LUNA Documentation

Release 0.10.1

Alexandre Fassio

May 02, 2022

CONTENTS

Release 0.11.4

Date May 02, 2022

CHAPTER

ONE

CONTENTS

1.1 Overview of LUNA

1.1.1 Introduction

LUNA is an object-oriented Python 3 toolkit for drug design that makes it easy to analyze very large data sets of 3D molecular structures and complexes, and that allows identifying, filtering, and visualizing atomic interactions. LUNA implements several features geared towards the analysis of molecular complexes, such as: a) accepting any molecular complex type, including protein-ligand and protein-protein; b) accepting multiple file formats, including PDB, MOL, and MOL2; c) providing pre- and post-processing functions to control how interactions are calculated or selected based on geometric constraints; and d) providing several functions to summarize, characterize, and visualize molecular interactions in Pymol.

LUNA also implements three hashed interaction fingerprints (IFP): Extended Interaction FingerPrint (EIFP), Functional Interaction FingerPrint (FIFP), and Hybrid Interaction FingerPrint (HIFP) – inspired by ECFP, FCFP[1], and E3FP[2]. While EIFP encodes explicit atomic substructures, FIFP only encodes more ''coarse-grained" pharmacophoric properties. HIFP adopts a ''hybrid" approach, encoding pharmacophoric properties for atom groups and precise environment information for atoms. All these IFPs are RDKit-compatible and their features represent interactions and contacts between protein residues, ligand atoms, and water molecules, by detecting their presence or absence (bit FPs), or their frequency (count FPs). Besides, these IFPs encode molecular interactions at different levels of detail and are fully interpretable, providing several functionalities to trace individual bits back to their original atomic substructures in the context of the binding site.

LUNA is developed by the Keiser Lab at UCSF and maintained primarily by Alexandre Fassio.

For a thorough description of LUNA and the three IFPs, please consult the original paper[3] and paper repository or *Usage and Examples*.

[1] Rogers D & Hahn M. Extended-connectivity fingerprints. *J. Chem. Inf. Model.* **50**: 742-54 (2010). doi 10.1021/ci100050t

[2] Axen SD, Huang XP, Caceres EL, Gendelev L, Roth BL, Keiser MJ. A Simple Representation Of Three-Dimensional Molecular Structure. *J. Med. Chem.* **60** (17): 7393–7409 (2017). doi 10.1021/acs.jmedchem.7b00696 bioRxiv 136705 F1000Prime RECOMMENDED

[3] TO BE DEFINED (2022).

1.1.2 Contributing

Development occurs on GitHub. Contributions, feature requests, and bug reports are greatly appreciated. Please consult the issue tracker.

1.1.3 License

LUNA is released under the MIT License.

Briefly, this means LUNA can be used in any manner without modification, with proper attribution. However, if the source code is modified for an application, this modified source must also be released under MIT License so that the community may benefit.

1.1.4 Citing LUNA

To cite LUNA, please reference the original paper[Page 3, 3].

References

1.2 Installation

1.2.1 Dependencies

LUNA is compatible with Python 3.x. It additionally has the following dependencies:

- Biopython == 1.72
- colorlog
- Matplotlib
- mmh3 >= 2.5.1
- NetworkX
- NumPy
- Open Babel
- pandas
- Pymol
- RDKit
- SciPy
- Seaborn

- xopen

1.2.2 Installation

The following installation approaches are listed in order of recommendation. These approaches require a prior installation of the dependencies listed above.

Option 1: Pip

To install with pip, run:

```
pip install -U luna
```

We recommend using a virtual environment for this.

Option 2: Build from source

1. Download LUNA repository to your machine.
   - Clone it to your machine with

     ```
     $ git clone https://github.com/keiserlab/LUNA.git
     ```

   - OR download an archive by navigating to the repository and clicking "Download ZIP". Extract the archive.
2. Install with

   ```
   $ cd LUNA
   $ python setup.py build
   $ python setup.py install
   ```

We recommend using a virtual environment for this.

1.3 Usage and Examples

To facilitate flexible use of the LUNA package, we provide an interface to run LUNA as a command line.

1.3.1 Command Line Interface

Command line interfaces (CLI) are provided for the most common task: calculate interactions for a series of protein-ligand complexes.

In the below examples, we assume the LUNA repository is located at $LUNA_REPO and that the current working directory is $LUNA_REPO/example.

run.py

This CLI provides options to calculate protein-ligand interactions using LUNA default methods, save Pymol sessions to analyze interactions, generate interaction fingerprints, and calculate the similarity between ligand binding modes using these generated fingerprints.

To see all available options, run:

```
$ python $LUNA_REPO/luna/run.py --help

usage: run.py [-h] -p PDB_FILE -e ENTRIES_FILE -l LIGAND_FILE -w WORKING_PATH
              [--out_ifp] [-L IFP_NUM_LEVELS] [-R IFP_RADIUS_STEP]
              [-S IFP_LENGTH] [-T {EIFP,HIFP,FIFP}] [-B] [-O IFP_OUTPUT]
              [--sim_matrix_output SIM_MATRIX_OUTPUT]
              [--filter_binding_modes BINDING_MODES_FILE] [--out_pse]
              [--pse_path PSE_PATH] [--overwrite] [--nproc NPROC]

optional arguments:
  -h, --help            show this help message and exit
  -p PDB_FILE, --prot PDB_FILE
                        the protein PDB file
  -e ENTRIES_FILE, --entries ENTRIES_FILE
                        an input file containing a list of ligand ids to process
  -l LIGAND_FILE, --lig LIGAND_FILE
                        a molecular file containing 1 or more ligands
  -w WORKING_PATH       the path where the project and its results will be saved
  --out_ifp             defines whether it should generate LUNA interaction fingerprints
  -L IFP_NUM_LEVELS     the number of level defines the number of iterations to construct
                        the fingerprint. Default: 2
  -R IFP_RADIUS_STEP    the radius growth rate defines the multiplier to increase the␣
↪sphere
                        size at each level. Default: 5.73171
  -S IFP_LENGTH         the fingerprint length. Default: 4096
  -T {EIFP,HIFP,FIFP}   the fingerprint type. Default: EIFP
  -B                    defines whether it should use bit fingerprints. The default value
                        is False, which implies that count fingerprints are used instead
  -O IFP_OUTPUT         the fingerprint output file.
                        Default: <WORKING_PATH>/results/fingerprints/ifp.csv
  --sim_matrix_output SIM_MATRIX_OUTPUT
                        the path where the similarity matrix will be saved. If not␣
↪provided,
                        it won't be generated
  --filter_binding_modes BINDING_MODES_FILE
                        the path of a file containing binding modes to filter
  --out_pse             defines whether it should export interactions to Pymol
  --pse_path PSE_PATH   the path where Pymol sessions (PSE files) will be saved.
                        Default: <WORKING_PATH>/results/pse/
  --overwrite           defines whether it should overwrite an existing project
  -f FORK_PROJECT, --fork_project FORK_PROJECT
                        If provided, copy an existing project to <WORKING_PATH>
  --nproc NPROC         the number of processors to use
```

Calculate protein-ligand interactions

In the following example, we will calculate protein-ligand interactions for docked Dopamine D4 complexes. The input files are located at $LUNA_REPO/examples/inputs.

```
$ python $LUNA_REPO/luna/run.py -p inputs/protein.pdb -l inputs/ligands.mol2 -e inputs/
↪entries.txt -w dopamine_results
```

Generate interaction fingerprints

To generate interaction fingerprints, you need to activate it with the flag `--out_ifp` and modify fingerprint parameters (options `-L`, `-R`, `-S`, `-T`, `-O`) as necessary.

```
$ python $LUNA_REPO/luna/run.py -p inputs/protein.pdb -l inputs/ligands.mol2 -e inputs/
↪entries.txt -w dopamine_results
                                --out_ifp -L 2 -R 5.73 -S 4096 -T EIFP -O dopamine_
↪results/results/new_fp.csv
```

Note: Note that if you have an existing project, you can provide its working path (`-w`) and then LUNA will automatically load the entire project results, which allows you to generate different fingerprints without reprocessing everything from scratch.

Generate similarity matrix

To compute the Tanimoto similarity between interaction fingerprints (IFPs) and create a similarity matrix, you need to use the flag `--sim_matrix_output` in conjunction with `--out_ifp`.

```
$ python $LUNA_REPO/luna/run.py -p inputs/protein.pdb -l inputs/ligands.mol2 -e inputs/
↪entries.txt -w dopamine_results
                                --out_ifp --sim_matrix_output sim_matrix.csv
```

Visualize interactions on Pymol

To depict interactions as Pymol sessions, you need to activate the flag `--out_pse`.

```
$ python $LUNA_REPO/luna/run.py -p inputs/protein.pdb -l inputs/ligands.mol2
                                -e inputs/entries.txt -w dopamine_results --out_pse
```

Filter interactions by binding mode

To filter interactions based on binding modes, you can use the option `--filter_binding_modes`. This option expects a configuration file that defines how interactions should be filtered. See an example from the configuration file `$LUNA_REPO/example/inputs/binding_modes.cfg`:

```
; To configurate an interaction type, create a new line and define the interaction: [New␣
↪interaction].
; Then you can define whether or not all interactions must be accepted by setting
↪'accept_only' to True or False.

; If you want to specify binding modes, use the variable 'accept_only', which expects a␣
↪list of strings \
in the format: <CHAIN ID>/<COMPOUND NAME>/<COMPOUND NUMBER>/<ATOM>
; Wildcards are accepted for the expected fields.
; For example, "*/HIS/*/*" represents all histidines' atoms from all chains.
;              "A/CBL/*/*" represents all ligands named CBL from chain A.
;              "B/HIS/*/N*" represents all histidines' nitrogens from chain B.

[Hydrogen bond]
accept_only=["A/LYS/245/*", "*/HIS/*/*"]

[Hydrophobic]
accept_all=True

[Cation-pi]
accept_only=["*"]
accept_all=False

[Weak hydrogen bond]
accept_all=False
accept_only=["*/THR/434/O*"]

[Face-to-edge pi-stacking]
accept_all=False

[Aromatic stacking]
accept_all=True

[*]
accept_all=False
```

```
$ python $LUNA_REPO/luna/run.py -p inputs/protein.pdb -l inputs/ligands.mol2
                                -e inputs/entries.txt -w dopamine_results --out_pse
                                --filter_binding_modes inputs/binding_modes.cfg
```

Warning: After executing the command above, existing results will be overwritten. If you want to keep the original results, you should fork the target project. To do so, use the option `-f` or `--filter_binding_modes`. Thus, filterings will only have an effect on the forked project.

