ChemCoord Documentation

Release 1.0.0

Oskar Weser

Sep 27, 2017

Contents

CHAPTER 1

Features

- You can use it as a python module.
- It reliably converts from Cartesian space (xyz-files) to internal coordinates (zmat-files) **without** introducing dummy atoms. Even in the case of linearity.
- The created zmatrix is not only a transformation to internal coordinates, it is a "chemical" zmatrix. By chemical I mean, that e.g. distances are along bonds or dihedrals are defined as you draw them in chemical textbooks.
- It derived from my own work and I heavily use it during the day. So all functions are tested and tailored around the workflow in theoretical chemistry.
- The classes are safe to inherit from and you can easily costumize it for the needs of your project.

CHAPTER 2

Contents

Installation Guide

You need a working python (both python2 and 3) installation together with some standard modules. You can use for example the anaconda3 installer.

The advantage of the anaconda3 installer is that you get a lot of additional modules and programs, that make it really easy to work with python. For example Ipython and the jupyter notebooks. I highly recommend to use those.

Unix

Just type in your terminal:

```
pip install chemcoord
```

This should also resolve all dependencies automatically.

Windows

I tested neither installation nor running the module on windows. As far as I know it should work as well if you use the `pip` manager. If you get it installed and running, please report it on the Github page.

Introduction and General Structure

What you need to know

I assume that you know python.

You can use chemcoord without knowing Pandas, but it gives you a great advantage. If you invest 1h for their tutorial you will greatly increase your productivity in scientific data analysis.

It also helps to know about numpy.

Internal representation of Data

This module uses pandas DataFrames to represent cartesian and internal coordinates. (I will refer to them in lab slang as xyz and zmat)

The xyz_frame has at least four columns `['atom', 'x', 'y', 'z']`.

The zmat_frame has at least seven columns `['atom', 'bond_with', 'bond', 'angle_with', 'angle', 'dihedral_with', 'dihedral']`.

Since they are normal pandas DataFrames you can do everything with them as long as you respect this structure. This means it is possible to append e.g. a column for the masses of each atom. Besides you can use all the capabilities of pandas.

If you want for example to get only the oxygen atoms of a xyz_frame you can use boolean slicing:

```
xyz_frame[xyz_frame['atom'] == 'O']
```

Main classes of this module

The "working horses" of this module are the `Cartesian` and the `Zmat` class. The have the methods to operate on their coordinates.

An methods of an instance of the `Cartesian` class usually return new instances of *Cartesian*. Besides all methods are **sideeffect free unless otherwise stated**.

Let's assume you have a `molecule1` and you want to cut a sphere around the origin which gives you `molecule2`:

```
molecule2 = molecule1.cutsphere()
```

If you try this, you will see that:

- molecule2 is a `Cartesian` instance.
- molecule1 remains unchanged.

Tutorial

Just follow the link to the Example notebook. If you want to have an interactive session, you have to download and open it with jupyter.

Documentation

Contents:

Cartesian coordinates

Cartesian(init)	The main class for dealing with cartesian Coordinates.

Continued on next page

Table 2.1 – continued from previous page

read_xyz(inputfile[, pythonic_index, get_bonds])	Reads a xyz file.
read_molden(inputfile[, pythonic_index, ...])	Reads a molden file.
write_molden(cartesian_list, outputfile)	Writes a list of Cartesians into a molden file.

Cartesian

class chemcoord.xyz_functions.**Cartesian**(*init*)

The main class for dealing with cartesian Coordinates.

Chemical Methods

get_bonds([modified_properties, ...])	Returns a dictionary representing the bonds.
to_zmat([buildlist, fragment_list, ...])	Transform to internal coordinates.
location([indexlist])	Returns the location of an atom.
bond_lengths(buildlist[, start_row])	Return the distances between given atoms.
angle_degrees(buildlist[, start_row])	Return the angles between given atoms.
dihedral_degrees(buildlist[, start_row])	Return the angles between given atoms.
move([vector, matrix, indices, copy])	Move a Cartesian.
basistransform(new_basis[, old_basis, ...])	Transforms the frame to a new basis.
add_data([list_of_columns, inplace])	Adds a column with the requested data.
total_mass()	Returns the total mass in g/mol.
barycenter()	Returns the mass weighted average location.
inertia()	Calculates the inertia tensor and transforms along rotation axes.
topologic_center()	Returns the average location.
cutcuboid([a, b, c, origin, outside_sliced, ...])	Cuts a cuboid specified by edge and radius.
cutsphere([radius, origin, outside_sliced, ...])	Cuts a sphere specified by origin and radius.
connected_to(index_of_atom[, exclude, ...])	Returns a Cartesian of atoms connected to the specified one.
distance_to([origin, ...])	Returns a Cartesian with a column for the distance from origin.
get_fragment(list_of_indextuples[, ...])	Get the indices of the atoms in a fragment.
fragmentate([give_only_index])	Get the indices of non bonded parts in the molecule.
make_similar(Cartesian2[, follow_bonds, ...])	Similarizes two Cartesians.
align(Cartesian2[, ignore_hydrogens])	Aligns two Cartesians.
change_numbering(rename_dict[, inplace])	Returns the reindexed version of Cartesian.
move_to(Cartesian2[, step, extrapolate])	Returns list of Cartesians for the movement from self to Cartesian2.
partition_chem_env([follow_bonds])	This function partitions the molecule into subsets of the same chemical environment.

get_bonds

Cartesian.**get_bonds**(*modified_properties=None*, *maximum_edge_length=25*, *difference_edge=6*, *use_valency=False*, *use_lookup=False*, *set_lookup=True*, *divide_et_impera=True*, *atomic_radius_data='atomic_radius_cc'*)

Returns a dictionary representing the bonds.

