moldf

Release 0.7.5

Ruibin Liu

Mar 13, 2024

CONTENTS

Important: This project is renamed from **pdbx2df**. Please go to its documentation for historical features.

MolDF reads structure files like PDB, PDBx/mmCIF, and MOL2 used in biology and chemistry into dictionaries of Pandas DataFrame s. With such a data structure, relatively loosely coupled data are separated into different `DataFrame` objects but are still linked to each other in the same Python dict. For the `DataFrame` objects, cheminformatians, bioinformaticans, and machine learning researchers should feel very comfortable to work with. It's easy to inspect, visualize, group, filter, manipulate, and export to other portable formats. Moreover, most machine learning frameworks support `DataFrame` s as inputs directly. This library makes it easy, intuitive, and fast to read those files into `DataFrame` s.

The PDBx/mmCIF format is the easiest to parse into a dict of `DataFrame` in that we can just use the provided category names as dict keys and the provided attribute names as column names in the `DataFrame`. Indeed, many mmCIF parsers just parse them into dicts.

The MOL2 format is also quite straightforward to parse because different category of data are well separated by definition. The category names and column names are also provided by the Tripos document. The minor difficulty comes from the fact that many categories have unstructured and/or optional data.

The PDB format is harder to parse compared to the other two. Except for a few categories like `SEQRES` which are self contained, many categories can be misleading if parsed into different `DataFrame` s. As such, I arbitrarily created some coarse-grained category names to group several categories together. As a result, the `_atom_site` category, mimicking the PDBx/mmCIF `_atom_site` category, is handy to work with for most use cases.

There are many other PDBx/PDB/MOL2 parsers, like Biopython PDBParser and OpenMM PDBFile, but most mainly parse the coordinates, and make the whole molecule into a python object of objects. It can be convenient in several use cases, but not so intuitive to visualize individual entries, select atoms, merge molecules, or export to other formats. And since they might need to build many python objects and not take advantage of the underlying structure of those structured data, they can be slow in large scale data processing. Moreover, those python objects are not so convenient to transfer to other platform or programming languages.

There are other python packages that can parse PDB files into `DataFrame` s. CPDB is the fastest by using Cython according to the author's benchmarks. But it can only parse PDB files not the other formats, and no writing back to PDB files. BioPandas can parse PDBx, PDB, and MOL2 files, but it is slow by the same benchmarks. According to my benchmark (coming soon!), **moldf** is also much faster than `BioPandas` and only slightly slower than `CPDB`.

Other than the lightweight and speedy parts, perhaps the provided *PDBDataFrame* `class`, which is a `Pandas DataFrame` subclass, is the most useful feature when we need to access common atom groups or select atoms finely. The `PDBDataFrame` class provides an easy to use `.` syntax to access common atom groups like `backbone`, `side_chain`, `water`, and `heavy_atoms`. It also implements atom selection language in a pythonic way that we can select by `atom_numbers`, `atom_names`, `chain_ids`, `residue_names`, `residue_numbers`, `x_coord`, `y_coord`, `z_coord`, `b_factor`, and others. We can even select by `distances` in a very flexible way. Check the documents for detailed information.

CHAPTER

ONE

CONTENTS

1.1 Install

Important: This project is renamed from **pdbx2df**. Please go to its documentation for historical features.

1.1.1 Install from PyPI

```
$ pip install moldf
```

The latest **stable** version matching your Python and OS versions will be installed. Python >= 3.7 versions are supported. Tests are done for Python 3.9, 3.10, and 3.11 versions on the latest Ubuntu, Windows, and Mac OS. Please report issues or problems in the GitHub issue tracker.

1.1.2 Install from source

```
$ pip install git+https://github.com/Ruibin-Liu/MolDF
```

The latest **development** version will be installed.

1.1.3 For contributors

```
$ git clone https://github.com/Ruibin-Liu/MolDF
$ cd MolDF
$ python -m venv .venv && source activate .venv/bin/activate  # recommended
$ pip install -r requirements_dev.txt   # for pre-commit hooks and pytest
$ pip install -r docs/requirements.txt  # for docs
$ pip install -e .                      # for the MolDF package itself
```

Python >= 3.10 is used for development. We use isort, black, mypy, and flake8 for coding style guide. and they are hooked into pre-commit, which means using `pre-commit` command after staging (`git add`) the commits. And we use pytest for automatic tests.

For documentation, we follow the Google Style Python Docstrings and use sphinx to generate these docs.

1.2 Quick Start

Important: This project is renamed from **pdbx2df**. Please go to its documentation for historical features.

This quick start tutorial will guide you to use the **MolDF** functions and classes to read, manipulate, and write PDBx, PDB, and MOL2 files. You will learn by going through some basic but useful examples.

1.2.1 1. Read a PDB file

To read a PDB file, you can use the `moldf.read_pdb` function:

One of the `pdb_file` and `pdb_id` parameters should be given. Otherwise, `moldf.read_pdb` will raise an exception. If `pdb_file` is given, `pdb_id` is ignored.

For example:

```
>>> from moldf import read_pdb
>>> pdb = read_pdb(pdb_id='1vii')
>>> pdb['_atom_site'].columns
Index(['record_name', 'atom_number', 'atom_name', 'alt_loc', 'residue_name',
    'chain_id', 'residue_number', 'insertion', 'x_coord', 'y_coord',
    'z_coord', 'occupancy', 'b_factor', 'segment_id', 'element_symbol',
    'charge'],
   dtype='object')
>>> pdb.keys()
dict_keys(['_atom_site'])
```

By default, a `1vii.pdb` file is downloaded to the `./PDB_files` directory from RCSB 1VII.

If you have a local PDB file `test.pdb` under your current directory. You can read it as:

```
>>> pdb = read_pdb(pdb_file='test.pdb')
```

1.2.2 2. Select atoms using PDBDataFrame

To select rows in the `_atom_site` `DataFrame`, you can of course just use standard filter operations in `Pandas`. For example, you might have thought of something as below:

```
>>> pdb_df = pdb['_atom_site']
>>> ca_atoms = pdb_df[(pdb_df.atom_name.str.strip().isin(['CA'])) & (pdb_df.element_
↪symbol.str.strip().isin(['C']))]
```

But it is obvious to see the cumbersomeness and error-proneness if you need more complex selections. And it should be noted that the condition as to `element_symbol` is necessary because only using the condition as to `atom_name` could give out a `DataFrame` containing calcium atoms if there is any, because calcium atoms also have `atom_name` as `CA`.

Instead, you could use the selection language implemented in the *PDBDataFrame* `class`. For the same selection, it is simply:

```
>>> from moldf import PDBDataFrame
>>> pdb_df = PDBDataFrame(pdb_df)  # Just adding a few methods to the standard Pandas␣
↪DataFrame
>>> ca_atoms = pdb_df.ca_atoms
```

Or equally,

```
>>> ca_atoms = pdb_df.atom_names(['CA'])  # If you want calcium atoms, you have to add 'CA
↪' to the 'names_2c' keyword herej.
```

The first method uses the build-in `ca_atoms` python `property` so that you can use the familiar `.` syntax. Check the *PDBDataFrame* `class` documentation for other convenient properties.