See an example in the next section.

Fork an existing project

This option allows you to fork an existing project to apply filterings without modifying the original project. To do so, you should use the option `-f` or `--filter_binding_modes`.

```
$ python $LUNA_REPO/luna/run.py -p inputs/protein.pdb -l inputs/ligands.mol2 -e inputs/
↪entries.txt
                                -w filtered_dopamine_results -f dopamine_results
                                --filter_binding_modes inputs/binding_modes.cfg --out_pse
```

Note: If you fork a project without the filtering option, it will only create a copy of the original project. Later, if you decide to filter interactions, you don't need to use the fork option again. Just use the filtering option directly on the forked directory and it will be overwritten.

1.4 Developer notes

1.4.1 Ligands in PDB files

LUNA works with both Open Babel and RDKit. You can switch between both libraries when setting a new project. So, the choice between Open Babel and RDKit depends on the type of molecular file you are working with. If you have ligands in MOL or MOL2 files, for instance, you can use RDKit for your convenience as you will be able to apply RDKit functions directly on the molecular object stored at *LUNA entries*. However, if you frequently work with **ligands** in **PDB files**, you may be especially interested in reading this note.

Although each library has its own difficulties at parsing PDB files, we identified that Open Babel works better at parsing ligands from PDB files, especially what concerns ligands containing aromatic rings. That happens because PDB files don't contain information about atomic charges, valences, and bond types. Consequently, it may occur that some ligands are incorrectly perceived by Open Babel and RDKit.

For this reason, when working with ligands in PDB files, no matter you choose Open Babel or RDKit, we use Open Babel under the hood for parsing, amending, and converting ligands so that both final Open Babel and RDKit objects reflect at most the original and expected molecular structure.

However, in the past, I identified different errors in different Open Babel versions, which can be seen in this issue thread. During that time, I identified that Open Babel 2.3.2 works better for parsing PDB files. For the most common errors identified in this version, I implemented *amending solutions* like charge correction based on the *OpenEye charge model*, or valence correction. Since this solution was taken on basis of Open Babel 2.3.2, some errors from other versions may still persist.

Having that in mind, I usually prefer to use two Open Babel versions in LUNA projects: **2.3.2** for parsing/amending/converting molecules and **3.1.1** as the standard Python library to wrap molecules as Python objects. However, by default, LUNA is configurated to use only the version 3.1.1 of Open Babel. So, if you work with PDB files and are also interested to use this strategy, you can change the OPENBABEL default executable at **luna/util/default_values.py**. There, modify the variable `OPENBABEL` to the executable of your preference. For example, in my local environment, you will find `OPENBABEL` set to "/usr/bin/obabel" that is the version 2.3.3.

1.4.2 Authoring code

We welcome contributions to LUNA!!! These notes are designed to help developers contribute code.

Code Formatting

LUNA's code should be *readable*. To ensure this, we rigorously follow the PEP8 style conventions and PEP257 docstring conventions, which maximize readability of the code and ease of future development. You may check your code for conformation to these conventions with the pycodestyle and pydocstyle utilities, respectively. Where the code is necessarily complicated, inline comments should reorient the reader.

Errors

LUNA-specific errors should inherit `luna.util.exceptions.LUNAError` base class. However, several built-in exceptions are also included in `luna.util.exceptions`.

Contributing Code

Before contributing code to LUNA, it is advisable for major modifications to submit an issue to the issue tracker to enable other developers to contribute to the design of the code and to reduce the amount of work necessary to conform the code to LUNA's standards. After writing the code, create a pull request. This is best even if you have push access to the LUNA repo, as it enables the test suite to be run on the new code prior to merging it with the remaining code base.

Versioning

LUNA versioning system is inspired by the Semantic Versioning Specification (SemVer), in which the version number takes on the following pattern MAJOR.MINOR.PATCH.

Given a version number, increment:

- **MAJOR** version when you make big scientific updates, which will make results produced by LUNA backward-incompatible;
- **MINOR** version when you make backward-incompatible updates that do not involve big scientific updates (e.g., modify class and function parameters);
- **PATCH** version when you make backward-compatible bug fixes or add new functionalities.

Writing Tests

The standard in LUNA is to commit a test for new functionality simultaneously with the new functionality or within the same pull request. While this slows development, it prevents building a large backlog of untested methods and classes.

These should ideally be unit tests, though for some complicated functionalities, integration tests are also necessary. For these complicated functions, specific units may still be tested using `unittest.mock`.

Continuous Integration

LUNA uses Travis CI for continuous integration. This ensures that each commit and pull request passes all tests on a variety of a systems and for all supported versions of Python. Additionally, Travis CI updates code coverage on Coveralls and tests all usage examples in the documentation using `doctest`.

1.4.3 Documentation

In general, it is best to document the rationale and basic usage of a module, class, or method in its docstring instead of in a separate documentation file. See, for example, the docstring for *`InteractionCalculator`*. We use a variety of tools to ensure that our documentation is always up-to-date. The official documentation is hosted on ReadtheDocs and is automatically generated when new code is committed to the repository.

Documenting Code

LUNA uses NumPy's docstring conventions for all docstrings. These are parsed by Sphinx using Napoleon.

The purpose of a docstring is to explain the purpose of a class/method, any relevant implementation details, its parameters, its attributes, its outputs, and its usage. The goal is clarity. For self-evident methods with descriptive variables, a simple one-line summary is all that is needed. For complicated use cases, often involving other methods/classes, it is better to document the usage elsewhere in the documentation.

1.5 LUNA API

1.5.1 LUNA package

Subpackages

luna.align package

Submodules

luna.align.tmalign module

class TMAlignment(*score*, *records*, ***kwargs*)

Bases: `Bio.Align.MultipleSeqAlignment`

Store TM-align results, including sequence alignment and TM-score.

Parameters

- **score** (*float*) – TM-score.
- **records** (iterable of `Bio.SeqRecord.SeqRecord`) – Same length protein sequences. This may be an empty list.
- ****kwargs** (*dict, optional*) – Extra arguments to *`TMAlignment`*. Refer to `Bio.Align.MultipleSeqAlignment` documentation for a list of all possible arguments.

align_structures(*pdb_to_align*, *ref_pdb*, *output_path=None*, *tmalign=None*)

Align two PDB files with TM-align.

> **Warning:** TM-align performs pair-wise structural alignments. Therefore, if your PDB file contains multiple structures, it is recommended you extract the chains first and align them separately, otherwise the alignment may not produce the expected results.
>
> To extract chains you can use *Extractor*.

Parameters

- **pdb_to_align** (*str*) – The PDB structure that will be aligned to `ref_pdb`.
- **ref_pdb** (*str*) – Reference PDB file, i.e., align `pdb_to_align` to `ref_pdb`.
- **output_path** (*str*) – Where to save TM-align output structures.
- **tmalign** (*str*) – Pathname to TM-align binary. The default value is '/bin/tmalign'.

Returns

- *TMAlignment*
- *Results of TM-align, including sequence alignment and TM-score.*

Examples

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.align.tmalign import align_structures
>>> pdb1 = "{LUNA_PATH}/tutorial/inputs/3QQF.pdb"
>>> pdb2 = "{LUNA_PATH}/tutorial/inputs/3QQK.pdb"
>>> alignment = align_structures(pdb1, pdb2, "./", tmalign="/media/data/Workspace/
↪Softwares/TMalignc/TMalign")
>>> print(alignment.score)
0.98582
```

extract_chain_from_sup(*sup_file*, *extract_chain*, *output_file*, *new_chain_id=None*, *QUIET=True*)

Extract a chain from the superposition file generated by TM-align.

> **Warning:** TM-align modifies the id of the original chains, so it is highly recommended to rename it to match the original chain id. To do so, use the parameter `new_chain_id`.

Parameters

- **sup_file** (*str*) – A superposition file generated by TM-align.
- **extract_chain** (*{'A', 'B'}*) – Target chain id to be extracted. TM-align performs pair-wise structural alignment, so the output file will always contain two chains 'A' and 'B', where 'A' and 'B' are the aligned and reference structures, respectively.
- **output_file** (*str*) – Save the extracted chain to this file.
- **new_chain_id** (*str, optional*) – The new chain id of the extracted chain. If not provided, the chain ids will be maintained as it is in the TM-align output.
- **QUIET** (*bool*) – If True (the default), mute warning messages generated by Biopython.

get_seq_records(*tm_output*, *aligned_pdb_id*, *ref_pdb_id*)

Create a pair of `Bio.SeqRecord.SeqRecord` from a TM-align output.

Parameters

- **tm_output** (*str*) – The output produced by TM-align.
- **aligned_pdb_id** (*str*) – An identifier for the aligned PDB file.
- **ref_pdb_id** (*str*) – An identifier for the reference PDB file.

`remove_sup_files`(*path*)
Remove all superposition files created by TM-align at a defined directory `path`.

Module contents

luna.analysis package

Submodules

luna.analysis.residues module

`generate_residue_matrix`(*interactions_mngrs*, *by_interaction=True*)
Generate a matrix to count interactions per residue.

Parameters

- **interactions_mngrs** (iterable of *InteractionsManager*) – A sequence of *InteractionsManager* objects from where interactions will be recovered.
- **by_interaction** (*bool*) – If True (the default), count the number of each interaction type per residue. Otherwise, count the overall number of interactions per residue.

Return type `pandas.DataFrame`

`heatmap`(*data_df*, *figsize=None*, *cmap='Blues'*, *heatmap_kw=None*, *gridspec_kw=None*)
Plot a residue matrix as a color-encoded matrix.

Parameters

- **data_df** (`pandas.DataFrame`) – A residue matrix produced with *generate_residue_matrix()*.
- **figsize** (*tuple, optional*) – Size (width, height) of a figure in inches.
- **cmap** (*str, iterable of str*) – The mapping from data values to color space. The default value is 'Blues'.
- **heatmap_kw** (*dict, optional*) – Keyword arguments for `seaborn.heatmap()`.
- **gridspec_kw** (*dict, optional*) – Keyword arguments for `matplotlib.gridspec.GridSpec`. Used only if the residue matrix (`data_df`) contains interactions.

Return type `matplotlib.axes.Axes` or `numpy.ndarray` of `matplotlib.axes.Axes`

luna.analysis.summary module

`count_interaction_types`(*interactions*, *must_have_target=True*, *compounds=None*, *key_map={}*)
Count the number of each type of interaction in `interactions`.

Parameters

- **interactions** (*iterable*) – An iterable object containing a sequence of interactions (`InteractionType`).
- **must_have_target** (*bool*) – If True, count only interactions involving the target ligand. The default value is True.
- **compounds** (*iterable, optional*) – Only count interactions involving the compounds in `compounds`. The default value is None, which implies that all compounds will be considered.
- **key_map** (*dictionary, optional*) – A dictionary to control which interactions to count and how to aggregate them. The keys are the interaction types to be considered and the values are the final interaction type, which can be used to aggregate interactions. If a value is None, the interaction will be ignored.

  For example, to aggregate all covalent interactions, `key_map` could be defined as follows:

```
key_map = {"Single bond": "Covalent bond",
           "Double bond": "Covalent bond",
           "Triple bond": "Covalent bond",
           "Aromatic bond": "Covalent bond"}
```

  Now, if Ionic interactions should be ignored, `key_map` could be defined as follows:

```
key_map = {"Ionic": None}
```

Returns **interaction_types_count** – The count of each interaction type.

Return type dict

Module contents

luna.interaction package

Submodules

luna.interaction.calc module

class InteractionCalculator(*inter_config={'boundary_cutoff': 6.2, 'max_an_ey_ang_ortho_multipolar_inter': 110, 'max_an_ey_ang_para_multipolar_inter': 25, 'max_cc_dist_amide_pi_inter': 4.5, 'max_cc_dist_pi_pi_inter': 6, 'max_da_dist_hb_inter': 3.9, 'max_da_dist_whb_inter': 4, 'max_dc_dist_whb_inter': 4.5, 'max_dihed_ang_amide_pi_inter': 30, 'max_dihed_ang_slope_pi_pi_inter': 60, 'max_disp_ang_ion_multipole_inter': 40, 'max_disp_ang_multipolar_inter': 40, 'max_disp_ang_offset_pi_pi_inter': 60, 'max_disp_ang_pi_pi_inter': 30, 'max_disp_ang_whb_inter': 40, 'max_disp_ang_xbond_inter': 60, 'max_disp_ang_ybond_inter': 60, 'max_dist_attract_inter': 6, 'max_dist_cation_pi_inter': 6, 'max_dist_hydrop_inter': 4.5, 'max_dist_proximal': 6, 'max_dist_repuls_inter': 6, 'max_ha_dist_hb_inter': 2.8, 'max_ha_dist_whb_inter': 3, 'max_hc_dist_whb_inter': 3.5, 'max_id_dist_ion_multipole_inter': 4.5, 'max_ne_dist_multipolar_inter': 4, 'max_ney_ang_multipolar_inter': 110, 'max_xa_dist_xbond_inter': 4, 'max_xc_dist_xbond_inter': 4.5, 'max_ya_dist_ybond_inter': 4, 'max_yc_dist_ybond_inter': 4.5, 'min_an_ey_ang_antipara_multipolar_inter': 155, 'min_an_ey_ang_ortho_multipolar_inter': 70, 'min_bond_separation': 3, 'min_bond_separation_for_clash': 4, 'min_cxa_ang_xbond_inter': 120, 'min_dar_ang_hb_inter': 90, 'min_dar_ang_whb_inter': 90, 'min_dha_ang_hb_inter': 90, 'min_dha_ang_whb_inter': 110, 'min_dhc_ang_whb_inter': 120, 'min_dihed_ang_slope_pi_pi_inter': 30, 'min_disp_ang_offset_pi_pi_inter': 30, 'min_dist_proximal': 2, 'min_har_ang_hb_inter': 90, 'min_har_ang_whb_inter': 90, 'min_idy_ang_ion_multipole_inter': 60, 'min_inter_atom_in_surf': 1, 'min_ney_ang_multipolar_inter': 70, 'min_rya_ang_ybond_inter': 120, 'min_surf_size': 1, 'min_xar_ang_xbond_inter': 80, 'min_yan_ang_ybond_inter': 80, 'vdw_clash_tolerance': 0.6, 'vdw_tolerance': 0.1}*, *inter_filter=None*, *inter_funcs=None*, *add_non_cov=True*, *add_cov=True*, *add_proximal=False*, *add_atom_atom=True*, *add_dependent_inter=False*, *add_h2o_pairs_with_no_target=False*, *strict_donor_rules=True*, *strict_weak_donor_rules=True*, *lazy_comps_list=['HOH', 'DOD', 'WAT', 'H2O', 'OH2']*)

Bases: `object`

Calculate interactions.

Note: This class provides default LUNA methods to calculate interactions. However, one can provide their own methods without modifying this class. In the **Examples** section, we will show how to define custom functions.

Note: In case you want to disable specific parameters (e.g., angles) used during the calculation of interactions, you do not need to define a custom function for it. You could just delete the parameter from the configuration and LUNA will automatically recognize that a given parameter is not necessary anymore.

Check **Examples 3** to see how to do it and how to implement this automatic behavior on your custom functions.

Parameters

- **inter_config** (*InteractionConfig*) – An *InteractionConfig* object with all parameters and cutoffs necessary to compute interactions defined in `inter_funcs`. If not provided, the

default LUNA configuration will be used instead (*DefaultInteractionConfig*).

- **inter_filter** (*InteractionFilter*, optional) – An *InteractionFilter* object to filter out interactions on-the-fly. The default value is None, which implies no interaction will be filtered out.
- **inter_funcs** (*dict of {tuple : iterable of callable}*) – A dict to define custom functions to calculate interactions, where keys are tuples of feature names (e.g. `("Hydrophobic", "Hydrophobic")`) and values are lists of references to custom functions (see Examples for more details). If not provided, the default LUNA methods will be used instead.
- **add_non_cov** (*bool*) – If True (the default), compute non-covalent interactions. If you are providing custom functions to compute non-covalent interactions and want to make them controllable by this flag, make sure to verify the state of `add_non_cov` at the beginning of the function and return an empty list in case it is False.
- **add_cov** (*bool*) – If True (the default), compute covalent interactions. If you are providing custom functions to compute covalent interactions and want to make them controllable by this flag, make sure to verify the state of `add_cov` at the beginning of the function and return an empty list in case it is False.
- **add_proximal** (*bool*) – If True, compute proximal interactions, which are only distance-based contacts between atoms or atom groups that, therefore, only imply proximity. The default value is False. If you are providing custom functions to compute proximal interactions and want to make them controllable by this flag, make sure to verify the state of `add_proximal` at the beginning of the function and return an empty list in case it is False.
- **add_atom_atom** (*bool*) – If True (the default), compute atom-atom interactions, which, as the name suggests, are interactions that only involve atoms no matter their features. If you are providing custom functions to compute atom-atom interactions and want to make them controllable by this flag, make sure to verify the state of `add_atom_atom` at the beginning of the function and return an empty list in case it is False.

Note: In LUNA, we consider the following interactions as atom-atom: *Van der Waals*, *Van der Waals clash*, and *Atom overlap*. We opted to separate *Van der Waals* from other non-covalent interactions because LUNA may generate an unnecessary number of additional interactions that are usually already represented by other non-covalent interactions as weak hydrogen bonds, hydrophobic, or dipole-dipole interactions. Thus, to give users a fine-grain control over which interactions to calculate, we provided this additional flag to turn off the calculation of Van der Waals interactions.

- **add_dependent_inter** (*bool*) – If True, compute interactions that depend on other interactions. Currently, only water-bridged hydrogen bonds and salt bridges have a dependency on other interactions. The first, depends on two or more hydrogen bonds, while the second depends on an ionic and a hydrogen bond. The default value is False, which implies no dependent interaction will be computed.
- **add_h2o_pairs_with_no_target** (*bool*) – If True, keep interactions of water with atoms and atom groups that do not belong to the target of LUNA's analysis, which are chains or molecules defined as an *Entry* instance. For example, if the target is a ligand and `add_h2o_pairs_with_no_target` is False, then water-water and water-residue hydrogen bonds will be removed because the ligand is not participating in the interactions. The default value is False.
- **strict_donor_rules** (*bool*) – If True (the default), hydrogen bonds will only be considered for donor atoms with explicit hydrogens bound to them. In that case, angles and distances will be evaluated. However, if the molecule containing the donor atom is in `lazy_comps_list`,

then angles and hydrogens will be ignored and LUNA will proceed with the determination of hydrogen bonds based only on donor-acceptor distances. Another exception occurs for solvent molecules in which the donor atom is only bound to hydrogens atoms (e.g., water, ammonia, and hydrogen sulfide). In that case, hydrogens can be positioned in many different ways by Open Babel, which may cause LUNA to detect different hydrogen bonds at each run. So, to circumvent this problem, by default, LUNA always ignores the explicit hydrogen position for donor atoms that only contain hydrogens bound to it.

- **strict_weak_donor_rules** (*bool*) – If True (the default), weak hydrogen bonds will only be considered for donor atoms with explicit hydrogens bound to them. In that case, angles and distances will be evaluated. The same exceptions described for `strict_donor_rules` apply here.
- **lazy_comps_list** (*iterable*) – A sequence of molecule names to ignore explicit hydrogen position during the calculation of hydrogen bonds and weak hydrogen bonds. The default list is ['HOH', 'DOD', 'WAT', 'H2O', 'OH2'], which only contains water molecule names, including variations used by different programs.

Examples

Example 1) How to define custom interactions:

In this example, we will define a custom function to calculate hydrogen bonds.

First, let's start importing the classes and the function we will use.

