authors list at `docs/AUTHORS.rst`.

1.4.9 Pull Request

Once you are finished, you can Pull Request you additions to the main repository and engage with the community. Please read the *docs/PULLREQUEST.rst* guidelines first, you will see them when you open a PR.

Before submitting a Pull Request, verify your development branch passes all tests as *described* **. If you are developing new code you should also implement new test cases.**

Also, before PR, update your development branch to the upstream main branch to certify there are no incompatibilities:

```
git checkout main
git pull
git checkout <my-new-branch-with-a-nice-name>
git merge --no-ff main
```

Correct any conflicts that may appear. It there are no conflicts, you are good to go (Pull Request).

1.5 API library documentation

The API library documentation references the internals of *mdacli*. Only advanced users with very specific development goals might need to use such interfaces. A regular user wanting to use *mdacli* from the command-line does not need to refer to these pages, refer to *Usage* instead.

1.5.1 High-Level API

The toplevel command line interface.

`mdacli.cli.cli`(*name*, *module_list*, *base_class=<Mock name='mock.AnalysisBase' id='139991830367728'>*, *version=''*, *description=''*, *skip_modules=None*, *ignore_warnings=False*)

Create the command-line interface.

This function creates a command line interface with a given *name* based on a module list.

Parameters

- **name** (*str*) – name of the interface
- **module_list** (*list*) – list of module from which the cli is build up.
- **base_class** (*cls*) – Class or list of classes that the Anaylsis modules belong to
- **description** (*str*) – description of the cli
- **skip_modules** (*list*) – list of strings containing modules that should be ommited
- **ignore_warnings** (*bool*) – ignore warnings when importing modules

Examples

This example code creates the command line interface of MDAnalysis:

```
from MDAnalysis.analysis import __all__

import mdacli

skip_mods = ['AnalysisFromFunction',
             'HydrogenBondAnalysis',
             'WaterBridgeAnalysis',
             'Contacts',
             'Dihedral',
             'PersistenceLength',
             'InterRDF_s']

mdacli.cli(name="MDAnalysis",
           module_list=[f'MDAnalysis.analysis.{m}' for m in __all__],
           version=mdacli.__version__,
           description=__doc__,
           skip_modules=skip_mods,
           ignore_warnings=True)
```

1.5.2 Support functions for CLI

Functionalities that support the creation of the command lines interface.

class mdacli.libcli.**KwargsDict**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Convert input string to a dictionary.

If string points to a ".json" file, reads the file. Else, attempts to convert string to dictionary using json.loads.

mdacli.libcli.**add_cli_universe**(*parser*, *name=''*)

Add universe parameters to an given argparse.ArgumentParser.

instance. The parameters *topology*, *topology_format*, *atom_style*, *coordinates* and *trajectory_format* are added to the parse.

Parameters

- **analysis_class_parser** (*argparse.ArgumentParser*) – The ArgumentsParser instance to which the run grorup is added
- **name** (*str*) – suffix for the argument names

mdacli.libcli.**add_output_group**(*analysis_class_parser*)

Add output group parameters to argparse.ArgumentParser instance.

The run group adds the parameters *output_prefix* and *output_directory* to the parser.

Parameters

analysis_class_parser (*argparse.ArgumentParser*) – The ArgumentsParser instance to which the run grorup is added

mdacli.libcli.**add_run_group**(*analysis_class_parser*)

Add run group parameters to an given argparse.ArgumentParser instance.

The run group adds the parameters *start*, *stop*, *step*, *verbose* to the parser.

Parameters

analysis_class_parser (*argparse.ArgumentParser*) – The ArgumentsParser instance to which the run grorup is added

mdacli.libcli.**convert_analysis_parameters**(*analysis_callable*, *analysis_parameters*, *reference_universe=None*)

Convert parameters from the command line suitable for anlysis.

Special types (i.e AtomGroups, Universes) are converted from the command line strings into the correct format. Parameters are changed inplace. Note that only keys are converted and no new key are added if present in the doc of the *analysis_callable* but not in the *analysis_parameters* dict.