The build-in properties are handy but not so flexible nor powerful. The second method is much more flexible in that you can select atoms providing a list of `atom_name` s to the `atom_names` method and optionally specifying metal atoms in the `names_2c` keyword. You can also invert the selection by:

```
>>> not_ca_atoms = pdb_df.atom_names(['CA'], invert=True)
```

All columns in the `PDBDataFrame` are supported for such an atom selection language, simply by use the plural forms of the column names as methods for selecting the corresponding columns. Another example:

```
>>> x_coord_larger_than_zero = pdb_df.x_coords(0, relation='>') # all atoms whose 'x_coord
↪' > 0
```

Here it shows you can use the `relation` keywords to control the relationship between the target variable and the reference value if it is a numerical column like *x_coord* or *atom_number* etc.

Selection based on `distance` can be done easily through the `distances` method, e.g.:

```
>>> close_to_origin = pdb_df.distances([0.0, 0.0, 0.0], cut_off=10.0, relation='<=')
```

which gives you all atoms within 10.0 Å of the point [0.0, 0.0, 0.0].

Even more, you can chain and make arbitrary combinations of them to get very complex selections.

```
>>> complex_selection = pdb_df.chain_ids(['A']).backbone.atom_names(['N']).residue_
↪names(['Lys', 'His', 'Arg']).distances([0.0, 0.0, 0.0], cut_off=10.0, relation='<=')
```

which gives you all the nitrogen atoms in the backbone of Lys, His, and Arg residues of 1vii's chain A that are within 10.0 Å of the origin point. For such a selection, using vanilla `Pandas` filter language can be very time-consuming, error-prone, and thus frustrating. Fortunately, `moldf` can help you save a lot of effort.

1.2.3 3. Write DataFrames back to a PDB file

Writing back to a PDB file is simply:

```
>>> from moldf import write_pdb
>>> write_pdb(pdb, 'output.pdb')
```

Remember to use the `pdb` object, not the `pdb_df`, or it will error out. An `output.pdb` file is saved to your working directory.

If you want to save the selected atoms (e.g. the `complex_selection` example above) only, you can:

```
>>> pdb_out = {'_atom_site': complex_selection}
>>> write_pdb(pdb_out, 'complex_selection.pdb')
```

and the `complex_selection.pdb` has all and only the atoms in the `complex_selection`.

1.2.4 4. Read a mmCIF/PDBx file

To read a PDBx file, you can use the `moldf.read_pdbx` function:

One of the `pdbx_file` and `pdb_id` parameters should be given. Otherwise, `moldf.read_pdbx` will raise an exception. If `pdbx_file` is given, `pdb_id` is ignored.

For example:

```
>>> from moldf import read_pdbx
>>> pdbx = read_pdbx(pdb_id='1vii')
>>> pdbx['_atom_site'].columns
Index(['group_PDB', 'id', 'type_symbol', 'label_atom_id', 'label_alt_id',
       'label_comp_id', 'label_asym_id', 'label_entity_id', 'label_seq_id',
       'pdbx_PDB_ins_code', 'Cartn_x', 'Cartn_y', 'Cartn_z', 'occupancy',
       'B_iso_or_equiv', 'pdbx_formal_charge', 'auth_seq_id', 'auth_comp_id',
       'auth_asym_id', 'auth_atom_id', 'pdbx_PDB_model_num'],
      dtype='object')
>>> pdbx.keys()
dict_keys(['_entry', '_audit_conform', '_database_2', '_pdbx_database_status', '_audit_
↪author', '_citation',
           '_citation_author', '_cell', '_symmetry', '_entity', '_entity_name_com', '_
↪entity_poly',
           '_entity_poly_seq', '_entity_src_gen', '_struct_ref', '_struct_ref_seq', '_
↪chem_comp', '_pdbx_nmr_exptl',
           '_pdbx_nmr_exptl_sample_conditions', '_pdbx_nmr_spectrometer', '_pdbx_nmr_
↪refine', '_pdbx_nmr_ensemble',
           '_pdbx_nmr_software', '_exptl', '_struct', '_struct_keywords', '_struct_asym',
↪ '_struct_biol',
           '_struct_conf', '_struct_conf_type', '_struct_site', '_struct_site_gen', '_
↪database_PDB_matrix',
           '_atom_sites', '_atom_type', '_atom_site', '_pdbx_poly_seq_scheme', '_pdbx_
↪struct_assembly',
           '_pdbx_struct_assembly_gen', '_pdbx_struct_oper_list', '_pdbx_audit_revision_
↪history',
           '_pdbx_audit_revision_details', '_pdbx_audit_revision_group', '_pdbx_audit_
↪revision_category',
           '_pdbx_audit_revision_item', '_software', '_pdbx_validate_close_contact', '_
↪pdbx_validate_torsion'])
```

By default, a `1vii.cif` file is downloaded to the `./PDBx_files` from RCSB 1VII.

Similarly to the `read_pdb` case, you can read a local `test.cif` file as well:

```
>>> pdbx = read_pdbx(pdbx_file='test.cif')
```

1.2.5 5. Write DataFrames back to a PDBx file

Similar to the above *writing back to PDB file* example, you can write back to a PDBx file like:

```
>>> from moldf import write_pdbx
>>> write_pdbx(pdbx, 'output.cif')
```

Here the `pdbx` object is the one generated in the *PDBx reading* example. An `output.cif` file is saved to your working directory.

Perhaps a useful case is that you want to keep only some categories but removing the other redundant ones:

```
>>> to_keep = ['_atom_site', '_entity_poly']
>>> pdbx_keep = {k: v for k, v in pdbx.items() if k in keep}
>>> write_pdbx(pdbx_keep, 'to_keep.cif')
```

And thus only the `_atom_site` and `_entity_poly` categories are saved to your working directory as `to_keep.cif`.

1.2.6 6. Read a MOL2 file

To read a Tripos MOL2 file, you can use the `moldf.read_mol2` function:

Let's download an example MOL2 file from LigandBox first. The example ligand is D00217 or Tylenol.