```
>>> from luna.interaction.type import InteractionType
>>> from luna.interaction.calc import InteractionCalculator
>>> from luna.util.math import euclidean_distance
```

Now, we define the custom function, which simply calculates hydrogen bonds based on donor-acceptor distances. If it is less than 3.5, then a new *InteractionType* object is created with type *Hydrogen bond*.

```
def custom_hbond_function(self, params):
    if not self.add_non_cov:
        return []

    group1, group2, feat1, feat2 = params
    interactions = []

    cc_dist = euclidean_distance(group1.centroid, group2.centroid)
    if cc_dist <= 3.5:
        params = {"dist_hbond_inter": cc_dist}
        inter = InteractionType(group1, group2, "Hydrogen bond", params=params)
        interactions.append(inter)
    return interactions
```

Note: Observe that the function checks if `add_non_cov` has been turned off and if so returns an empty list. That's a recommended strategy because it allows one to turn off all non-covalent interactions with a single flag.

Also, observe that *InteractionCalculator* always expects functions to return a list at the end. That means multiple interactions may be detected for a single pair of *AtomGroup* objects. For example, a donor atom containing 2 hydrogens could, in theory, form two different hydrogen bonds with an acceptor atom.

Now, we have two options to set the custom function to an *InteractionCalculator* object:

1) Define a new dict with the custom functions:

```
>>> custom_funcs = {("Donor", "Acceptor"): [custom_hbond_function]}
>>> ic = InteractionCalculator(inter_funcs=custom_funcs)
```

2) Overwrite the default dict in InteractionCalculator:

```
>>> ic = InteractionCalculator()
>>> ic.funcs[("Donor", "Acceptor")] = [custom_hbond_function]
```

Example 2) How to modify parameters to calculate interactions:

If you just want to modify specific values from the default configuration, you can create a new *DefaultInteractionConfig*, alter parameters, and pass it to *InteractionCalculator*.

```
>>> from luna.interaction.calc import InteractionCalculator
>>> from luna.interaction.config import DefaultInteractionConfig
>>> custom_config = DefaultInteractionConfig()
>>> custom_config.config["min_dha_ang_hb_inter"] = 120
>>> ic = InteractionCalculator(inter_config=custom_config)
```

Alternatively, you can initiate a new *InteractionCalculator* without providing an *InteractionConfig* object, which will cause *InteractionCalculator* to initiate the default configuration. Then, you can modify it directly as we did before.

```
>>> from luna.interaction.calc import InteractionCalculator
>>> ic = InteractionCalculator()
>>> print(ic.inter_config["min_dha_ang_hb_inter"])
90
>>> ic.inter_config["min_dha_ang_hb_inter"] = 120
>>> print(ic.inter_config["min_dha_ang_hb_inter"])
120
```

Finally, if you want to define a custom configuration that will be used in your custom functions, you first need to define the parameters as a dict and then initiate a new *InteractionConfig*. See below:

```
>>> from luna.interaction.config import InteractionConfig
>>> config = {"param1": 2.5, "param2": 90}
>>> custom_config = InteractionConfig(config)
>>> ic = InteractionCalculator(inter_config=custom_config)
>>> print(ic.inter_config["param1"])
2.5
>>> print(ic.inter_config["param2"])
90
```

Example 3) How to disable specific parameters and how to enable automatic recognition of disabled parameters in custom functions:

To automatically disable, for instance, verification of angles during the calculation of interactions using LUNA's default functions, we just need to remove the parameters related to angles from `inter_config`. Let's see an example where we disable angles from hydrogen bonds:

```
>>> ic = InteractionCalculator()
>>> del ic.inter_config["min_dha_ang_hb_inter"]
```

(continues on next page)

(continued from previous page)

```
>>> del ic.inter_config["min_har_ang_hb_inter"]
>>> del ic.inter_config["min_dar_ang_hb_inter"]
```

This simple behavior is possible thanks to the function *is_within_boundary*, which always returns True if the parameter does not exist in `inter_config`. Thus, you can take advantage of this system when implementing custom functions so others will also have the possibility to turn off specific parameters without modifying the code directly. Let's see that in practice.

First, let's start importing the classes and the function we will use.

```
>>> from luna.interaction.config import InteractionConfig
>>> from luna.interaction.type import InteractionType
>>> from luna.interaction.calc import InteractionCalculator
>>> from luna.util.math import euclidean_distance
>>> from operator import le
```

Now, we define a custom function that calculates hydrogen bonds based on donor-acceptor distances only. Observe at line #9 that instead of checking the cutoff directly, we call *is_within_boundary* with the value, parameter, and a comparison function (`le`: less than or equal), which will make it possible to modify or even disable the parameter automatically.

```
1  def custom_hbond_function(self, params):
2      if not self.add_non_cov:
3          return []
4
5      group1, group2, feat1, feat2 = params
6      interactions = []
7
8      cc_dist = euclidean_distance(group1.centroid, group2.centroid)
9      if self.is_within_boundary(cc_dist, "max_hb_dist", le):
10         params = {"dist_hbond_inter": cc_dist}
11         inter = InteractionType(group1, group2, "Hydrogen bond", params=params)
12         interactions.append(inter)
13     return interactions
```

Finally, we are ready to provide the function and custom parameters to *InteractionCalculator*.

```
>>> custom_config = InteractionConfig({"max_hb_dist": 3})
>>> custom_funcs = {("Donor", "Acceptor"): [custom_hbond_function]}
>>> ic = InteractionCalculator(inter_funcs=custom_funcs, inter_config=custom_config)
```

By doing so, we can now alter the new parameter or turn it off.