Warning: This function is **not sideeffect free**, since it assigns the output to a variable `self.__bond_dic` if `set_lookup` is `True` (which is the default). This is necessary for performance reasons.

The Cartesian().get_bonds() method will use or not use a lookup depending on `use_lookup`. Greatly increases performance if True, but could introduce bugs in certain situations.

Just imagine a situation where the `Cartesian().frame` is changed manually. If you apply lateron a method e.g. `to_zmat()` that makes use of `get_bonds()` the dictionary of the bonds may not represent the actual situation anymore.

You have two possibilities to cope with this problem. Either you just re-execute `get_bonds` on your specific instance, or you change the `internally_use_lookup` option in the settings submodule. Please note that the internal use of the lookup variable greatly improves performance.

Parameters

- **modified_properties** (*dic*) – If you want to change the van der Vaals radius or valency of one or more specific atoms, pass a dictionary that looks like:

```
modified_properties = {index1 :
    {'atomic_radius' : 1.5, 'valency' : 8}, ...}
```

  For global changes use the constants.py module.

- **maximum_edge_length** (*float*) – Maximum length of one edge of a
- **if divide_et_impera is True.** (*cuboid*) –
- **difference_edge** (*float*) –
- **use_valency** (*bool*) – If `True` atoms can't have more bonds than their valency. This means that the bonds, exceeding the number of valency, with lowest overlap will be cut, although the van der Waals radii overlap.
- **use_lookup** (*bool*) –
- **set_lookup** (*bool*) –
- **divide_et_impera** (*bool*) – Since the calculation of overlaps or distances between atoms scale with $O(n^2)$, it is recommended to split the molecule in smaller cuboids and calculate the bonds in each cuboid. The scaling becomes then $O(n \log(n))$. This approach can lead to problems if `use_valency` is `True`. Bonds from one cuboid to another can not be counted for the valency.. This means that in certain situations some atoms can be oversaturated, although `use_valency` is `True`.
- **atomic_radius_data** (*str*) – Defines which column of `constants.elements` is used. The default is `atomic_radius_cc` and can be changed with `settings.atomic_radius_data`. Compare with *add_data()*.

Returns Dictionary mapping from an atom index to the indices of atoms bonded to.

Return type dict

to_zmat

`Cartesian.`**`to_zmat`**(*buildlist=None*, *fragment_list=None*, *check_linearity=True*)

Transform to internal coordinates.

Transforming to internal coordinates involves basically three steps:

1. Define an order of how to build.
2. **Check for problematic local linearity. In this algorithm an** angle with `170 < angle < 10` is assumed to be linear. This is not the mathematical definition, but makes it safer against "floating point noise"
3. **Calculate the bond lengths, angles and dihedrals using the** references defined in step 1 and 2.

In the first two steps a so called `buildlist` is created. This is basically a `np.array` of shape `(n_atoms, 4)` and

integer type.

The four columns are ''['own_index', 'bond_with', 'angle_with', 'dihedral_with']''.

This means that usually the upper right triangle can be any number, because for example the first atom has no other atom as reference.

It is important to know, that getting the buildlist is a very costly step since the algoritym tries to make some guesses based on the connectivity to create a "chemical" zmatrix.

If you create several zmatrices based on the same references you can save the buildlist of a zmatrix with `Zmat.build_list()`.

If you then pass the buildlist as argument to `to_zmat`, then the algorithm directly starts with step 3.

Another thing is that you can specify fragments. For this purpose the function *`Cartesian.get_fragment()`*

is quite handy.

An element of fragment_list looks like:

```
(fragment, connections)
```

Fragment is a `Cartesian` instance and connections is a `(3, 4)` numpy integer array, that defines how the fragment is connected to the molecule.

Parameters

- **`buildlist`** (*`np.array`*) –
- **`fragment_list`** (*`list`*) –
- **`check_linearity`** (*`bool`*) –

Returns A new instance of `Zmat`.

Return type *Zmat*

location

`Cartesian.`**`location`**(*indexlist=None*)

Returns the location of an atom.

You can pass an indexlist or an index.

Parameters

- **frame** (*pd.dataframe*) –
- **indexlist** (*list*) – If indexlist is None, the complete index is used.

Returns A matrix of 3D rowvectors of the location of the atoms specified by indexlist. In the case of one index given a 3D vector is returned one index.

Return type np.array

bond_lengths

Cartesian.**bond_lengths**(*buildlist*, *start_row=0*)

Return the distances between given atoms.

In order to know more about the buildlist, go to *to_zmat()*.

Parameters

- **buildlist** (*np.array*) –
- **start_row** (*int*) –

Returns

Vector of the distances between the first and second atom of every entry in the buildlist.

Return type list

angle_degrees

Cartesian.**angle_degrees**(*buildlist*, *start_row=0*)

Return the angles between given atoms.

In order to know more about the buildlist, go to *to_zmat()*.

Parameters

- **buildlist** (*list*) –
- **start_row** (*int*) –

Returns

List of the angle between the first, second and third atom of every entry in the buildlist.

Return type list

dihedral_degrees

Cartesian.**dihedral_degrees**(*buildlist*, *start_row=0*)

Return the angles between given atoms.

In order to know more about the buildlist, go to *to_zmat()*.

Parameters

- **buildlist** (*list*) –
- **start_row** (*int*) –

Returns

List of the dihedral between the first, second, third and fourth atom of every entry in the buildlist.

Return type list

move

Cartesian.**move**(*vector=[0, 0, 0], matrix=array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]), indices=None, copy=False*)

Move a Cartesian.

The Cartesian is first rotated, mirrored... by the matrix and afterwards translated by the vector

Parameters

- **vector** (*np.array*) – default is np.zeros(3)
- **matrix** (*np.array*) – default is np.identity(3)
- **indices** (*list*) – Indices to be moved.
- **copy** (*bool*) – Atoms are copied or translated to the new location.