You can read it as:

```
>>> from moldf import read_mol2
>>> mol2 = read_mol2(mol2_file='./D00217-01.mol2')
>>> mol2['ATOM'].columns
Index(['atom_id', 'atom_name', 'x', 'y', 'z', 'atom_type', 'subst_id',
       'subst_name', 'charge'],
      dtype='object')
>>> mol2.keys()
dict_keys(['ATOM', 'MOLECULE', 'BOND'])
```

1.2.7 7. Write a MOL2 file

You might need to do some manipulation to a *mol2* file and then write back. One example is ParmEd needs the input *mol2* file grouping the atoms in a same residue (can be accessed by the *subst_name* column) together if there are many, so that it can build the correct topology of the system. One solution is to read the *mol2* file, group the residues by *subst_name*, and then write back.

```
>>> from moldf import read_mol2, write_mol2
>>> mol2 = read_mol2(mol2_file='glutathione.mol2')
>>> mol2['ATOM'].sort_values(by=['subst_name', 'atom_id'], inplace=True)
>>> write_mol2(mol2, file_name='glutathione_moldf.mol2')
```

In the *glutathione_moldf.mol2* file, the atoms belonging to the same residue are together.

1.2.8 8. RMSD, radius of gyration, and distance matrix

In moldf, it is very intuitive and convenient to do atom selection as shown above, thanks to the *PDBDataFrame* class. In fact, the class has more than that. We can use it to calculate RMSD, radius of gyration, and distance matrix easily.

```
>>> from moldf import read_pdb, PDBDataFrame
>>> pdb = read_pdb(pdb_id='1g03')
>>> df = pdb['_atom_site']
>>> df = PDBDataFrame(df)
>>> all_rmsd = df.rmsd()  # all_rmsd contains all RMSDs between NMR models 2-20 and 1
>>> model_1 = df.nmr_models(1)
>>> m1_rgyr = model_1.radius_of_gyration  # model 1's radius of gyration
>>> m1_dis_mat = model_1.distance_matrix  # model 1's distance matrix in condensed form
```

Check the API reference for *PDBDataFrame* for more options in the *rmsd* method. For example, *align* can be set as *False* so that the calculated RMSD values are based on the original coordinates.

For *distance_matrix*, we can set *df.use_squared_distance=False* and *df.use_square_form=True* so that the returned distance matrix is a truly squared matrix whose elements are distances, not distance squared values. The default settings can save computation time and RAM usage, recommended for large scale processing where squared distances are not required.

1.3 API References

Important: This project is renamed from **pdbx2df**. Please go to its documentation for historical features.

1.3.1 PDBx Reader

PDBx/mmCIF format reading.

Reads a PDBx file into a dict of `Pandas DataFrame` s. The dict keys are read from the mmCIF category names automatically. For each category, the column names of the corresponding `DataFrame` are read form the category attributes automatically.

For example:

```
_audit_conform.dict_name       mmcif_pdbx.dic
_audit_conform.dict_version    5.355
_audit_conform.dict_location   http://mmcif.pdb.org/dictionaries/ascii/mmcif_pdbx.dic
```

In this category, the category name is `_audit_conform`, so the returned dict key is `_audit_conform`. The category attributes are `dict_name`, `dict_version`, and `dict_location`, so the returned dict value as a `DataFrame` has the exact column names.

`moldf.read_pdbx.AF2_MODEL = 4`

For AlphaFold structures, the version to use.

`moldf.read_pdbx.read_pdbx`(*pdbx_file: str | PathLike[str] | None = None*, *pdb_id: str | None = None*, *save_pdbx_file: bool = True*, *pdbx_file_dir: str | PathLike | None = None*, *category_names: list | None = None*, *convert_dtype: bool = False*) → dict[str, DataFrame]

Reads a `.cif` file's categories into a `dict` of `Pandas DataFrame` s.

Parameters

- **pdb_id** (*optional*) – PDB/Uniprot ID. Required if `pdbx_file` is `None`. Defaults to **None**.
- **pdbx_file** (*optional*) – file name for a PDBx/mmCIF file. Used over *pdb_id*. Defaults to **None**.
- **category_names** (*optional*) – a list of categories in the mmCIF file format. If `None`, `all` is used and all categories will be processed. Defaults to **None**.
- **save_pdbx_file** (*optional*) – whether to save the fetched PDBx file from RCSB to `pdbx_file_dir`. Defaults to **False**.
- **pdbx_file_dir** (*optional*) – directory to save fetched PDBx files. If `None` but `save_pdbx_file` is `True`, './PDBx_files' is used. Defaults to **None**.
- **convert_dtype** (*optional*) – whether to convert the data types according to the RCSB mmcif specifications. Defaults to **False**.

Returns

A dict of `Pandas DataFrame` s corresponding to required categories.

Raises

- `ValueError` – if none of `pdb_id` or `pdbx_file` is provided, or if `pdb_id` is given but cannot the PDB file cannot be downloaded from RCSB, or the PDB file is corrupted like no end-line symbol, or some content is irregular.
- `FileNotFoundError` – if `pdbx_file` cannot be found.

1.3.2 PDB Reader

PDB format reading.

Reads a PDB file, including Chimera compatible ones, into a dict of `Pandas DataFrame` s.

Atom coordinates and sequences are currently supported by the following category names:

`_atom_site`: `ATOM`, `HETATM`, `TER`, `NUMMDL`, `MODEL`, and `ENDMDL` lines.

`_seq_res`: `SEQRES` lines.

`moldf.read_pdb.IMPLEMENTED_PDB_CATS = ['_atom_site', '_seq_res', '_chem_comp']`

PDB categories that are currently implemented.

`moldf.read_pdb.ATOM_SITE = ('ATOM', 'HETATM', 'TER')`

`_atom_site` primary lines.

`moldf.read_pdb.NMR_MDL = ('NUMMDL', 'MODEL', 'ENDMDL')`

`_atom_site` additional lines.

`moldf.read_pdb.CHEM_COMP = ('HET ', 'HETNAM', 'FORMUL', 'HETSYN')`

`_chem_comp` lines.

`moldf.read_pdb.AF2_MODEL = 4`

For AlphaFold structures, the version to use.

`moldf.read_pdb.read_pdb`(*pdb_file: str | PathLike[str] | None = None*, *pdb_id: str | None = None*, *category_names: list | None = None*, *save_pdb_file: bool = True*, *pdb_file_dir: str | PathLike | None = None*, *allow_chimera: bool = True*, *need_ter_lines: bool = True*) → dict[str, DataFrame]

Reads a `.pdb` file's categories into a `dict` of `Pandas DataFrame` s.

Parameters

- **pdb_id** (*optional*) – PDB/Uniprot ID. Required if `pdb_file` is `None`. Defaults to **None**.
- **pdb_file** (*optional*) – file name for a PDB file. Used over *pdb_id*. Defaults to **None**.
- **category_names** (*optional*) – a list of categories similar to the mmCIF format. If `None`, `_atom_site` is used. To be consistent with the PDBx format, the following category names are used to refer to block(s) in a PDB file and only they are supported:

  1. `_atom_site`: `ATOM`, `HETATM`, and `TER` lines and possible `NUMMDL`, `MODEL`, and `ENDMDL` lines.
  2. `_seq_res`: `SEQRES` lines.
  3. **`_chem_comp`: all lines in `CHEM_COMP` above. However, only** compounds included in the ``HET `` lines are extracted which usually means the water molecule is not included.