AtomGroup selection with type None are ignored since these could be default arguments.

The following types are converted:

- AtomGroup: Select atoms based on `universe.select_atoms`
- list[AtomGroup]: Select atoms based on `universe.select_atoms` for every element in list
- Universe: Created from parameters.

Parameters

- **analysis_callable** (*function*) – Analysis class for which the analysis should be performed.
- **analysis_parameters** (*dict*) – parameters to be processed
- **reference_universe** (*MDAnalysis.Universe*) – Universe from which the AtomGroup selection are done.

Returns

universe (*Universe*) – The universe created from the anaylysis parameters or None of no ine is created

Raises

`ValueError` – If an Atomgroup does not contain any atoms

mdacli.libcli.**create_cli**(*sub_parser*, *interface_name*, *parameters*)

Add subparsers to *cli_parser*.

Subparsers parameters are divided in the following categories:

1. **Analysis Run parameters**
   time frame as begin, end, step and vebosity
2. **Saving Parameters**
   output_prefix and output_directory
3. **Mandatory Parameters**
   mandatory parameters are defined in the CLI as named parameters as per design
4. **Optional Parameters**
   Named parameters in the Analysis class
5. **Reference Universe Parameters**
   A reference Universe for selection commands. Only is created if AtomGroup arguments exist.

All CLI's parameters are named parameters.

Parameters

- **sub_parser** (*argparse.sub_parser*) – A sub parser where the new parser will be added.
- **interface_name** (*str*) – Name of the interface.
- **parameters** (*dict*) – Parameters needed to fill the argparse requirements for the CLI interface.

Returns

None

mdacli.libcli.**create_universe**(*topology*, *coordinates=None*, *topology_format=None*, *trajectory_format=None*, *atom_style=None*, *dimensions=None*)

Initilize a MDAnalysis universe instance.

Parameters

- **topology** (str, stream, *~MDAnalysis.core.topology.Topology*, *np.ndarray*) – A CHARMM/XPLOR PSF topology file, PDB file or Gromacs GRO file; used to define the list of atoms. If the file includes bond information, partial charges, atom masses, ... then these data will be available to MDAnalysis. Alternatively, an existing `MDAnalysis.core.topology.Topology` instance may be given, numpy coordinates, or None for an empty universe.
- **coordinates** (*str, stream, list of str, list of stream*) – Coordinates can be provided as files of a single frame (eg a PDB, CRD, or GRO file); a list of single frames; or a trajectory file (in CHARMM/NAMD/LAMMPS DCD, Gromacs XTC/TRR, or generic XYZ format). The coordinates must be ordered in the same way as the list of atoms in the topology. See Table of supported coordinate formats for what can be read as coordinates. Alternatively, streams can be given.
- **topology_format** (*str, None*) – Provide the file format of the topology file; `None` guesses it from the file extension. Can also pass a subclass of `MDAnalysis.topology.base.TopologyReaderBase` to define a custom reader to be used on the topology file.
- **trajectory_format** (*str or list or object*) – provide the file format of the coordinate or trajectory file; `None` guesses it from the file extension. Note that this keyword has no effect if a list of file names is supplied because the "chained" reader has to guess the file format for each individual list member [`None`]. Can also pass a subclass of `MDAnalysis.coordinates.base.ProtoReader` to define a custom reader to be used on the trajectory file.
- **atom_style** (*str*) – Customised LAMMPS *atom_style* information. Only works with *topology_format* = *data*
- **dimensions** (*iterable of floats*) – vector that contains unit cell lengths and probable angles. Expected shapes are eithere (6, 0) or (1, 6) or for shapes of (3, 0) or (1, 3) all angles are set to 90 degrees.

Raises

`IndexError` – If the dimesions of the *dimensions* argument are not 3 or 6.

Returns

MDAnalysis.Universe

mdacli.libcli.**find_classes_in_modules**(*cls*, **module_names*)

Find classes that belong to cls in modules.

A series of names can be given as arguments.