```
>>> ic.inter_config["max_hb_dist"] = 3.5
>>> del ic.inter_config["max_hb_dist"]
```

static calc_amide_pi(*self*, *params*)

Default method to calculate amide-pi stackings.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, `ChemicalFeature`, `ChemicalFeature`)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (`ChemicalFeature` objects), respectively.

Return type list

static calc_atom_atom(*self*, *params*)
Default method to calculate atom-atom interactions, which include covalent bonds, Van der Waals, Van der Waals clash, and atom overlap.

Note that covalent bonds are controlled by the flag `add_cov`, while the other three interactions are controlled by the flag `add_atom_atom`.

Note: We opted to separate *Van der Waals* from other non-covalent interactions because LUNA may generate an unnecessary number of additional interactions that are usually already represented by other non-covalent interactions as weak hydrogen bonds, hydrophobic, or dipole-dipole interactions. Thus, to give users a fine-grain control over which interactions to calculate, we provided this additional flag to turn off the calculation of Van der Waals interactions.

Parameters **params** (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_cation_pi(*self*, *params*)
Default method to calculate cation-pi interactions.

Parameters **params** (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_chalc_bond(*self*, *params*)
Default method to calculate chalcogen bonds.

Parameters **params** (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_chalc_bond_pi(*self*, *params*)
Default method to calculate chalcogen bonds between chalcogens and aromatic rings.

Parameters **params** (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_hbond(*self*, *params*)
Default method to calculate hydrogen bonds.

Parameters **params** (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_hbond_pi(*self*, *params*)
Default method to calculate hydrogen bonds between (weak) donors and aromatic rings.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_hydrop(*self*, *params*)
Default method to calculate hydrophobic interactons.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

calc_interactions(*trgt_atm_grps*, *nb_atm_grps=None*)
Calculate interactions established by atoms and atoms groups in trgt_atm_grps using methods available in funcs.

The functions in funcs are chosen based on the features of each atom or atom group. For example, consider that a pair of *AtomGroup* objects have both the features 'Hydrophobic'. Then, *calc_interactions* will call any interaction function defined for the tuple ("Hydrophobic", "Hydrophobic") in funcs. Consider now a pair of *AtomGroup* objects whose features are 'Donor' and 'Hydrophobic'. Once again, *calc_interactions* will evaluate if there is any function defined for the tuple ("Donor", "Hydrophobic") (the order does not matter). If there is none, nothing is done and the pair is skipped.

Parameters

- **trgt_atm_grps** (iterable of *AtomGroup*) – Compute interactions involving these *AtomGroup* objects.
- **nb_atm_grps** (iterable of *AtomGroup*) – If defined, only compute interactions between *AtomGroup* objects from trgt_atm_grps with *AtomGroup* objects from nb_atm_grps. If not provided, set nb_atm_grps to be the same as trgt_atm_grps, which implies that interactions will be calculated using only pairs of *AtomGroup* objects from trgt_atm_grps.

static calc_ion_multipole(*self*, *params*)
Default method to calculate favorable and unfavorable ion-dipole interactions.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_ionic(*self*, *params*)
Default method to calculate attractive ionic interactions.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_multipolar(*self*, *params*)
Default method to calculate favorable and unfavorable dipole-dipole interactions.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_pi_pi(*self*, *params*)
Default method to calculate aromatic stackings.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_proximal(*self*, *params*)
Default method to calculate proximal interactions.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_repulsive(*self*, *params*)
Default method to calculate repulsive ionic interactions.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_weak_hbond(*self*, *params*)
Default method to calculate weak hydrogen bonds.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_xbond(*self*, *params*)
Default method to calculate halogen bonds.

Parameters params (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

static calc_xbond_pi(*self*, *params*)
Default method to calculate halogen bonds between halogens and aromatic rings.

Parameters **params** (tuple of (*AtomGroup*, *AtomGroup*, ChemicalFeature, ChemicalFeature)) – The tuple follows the order (A, B, A_f, B_f), where A and B are two *AtomGroup* objects, and A_f and B_f are their features (ChemicalFeature objects), respectively.

Return type list

find_dependent_interactions(*interactions*)

Compute interactions that depend on other interactions. Currently, only water-bridged hydrogen bonds and salt bridges have a dependency on other interactions. The first, depends on two or more hydrogen bonds, while the second depends on an ionic and a hydrogen bond. The default value is False, which implies no dependent interaction will be computed.

Parameters **interactions** (*InteractionType*) – Use these interactions to compute dependent interactions.

property funcs

The dict that defines functions to calculate interactions.

Type dict

get_functions(*feat1*, *feat2*)

Get the functions to calculate interactions for the given features.

Parameters **feat1, feat2** (ChemicalFeature)

Return type iterable of callable

is_feature_pair_valid(*feat1*, *feat2*)

Check if the provided pair of features is valid or not.

It will be valid if the pair exists in funcs, i.e., there is one or more functions to calculate interactions defined for that given pair of features.

It also return False if non-covalent interactions is turned off (add_non_cov = False) and at least one of the features is not *Atom*. This is useful to save processing time as it skips pairs that have functions to calculate non-covalent interactions right away.

Parameters **feat1, feat2** (ChemicalFeature)

Return type bool

is_within_boundary(*value*, *key*, *func*)

Check if a value is within the boundary defined for a given parameter.

Note: It will always return True if the parameter does not exist in inter_config.

Parameters

- **value** (*any*) – The value to be evaluated.
- **key** – A parameter defined in inter_config.
- **func** (*callable*) – The function that evaluates if value is within the boundaries defined for the parameter key.

  Usually, the comparison functions (e.g., lt, le, ge, gt) available in the Python module operator are enough for number comparisons. If you need custom comparison functions, just provide them here.

Return type bool

remove_h2o_pairs_with_no_target(*interactions*)
Remove interactions of water with atoms and atom groups that do not belong to the target of LUNA's analysis, which are chains or molecules defined as an *Entry* instance.

Parameters **interactions** (*InteractionType*)

remove_inconsistencies(*interactions*)
Remove conflicts between interactions in `interactions`.

Note: By default, LUNA defines conflicts as any unfavorable dipole interaction involving an atom establishing a hydrogen bond. Due to the strength of a hydrogen bond, atoms may approximate more to each other, which sometimes may cause unfavorable interactions involving dipoles to be detected. To avoid such conflicts, LUNA removes the unfavorable interactions.

Also, it may occur that unfavorable dipole interactions are detected for amide-aromatic stackings, in which the aromatic ring contains heteroatoms. To avoid such conflicts, LUNA also removes those unfavorable interactions.

Parameters **interactions** (*InteractionType*)

set_functions_to_pair(*pair*, *funcs*)
Set functions to calculate interaction for the given pair of features.

Parameters

- **pair** (tuple of (`ChemicalFeature`, `ChemicalFeature`))
- **funcs** (*iterable of callable*)

class InteractionsManager(*interactions=None*, *entry=None*)
Bases: `object`

Store and manage *InteractionType* objects.

Parameters

- **interactions** (iterable of *InteractionType*, optional) – An initial sequence of *InteractionType* objects.
- **entry** (*Entry*, optional) – The chain or compound used as reference to calculate interactions.

add_interactions(*interactions*)
Add one or more *InteractionType* objects to `interactions`.

count_interations(*must_have_target=False*)
Count the number of each type of interaction in `interactions`.

Parameters **must_have_target** (*bool*) – If True, count only interactions involving the target ligand. The default value is False, which implies all interactions will be considered.

Return type dict

filter_by_types(*types*)
Filter *InteractionType* objects by their types.

Parameters **types** (*iterable of str*) – A sequence of interaction types.

Yields *InteractionType*

filter_out_by_binding_mode(*binding_modes_filter*)
Filter out interactions based on binding modes.

Note: this method modifies `interactions`.

Parameters **binding_modes_filter** (*BindingModeFilter*) – A *BindingModeFilter* object that defines binding mode conditions to decide which interactions are valid.

Returns The interactions that were filtered out.

Return type set of *InteractionType*

`get_all_atm_grps()`
Get all atom groups establishing interactions.

Return type set of *AtomGroup*

`property interactions`
The list of interactions. Additional interactions should be added using the method *add_interactions()*.

Type list of *InteractionType*, read-only

`static load(input_file)`
Load the pickled representation of an *InteractionsManager* object saved at the file `input_file`.

Returns The reconstituted *InteractionsManager* object.

Return type *InteractionsManager*

Raises **PKLNotReadError** – If the file could not be loaded.

`remove_interactions(interactions)`
Remove one or more *InteractionType* objects from `interactions`.

Any recursive references to the removed objects will also be cleared.

`save(output_file, compressed=True)`
Write the pickled representation of the *InteractionsManager* object to the file `output_file`.

Parameters

- **output_file** (*str*) – The output file.
- **compressed** (*bool, optional*) – If True (the default), compress the pickled representation as a gzip file (.gz).

Raises **FileNotCreated** – If the file could not be created.

`property size`
The number of interactions.

Type int, read-only

`to_csv(output_file)`
Write interactions to a comma-separated values (csv) file.

Parameters **output_file** (*str*) – The output CSV file.

`to_json(output_file=None, indent=None)`
Write interactions to a_initial_shell_data JSON file.

Parameters

- **output_file** (*str*) – The output JSON file.
- **indent** (*int or str, optional*) – Indent level for pretty-printed JSON files. An indent level of 0, negative, or '' only insert newlines. Positive integers indent that many spaces per level. If a string is provided (e.g., '\t'), it will be used to indent each level. The default value is None, which selects the most compact representation.

luna.interaction.config module

class DefaultInteractionConfig
Bases: *luna.interaction.config.InteractionConfig*

Default parameters for calculating interactions in LUNA.

class InteractionConfig(*config=None*)
Bases: dict

Generic class to define parameters for interactions.

Parameters **config** (*dict, optional*) – A dict containing parameters for calculating interactions.

Examples

```
>>> config = {"max_ha_dist_hb_inter": 2.5}
>>> inter_config = InteractionConfig(config)
>>> print(inter_config["max_ha_dist_hb_inter"])
2.5
inter_config["max_ha_dist_hb_inter"] = 3
>>> print(inter_config["max_ha_dist_hb_inter"])
3
```

property params
The list of parameters.

Type list

luna.interaction.contact module

get_all_contacts(*entity*, *radius=6.2*, *level='A'*)
Recover all residue-residue or atom-atom contacts in `entity`.

Parameters

- **entity** (`Entity`) – The PDB object from where atoms and residues will be recovered.
- **radius** (*float*) – The cutoff distance (in Å) for defining contacts. The default value is 6.2.
- **level** (*{'R', 'A'}*) – Return residues ('R') or atoms ('A') in contact with `source`.

Returns Each tuple contains either a pair of residues or atoms in contact.

Return type list of tuple of (`Residue` or `Atom`, `Residue` or `Atom`)

Raises **EntityLevelError** – If `level` is neither 'R' nor 'A'.

Examples

In this example, we will identify all residue-residue contacts within 2.5 Å in the PDB 3QQK. So, let's first parse the PDB file.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.MyBio.PDB.PDBParser import PDBParser
>>> pdb_parser = PDBParser(PERMISSIVE=True, QUIET=True)
>>> structure = pdb_parser.get_structure("Protein", f"{LUNA_PATH}/tutorial/inputs/
↪3QQK.pdb")
```

Then, to recover all residue-residue contacts within 2.5 Å use *get_all_contacts()* with `level` set to 'R' and radius set to 2.5.

```
>>> from luna.interaction.contact import get_all_contacts
>>> contacts = get_all_contacts(structure, radius=2.5, level="R")
>>> print(len(contacts))
314
```

get_contacts_with(*entity*, *source*, *target=None*, *radius=6.2*, *level='A'*)

Recover atoms or residues in contact with `source`.

Parameters

- **entity** (`Entity`) – The PDB object from where atoms and residues will be recovered.
- **source** (`Entity`) – The reference, which can be any `Entity` instance (structure, model, chain, residue, or atom).
- **target** (`Entity`, optional) – If provided, only contacts with the `target` will be considered.
- **radius** (*float*) – The cutoff distance (in Å) for defining contacts. The default value is 6.2.
- **level** (*{'R', 'A'}*) – Return residues ('R') or atoms ('A') in contact with `source`.

Returns Each tuple contains two items: the first corresponds to a residue/atom from the `source`, and the second corresponds to a residue/atom in contact with `source`.

Return type set of tuple of (`Residue` or `Atom`, `Residue` or `Atom`)

Raises **EntityLevelError** – If `level` is neither 'R' nor 'A'.

Examples

Example 1) In this example, we will identify residue-residue contacts between a ligand and nearby residues.

First, let's parse a PDB file to work with.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.MyBio.PDB.PDBParser import PDBParser
>>> pdb_parser = PDBParser(PERMISSIVE=True, QUIET=True)
>>> structure = pdb_parser.get_structure("Protein", f"{LUNA_PATH}/tutorial/inputs/
↪3QQK.pdb")
```

Now, we define the target ligand.

```
>>> ligand = structure[0]["A"][('H_X02', 497, ' ')]
```

Then, to recover contacts between this ligand and nearby residues we use *get_contacts_with()* with `level` set to 'R'.