  Defaults to **None**.
- **save_pdb_file** (*optional*) – whether to save the fetched PDB file from RCSB to `pdb_file_dir`. Defaults to **True**.
- **pdb_file_dir** (*optional*) – directory to save fetched PDB files. If `None` but `save_pdb_file` is `True`, './PDB_files' is used. Defaults to **None**.
- **allow_chimera** (*optional*) – whether to allow Chimera-formatted PDB files. Defaults to **True**.
- **need_ter_lines** (*optional*) – whether to read the `TER` lines into the `DataFrame`. Defaults to **True**.

Returns

A dict of `Pandas DataFrame` s corresponding to required categories.

Raises

- `ValueError` – if none of `pdb_id` or `pdbx_file` is provided, or if `pdb_id` is given but cannot the PDB file cannot be downloaded from RCSB,
- `NotImplementedError` – if *category_names* not a subset of allowed names.
- `FileNotFoundError` – if `pdbx_file` cannot be found.

`moldf.read_pdb._split_atom_line`(*line: str*, *nmr_model: int = -1*, *allow_chimera: bool = True*, *is_ter_line: bool = False*) → tuple

Internal function to parse a single line belonging to `ATOM`, `HETATM`, or `TER` lines.

Parameters

- **line** (*required*) – A `ATOM`, `HETATM`, or `TER` line.
- **nmr_model** (*optional*) – the NMR model number for the line; `-1` means not an NMR model. Defaults to **-1**.
- **allow_chimera** (*optional*) – try to parse as a Chimera-formatted PDB file. Defaults to **True**
- **is_ter_line** (*optional*) – whether the line starts with `TER`. Defaults to **False**.

Returns

parsed values as a tuple.

1.3.3 MOL2 Reader

Mol2 format reading.

Read a Tripos `.mol2` file into a dictionary of `pandas DataFrames`. Different categories like 'ATOM' and 'BOND' are read into different DataFrame objects.

`moldf.read_mol2.IMPLEMENTED_MOL2_CATS = ['ATOM', 'MOLECULE', 'BOND', 'HEADER']`

MOL2 categories that are currently implemented.

`moldf.read_mol2.ATOM_COL_NAMES = ('atom_id', 'atom_name', 'x', 'y', 'z', 'atom_type', 'subst_id', 'subst_name', 'charge', 'status_bit')`

MOL2 `ATOM` column names.

`moldf.read_mol2.BOND_COL_NAMES = ('bond_id', 'origin_atom_id', 'target_atom_id', 'bond_type', 'status_bit')`

MOL2 `BOND` column names.

`moldf.read_mol2.read_mol2`(*mol2_file: str | PathLike*, *category_names: list | None = None*) → dict[str, DataFrame]

Reads a `.mol2` file's categories into a `dict` of `Pandas DataFrame` s.

Parameters

- **mol2_file** (*required*) – file name for a PDB file.
- **category_names** (*optional*) – a list of categories as to the `.mol2` file format. If `None`, `['ATOM', 'MOLECULE', 'BOND', 'HEADER']` is used. Defaults to **None**.

Returns

A dict of `category_name` as keys(s) and `pd.DataFrame` as values.

Raises

`NotImplementedError` – if `category_names` not a subset of `['ATOM', 'MOLECULE', 'BOND', 'HEADER']`

`moldf.read_mol2._get_header_df`(*header_lines: list[tuple]*) → DataFrame

Turns the `HEADER` lines into a `Pandas DataFrame`. The `HEADER` lines are those starting with one or multiple `#` symbols.

Parameters

header_lines (*required*) – a list of tuples corresponding to each line's content. Tuples are generate by splitting the lines by `:`.

Returns

`Pandas DataFrame` of The `HEADER` category

`moldf.read_mol2._get_molecule_df`(*molecule_lines: list[tuple]*) → DataFrame

Turns the `MOLECULE` lines into a `Pandas DataFrame`.

Parameters

molecule_lines (*required*) – a list of tuples corresponding to each line's content.

Returns

`Pandas DataFrame` of The `MOLECULE` category

`moldf.read_mol2._set_atom_df_dtypes`(*data_df: DataFrame*) → DataFrame

Sets the data types for the `ATOM` category.

Parameters

data_df (*required*) – original `Pandas DataFrame` for the `ATOM` category with all strings.

Returns
Pandas DataFrame of The ATOM category

moldf.read_mol2.**_set_bond_df_dtypes**(*data_df: DataFrame*) → DataFrame

Sets the data types for the BOND category

Parameters
data_df (*required*) – original Pandas DataFrame for the BOND category with all strings.

Returns
Pandas DataFrame of The BOND category

1.3.4 JCSV Reader

JCSV format reading.

Reads a JCSV file into a dict of Pandas DataFrame s. It is not limited to any molecular format.

moldf.read_jcsv.**read_jcsv**(*jcsv_file: str | PathLike*, *category_names: list | None = None*) → dict[str, DataFrame]

Reads a JCSV file by name.

Currently no molecular file repository has JCSV files so we can only read from a file name/path.

Parameters

- **jcsv_file** (*required*) – JCSV file name/path.
- **category_names** (*optional*) – a list of category names. If None, all categories are read. Defaults to **None**.

Returns
a dict of Pandas DataFrames for each category.

Raises

- **TypeError** – if category_names is not a list of strings.
- **ValueError** – if any of the category_names has double quotes or if the number of items in any line does not match the number of column names in the same category.

moldf.read_jcsv.**_read_jcsv_by_line**(*jcsv_file: str | PathLike*, *category_names: list | None = None*) → dict[str, DataFrame]

Reads JCSV file line by line when the file has no meta data to select blocks.

Parameters

- **jcsv_file** (*required*) – JCSV file name/path.
- **category_names** (*optional*) – a list of category names. If None, all categories are read. Defaults to **None**. It is passed by the read_jcsv caller, so it is not sanitized here.

Returns
a dict of Pandas DataFrames for each category.

Raises
ValueError – if the number of items in any line does not match the number of column names in the same category.

moldf.read_jcsv._count_n_lines(*file_name: str | PathLike*)

Gets the number of lines in a file. From https://stackoverflow.com/a/68385697/10094189

Parameters

file_name (*required*) – file name or path.

1.3.5 PDBDataFrame Class

PDBDataFrame as a subclass of Pandas DataFrame.

Several features are added to make PDB data more accessible and selectable:

1. Properties like sequences, heavy_atoms, backbone, and water are directly accessed by . operation.

2. Atom selection by using methods whose names are just the column names plus s (plural form). For example, selecting atoms by names is simply df.atom_names([names]) where atom_name is the column name and atom_names is the selection function. Each selection returns a PDBDataFrame object as well, which means we can chain selections one by one like df.atom_names([names]).residue_numbers([numbers]).