Parameters

- **cls** (*single class or list of classes*) – parent reference class type to search for

- **module_names** (*str*) – module to import import in absolute or relative terms (e.g. either pkg.mod or ..mod).

Returns

- *list*
- *list of found class objects. If no classes are found, return None.*

mdacli.libcli.**find_cls_members**(*cls*, *modules*, *ignore_warnings=False*)

Find members of a certain class in modules.

Parameters

- **cls** (*class or list of classes*) – parent reference class or list of classes to be searched for
- **modules** (*list*) – list of modules for which members should be searched for
- **ignore_warnings** (*bool*) – Flag to ignore warnings

mdacli.libcli.**init_base_argparse**(*name*, *version*, *description*)

Create a basic *ArgumentParser*.

The parser has options for printing the version, running in debug mode and with a logfile. Note that the funtion only adds the options to the parser but not the logic for actually running in debug mode nor how to store the log file.

Parameters

- **name** (*str*) – Name of the cli program
- **version** (*str*) – Version of the cli program
- **description** (*str*) – Description of the cli program

Returns

ArgumentParser

mdacli.libcli.**run_analysis**(*analysis_callable*, *mandatory_analysis_parameters*, *optional_analysis_parameters=None*, *reference_universe_parameters=None*, *run_parameters=None*, *output_parameters=None*)

Perform main client logic.

Parameters

- **analysis_callable** (*function*) – Analysis class for which the analysis is performed.
- **mandatory_analysis_parameters** (*dict*) – Mandatory parameters for executing the analysis
- **optional_analysis_parameters** (*dict*) – Optional parameters for executing the analysis
- **run_parameters** (*dict*) – time frame parameters: start, stop, step, verbose
- **output_parameters** (*dict*) – output_prefix and output_directory

Returns

MDAnalysis.analysis.base.AnalysisBase – AnalysisBase instance of the given `analysis_callable` after run.

mdacli.libcli.**setup_clients**(*ap*, *title*, *members*)

Set up analysis clients for an ArgumentParser instance.

Parameters

- **ap** (*argparse.ArgumentParser*) – Argument parser instance

- **title** (*str*) – title of the parser
- **members** (*list*) – list containing Analysis classes for setting up the parser

mdacli.libcli.**split_argparse_into_groups**(*parser*, *namespace*)

Split the populated namespace of argparse into groups.

https://stackoverflow.com/questions/31519997/is-it-possible-to- only-parse-one-argument-groups-parameters-with-argparse

Parameters

- **parse** (*argparse.ArgumentParser*) – argument parser instance
- **namespace** (*argparse.Namespace*) – instance storing the parameters

Returns

arg_grouped_dict (*dict*) – Dictionary containing parameters split according to their groups

1.5.3 Results Saving

Manage data saving.

mdacli.save.**get_1D_arrays**(*results*)

Get items from dict which correspond to np.ndarrays one dim.

mdacli.save.**get_cli_input**()

Return a proper fomatted string of the command line input.

mdacli.save.**is_1d_array**(*arr*, ***, *ndim=1*)

Assert value is array and of certain dimension.

mdacli.save.**is_2d_array**(*arr*, ***, *ndim=2*)

Assert value is array and of certain dimension.

mdacli.save.**is_3d_array**(*arr*, ***, *ndim=3*)

Assert value is array and of certain dimension.

mdacli.save.**is_dimension_array**(*arr*, *ndim*)

Assert value is array and of certain dimension.

mdacli.save.**is_higher_dimension_array**(*arr*, *ndim*)

Assert value is array and of certain dimension.

mdacli.save.**is_serializable**(*value*)

Assert if value is json serializable.

mdacli.save.**remove_files**(*files*)

Remove files from disk.

mdacli.save.**return_with_remove**(*ddict*, *keys*, *remove*)

Serve all saving functions.

If remove is true, Returns subset of keys from dict. Removes keys subset from original dict.

Else, return None.

mdacli.save.**save**(*results*, *fprefix='mdacli_results'*)

Save the attributes of a results instance to disk.