Returns

Return type *Cartesian*

basistransform

Cartesian.**basistransform**(*new_basis, old_basis=array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]), rotate_only=True*)

Transforms the frame to a new basis.

This function transforms the cartesian coordinates from an old basis to a new one. Please note that old_basis and new_basis are supposed to have full Rank and consist of three linear independent vectors. If rotate_only is True, it is asserted, that both bases are orthonormal and right handed. Besides all involved matrices are transposed instead of inverted.

In some applications this may require the function `utilities.orthonormalize()` as a previous step.

Parameters

- **old_basis** (*np.array*) –
- **new_basis** (*np.array*) –
- **rotate_only** (*bool*) –

Returns The transformed molecule.

Return type *Cartesian*

add_data

Cartesian.**add_data**(*list_of_columns=None, inplace=False*)

Adds a column with the requested data.

If you want to see for example the mass, the colormap used in jmol and the block of the element, just use:

```
['mass', 'jmol_color', 'block']
```

The underlying `pd.DataFrame` can be accessed with `constants.elements`.

To see all available keys use `constants.elements.info()`.

The data comes from the module 'mendeleev <http://mendeleev.readthedocs.org/en/latest/>'_ written by Lukasz Mentel.

Please note that I added three columns to the mendeleev data:

```
['atomic_radius_cc', 'atomic_radius_gv', 'gv_color',
    'valency']
```

The `atomic_radius_cc` is used by default by this module for determining bond lengths.

The three others are taken from the MOLCAS grid viewer written by Valera Veryazov.

Parameters

- **`list_of_columns`** (*`str`*) – You can pass also just one value. E.g. `'mass'` is equivalent to `['mass']`. If `list_of_columns` is `None` all available data is returned.
- **`inplace`** (*`bool`*) –

Returns

Return type *Cartesian*

total_mass

`Cartesian.`**`total_mass`**`()`

Returns the total mass in g/mol.

Parameters **`None`** –

Returns

Return type float

barycenter

`Cartesian.`**`barycenter`**`()`

Returns the mass weighted average location.

Parameters **`None`** –

Returns

Return type np.array

inertia

`Cartesian.`**`inertia`**`()`

Calculates the inertia tensor and transforms along rotation axes.

This function calculates the inertia tensor and returns a 4-tuple.

Parameters **None** –

Returns

The returned dictionary has four possible keys:

`transformed_Cartesian`: A frame that is transformed to the basis spanned by the eigenvectors of the inertia tensor. The x-axis is the axis with the lowest inertia moment, the z-axis the one with the highest. Contains also a column for the mass

`diag_inertia_tensor`: A vector containing the sorted inertia moments after diagonalization.

`inertia_tensor`: The inertia tensor in the old basis.

`eigenvectors`: The eigenvectors of the inertia tensor in the old basis.

Return type dict

topologic_center

`Cartesian.`**`topologic_center`**`()`

Returns the average location.

Parameters **None** –

Returns

Return type np.array

cutcuboid

`Cartesian.`**`cutcuboid`**`(`*a=20, b=None, c=None, origin=[0, 0, 0], outside_sliced=True, preserve_bonds=False*`)`

Cuts a cuboid specified by edge and radius.

Parameters

- **a** (*float*) – Value of the a edge.
- **b** (*float*) – Value of the b edge. Takes value of a if None.
- **c** (*float*) – Value of the c edge. Takes value of a if None.
- **origin** (*list*) – Please note that you can also pass an integer. In this case it is interpreted as the index of the atom which is taken as origin.
- **outside_sliced** (*bool*) – Atoms outside/inside the sphere are cut out.
- **preserve_bonds** (*bool*) – Do not cut covalent bonds.

Returns

Return type *Cartesian*

cutsphere

`Cartesian.`**`cutsphere`**`(`*radius=15.0, origin=[0.0, 0.0, 0.0], outside_sliced=True, preserve_bonds=False*`)`

Cuts a sphere specified by origin and radius.

Parameters

- **radius** (*float*) –
- **origin** (*list*) – Please note that you can also pass an integer. In this case it is interpreted as the index of the atom which is taken as origin.
- **outside_sliced** (*bool*) – Atoms outside/inside the sphere are cut out.
- **preserve_bonds** (*bool*) – Do not cut covalent bonds.

Returns

Return type *Cartesian*

connected_to

Cartesian.**connected_to**(*index_of_atom*, *exclude=None*, *give_only_index=False*, *follow_bonds=None*)

Returns a Cartesian of atoms connected to the specified one.

Connected means that a path along covalent bonds exists.

Parameters

- **index_of_atom** (*int*) –
- **exclude** (*list*) – Indices in this list are omitted.
- **give_only_index** (*bool*) – If `True` a set of indices is returned. Otherwise a new Cartesian instance.
- **follow_bonds** (*int*) – This option determines how many branches the algorithm follows. If `None` it stops after reaching the end in every branch. If you have a single molecule this usually means, that the whole molecule is recovered.

Returns A set of indices or a new Cartesian instance.

distance_to

Cartesian.**distance_to**(*origin=[0, 0, 0]*, *indices_of_other_atoms=None*, *sort=False*)

Returns a Cartesian with a column for the distance from origin.

get_fragment

Cartesian.**get_fragment**(*list_of_indextuples*, *give_only_index=False*)

Get the indices of the atoms in a fragment.

The list_of_indextuples contains all bondings from the molecule to the fragment. `[(1,3), (2, 4)]` means for example that the fragment is connected over two bonds. The first bond is from atom 1 in the molecule to atom 3 in the fragment. The second bond is from atom 2 in the molecule to atom 4 in the fragment.