3. Distance matrix as a @property and @classmethod.

moldf.pdb_dataframe.**RESIDUE_CODES** = {'ALA': 'A', 'ARG': 'R', 'ASH': 'D', 'ASN': 'N', 'ASP': 'D', 'CYM': 'C', 'CYS': 'C', 'CYX': 'C', 'GLH': 'E', 'GLN': 'Q', 'GLU': 'E', 'GLY': 'G', 'HID': 'H', 'HIE': 'H', 'HIP': 'H', 'HIS': 'H', 'ILE': 'I', 'LEU': 'L', 'LYN': 'K', 'LYS': 'K', 'MET': 'M', 'PHE': 'F', 'PRO': 'P', 'PYL': 'O', 'SEC': 'U', 'SER': 'S', 'THR': 'T', 'TRP': 'W', 'TYR': 'Y', 'VAL': 'V'}

dict[str, str], turn 3-, 2-, and 1-letter residue codes to 1-letter codes.

moldf.pdb_dataframe.**PDBX_COLS** = {'alt_loc': 'label_alt_id', 'atom_name': 'label_atom_id', 'atom_number': 'id', 'b_factor': 'B_iso_or_equiv', 'chain_id': 'label_asym_id', 'charge': 'pdbx_formal_charge', 'element_symbol': 'type_symbol', 'insertion': 'pdbx_PDB_ins_code', 'nmr_model': 'pdbx_PDB_model_num', 'occupancy': 'occupancy', 'record_name': 'group_PDB', 'residue_name': 'label_comp_id', 'residue_number': 'label_seq_id', 'segment_id': 'label_entity_id', 'x_coord': 'Cartn_x', 'y_coord': 'Cartn_y', 'z_coord': 'Cartn_z'}

dict[str, str], PDB and mmCIF column name dictionary.

class moldf.pdb_dataframe.**PDBDataFrame**(**args*, *pdb_format: str | None = None*, *use_squared_distance: bool = True*, *use_square_form: bool = False*, ***kwargs*)

Bases: DataFrame

Pandas DataFrame with extended attributes and methods for PDB data.

It enables Pythonic atom selection methods and convenient . accessing to common PDB structure properties.

Parameters

- ***args** – all pd.DataFrame positional arguments. For example, the _atom_site dataframe returned by reading a PDB file.
- **pdb_format** (*optional*) – PDB format in the underlying provided data. If None, PDB is assumed. Defaults to **None**.
- **use_squared_distance** (*optional*) – whether to use squared distance when calculating distance matrix. Defaults to **True**.
- **use_square_form** (*optional*) – whether to use a square matrix for the distance matrix. Defaults to **False**.
- ****kwargs** – all pd.DataFrame acceptable keyword arguments.

Returns
A `PDBDataFrame` instance.

Example

```
>>> from moldf import read_pdb, PDBDataFrame
>>> pdb = read_pdb(pdb_id='1vii')
>>> pdb_df = pdb['_atom_site']
>>> pdb_df = PDBDataFrame(pdb_df)
```

Warning: This subclass uses a custom `__hash__` function for caching some calculations. And thus a custom `__eq__` function is also implemented. For other typical `DataFrame` operations, use those `.all()`, `.any()`, `.bool()` functions to do comparison.

_metadata: list[str] = ['_use_squared_distance', '_use_square_form', '_is_chimera', '_RESIDUE_CODES', '_ELEMENT_MASSES', '_pdb_format']

property _constructor

Used when a manipulation result has the same dimensions as the original.

_pdbx_to_pdb(*keep_original: bool = False*)

Converts PDBx '_atom_site' DataFrame to PDB format.

Parameters
keep_original (*optional*) – whether to keep the original columns in the PDBx '_atom_site' DataFrame. Defaults to **False**.

property pdb_format: str

The format of the current PDBDataFrame.

property RESIDUE_CODES: dict[str, str]

A dict of `residue_name` as keys and `residue_code` as values, where `residue_code` is a 1-character code used in sequences. **Settable**.

property ELEMENT_MASSES: dict[str, float]

A dict of `element_symbol` as keys and `element_mass` as values, where `element_mass` is taken from NIST. **Settable**.

property is_chimera: bool

Whether the original read-in PDB was Chimera compatible format. The main effect is the `residue_name` str width is 4 in the Chimera compatible format instead of 3 as in the standard PDB format. **Not settable**.

property hash_random_state: int

The `random_state` used in the `__hash__` function. **Settable**.

property use_squared_distance: bool

Whether R or R^2 is used in distance matrix calculations. Using R^2 saves computation time. **Settable**.

property use_square_form: bool

Whether the distance matrix will be in a square form. Using square form consumes less memory. **Settable**.

property atoms: Self

Gets atoms in the `ATOM` and `HETATM` entries. In other words, removing 'TER' lines.

Returns
sub `PDBDataFrame`.

property coords: `Self`

Gets the `x_coord`, `y_coord`, and `z_coord` columns only. Use `pdb_df.coords.values` to get the underlying Numpy array of the coordinates.

property element_set: `set`

Gets the set of element symbols.

property bonds: `dict`

Gets the list of bonds. Each bond is represented as a pair of
`atom_number` values.

Raises
`ValueError` – if the list of `atom_number` is not a set.

get_bonds_by_distance(*single_radii_set: str | None = None*, *need_non_covalent: bool = False*, *non_covalent_cutoff: float = 4.5*) → dict

Gets all the bonds purely by covalent radii constraints.

Parameters

- **single_radii_set** (*optional*) – radii sets to use. If `None`, `single_C` is used as to Cordero (PMID 18478144). Another option is `single_PA` which refers to Pyykkö's studies (PMID 19058281;19856342;15832398, and doi:10.1103/PhysRevB.85.024115). Defaults to **None**.
- **need_non_covalent** (*optional*) – whether non-covalent 'bonding' should be included. Defaults to **False**.
- **non_covalent_cutoff** (*optional*) – distance cutoff for non-covalent 'bonding'.

Raises
`ValueError` – if the list of `atom_number` is not unique or if the `single_radii_set` is not valid.

Returns

a dictionary of bonds with tuple of `atom_number` as keys and bond types
as values.

get_bonds_by_template() → dict

Gets covalent bonds based on residue/ligand templates.

property sequences: `dict[str, str]`

Gets the sequences for each chain as a dict of `chain_id` as key(s) and `chain_sequence` as value(s).

property chain_list: `list`

Gets all chain ids as a list.

property residue_list: `list[tuple]`

Gets all residues as a list of tuple (`chain_id`, `residue_name`, `residue_number`).

property backbone: `Self`

Gets backbone or N+CA+C+O atoms.

Returns
sub `PDBDataFrame`

property side_chain: Self

Gets side chain or NOT N+CA+C+O atoms.