```
>>> from luna.interaction.contact import get_contacts_with
>>> contacts = sorted(get_contacts_with(structure, ligand, radius=3, level="R"))
>>> for pair in contacts:
...     print(pair)
(<Residue X02 het=H_X02 resseq=497 icode= >, <Residue GLU het=  resseq=81 icode= >)
(<Residue X02 het=H_X02 resseq=497 icode= >, <Residue LEU het=  resseq=83 icode= >)
(<Residue X02 het=H_X02 resseq=497 icode= >, <Residue X02 het=H_X02 resseq=497␣
↪icode= >)
(<Residue X02 het=H_X02 resseq=497 icode= >, <Residue HOH het=W resseq=321 icode= >)
```

Example 2) In this example, we will identify atom-atom contacts between a ligand and a given residue.

First, let's parse a PDB file to work with.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.MyBio.PDB.PDBParser import PDBParser
>>> pdb_parser = PDBParser(PERMISSIVE=True, QUIET=True)
>>> structure = pdb_parser.get_structure("Protein", f"{LUNA_PATH}/tutorial/inputs/
↪3QQK.pdb")
```

Now, we define the target ligand and residue.

```
>>> ligand = structure[0]["A"][('H_X02', 497, ' ')]
>>> residue = structure[0]["A"][(' ', 81, ' ')]
```

Then, to recover contacts between these compounds we use *get_contacts_with()*. As we need atom-wise contacts, set `level` to 'A'.

```
>>> from luna.interaction.contact import get_contacts_with
>>> contacts = sorted(get_contacts_with(structure, ligand, target=residue, radius=4,
↪ level="A"))
>>> for pair in contacts:
...     print(pair)
(<Atom C7>, <Atom O>)
(<Atom N10>, <Atom C>)
(<Atom N10>, <Atom O>)
```

get_cov_contacts_with(*entity*, *source*, *target=None*)

Recover potential covalent bonds with `source`.

Covalent bonds between two nearby atoms A and B are determined as in Open Babel:

$$0.4 <= \overrightarrow{\|AB\|} <= A_{cov} + B_{cov} + 0.45$$

Where $\overrightarrow{\|AB\|}$ is the distance between atoms A and B, and A_{cov} and B_{cov} are the covalent radius of atoms A and B, respectively.

Parameters

- **entity** (`Entity`) – The PDB object from where atoms will be recovered.
- **source** (`Entity`) – The reference, which can be any `Entity` instance (structure, model, chain, residue, or atom).

- **target** (`Entity`, optional) – If provided, only covalent bonds with the `target` will be considered.

Returns Pairs of atoms with potential covalent bonds.

Return type set of tuple of (`Atom`, `Atom`)

Examples

In this example, we will identify all covalent bonds involving a given residue in the PDB 3QQK. So, let's first parse the PDB file.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.MyBio.PDB.PDBParser import PDBParser
>>> pdb_parser = PDBParser(PERMISSIVE=True, QUIET=True)
>>> structure = pdb_parser.get_structure("Protein", f"{LUNA_PATH}/tutorial/inputs/
↪3QQK.pdb")
```

Now, we select the residue of our interest.

```
>>> residue = structure[0]["A"][(' ', 81, ' ')]
```

Finally, let`s recover potential covalent bonds involving this residue with *get_cov_contacts_with()*. In the below snippet, pairs of atoms are sorted by atoms` serial number, and the residue information is printed together with the atom name.

```
>>> from luna.interaction.contact import get_cov_contacts_with
>>> cov_bonds = sorted(get_cov_contacts_with(structure, residue), key=lambda x:␣
↪(x[0].serial_number, x[1].serial_number))
>>> for atm1, atm2 in cov_bonds:
>>>     pair = ("%s%d/%s" % (atm1.parent.resname, atm1.parent.id[1], atm1.name),
...             "%s%d/%s" % (atm2.parent.resname, atm2.parent.id[1], atm2.name))
>>>     print(pair)
('PHE80/C', 'GLU81/N')
('GLU81/N', 'GLU81/CA')
('GLU81/CA', 'GLU81/C')
('GLU81/CA', 'GLU81/CB')
('GLU81/C', 'GLU81/O')
('GLU81/C', 'PHE82/N')
('GLU81/CB', 'GLU81/CG')
('GLU81/CG', 'GLU81/CD')
('GLU81/CD', 'GLU81/OE1')
('GLU81/CD', 'GLU81/OE2')
```

`get_proximal_compounds`(*source*, *radius=2.2*)

Recover proximal compounds to `source`.

Parameters

- **source** (`Residue`) – The reference compound.
- **radius** (*float*) – The cutoff distance (in Å) for defining proximity. The default value is 2.2, which may recover potential residues bound to `source` through covalent bonds.

Returns The list of proximal compounds always include `source`.

Return type list of `Residue`

Raises **IllegalArgumentError** – If source is not a Residue

Examples

In this example, we will identify all proximal compounds to a given residue in the PDB 3QQK. So, let's first parse the PDB file.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.MyBio.PDB.PDBParser import PDBParser
>>> pdb_parser = PDBParser(PERMISSIVE=True, QUIET=True)
>>> structure = pdb_parser.get_structure("Protein", f"{LUNA_PATH}/tutorial/inputs/
↪3QQK.pdb")
```

Now, we select the residue of our interest.

```
>>> residue = structure[0]["A"][(' ', 81, ' ')]
```

Finally, we call *get_proximal_compounds()*.

```
>>> from luna.interaction.contact import get_proximal_compounds
>>> compounds = get_proximal_compounds(residue)
>>> for c in compounds:
...     print(c)
<Residue PHE het=  resseq=80 icode= >
<Residue GLU het=  resseq=81 icode= >
<Residue PHE het=  resseq=82 icode= >
```

luna.interaction.filter module

class BindingModeCondition(*condition*)

Bases: object

Define binding mode conditions to filter interactions.

Parameters **condition** (*str*) – A string defining which chains, compounds, or atoms should be accepted. If condition is the wildcard '*', then all chains, compounds, and atoms will be considered valid. Otherwise, condition should have the format '<CHAIN ID>/<COMPOUND NAME>/<COMPOUND NUMBER>/<ATOM>'. Wildcards are accepted for each one of these fields. For example:

- '*/HIS/*/*': represents all histidines' atoms from all chains.
- 'A/CBL/*/*' represents all ligands named CBL from chain A.
- 'B/HIS/*/N*' represents all histidines' nitrogens from chain B.

Variables

- **~BindingModeCondition.accept_all** (*bool*) – If True, all chains, compounds, and atoms will be considered valid.
- **~BindingModeCondition.accept_all_chains** (*bool*) – If True, all chains will be considered valid.
- **~BindingModeCondition.accept_all_comps** (*bool*) – If True, all compound names will be considered valid.

- **~BindingModeCondition.accept_all_comp_nums** (*bool*) – If True, all compound numbers (residue sequence number in the PDB format) will be considered valid.
- **~BindingModeCondition.accept_all_atoms** (*bool*) – If True, all atoms will be considered valid.
- **~BindingModeCondition.chain_id** (*str or None*) – If provided, accept only chains whose id matches chain_id.
- **~BindingModeCondition.comp_name** (*str or None*) – If provided, accept only compounds whose name matches comp_name.
- **~BindingModeCondition.comp_num** (*int or None*) – If provided, accept only compounds whose sequence number matches comp_num.
- **~BindingModeCondition.comp_icode** (*str or None*) – If provided, accept only compounds whose insertion code matches comp_icode.
- **~BindingModeCondition.atom** (*str or None*) – If provided, accept only atoms whose name matches atom.

is_valid(*atm_grp*)

Check if an atom group is valid or not based on this condition.

atm_grp : *luna.mol.groups.AtomGroup*

class BindingModeFilter(*config*)

Bases: object

Filter interactions based on a set of binding mode conditions.

Parameters **config** (*dict of {str : iterable}*) – A dict defining binding modes and how interactions should be validated. Each key represents an interaction type and values are an iterable of *BindingModeCondition* instances.

classmethod from_config_file(*config_file*)

Initialize from a configuration file.

Parameters **``config_file``** (*str*) – The configuration file pathname.

Return type *BindingModeFilter*

Examples

It follows an example of a configuration file:

```
; To configurate an interaction type, create a new line and define the␣
↪interaction: [New interaction].
; Then you can define whether or not all interactions must be accepted by␣
↪setting 'accept_only' to True or False.

; If you want to specify binding modes, use the variable 'accept_only', which␣
↪expects a list of strings             in the format: <CHAIN ID>/<COMPOUND␣
↪NAME>/<COMPOUND NUMBER>/<ATOM>
; Wildcards are accepted for the expected fields.
; For example, "*/HIS/*/*" represents all histidines' atoms from all chains.
;              "A/CBL/*/*" represents all ligands named CBL from chain A.
;              "B/HIS/*/N*" represents all histidines' nitrogens from chain B.

[Hydrogen bond]
```

(continues on next page)

(continued from previous page)

```
accept_only=["A/LYS/245/*", "*/HIS/*/*"]

[Hydrophobic]
accept_all=True

[Cation-pi]
accept_only=["*"]
accept_all=False

[Weak hydrogen bond]
accept_all=False
accept_only=["*/THR/434/O*"]

[Face-to-edge pi-stacking]
accept_all=False

[Aromatic stacking]
accept_all=True

[*]
accept_all=False
```

is_valid(*inter*)

Check if an interaction is valid or not based on this binding mode configuration.

inter : *luna.interaction.type.InteractionType*

class InteractionFilter(*ignore_self_inter=True*, *ignore_intra_chain=True*, *ignore_inter_chain=True*, *ignore_res_res=True*, *ignore_res_nucl=True*, *ignore_res_hetatm=True*, *ignore_nucl_nucl=True*, *ignore_nucl_hetatm=True*, *ignore_hetatm_hetatm=True*, *ignore_h2o_h2o=True*, *ignore_any_h2o=False*, *ignore_multi_comps=False*, *ignore_mixed_class=False*)

Bases: `object`

Filter interactions based on their components.