Parameters

- **list_of_indextuples** (*list*) –
- **give_only_index** (*bool*) – If `True` a set of indices is returned. Otherwise a new Cartesian instance.

Returns A set of indices or a new Cartesian instance.

fragmentate

Cartesian.**fragmentate**(*give_only_index=False*)
Get the indices of non bonded parts in the molecule.

Parameters **give_only_index** (*bool*) – If `True` a set of indices is returned. Otherwise a new Cartesian instance.

Returns A list of sets of indices or new Cartesian instances.

Return type list

make_similar

Cartesian.**make_similar**(*Cartesian2*, *follow_bonds=4*, *prealign=True*)
Similarizes two Cartesians.

Returns a reindexed copy of `Cartesian2` that minimizes the distance for each atom in the same chemical environemt from `self` to `Cartesian2`.

Read more about the definition of the chemical environment in *Cartesian.partition_chem_env()*

> **Warning:**
>
> **Please check the result with e.g.** *Cartesian.move_to()*
>
> **It is probably necessary to use the function** *Cartesian.change_numbering()*.

Parameters

- **Cartesian2** (Cartesian) –
- **max_follow_bonds** (*int*) –
- **prealign** (*bool*) – The method *Cartesian.align()* is applied before reindexing.

Returns

Aligned copy of `self` and aligned + reindexed version of `Cartesian2`

Return type tuple

align

Cartesian.**align**(*Cartesian2*, *ignore_hydrogens=False*)
Aligns two Cartesians.

Searches for the optimal rotation matrix that minimizes the RMSD (root mean squared deviation) of `self` to Cartesian2.

Returns a tuple of copies of `self` and `Cartesian2` where both are centered around their topologic center and `Cartesian2` is aligned along `self`.

Uses the Kabsch algorithm implemented with `utilities.kabsch()`

Parameters

- **Cartesian2** (Cartesian) –
- **ignore_hydrogens** (*bool*) – Hydrogens are ignored for the RMSD.

Returns

Return type tuple

change_numbering

Cartesian.**change_numbering**(*rename_dict*, *inplace=False*)

Returns the reindexed version of Cartesian.

Parameters **rename_dict** (*dict*) – A dictionary mapping integers on integers.

Returns A renamed copy according to the dictionary passed.

Return type *Cartesian*

move_to

Cartesian.**move_to**(*Cartesian2*, *step=5*, *extrapolate=(0, 0)*)

Returns list of Cartesians for the movement from self to Cartesian2.

Parameters

- **Cartesian2** (Cartesian) –
- **step** (*int*) –
- **extrapolate** (*tuple*) –

Returns

The list contains self as first and Cartesian2 as last element.

The number of intermediate frames is defined by step. Please note, that for this reason: len(list) = (step + 1). The numbers in extrapolate define how many frames are

appended to the left and right of the list continuing the movement.

Return type list

partition_chem_env

Cartesian.**partition_chem_env**(*follow_bonds=4*)

This function partitions the molecule into subsets of the same chemical environment.

A chemical environment is specified by the number of surrounding atoms of a certain kind around an atom with a certain atomic number represented by a tuple of a string and a frozenset of tuples.

The follow_bonds option determines how many branches the algorithm follows to determine the chemical environment.

Example: A carbon atom in ethane has bonds with three hydrogen (atomic

number 1) and one carbon atom (atomic number 6).

If `follow_bonds=1` these are the only atoms we are interested in and the chemical environment is:

```
('C', frozenset([('H', 3), ('C', 1)]))
```

If `follow_bonds=2` we follow every atom in the chemical enviromment of `follow_bonds=1` to their direct neighbours.

In the case of ethane this gives:

```
('C', frozenset([('H', 6), ('C', 1)]))
```

In the special case of ethane this is the whole molecule; in other cases you can apply this operation recursively and

stop after `follow_bonds` or after reaching the end of branches.

Parameters **`follow_bonds`** (*int*) –

Returns

The output will look like this:

```
{ (element_symbol, frozenset([tuples]))  :
    set([indices]) }

A dictionary mapping from a chemical environment to
    the set of indices of atoms in this environment.
```

Return type dict

Technical Methods

__init__(init)	How to initialize a Cartesian instance.
index	Returns the index.
columns	Returns the columns.
replace([to_replace, value, inplace, limit, ...])	Replace values given in 'to_replace' with 'value'.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
set_index(keys[, drop, append, inplace, ...])	Set the DataFrame index (row labels) using one or more existing columns.
append(other[, ignore_index, verify_integrity])	Append rows of *other* to the end of this frame, returning a new object.
insert(loc, column, value[, ...])	Insert column into DataFrame at specified location.
sort_values(by[, axis, ascending, inplace, ...])	Sort by the values along either axis
write(outputfile[, sort_index])	Writes the Cartesian into a file.
read_xyz(inputfile[, pythonic_index, get_bonds])	Reads a xyz file.
read_molden(inputfile[, pythonic_index, ...])	Reads a molden file.

__init__

`Cartesian.__init__`(*init*)

How to initialize a Cartesian instance.

Parameters **`frame`** (*pd.DataFrame*) – A Dataframe with at least the columns `['atom', 'x', 'y', 'z']`. Where `'atom'` is a string for the elementsymbol.

Returns A new cartesian instance.

Return type *Cartesian*

index

Cartesian.**index**
Returns the index.

Assigning a value to it changes the index.

columns

Cartesian.**columns**
Returns the columns.

Assigning a value to it changes the columns.

replace

Cartesian.**replace**(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*, *axis=None*)
Replace values given in 'to_replace' with 'value'.

The description is taken from the pandas project.