1D, 2D and 3D numpy arrays are saved to csv files. 1D arrays of the same length are vertically stacked to create a table. 2D arrays are saved directly. 3D arrays are split into 2D arrays along the shortest dimension and one CSV is saved for each 2D array created, and resulting CSVs are stored together in a ZIP file. Note: higher dimensional arrays are ignored.

We try to save everything else in a JSON file. Non-serializable types are ignored.

Parameters

- **fprefix** (*str*) – prefix for all files saved
- **results** (*~MDAnalysis.analysis.base.Results*) – A Results instance from which the stored data is taken.

mdacli.save.**save_1D_arrays**(*results*, *fprefix='1darray'*, *remove=True*)

Save 1D arrays from results.

Parameters

- **results** (*dict-like*) – Dictionary containing results.
- **remove** (*bool*) – If true remove keys mapping to 1D numpy arrays.

mdacli.save.**save_2D_arrays**(*results*, *fprefix='2Darr'*, *remove=True*)

Save items of 2D array.

mdacli.save.**save_3D_array_to_2D_csv**(*item*, *arr_name='arr'*, *zipit=True*)

Save 3D array to 2D CSVs.

Has option to store all in a ZIP file.

mdacli.save.**save_3D_arrays**(*results*, *fprefix='3Darr'*, *remove=True*)

Save items of 2D array.

mdacli.save.**save_Results_object**(*results*, *fprefix='results'*, *remove=True*)

Save results if they are Results objects.

mdacli.save.**save_files_to_zip**(*files*, *zipname='thezip'*, *remove=True*)

Compress all files into a single zip archive.

Parameters

- **files** (*list-like*) – File names to save to the ZIP archive.
- **zipname** (*str*) – The name of the zip file without extension.
- **remove** (*bool, option, default True*) – Removes the original files.

mdacli.save.**save_higher_dim_arrays**(*results*, *fprefix='XDarr'*, *remove=True*, *min_ndim=4*)

Save items of multidimensional arrays to CSV.

mdacli.save.**save_json_serializables**(*results*, *remove=True*, ***jsonargs*)

Save serializable items to a JSON.

mdacli.save.**save_result_array**(*arr*, *fprefix='prefix'*)

Save array to disk accoring to num of dimensions.

mdacli.save.**save_to_json**(*json_dict*, *fname='jdict'*, *indent=4*, *sort_keys=True*)

Save dictionary to JSON file.

mdacli.save.**savetxt_w_command**(*fname*, *X*, *header=''*, *fsuffix='.csv'*, ***kwargs*)

Save CSV data with info about execution command.

Adds the command line input to the header and checks for a doubled defined filesuffix.

mdacli.save.**stack_1d_arrays_list**(*list_1D*, *extra_list=None*)

Stack a list of 1D numpy arrays of the same length vertically together.

The result is a list containing 2D arrays where each array got the same number of rows.

Parameters

- **list_1d** (*list*) – list of 1 dimensional numpy arrays
- **extra_list** (*list*) – additional list of numpy arrays on which the operations are executed as for `list_1d`

Returns

- **out_list** (*list*) – list of stacked 2D numpy arrays organized by their length
- **out_extra** (*list*) – list of stacked 2D numpy applied applied to the same operations as out_list

mdacli.save.**try_to_squeeze_me**(*arr*)

Squeeze the arr if is array.

1.5.4 Coloring

Emphaising strings with colors etc.

Taken from https://gist.github.com/tuvokki/14deb97bef6df9bc6553.

class mdacli.colors.**Emphasise**

Class for emphaising strings with colors etc.

Variables

- **bold** (*str*) – bold attribute
- **underline** (*str*) – underline attribute
- **gray** (*str*) – gray color
- **red** (*str*) – red color
- **green** (*str*) – green color
- **yellow** (*str*) – yellow color
- **blue** (*str*) – blue color
- **pink** (*str*) – pink color
- **turquoise** (*str*) – turquoise color

blue = '\x1b[94m'

bold = '\x1b[1m'

static debug(*message*)

Return a turquoise debug message.