Returns
sub PDBDataFrame.

property ca_atoms: Self

Gets the alpha carbon (CA) atoms.

Returns
sub PDBDataFrame.

property heavy_atoms: Self

Gets the heavy or NOT hydrogen atoms.

Returns
sub PDBDataFrame.

property hetero_atoms: Self

Gets the hetero (HETATM) atoms.

Returns
sub PDBDataFrame.

property residues: Self

Gets the residue (ATOM) atoms.

Returns
sub PDBDataFrame.

property water: Self

Gets all water atoms.

Returns
sub PDBDataFrame.

property n_atoms: int

Gets the number of atoms.

property n_residues: int

Gets the number of residues.

property n_chains: int

Gets the number of chains.

property n_segments: int

Gets the number of segments.

property n_models: int

Gets the number of models.

property center_of_geometry: ndarray

Gets the center of geometry as a (3,) np.ndarray.

property center_of_mass: ndarray

Gets the center of mass as a (3,) np.ndarray.

property radius_of_gyration: float

Gets the radius of gyration

get_masses() → ndarray

Gets the masses for all atoms in the current dataframe.

property distance_matrix: ndarray

Gets the distance matrix.

rmsd(*other: Self | ndarray | None = None*, *align: bool = True*, *weights: list | None = None*, *selection: Self | list | None = None*) → list | float

Calculates RMSD 1) among sets of coordinates in one `PDBDataFrame` with multiple `nmr_model` s or 2) two sets of coordinates in two `PDBDataFrames`.

Parameters

- **other** (*optional*) – the other `PDBDataFrame` or (N, 3) `numpy.ndarray` to calculate RMSD against. If `None`, `self` should contain at least two sets of coordinates (`nmr_model` has >= 2 unique values). Defaults to **None**.
- **align** (*optional*) – whether to align the structures before calculating RMSD. If `False`, the `weights` and `selection` keywords are ignored. Defaults to **True**.
- **weights** (*optional*) – a list of weights for all the atoms in `selection` to do structure alignment. If `None`, all coordinates in the `selection` or `self` have the same weights. Defaults to **None**.
- **selection** (*optional*) – a list of `atom_number` s in `self` or a PDBDataFrame after the filtering methods. If `None`, all coordinates in `self` are used for structure alignment. Defaults to **None**.

Returns

RMSD or a list of RMSD's.

Raises

- **ValueError** – if dimensionalities mismatch among `self`, `other`, `weights`, and `selection` if they are not `None`; or `atom_number` s in `self` are not unique.
- **TypeError** – if `other`, `weights`, and `selection` have unsupported types.

record_names(*names: list[str]*, *invert: bool = False*) → Self

Filter by `record_name`.

Parameters

- **names** (*required*) – a list of `record_name` s.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns

sub `PDBDataFrame`

atom_numbers(*numbers: list[int] | int*, *relation: str | None = None*, *invert: bool = False*) → Self

Filter by `atom_number`.

Parameters

- **numbers** (*required*) – one or a list of `atom_number` s.
- **relation** (*optional*) – `atom_number` relationship to `numbers`. If `numbers` is an integer, it has to be one of <, <=, =, >=, and >. If `None`, <= is used. Ignored if a list of integers are provided to `numbers`. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub PDBDataFrame

atom_names(*names: list[str]*, *names_2c: list[str] | None = None*, *invert: bool = False*, *suppress_warning: bool = False*) → Self

Filter by atom_name.

Parameters

- **names** (*required*) – a list of atom_name s whose element_symbols have only one character. Atoms in common residues and ligands should be provide here like C, H, O, N, S, P, F.
- **names_2c** (*optional*) – a list of atom_name s whose element_symbols have two characters like ion (FE) and chloride (CL). Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.
- **suppress_warning** – whether to suppress the warning message about possible conflicts between names and names_2c. Defaults to **False**.

Returns
sub PDBDataFrame

alt_locs(*locs: list[str]*, *invert: bool = False*) → Self

Filter by alt_loc.

Parameters

- **locs** (*required*) – a list of alt_loc s.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub PDBDataFrame

residue_names(*names: list[str]*, *invert: bool = False*) → Self

Filter by residue_names.

Parameters

- **names** (*required*) – a list of residue_name s
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub PDBDataFrame

chain_ids(*ids: list[str]*, *invert: bool = False*) → Self

Filter by chain_id.

Parameters

- **ids** (*required*) – a list of chain_id s.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub PDBDataFrame

residue_numbers(*numbers: list[int] | int*, *relation: str | None = None*, *invert: bool = False*) → Self

Filter by residue_number.

Parameters

- **numbers** (*required*) – one or a list of `residue_number` s.
- **relation** (*optional*) – `residue_number` relationship to `numbers`. If `numbers` is an integer, it has to be one of <, <=, =, >=, and >. If `None`, '<=' is used. Ignored if a list of integers are provided to `numbers`. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub `PDBDataFrame`

insertions(*codes: list[str]*, *invert: bool = False*) → Self

Filter by `insertion`.

Parameters

- **codes** (*required*) – a list of `insertion` codes.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub `PDBDataFrame`

x_coords(*value: float*, *relation: str | None = None*, *invert: bool = False*, *epsilon: float = 0.01*) → Self

Filter by `x_coord`.

Parameters

- **value** (*required*) – value to select `x_coord` s.
- **relation** (*optional*) – `x_coord` relationship to `value`. It has to be one of `'<'`, `'<='`, `'='`, `'>='`, and `'>'`. If `None`, `'<='` is used. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.
- **epsilon** (*optional*) – atoms `abs(x_coord - value) <= epsilon` are selected when `invert = False`. Defaults to **0.01**.

Returns
sub `PDBDataFrame`

y_coords(*value: float*, *relation: str | None = None*, *invert: bool = False*, *epsilon: float = 0.01*) → Self

Filter by `y_coord`.

Parameters

- **value** (*required*) – value to select `y_coord` s.
- **relation** (*optional*) – `y_coord` relationship to `value`. It has to be one of `'<'`, `'<='`, `'='`, `'>='`, and `'>'`. If `None`, `'<='` is used. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.
- **epsilon** (*optional*) – atoms `abs(y_coord - value) <= epsilon` are selected when `invert = False`. Defaults to **0.01**.

Returns
sub `PDBDataFrame`

z_coords(*value: float*, *relation: str | None = None*, *invert: bool = False*, *epsilon: float = 0.01*) → Self

Filter by `z_coord`.

Parameters

- **value** (*required*) – value to select `z_coord` s.

- **relation** (*optional*) – `z_coord` relationship to `value`. It has to be one of `'<'`, `'<='`, `'='`, `'>='`, and `'>'`. If `None`, `'<='` is used. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.
- **epsilon** (*optional*) – atoms `abs(z_coord - value) <= epsilon` are selected when `invert = False`. Defaults to **0.01**.