Parameters

- **to_replace** (*str, regex, list, dict, Series, numeric, or None*) –
  - str or regex:
    - str: string exactly matching *to_replace* will be replaced with *value*
    - regex: regexs matching *to_replace* will be replaced with *value*
  - list of str, regex, or numeric:
    - First, if *to_replace* and *value* are both lists, they **must** be the same length.
    - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
    - str and regex rules apply as above.
  - dict:
    - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
    - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - None:

* This means that the `regex` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or `Series`.

See the examples section for examples of each of these.

- **value** (*scalar, dict, list, str, regex, default None*) – Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.
- **inplace** (*boolean, default False*) – If True, in place. Note: this will modify any other views on this object (e.g. a column form a DataFrame). Returns the caller if this is True.
- **limit** (*int, default None*) – Maximum size gap to forward or backward fill
- **regex** (bool or same types as *to_replace*, default False) – Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Otherwise, *to_replace* must be `None` because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.
- **method** (*string, optional, {'pad', 'ffill', 'bfill'}*) – The method to use when for replacement, when `to_replace` is a `list`.

Returns filled

Return type *Cartesian*

Raises

- `AssertionError` – * If *regex* is not a `bool` and *to_replace* is not `None`.
- **TypeError – * If *to_replace* is a `dict` and *value* is not a `list`,** `dict`, `ndarray`, or `Series`

  – If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.

- **ValueError – * If *to_replace* and *value* are `list` s or `ndarray` s, but** they are not the same length.

Notes

•Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.

•Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.

•This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.

sort_index

Cartesian.**sort_index**(*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)

Sort object by labels (along an axis)

The description is taken from the pandas project.

Parameters

- **axis** (*index, columns to direct sorting*) –
- **level** (*int or level name or list of ints or list of level names*) – if not None, sort on values in specified index level(s)
- **ascending** (*boolean, default True*) – Sort ascending vs. descending
- **inplace** (*bool*) – if True, perform operation in-place
- **kind** ({*quicksort*, *mergesort*, *heapsort*}) – Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.
- **na_position** (*{'first', 'last'}*) – *first* puts NaNs at the beginning, *last* puts NaNs at the end
- **sort_remaining** (*bool*) – if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

Returns **sorted_obj**

Return type *Cartesian*

set_index

Cartesian.**set_index**(*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

The description is taken from the pandas project.

Parameters

- **keys** (*column label or list of column labels / arrays*) –
- **drop** (*boolean, default True*) – Delete columns to be used as the new index
- **append** (*boolean, default False*) – Whether to append columns to existing index
- **inplace** (*boolean, default False*) – Modify the DataFrame in place (do not create a new object)
- **verify_integrity** (*boolean, default False*) – Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

Examples

```
>>> indexed_df = df.set_index(['A', 'B'])
>>> indexed_df2 = df.set_index(['A', [0, 1, 2, 0, 1, 2]])
>>> indexed_df3 = df.set_index([[0, 1, 2, 0, 1, 2]])
```

Returns Cartesian

Return type *Cartesian*

append

`Cartesian.`**`append`**(*other*, *ignore_index=False*, *verify_integrity=False*)

Append rows of *other* to the end of this frame, returning a new object.

Columns not in this frame are added as new columns. The description is taken from the pandas project.

Parameters

- **other** (*DataFrame or Series/dict-like object, or list of these*) – The data to append.
- **ignore_index** (*boolean, default False*) – If True, do not use the index labels.
- **verify_integrity** (*boolean, default False*) – If True, raise ValueError on creating index with duplicates.

Returns appended

Return type *Cartesian*

Notes

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

See also:

pandas.concat() General function to concatenate DataFrame, Series or Panel objects

Examples

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

insert

Cartesian.**insert**(*loc*, *column*, *value*, *allow_duplicates=False*, *inplace=False*)

Insert column into DataFrame at specified location.

If *allow_duplicates* is False, raises Exception if column is already contained in the DataFrame.

Parameters

- **loc** (*int*) – Must have 0 <= loc <= len(columns)
- **column** (*object*) –
- **value** (*int, Series, or array-like*) –
- **inplace** (*bool*) –

sort_values

Cartesian.**sort_values**(*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)

Sort by the values along either axis

The description is taken from the pandas project.

Parameters

- **by** (*string name or list of names which refer to the axis items*) –
- **axis** (*index, columns to direct sorting*) –
- **ascending** (*bool or list of bool*) – Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.
- **inplace** (*bool*) – if True, perform operation in-place
- **kind** ({*quicksort*, *mergesort*, *heapsort*}) – Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.
- **na_position** (*{'first', 'last'}*) – *first* puts NaNs at the beginning, *last* puts NaNs at the end

Returns **sorted_obj**

Return type *Cartesian*

write

Cartesian.**write**(*outputfile*, *sort_index=True*)
Writes the Cartesian into a file.

If sort_index is true, the frame is sorted by the index before writing.

Note: Since it permamently writes a file, this function is strictly speaking **not sideeffect free**. The frame to be written is of course not changed.

Parameters

- **outputfile** (*str*) –
- **sort_index** (*bool*) –

Returns None

Return type None

read_xyz

classmethod Cartesian.**read_xyz**(*inputfile*, *pythonic_index=False*, *get_bonds=True*)
Reads a xyz file.

Parameters

- **inputfile** (*str*) –
- **pythonic_index** (*bool*) –

Returns

Return type *Cartesian*

read_molden

classmethod Cartesian.**read_molden**(*inputfile*, *pythonic_index=False*, *get_bonds=True*)
Reads a molden file.

Parameters

- **inputfile** (*str*) –
- **pythonic_index** (*bool*) –

Returns A list containing Cartesian is returned.

Return type list

Advanced methods

_divide_et_impera([maximum_edge_length, ...])	Returns a molecule split into cuboids.
_preserve_bonds(sliced_cartesian)	Is called after cutting geometric shapes.
_get_buildlist([fixed_buildlist])	Create a buildlist for a Zmatrix.