Parameters

message (*str*) – turquoise debug message to return

Returns
decorated_message (*str*) – decorated message

Example

```
>>> print(bcolors.debug("a=1"))
```

static emphasise(*str*, *style*)

Decorate a `str` with desired style.

The Style could be a color, bold or underline.

Parameters

- **message** (*str*) – message to print
- **style** (*str*) – emphasising style. See class attributes for available styles

Example

```
>>> print(Emphasise.emphasise("My colored message", Emphasise.blue))
```

static error(*message*)

Return a red error.

Parameters
message (*str*) – red error to return

Returns
decorated_message (*str*) – decorated message

Example

```
>>> print(bcolors.error("Potential Danger!"))
```

gray = '\x1b[90m'

green = '\x1b[92m'

static header(*message*)

Return a pink header.

Parameters
message (*str*) – pink header to return

Returns
decorated_message (*str*) – decorated message

Example

```
>>> print(bcolors.header("This is great"))
```

static info(*message*)

Return a blue info.

Parameters
message (*str*) – blue info to return

Returns
decorated_message (*str*) – decorated message

Example

```
>>> print(bcolors.info("Blue Yay!"))
```

static ok(*message*)

Return a green ok.

Parameters
message (*str*) – green ok to return

Returns
decorated_message (*str*) – decorated message

Example

```
>>> print(bcolors.ok("Yay!"))
```

pink = '\x1b[95m'

red = '\x1b[91m'

turquoise = '\x1b[96m'

underline = '\x1b[4m'

static warning(*message*)

Return a yellow warning.

Parameters
message (*str*) – yellow warning to print

Returns
decorated_message (*str*) – decorated message

Example

```
>>> print(bcolors.warning("Potential Danger!"))
```

yellow = '\x1b[93m'

1.5.5 Logging

Logging.

mdacli.logger.**setup_logging**(*logobj*, *logfile=None*, *level=30*)

Create a logging environment for a given logobj.

Parameters

- **logobj** (*logging.Logger*) – A logging instance
- **logfile** (*str*) – Name of the log file
- **level** (*int*) – Set the root logger level to the specified level. If for example set to *logging.DEBUG* detailed debug logs inludcing filename and function name are displayed. For :py:obj:`logging.INFO only the message logged from errors, warnings and infos will be displayed.

1.5.6 Utilities

Useful helper functions for running the cli.

mdacli.utils.**convert_str_time**(*x*, *dt*)

Convert a string *x* into a frame number based on given *dt*.

If *x* does not contain any units its assumed to be a frame number already.

See *split_time_unit()*.

Parameters

- **x** (*str*) – the input string
- **dt** (*float*) – the time step in ps

Returns

int – frame number

Raises

ValueError – The input does not contain any units but is not an integer.

mdacli.utils.**parse_callable_signature**(*callable_obj*)

Parse a callable signature to a convenient dictionary for CLI creation.

The parameters used in the CLI are a combination of the callable signature and the information in the callable docstring.

Parameters

callable_obj (*callable*) – The callable object to inspect. Details of this object required for the creation of a CLI are added to the *storage_dict*.

Returns

dict

mdacli.utils.**parse_docs**(*klass*)

Parse classes docstrings to a convenient dictionary.

This parser is based on NumpyDocString format, yet it is not so strict. Combined docstrings from class main docstring and *__init__* method.

Parameters
klass (*callable*) – A klass object from which a DOCSTRING can be extracted.

Returns

tuple (str, str, dict of dict) –

- One line summary description of the callable
- Extended description of the callable
- **dictionary where keys are parameter names and subdictinary**
  has keys "type" and "desc" for parameter type and description.

mdacli.utils.**split_time_unit**(*s*)

Split time and units.

Follows the regex:: https://regex101.com/r/LZAbil/2

Returns
tuple (float, str) – Value as float, units as str.

Raises
IndexError – Tuple could not be found. This happens when a number is not present in the start of the string.

1.6 Changelog

1.6.1 v0.1.30 (2024-04-03)

- Replace Boolean `debug` option in `setup_logging` by more flexible integer `level` parameter.