Returns
sub `PDBDataFrame`

occupancies(*value: float*, *relation: str | None = None*, *invert: bool = False*, *epsilon: float = 0.01*) → Self

Filter by `occupancy`.

Parameters

- **value** (*required*) – value to select `occupancy` s.
- **relation** (*optional*) – `occupancy` relationship to `value`. It has to be one of `'<'`, `'<='`, `'='`, `'>='`, and `'>'`. If `None`, `'<='` is used. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.
- **epsilon** (*optional*) – atoms `abs(occupancy - value) <= epsilon` are selected when `invert = False`. Defaults to **0.01**.

Returns
sub `PDBDataFrame`

b_factors(*value: float*, *relation: str | None = None*, *invert: bool = False*, *epsilon: float = 0.01*) → Self

Filter by `b_factor`.

Parameters

- **value** (*required*) – value to select `b_factor` s.
- **relation** (*optional*) – `b_factor` relationship to `value`. It has to be one of `'<'`, `'<='`, `'='`, `'>='`, and `'>'`. If `None`, `'<='` is used. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.
- **epsilon** (*optional*) – atoms `abs(b_factor - value) <= epsilon` are selected when `invert = False`. Defaults to **0.01**.

Returns
sub `PDBDataFrame`

segment_ids(*ids: list[str]*, *invert: bool = False*) → Self

Filter by `segment_id`.

Parameters

- **ids** (*required*) – a list of `segment_id` s.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub `PDBDataFrame`

element_symbols(*symbols: list[str]*, *invert: bool = False*) → Self

Filter by `element_symbol`.

Parameters

- **symbols** (*required*) – a list of `element_symbol` s.

- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub PDBDataFrame

charges(*charges: list[str]*, *invert: bool = False*) → Self

Filter by charge.

Parameters

- **charges** (*required*) – a list of charge s.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub PDBDataFrame

Notes: charge is 2-char string in the PDB specifications.

nmr_models(*models: list[int] | int*, *relation: str | None = None*, *invert: bool = False*) → Self

Filter by nmr_model.

Parameters

- **models** (*required*) – one or a list of nmr_model ids.
- **relation** (*optional*) – nmr_model relationship to models. If models is an integer, it has to be one of '<', '<=', '=', '>=', and '>'. If None, '<=' is used. Ignored if a list of integers are provided to models. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.

Returns
sub PDBDataFrame

distances(*other: ndarray | Self | Iterable*, *cut_off: float = inf*, *to: str | None = None*, *invert: bool = False*) → Self

Filter by distance to a reference point or group of atoms.

Parameters

- **other** – the other group's coordinate(s).
- **cut_off** – the distance cutoff to filter.
- **to** – if other is a group atoms, using which method to determine whether the atoms meet the cut_off distance. If None, COM or center of mass is used if other is PDBDataFrame, and COG or center of geometry is used if other is np.ndarray or Iterable. The following are allowed:

  com, center of mass, center_of_mass:
  use the center of mass for the other.

  cog, center of geometry, center_of_geometry:
  use the center of geometry for the other.

  all:
  whether all the pair-distances meet the cut_off distance criteria.

  any:
  whether any of the pair-distances meets the cut_off distance criteria.

- **invert** – whether to invert the selection. Defaults to **False**.

Returns
sub PDBDataFrame

_filter_num_col(*value: int | float | list[int]*, *num_col_name: str*, *relation: str | None = None*, *invert: bool = False*, *epsilon: float | int = 0.01*, *suppress_warning: bool = False*) → Self

Generic function to do filter by a numerical column.

Parameters

- **value** (*required*) – value(s) to select by the column given by the num_col_name input.
- **num_col_name** (*required*) – one of atom_number, residue_number, x_coord, y_coord, or z_coord, occupancy, b_factor, and nmr_model. Note: the charge column is not numerical by PDB format.
- **relation** (*optional*) – x/y/z_coord relationship to value. It has to be one of '<', '<=', '=', '>=', and '>'. If None, '<=' is used. Ignored if a list of integers are provided to value. Defaults to **None**.
- **invert** (*optional*) – whether to invert the selection. Defaults to **False**.
- **epsilon** (*optional*) – atoms abs``(``num_col_value - value) <= epsilon are selected when invert = False and relation = '='. Ignored if a list of integers are provided to value. Defaults to **0.01**.
- **suppress_warning** (*optional*) – whether to suppress warnings. Defaults to **False**.

Returns
sub PDBDataFrame

Raises
ValueError – if xyz not in [atom_number, residue_number, x_coord, y_coord, or z_coord, occupancy, b_factor, nmr_model] or relation not in ['<', '<=', '=', '>=', '>'] when selecting on float cols.

classmethod get_chain_list(*pdb_df: Self*) → list

Gets the list of chain ids given a PDBDataFrame object.

Parameters
pdb_df (*required*) – a PDBDataFrame object.

Returns
a list of chain ids.

> **Warning:**
>
> **This method relies on the original and standard numbering of the atoms and**
> residues in the DataFrame. Therefore, any selecting or sorting of the it can lead to wrong results.

classmethod get_residue_list(*pdb_df: Self*, *include_heteros: bool = False*) → list[tuple]

Gets the list of residues given a PDBDataFrame object.

Parameters

- **pdb_df** (*required*) – a PDBDataFrame object.
- **include_heteros** (*optional*) – whether to include hetero ligands. Defaults to **False**.

Returns
a list of residues as (`chain_id`, `residue_name`, `residue_number`).

> **Warning:**
>
> **This method relies on the original and standard numbering of the atoms and**
> residues in the DataFrame. Therefore, any selecting or sorting of the it can lead to wrong results.

classmethod get_distance_matrix(*pdb_df: Self*, *other_data: Self | tuple | None = None*, *use_r2: bool = True*, *square_form: bool = False*) → ndarray

Calculates the distance matrix given a `PDBDataFrame` object and (optional) reference data.

Parameters

- **pdb_df** (*required*) – a `PDBDataFrame` object.
- **other_data** (*optional*) – the coordinates of to calculate the distances against. Defaults to **None**.
- **use_r2** (*optional*) – whether to use r^2 or r for distance matrix. Defaults to **True**.
- **square_form** (*optional*) – whether to output a square form of the density matrix. If two `PDBDataFrame``s are different or ``other_data` is not a `PDBDataFrame`, `square_form` is ignored. Defaults to **False**.

Returns
distance matrix (squared or condensed form)

Raises
ValueError – if `other_data` is not of `PDBDataFrame|tuple|None` type or wrong shape if it is a `tuple`.

1.3.6 PDBx Writer

PDBx/mmCIF format writing.

Write a dict of `Pandas DataFrame` back to a PDBx file.

moldf.write_pdbx.**write_pdbx**(*pdbx: dict[str, DataFrame]*, *file_name: str | PathLike | None = None*) → None

Writes a `dict` of `Pandas DataFrame` s into a PDBx file.