Continued on next page

Table 2.4 – continued from previous page

_clean_dihedral(buildlist_to_check)	Reindexes the dihedral defining atom if colinear.
_build_zmat(buildlist)	Creates the zmatrix from a buildlist.

_divide_et_impera

Cartesian.**_divide_et_impera**(*maximum_edge_length=25.0*, *difference_edge=6.0*)

Returns a molecule split into cuboids.

If your algorithm scales with $O(n^2)$. You can use this function as a preprocessing step to make it scaling with $O(n \log(n))$.

Parameters

- **maximum_edge_length** (*float*) – Maximum length of one edge
- **a cuboid. difference_edge** (*of*) –

Returns

A dictionary mapping from a 3 tuple of integers to a 2 tuple of sets. The 3 tuple gives the integer numbered coordinates of the cuboids. The first set contains the indices of atoms lying in the cube with a maximum edge length of `maximum_edge_length`. They are pairwise disjunct and are referred to as small cuboids. The second set contains the indices of atoms lying in the cube with `maximum_edge_length + difference_edge`. They are a bit larger than the small cuboids and overlap with `difference_edge / 2`.

Return type dict

_preserve_bonds

Cartesian.**_preserve_bonds**(*sliced_cartesian*)

Is called after cutting geometric shapes.

If you want to change the rules how bonds are preserved, when applying e.g. *Cartesian.cutsphere()* this is the function you have to modify.

It is recommended to inherit from the Cartesian class to tailor it for your project, instead of modifying the source code of ChemCoord.

Parameters **sliced_frame** (Cartesian) –

Returns

Return type *Cartesian*

_get_buildlist

Cartesian.**_get_buildlist**(*fixed_buildlist=None*)

Create a buildlist for a Zmatrix.

Parameters **fixed_buildlist** (*np.array*) – It is possible to provide the beginning of the buildlist. The rest is "figured" out automatically.

Returns buildlist

Return type np.array

_clean_dihedral

Cartesian.**_clean_dihedral**(*buildlist_to_check*)
Reindexes the dihedral defining atom if colinear.

Parameters **buildlist** (*np.array*) –

Returns modified_buildlist

Return type np.array

_build_zmat

Cartesian.**_build_zmat**(*buildlist*)
Creates the zmatrix from a buildlist.

Parameters **buildlist** (*np.array*) –

Returns A new instance of Zmat.

Return type *Zmat*

read_xyz

chemcoord.xyz_functions.**read_xyz**(*inputfile*, *pythonic_index=False*, *get_bonds=True*)
Reads a xyz file.

Note: This function calls in the background *Cartesian.read_xyz()*. If you inherited from *Cartesian* to tailor it for your project, you have to use this method as a constructor. Otherwise you can choose.

Parameters

- **inputfile** (*str*) –
- **pythonic_index** (*bool*) –

Returns

Return type *Cartesian*

read_molden

chemcoord.xyz_functions.**read_molden**(*inputfile*, *pythonic_index=False*, *get_bonds=True*)
Reads a molden file.

Parameters

- **inputfile** (*str*) –
- **pythonic_index** (*bool*) –

Returns A list containing Cartesian is returned.

Return type list

write_molden

chemcoord.xyz_functions.**write_molden**(*cartesian_list*, *outputfile*)

Writes a list of Cartesians into a molden file.

Note: Since it permamently writes a file, this function is strictly speaking **not sideeffect free**. The frame to be written is of course not changed.

Parameters

- **cartesian_list** (*list*) –
- **outputfile** (*str*) –

Returns

Return type None

Utilities for euclidean geometry

rotation_matrix(axis, angle)	Returns the rotation matrix.
give_angle(Vector1, Vector2)	Calculate the angle in degrees between two vectors.
orthormalize(basis)	Orthonormalizes a given basis.
normalize(vector)	Normalizes a vector
distance(vector1, vector2)	Calculates the distance between vector1 and vector2
kabsch(P, Q)	The optimal rotation matrix U is calculated and then used to rotate matrix P unto matrix Q so the minimum root-mean-square deviation (RMSD) can be calculated.

rotation_matrix

chemcoord.utilities.**rotation_matrix**(*axis*, *angle*)

Returns the rotation matrix.

This function returns a matrix for the counterclockwise rotation around the given axis. The Input angle is in radians.

Parameters

- **axis** (*vector*) –
- **angle** (*float*) –

Returns

Return type Rotation matrix (np.array)

give_angle

chemcoord.utilities.**give_angle**(*Vector1*, *Vector2*)

Calculate the angle in degrees between two vectors. The vectors do not have to be normalized.

orthormalize

chemcoord.utilities.**orthormalize**(*basis*)

Orthonormalizes a given basis.

This functions returns a right handed orthormalized basis. Since only the first two vectors in the basis are used, it does not matter if you give two or three vectors.

Right handed means, that:

- np.cross(e1, e2) = e3
- np.cross(e2, e3) = e1
- np.cross(e3, e1) = e2

Parameters **basis** (*np.array*) – An array of shape = (3,2) or (3,3)

Returns A right handed orthonormalized basis.

Return type new_basis (np.array)

normalize

chemcoord.utilities.**normalize**(*vector*)

Normalizes a vector

distance

chemcoord.utilities.**distance**(*vector1*, *vector2*)

Calculates the distance between vector1 and vector2

kabsch

chemcoord.utilities.**kabsch**(*P*, *Q*)

The optimal rotation matrix U is calculated and then used to rotate matrix P unto matrix Q so the minimum root-mean-square deviation (RMSD) can be calculated.

Using the Kabsch algorithm with two sets of paired point P and Q, centered around the center-of-mass. Each vector set is represented as an NxD matrix, where D is the the dimension of the space.