Parameters

- **ignore_self_inter** (*bool*) – If True, ignore interactions involving atoms of the same compound.
- **ignore_intra_chain** (*bool*) – If True, ignore intra-chain interactions (e.g., interactions between residues in the same protein chain).
- **ignore_inter_chain** (*bool*) – If True, ignore interactions between different chains.
- **ignore_res_res** (*bool*) – If True, ignore residue-residue interactions.
- **ignore_res_nucl** (*bool*) – If True, ignore residue-nucleotide interactions.
- **ignore_res_hetatm** (*bool*) – If True, ignore residue-ligand interactions.
- **ignore_nucl_nucl** (*bool*) – If True, ignore nucleotide-nucleotide interactions.
- **ignore_nucl_hetatm** (*bool*) – If True, ignore nucleotide-ligand interactions.
- **ignore_hetatm_hetatm** (*bool*) – If True, ignore ligand-ligand interactions.
- **ignore_h2o_h2o** (*bool*) – If True, ignore water-water interactions.

- **ignore_any_h2o** (*bool*) – If True, ignore all interactions involving water.
- **ignore_multi_comps** (*bool*) – If True, ignore interactions established by atom groups composed of multiple compounds (e.g.: amides formed by peptide bonds involve two residues).
- **ignore_mixed_class** (*bool*) – If True, ignore interactions established by atom groups comprising mixed compound classes (e.g. residues and ligands bound by a covalent bond).

is_valid_pair(*src_grp*, *trgt_grp*)

Evaluate if a pair of atom groups are valid according to the flags defined in this class.

src_grp, trgt_grp : *luna.mol.groups.AtomGroup*

classmethod new_nli_filter(*ignore_nucl_hetatm=False*, *ignore_hetatm_hetatm=False*, *ignore_any_h2o=False*, *ignore_self_inter=False*, ***kwargs*)

Initialize the default filter for nucleotide-ligand interactions.

Return type *InteractionFilter*

classmethod new_nni_filter(*ignore_nucl_nucl=False*, *ignore_inter_chain=False*, *ignore_intra_chain=False*, *ignore_any_h2o=False*, *ignore_self_inter=False*, ***kwargs*)

Initialize the default filter for nucleotide-nucleotide interactions.

Return type *InteractionFilter*

classmethod new_pli_filter(*ignore_res_hetatm=False*, *ignore_hetatm_hetatm=False*, *ignore_any_h2o=False*, *ignore_self_inter=False*, ***kwargs*)

Initialize the default filter for protein-ligand interactions.

Return type *InteractionFilter*

classmethod new_pni_filter(*ignore_res_nucl=False*, *ignore_inter_chain=False*, *ignore_intra_chain=False*, *ignore_any_h2o=False*, *ignore_self_inter=False*, ***kwargs*)

Initialize the default filter for protein-nucleotide interactions.

Return type *InteractionFilter*

classmethod new_ppi_filter(*ignore_res_res=False*, *ignore_inter_chain=False*, *ignore_intra_chain=False*, *ignore_any_h2o=False*, *ignore_self_inter=False*, ***kwargs*)

Initialize the default filter for protein-protein interactions.

Return type *InteractionFilter*

luna.interaction.type module

class InteractionType(*src_grp*, *trgt_grp*, *inter_type*, *src_interacting_atms=None*, *trgt_interacting_atms=None*, *src_centroid=None*, *trgt_centroid=None*, *directional=False*, *params=None*)

Bases: `object`

Define an interaction type.

Parameters

- **src_grp** (*AtomGroup*) – The interaction's first atom or atom group. For directional interactions such as hydrogen bonds, `src_grp` represents donor atoms and atom groups that act as nucleophiles, i.e., from where the interaction "comes from".

- **trgt_grp** (*AtomGroup*) – The interaction's second atom or atom group. For directional interactions such as hydrogen bonds, `trgt_grp` represents acceptor atoms and atom groups that act as electrophiles, i.e., the interaction "receiver".
- **inter_type** (*str*) – The interaction type.
- **src_interacting_atms** (iterable of *ExtendedAtom*, optional) – If provided, represent the set of atoms from `src_grp` that in fact participate in the interaction. For example, in hydrophobic islands, not all of their atoms are in direct contact with another hydrophobic surface.
- **trgt_interacting_atms** (iterable of *ExtendedAtom*, optional) – If provided, represent the set of atoms from `trgt_grp` that in fact participate in the interaction. For example, in hydrophobic islands, not all of their atoms are in direct contact with another hydrophobic surface.
- **src_centroid** (*array_like of float (size 3), optional*) – Atomic coordinates (x, y, z) of the centroid of `src_grp`. If not provided, it will be calculated automatically from `src_grp` or `src_interacting_atms`.
- **trgt_centroid** (*array_like of float (size 3), optional*) – Atomic coordinates (x, y, z) of the centroid of `trgt_grp`. If not provided, it will be calculated automatically from `trgt_grp` or `trgt_interacting_atms`.
- **directional** (*bool*) – Indicate if the interaction has a direction as in hydrogen bonds and multipolar interactions.
- **params** (*dict, optional*) – Interaction parameters (distances, angles, etc).

`as_json()`

Represent this interaction as a dict containing the interaction type, flags indicating if its directional or not and if it is an intra- or intermolecular interaction, its default color for visual representations, and information related to each involved atom group.

The dict is defined as follows:

- `type` (str): the interaction type;
- `is_directional` (bool`): if it is a directional interaction;
- `is_intramol_interaction` (bool`): if it is an intramolecular interaction;
- `is_intermol_interaction` (bool`): if it is an intermolecular interaction;
- `color` (str): the default interaction hex color for visual representations;
- `src_grp` (dict): information related to the interaction's first atom or atom group. The dict structure is defined in *luna.mol.groups.AtomGroup.as_json()*. Additional keys:
  - `add_pseudo_group` (bool): if True, it means a pseudo-atom (centroid) should represent `src_grp` as it comprises multiple atoms.
  - `centroid` (list of float): the atomic coordinates (x, y, z) of the centroid of `src_grp`
  - `show_centroid` (str): if the centroid should be shown or not. It will be False only for nucleophiles and electrophiles.
- `trgt_grp` (dict): information related to the interaction's second atom or atom group. The dict structure is defined in *luna.mol.groups.AtomGroup.as_json()*. Additional keys:
  - `add_pseudo_group` (bool): if True, it means a pseudo-atom (centroid) should represent `trgt_grp` as it comprises multiple atoms.
  - `centroid` (list of float): the atomic coordinates (x, y, z) of the centroid of `trgt_grp`
  - `show_centroid` (str): if the centroid should be shown or not. It will be False only for nucleophiles and electrophiles.

clear_refs()
References to this *InteractionType* instance will be removed from the list of interactions of src_grp and trgt_grp.

get_partner(*comp*)
Get the partner atom group that forms this interaction with comp. Return None if comp is neither the src_grp nor trgt_grp.

Parameters **comp** (*AtomGroup*) – Get the partner of this atom group.

Return type *AtomGroup* or None

is_directional()
Indicate if the interaction has a direction as in hydrogen bonds and multipolar interactions.

Return type bool

is_intermol_interaction()
Indicate if the interaction involves atoms from different molecules.

Return type bool

is_intramol_interaction()
Indicate if the interaction involves atoms from the same molecule.

Return type bool

property params
Interaction parameters (distances, angles, etc).

Type dict

property required_interactions
If this interaction depends on other interactions, then return them as a list. Currently, by default, only water-bridged hydrogen bonds and salt bridges have a dependency on other interactions. The first, depends on two or more hydrogen bonds, while the second depends on an ionic and a hydrogen bond.

Type list of *InteractionType*

property src_centroid
Atomic coordinates (x, y, z) of the centroid of src_grp. If it is not provided during the initialization of this class, then it will be calculated automatically from src_grp or src_interacting_atms.

Type array_like of float (size 3)

property src_grp
The interaction's first atom or atom group.

Type *AtomGroup*

property src_interacting_atms
The set of atoms from src_grp that in fact participate in the interaction. If a sequence of atoms is not provided during the initialization of this class, then all atoms from src_grp are returned.

Type iterable of *ExtendedAtom*

property trgt_centroid
Atomic coordinates (x, y, z) of the centroid of trgt_grp. If it is not provided during the initialization of this class, then it will be calculated automatically from trgt_grp or trgt_interacting_atms.

Type array_like of float (size 3)

property trgt_grp
The interaction's second atom or atom group.

Type *AtomGroup*

property trgt_interacting_atms
The set of atoms from `trgt_grp` that in fact participate in the interaction. If a sequence of atoms is not provided during the initialization of this class, then all atoms from `trgt_grp` are returned.

Type iterable of *ExtendedAtom*

property type
The interaction type.

Type str

luna.interaction.view module

class InteractionViewer(*show_hydrop_surface=False*, ***kwargs*)
Bases: *luna.wrappers.pymol.PymolSessionManager*

Class that inherits from *PymolSessionManager* and implements *set_view()* to depict interactions in Pymol and save the view as a Pymol session.

This class can be used to visualize multiple complexes into the same Pymol session, for instance, to compare binding modes. To do so, it is recommended that the protein structures are in the same coordinate system, i.e., they should be aligned first. This can be achieved with `luna.align.tmalign.align_2struct()` or any other tool of your preference.

Parameters

- **show_hydrop_surface** (*bool*) – If True, highlight hydrophobic surfaces. The default value is False
- ****kwargs** (*dict, optional*) – Extra arguments to *InteractionViewer*. Refer to *PymolSessionManager* documentation for a list of all possible arguments.

Examples

In the below examples, we will assume a LUNA project object named `proj_obj` already exists.

Example 1) In the first example, we will visualize all interactions identified in the first protein-ligand complex. To do so, we will provide a tuple containing an *InteractionsManager* from where the interactions will be recovered.