Parameters

- **pdbx** (*required*) – a `dict` of `Pandas DataFrame` s to write.
- **file_name** (*optional*) – file name to write a PDBx file. If `None`, `moldf_output.cif` will be used as the file name. Defaults to **None**.

Raises
TypeError – if `pdbx` is not a valid dict of `DataFrame`.

1.3.7 PDB Writer

PDB format writing.

Write a dict of `Pandas DataFrame` back to a PDB file.

Currently, only the `_atom_site` category can be written back.

`moldf.write_pdb.IMPLEMENTED_PDB_CATS = ['_atom_site']`

PDB categories that are currently implemented.

`moldf.write_pdb.write_pdb(pdb: dict[str, DataFrame], file_name: str | PathLike | None = None, allow_chimera: bool = False) → None`

Write a dict of `Pandas DataFrame` s into a PDB file.

Parameters

- **pdb** (*required*) – a `dict` of `Pandas DataFrame` s to write.
- **file_name** (*optional*) – file name to write a PDB file. If `None`, `moldf_output.pdb` will be used as the file name. Defaults to **None**.
- **allow_chimera** (*optional*) – whether to allow writing to Chimera-formatted PDB files. Defaults to **False**.

Raises

- `TypeError` – if `pdb` is not a valid dict of `DataFrame`.
- `ValueError` – if the `pdb` contains other than supported categories.

1.3.8 MOL2 Writer

MOL2 format writing.

Write a dict of `Pandas DataFrame` back to a MOL2 file.

Currently, only the `MOLECULE`, `ATOM`, and `BOND` categories can be written back.

`moldf.write_mol2.IMPLEMENTED_MOL2_CATS = ['MOLECULE', 'ATOM', 'BOND', 'HEADER']`

MOL2 categories that are currently implemented.

`moldf.write_mol2.write_mol2(mol2: dict[str, DataFrame], file_name: str | PathLike | None = None) → None`

Write a dict of `Pandas DataFrame` s into a MOL2 file. See https://is.muni.cz/th/fzk5s/dp_jakub_Vana.pdf p19 for column definitions.

Parameters

- **mol2** (*required*) – a `dict` of `Pandas DataFrame` s to write.
- **file_name** (*optional*) – file name to write a MOL2 file. If `None`, `moldf_output.mol2` will be used as the file name. Defaults to **None**.

Raises

- `TypeError` – if `mol2` is not a valid dict of `DataFrame`.
- `ValueError` – if the `mol2` contains other than supported categories.

1.3.9 JCSV Writer

Write any dict of Pandas DataFrame to JCSV.

moldf.write_jcsv.**write_jcsv**(*data: dict[str, DataFrame]*, *file_name: str | PathLike | None = None*, *write_meta: bool = True*, ***kwargs*) → None

Write a dict of `Pandas DataFrame` s into a JCSV file. See https://github.com/Ruibin-Liu/JCSV for definitions.

Parameters

- **data** (*required*) – a `dict` of `Pandas DataFrame` s to write.
- **file_name** (*optional*) – file name to write a JCSV file. If `None`, `moldf_output.jcsv` will be used as the file name if `path_or_buf` is not specified in `**kwargs`. Defaults to **None**.
- **write_meta** (*optional*) – whether to write meta data into the **first** category. Currently, only the first line number for each category is recorded. Defaults to **True**.
- ****kwargs** – keyword arguments for `pd.DataFrame.to_csv`. Invalid ones are ignored. Check https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html

Raises

- `TypeError` – if `data` is not a valid dict of `DataFrame`.
- `ValueError` – if `"` in any of the column names.

1.3.10 Covalent Bond

Gets covalent bonds by covalent radii cutoff or ligand templates.

moldf.covalent_bond.**get_covalent_bond_cutoffs**(*element_symbols: list | set*, *single_radii_set: str | None = None*) → tuple[dict, dict, dict]

Gets the DataFrame for additive covalent radii of relevant elements.

Parameters

- **element_symbols** (*required*) – a list or set of element symbols whose covalent radii cutoffs are returned.
- **single_radii_set** (*optional*) – radii sets to use. If `None`, `single_C` is used as to Cordero (PMID 18478144). Another option is `single_PA` which refers to Pyykkö's studies (PMID 19058281;19856342;15832398, and doi:10.1103/PhysRevB.85.024115). Defaults to **None**.

Raises

`ValueError` – if the `single_radii_set` is not valid.

Returns

a tuple of the three dictionaries of the distance cutoffs for the pairs of the queried element symbols: (single_bonds, double_bonds, triple_bonds).

moldf.covalent_bond.**get_residue_template**(*residue_name: str*, *parent_name: str | None = None*, *residue_template_file: str | PathLike | None = None*, *save_template_file: bool = True*, *template_file_dir: str | PathLike | None = None*) → dict

Gets the intra- covalent bonds of a residue or ligand.

Parameters

- **residue_name** (*required*) – residue or ligand name to query.
- **parent_name** (*optional*) – parent name of the residue or ligand. This is useful if the `residue_name` in the PDB file is used by RCSB for a different residue or ligand. If `None`, it is the same as `residue_name`. Defaults to **None**.
- **residue_template_file** (*optional*) – residue or ligand template file name/path. If `None`, this function queries RCSB by the `parent_name`. If it's not a path, this function looks at the current working directory first and then `template_file_dir` if it is not `None`. Defaults to **None**.
- **save_template_file** (*optional*) – whether to save the downloaded (from RCSB) template file. Defaults to **True**.
- **template_file_dir** (*optional*) – directory to save fetched template files. If `None` but `save_template_file` is `True`, './template_files' directory is used. Defaults to **None**.

Returns

a dictionary of bonds for the residue. The dictionary key format is
(atom_id_1: str, atom_id_2: str), and the dictionary value format is (bond_order: str, is_aromatic: bool, stereo_flag: str)

Raises

- `ValueError` – if a `.cif` file for the residue/ligand cannot be downloaded from RCSB for any reason if `residue_template_file` is `None`. Check the url provided in the error message to download manually if possible.
- `FileNotFoundError` – if `residue_template_file` cannot be found if not `None`.

Warning: RuntimeWarning: if the template file contains other residue/ligands.

1.3.11 PDBx Line Splitter

`moldf.split_line`(*line: str*, *delimiter: str | None = None*) → list

Splits a string line into tokens separated by `delimiter` s, assuming all `'` and `"` in the start character or following a *delimiter* are paired to quote a token.

Parameters

- **line** (*required*) – line as a string
- **delimiter** (*optional*) – `delimiter` to split the line. If `None`, `' '` (one space) is used. Defaults to **None**.

Returns
A list of tokens

CHAPTER

TWO

INDICES AND TABLES

- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

G

H

I

M

N

O

P

R

S

U

W

X

Y

Z