The algorithm works in three steps: - a translation of P and Q - the computation of a covariance matrix C - computation of the optimal rotation matrix U

http://en.wikipedia.org/wiki/Kabsch_algorithm

Parameters: P – (N, number of points)x(D, dimension) matrix Q – (N, number of points)x(D, dimension) matrix

Returns: U – Rotation matrix

Functions for internal coordinates

Zmat(init)	The main class for dealing with internal coordinates.

Zmat

class chemcoord.zmat_functions.**Zmat**(*init*)
The main class for dealing with internal coordinates.

Chemical Methods

build_list()	Return the buildlist which is necessary to create this Zmat
change_numbering([new_index, inplace])	Change numbering to a new index.
add_data([list_of_columns, inplace])	Adds a column with the requested data.
read_zmat(inputfile[, implicit_index])	Reads a zmat file.
to_xyz([SN_NeRF])	Transforms to cartesian space.
write(outputfile[, implicit_index])	Writes the zmatrix into a file.

build_list

Zmat.**build_list**()
Return the buildlist which is necessary to create this Zmat

Parameters **None** –

Returns Buildlist

Return type np.array

change_numbering

Zmat.**change_numbering**(*new_index=None*, *inplace=False*)
Change numbering to a new index.

Changes the numbering of index and all dependent numbering (bond_with...) to a new_index.

The user has to make sure that the new_index consists of distinct elements.

Parameters

- **new_index** (*list*) – If None the new_index is taken from 1 to the
- **of atoms.** (*number*) –

Returns Reindexed version of the zmatrix.

Return type *Zmat*

add_data

Zmat.**add_data**(*list_of_columns=None*, *inplace=False*)
Adds a column with the requested data.

If you want to see for example the mass, the colormap used in jmol and the block of the element, just use:

```
['mass', 'jmol_color', 'block']
```

The underlying `pd.DataFrame` can be accessed with `constants.elements`.

To see all available keys use `constants.elements.info()`.

The data comes from the module 'mendeleev <http://mendeleev.readthedocs.org/en/latest/>'_ written by Lukasz Mentel.

Please note that I added three columns to the mendeleev data:

```
['atomic_radius_cc', 'atomic_radius_gv', 'gv_color',
    'valency']
```

The `atomic_radius_cc` is used by default by this module for determining bond lengths.

The three others are taken from the MOLCAS grid viewer written by Valera Veryazov.

Parameters

- **`list_of_columns`** (*str*) – You can pass also just one value. E.g. `'mass'` is equivalent to `['mass']`. If `list_of_columns` is `None` all available data is returned.
- **`inplace`** (*bool*) –

Returns

Return type *Cartesian*

read_zmat

classmethod `Zmat.`**`read_zmat`**(*inputfile*, *implicit_index=True*)

Reads a zmat file.

Lines beginning with # are ignored.

Parameters

- **`inputfile`** (*str*) –
- **`implicit_index`** (*bool*) – If this option is true the first column
- **`to be the element symbols for the atoms.`** (*has*) – The row number is used to determine the index.

Returns

Return type *Zmat*

to_xyz

`Zmat.`**`to_xyz`**(*SN_NeRF=False*)

Transforms to cartesian space.

Parameters **`SN_NeRF`** (*bool*) – Use the **Self-Normalizing Natural Extension Reference Frame** algorithm[1]. In theory this means 30 % less floating point operations, but since this

[1] Parsons J, Holmes JB, Rojas JM, Tsai J, Strauss CE (2005). Practical conversion from torsion space to Cartesian space for in silico protein synthesis. J Comput Chem. 26(10) , 1063-8. doi:10.1002/jcc.20237

module is in python, floating point operations are not the rate determining step. Nevertheless it is a more elegant method than the "intuitive" conversion. Could make a difference in the future when certain functions will be implemented in `Fortran`.

Returns Reindexed version of the zmatrix.

Return type *Cartesian*

write

`Zmat.`**`write`**(*outputfile*, *implicit_index=True*)

Writes the zmatrix into a file.

Note: Since it permamently writes a file, this function is strictly speaking **not sideeffect free**. The frame to be written is of course not changed.

Parameters

- **outputfile** (*str*) –
- **implicit_index** (*bool*) – If implicit_index is set, the zmat indexing is changed to range(1, number_atoms+1). Besides the index is omitted while writing which means, that the index is given implicitly by the row number.

Returns None

Return type None

Technical Methods

__init__(init)	How to initialize a Zmat instance.
index	Returns the index.
columns	Returns the columns.
replace([to_replace, value, inplace, limit, ...])	Replace values given in 'to_replace' with 'value'.
sort_index([axis, level, ascending, ...])	Sort object by labels (along an axis)
set_index(keys[, drop, append, inplace, ...])	Set the DataFrame index (row labels) using one or more existing columns.
append(other[, ignore_index, verify_integrity])	Append rows of *other* to the end of this frame, returning a new object.
insert(loc, column, value[, ...])	Insert column into DataFrame at specified location.
sort_values(by[, axis, ascending, inplace, ...])	Sort by the values along either axis

__init__

`Zmat.`**`__init__`**(*init*)

How to initialize a Zmat instance.

Parameters **init** (*pd.DataFrame*) – A Dataframe with at least the columns `['atom', 'bond_with', 'bond', 'angle_with', 'angle', 'dihedral_with', 'dihedral']`. Where `'atom'` is a string for the elementsymbol.

Returns A new zmat instance.

Return type *Zmat*

index

`Zmat.`**`index`**
Returns the index.

Assigning a value to it changes the index.

columns

`Zmat.`**`columns`**
Returns the columns.

Assigning a value to it changes the columns.

replace

`Zmat.`**`replace`**(*to_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*, *axis=None*)
Replace values given in 'to_replace' with 'value'.

The description is taken from the pandas project.