1.6.2 v0.1.29 (2024-03-21)

- Change handling of lowercase module names

1.6.3 v0.1.28 (2023-09-29)

- Make choices style parsable (#114)

1.6.4 v0.1.27 (2023-08-25)

- Reenable docs on https://mdacli.mdanalysis.org

1.6.5 v0.1.26 (2023-06-27)

- Make the canonical names of mda types parsable.

1.6.6 v0.1.25 (2023-02-21)

- Add integration test running a complete analysis.
- Rename run_analsis to run_analysis
- Move *-nt* to base args (Fixes #109)
- set *long-description* to single file
- remove Py3.8
- Fixing codecov upload
- Translate `setup.py` into `pyproject.toml`

1.6.7 v0.1.24 (2023-01-27)

- Fix classifier in setup.cfg

1.6.8 v0.1.23 (2023-01-27)

- Disable docs deploy on Github Pages
- Add Python 3.11 to CI matrix

1.6.9 v0.1.22 (2023-01-27)

- Limit number of threads used by analysis

1.6.10 v0.1.21 (2022-04-22)

- Allow None as start/stop values (need for 1 frame trajectories)

1.6.11 v0.1.20 (2022-04-13)

- Fix isort dependency
- Added tests for Python 3.10
- Introduce new setup.cfg and pyproject.toml
- Update __authors__

1.6.12 v0.1.19 (2022-04-12)

- Update README.rst for MDAnalysis 2.1.0 modules

1.6.13 v0.1.18 (2022-04-12)

- Fixed typo in libcli.py

1.6.14 v0.1.17 (2022-04-07)

- Added dihedral module

1.6.15 v0.1.16 (2022-02-25)

- Do not convert None types

1.6.16 v0.1.15 (2022-02-25)

- Set positional arguments as required in cli

1.6.17 v0.1.14 (2022-02-17)

- corrects bump2version changelog update

1.6.18 v0.1.13 (2022-02-16)

- Added conda package install instructions (#88)

1.6.19 v0.1.12 (2022-01-19)

- Support list of AtomGroups as parameters (#82)
- Simplify *add_argument* logic in *create_CLI* (#82)
- Allow list of reference classes in module detection (#82)
- Support for generic classes as reference in module detection (#82)
- Rename *save_results `* to *save* (#82)
- More tests for docstring parsing and CLI creation (#82)

1.6.20 v0.1.11 (2022-01-19)

- Improved help for run parameters (#83)

1.6.21 v0.1.10 (2022-01-18)

- Removed conda dependency from CI and tox (#86)

1.6.22 v0.1.9 (2022-01-16)

- Fix test banner in README.rst (#85)

1.6.23 v0.1.8 (2022-01-16)

- Use Github actions matrix for tests (#68)
- Fix Conda permissions on MacOS (#68)
- Fix Tests failing on Windows (#68)

1.6.24 v0.1.7 (2021-12-18)

- Improves regex to convert from time to frame (#81)

1.6.25 v0.1.6 (2021-12-01)

- Fixed URL in docs (#80)

1.6.26 v0.1.5 (2021-12-01)

- Add doc deployment to CI (#78)

1.6.27 v0.1.4 (2021-11-24)

- Link docs to mdacli.mdanalysis.org (#75)

1.6.28 v0.1.3 (2021-11-24)

- MDA-style documentation pages (#70)

1.6.29 v0.1.2 (2021-11-18)

- Added option to manually set box dimensions (#65)

1.6.30 v0.1.1 (2021-11-18)

- corrects .bumpversion.cfg for CHANGELOG
- updates docs/CONTRIBUTING.rst accordingly

1.6.31 v0.1.0 (2021-11-18)

- Initial release

1.7 Authors

- Philip Loche (github)
- Joao M. C. Teixeira (webpage, github)
- Oliver Beckstein (webpage, github)
- Lily Wang (github)
- Henrik Jäger (github)

PYTHON MODULE INDEX

m

INDEX