Parameters

- **to_replace** (*str, regex, list, dict, Series, numeric, or None*) –
  - str or regex:
    - str: string exactly matching *to_replace* will be replaced with *value*
    - regex: regexs matching *to_replace* will be replaced with *value*
  - list of str, regex, or numeric:
    - First, if *to_replace* and *value* are both lists, they **must** be the same length.
    - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
    - str and regex rules apply as above.
  - dict:
    - Nested dictionaries, e.g., {'a': {'b': nan}}, are read as follows: look in column 'a' for the value 'b' and replace it with nan. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
    - Keys map to column names and values map to substitution values. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - None:

  - This means that the `regex` argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also `None` then this **must** be a nested dictionary or `Series`.

  See the examples section for examples of each of these.

- **value** (*scalar, dict, list, str, regex, default None*) – Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.
- **inplace** (*boolean, default False*) – If True, in place. Note: this will modify any other views on this object (e.g. a column form a DataFrame). Returns the caller if this is True.
- **limit** (*int, default None*) – Maximum size gap to forward or backward fill
- **regex** (bool or same types as *to_replace*, default False) – Whether to interpret *to_replace* and/or *value* as regular expressions. If this is `True` then *to_replace must* be a string. Otherwise, *to_replace* must be `None` because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.
- **method** (*string, optional, {'pad', 'ffill', 'bfill'}*) – The method to use when for replacement, when `to_replace` is a `list`.

Returns **filled**

Return type *Cartesian*

Raises

- `AssertionError` – * If *regex* is not a `bool` and *to_replace* is not `None`.
- **`TypeError` – * If *to_replace* is a `dict` and *value* is not a `list`,** `dict`, `ndarray`, or `Series`

  – If *to_replace* is `None` and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.

- **`ValueError` – * If *to_replace* and *value* are `list` s or `ndarray` s, but** they are not the same length.

Notes

•Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.

•Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.

•This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.

sort_index

Zmat.**sort_index**(*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *by=None*)

Sort object by labels (along an axis)

The description is taken from the pandas project.

Parameters

- **axis** (*index, columns to direct sorting*) –
- **level** (*int or level name or list of ints or list of level names*) – if not None, sort on values in specified index level(s)
- **ascending** (*boolean, default True*) – Sort ascending vs. descending
- **inplace** (*bool*) – if True, perform operation in-place
- **kind** ({*quicksort*, *mergesort*, *heapsort*}) – Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.
- **na_position** (*{'first', 'last'}*) – *first* puts NaNs at the beginning, *last* puts NaNs at the end
- **sort_remaining** (*bool*) – if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

Returns **sorted_obj**

Return type *Cartesian*

set_index

Zmat.**set_index**(*keys*, *drop=True*, *append=False*, *inplace=False*, *verify_integrity=False*)

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

The description is taken from the pandas project.

Parameters

- **keys** (*column label or list of column labels / arrays*) –
- **drop** (*boolean, default True*) – Delete columns to be used as the new index
- **append** (*boolean, default False*) – Whether to append columns to existing index
- **inplace** (*boolean, default False*) – Modify the DataFrame in place (do not create a new object)
- **verify_integrity** (*boolean, default False*) – Check the new index for duplicates. Otherwise defer the check until necessary. Setting to False will improve the performance of this method

Examples

```
>>> indexed_df = df.set_index(['A', 'B'])
>>> indexed_df2 = df.set_index(['A', [0, 1, 2, 0, 1, 2]])
>>> indexed_df3 = df.set_index([[0, 1, 2, 0, 1, 2]])
```

Returns Cartesian

Return type *Cartesian*

append

Zmat.**append**(*other*, *ignore_index=False*, *verify_integrity=False*)

Append rows of *other* to the end of this frame, returning a new object.

Columns not in this frame are added as new columns. The description is taken from the pandas project.

Parameters

- **other** (*DataFrame or Series/dict-like object, or list of these*) – The data to append.
- **ignore_index** (*boolean, default False*) – If True, do not use the index labels.
- **verify_integrity** (*boolean, default False*) – If True, raise ValueError on creating index with duplicates.

Returns appended

Return type *Cartesian*

Notes

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

See also:

pandas.concat() General function to concatenate DataFrame, Series or Panel objects

Examples

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With *ignore_index* set to True:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

insert

Zmat.**insert**(*loc*, *column*, *value*, *allow_duplicates=False*, *inplace=False*)

Insert column into DataFrame at specified location.

If *allow_duplicates* is False, raises Exception if column is already contained in the DataFrame.

Parameters

- **loc** (*int*) – Must have 0 <= loc <= len(columns)
- **column** (*object*) –
- **value** (*int, Series, or array-like*) –
- **inplace** (*bool*) –

sort_values

Zmat.**sort_values**(*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*)

Sort by the values along either axis

The description is taken from the pandas project.

Parameters

- **by** (*string name or list of names which refer to the axis items*) –
- **axis** (*index, columns to direct sorting*) –
- **ascending** (*bool or list of bool*) – Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.
- **inplace** (*bool*) – if True, perform operation in-place
- **kind** ({*quicksort*, *mergesort*, *heapsort*}) – Choice of sorting algorithm. See also ndarray.np.sort for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.
- **na_position** (*{'first', 'last'}*) – *first* puts NaNs at the beginning, *last* puts NaNs at the end

Returns **sorted_obj**

Return type *Cartesian*

References

Bugreports and development.

If you request new feautures or want to report bugs please open an issue on the github project page.

If you want to contribute in the development, feel free to contact me as well over the github project page.

Previous Contribution

- Main Work: Oskar Weser
- Python2 compatibility: Keld Lundgaard

License

GNU LESSER GENERAL PUBLIC LICENSE Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

2. Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

2. Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

4. Do one of the following:

0. Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

   1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

   e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Index

Symbols

A

B

C

D

F

G

I