First access the property `interactions_mngrs` to get an iterable of *InteractionsManager* objects and get the first one.

```
>>> interactions_mngr = list(proj_obj.interactions_mngrs)[0]
```

As *InteractionViewer* expects a list of tuples, we will define it now. The first item of the tuple is the *Entry* instance, which represents a ligand. This can be obtained directly from the *InteractionsManager* object. The *Entry* instance is necessary because the second item in the tuple (interactions) may be an iterable of *InteractionType* from where such information cannot be recovered. Finally, the third item can be either a PDB file or a directory. In this example, we will use the PDB directory defined during the LUNA project initialization.

```
>>> inter_tuples = [(interactions_mngr.entry, interactions_mngr, proj_obj.pdb_path)]
```

Now, we create a new *InteractionViewer* object and call *new_session()*, which will initialize the session, depict the interactions, and save the session to an output PSE file.

```
>>> inter_view = InteractionViewer()
>>> inter_view.new_session(inter_tuples, "output.pse")
```

Example 2) In this example, we will create a new Pymol session where a given set of interactions will be shown. Let's say, for instance, we only want to visualize hydrogen bonds.

To do so, instead of defining a tuple with an *InteractionsManager* instance, we will define a list of *InteractionType* objects, which will contain only hydrogen bonds.

Let's start with the selection of hydrogen bonds. Here, we will use the built-in method *luna.interaction.calc.InteractionsManager.filter_by_types()*, which permits to filter interactions by type.

```
>>> interactions_mngr = list(proj_obj.interactions_mngrs)[0]
>>> hydrogen_bonds = interactions_mngr.filter_by_types(['Hydrogen bond'])
```

Now, we just create the tuple as we did before and define the list of interactions we want to depict in the Pymol session.

```
>>> inter_tuples = [(interactions_mngr.entry, hydrogen_bonds, proj_obj.pdb_path)]
```

Finally, we create a new *InteractionViewer* object and call `InteractionViewer.new_session()`, which will initialize the session, depict the interactions, and save the session to an output PSE file.

```
>>> inter_view = InteractionViewer()
>>> inter_view.new_session(inter_tuples, "output.pse")
```

Example 3) In this final example, we will create a new Pymol session to visualize all complexes in a LUNA project. To do so, we create a list with one tuple for each complex:

```
>>> inter_tuples = [(im.entry, im, proj_obj.pdb_path) for im in proj_obj.
↪interactions_mngrs]
```

Then, as we did before, just create a new *InteractionViewer* object and call `InteractionViewer.new_session()`.

```
>>> inter_view = InteractionViewer()
>>> inter_view.new_session(inter_tuples, "output.pse")
```

Note: it is recommended that all complexes have the same atomic coordinates to make the analysis and comparisons easier. If that's not the case, you may want to align the structures first. To do so, you can use `luna.align.tmalign.align_2struct()` or any other tool of your preference.

set_view(*inter_tuples*)

Depict interactions into the current Pymol session.

Parameters **inter_tuples** (*iterable of tuple*) – Each tuple must contain three items: an *Entry* instance, an iterable of *InteractionType* or an *InteractionsManager*, and a PDB file or the directory where the PDB file is located.

Subpackages

luna.interaction.fp package

Submodules

luna.interaction.fp.fingerprint module

class CountFingerprint(*indices=None*, *counts=None*, *fp_length=4294967296*, *unfolded_fp=None*, *unfolding_map=None*, *props=None*)

Bases: *luna.interaction.fp.fingerprint.Fingerprint*

A fingerprint that stores the number of occurrences of each index.

Parameters

- **indices** (*array_like of int, optional*) – Indices of "on" bits. It is optional if `counts` is provided.
- **counts** (*dict, optional*) – Mapping between each index in `indices` to the number of counts. If not provided, the default count value of 1 will be used instead.
- **fp_length** (*int*) – The fingerprint length (total number of bits). The default value is 2^{32}.
- **unfolded_fp** (*Fingerprint* or None) – The unfolded version of this fingerprint. If None, this fingerprint may have not been folded yet.
- **unfolding_map** (*dict, optional*) – A mapping between current indices and indices from the unfolded version of this fingerprint what makes it possible to trace folded bits back to the original shells (features).
- **props** (*dict, optional*) – Custom properties of the fingerprint, consisting of a string keyword and some value. It can be used, for instance, to save the ligand name and parameters used to generate shells (IFP features).

property counts

Mapping between each index in `indices` to the number of counts.

Type dict, read-only

fold(*new_length=4096*)

Fold this fingerprint to size `new_length`.

Parameters **new_length** (*int*) – Length of the new fingerprint, ideally multiple of 2. The default value is 4096.

Returns Folded *Fingerprint*.

Return type *Fingerprint*

Raises **BitsValueError** – If the new fingerprint length is not a multiple of 2 or is greater than the existing fingerprint length.

Examples

```
>>> from luna.interaction.fp.fingerprint import CountFingerprint
>>> import numpy as np
>>> np.random.seed(0)
>>> on_bits = 8
>>> fp_length = 32
>>> indices, counts = np.unique(np.random.randint(0, fp_length, on_bits),␣
↪return_counts=True)
>>> counts = dict(zip(indices, counts))
>>> print(counts)
{0: 1, 3: 2, 7: 1, 12: 1, 15: 1, 21: 1, 27: 1}
>>> fp = CountFingerprint.from_indices(indices, counts=counts, fp_length=fp_
↪length)
>>> print(fp.indices)
[ 0  3  7 12 15 21 27]
>>> print(fp.to_vector(compressed=False))
[1 0 0 2 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0]
>>> folded_fp = fp.fold(8)
>>> print(folded_fp.indices)
[0 3 4 5 7]
>>> print(folded_fp.to_vector(compressed=False))
[1 0 0 3 1 1 0 2]
```

classmethod from_bit_string(*bit_string*, *counts=None*, *fp_length=None*, ***kwargs*)

Initialize from a bit string (e.g. '0010100110').

Parameters

- **bit_string** (*str*) – String of 0s and 1s.
- **counts** (*dict, optional*) – Mapping between each index in `indices` to the number of counts. If not provided, the default count value of 1 will be used instead.
- **fp_length** (*int, optional*) – The fingerprint length (total number of bits). If not provided, the fingerprint length will be defined based on the string length.
- ****kwargs** (*dict, optional*) – Extra arguments to *Fingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *CountFingerprint*

Examples

```
>>> from luna.interaction.fp.fingerprint import CountFingerprint
>>> fp = CountFingerprint.from_bit_string("0010100110000010",
...                                       counts={2: 5, 4: 1, 7: 3, 8: 1, 14: 2}
↪)
>>> print(fp.indices)
[ 2  4  7  8 14]
>>> print(fp.counts)
{2: 5, 4: 1, 7: 3, 8: 1, 14: 2}
```

classmethod from_counts(*counts*, *fp_length=4294967296*, ***kwargs*)

Initialize from a counting map.

Parameters

- **counts** (*dict*) – Mapping between each index in `indices` to the number of counts.
- **fp_length** (*int*) – The fingerprint length (total number of bits). The default value is 2^{32}.
- ****kwargs** (*dict, optional*) – Extra arguments to *CountFingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *CountFingerprint*

Examples

```
>>> from luna.interaction.fp.fingerprint import CountFingerprint
>>> import numpy as np
>>> np.random.seed(0)
>>> on_bits = 8
>>> fp_length = 32
>>> counts = dict(zip(*np.unique(np.random.randint(0, fp_length, on_bits),
...                              return_counts=True)))
>>> print(counts)
{0: 1, 3: 2, 7: 1, 12: 1, 15: 1, 21: 1, 27: 1}
>>> fp = CountFingerprint.from_counts(counts=counts, fp_length=fp_length)
>>> print(fp.indices)
[ 0  3  7 12 15 21 27]
>>> print(fp.to_vector(compressed=False))
1 0 0 2 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0]
```

classmethod from_fingerprint(*fp*, ***kwargs*)

Initialize from an existing fingerprint.

Parameters

- **fp** (*Fingerprint*) – An existing fingerprint.
- ****kwargs** (*dict, optional*) – Extra arguments to *Fingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *CountFingerprint*

classmethod from_indices(*indices=None*, *counts=None*, *fp_length=4294967296*, ***kwargs*)

Initialize from an array of indices.

Parameters

- **indices** (*array_like of int, optional*) – Indices of "on" bits. It is optional if `counts` is provided.
- **counts** (*dict, optional*) – Mapping between each index in `indices` to the number of counts. If not provided, the default count value of 1 will be used instead.
- **fp_length** (*int*) – The fingerprint length (total number of bits). The default value is 2^{32}.
- ****kwargs** (*dict, optional*) – Extra arguments to *CountFingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *CountFingerprint*

Examples

```
>>> from luna.interaction.fp.fingerprint import CountFingerprint
>>> import numpy as np
>>> np.random.seed(0)
>>> on_bits = 8
>>> fp_length = 32
>>> indices, counts = np.unique(np.random.randint(0, fp_length, on_bits),␣
↪return_counts=True)
>>> counts = dict(zip(indices, counts))
>>> print(counts)
{0: 1, 3: 2, 7: 1, 12: 1, 15: 1, 21: 1, 27: 1}
>>> fp = CountFingerprint.from_indices(indices, counts=counts, fp_length=fp_
↪length)
>>> print(fp.indices)
[ 0  3  7 12 15 21 27]
>>> print(fp.to_vector(compressed=False))
[1 0 0 2 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0]
```

classmethod from_vector(*vector*, *fp_length=None*, ***kwargs*)

Initialize from a vector.

Parameters

- **vector** (numpy.ndarray or scipy.sparse.csr_matrix) – Array of counts.
- **fp_length** (*int, optional*) – The fingerprint length (total number of bits). If not provided, the fingerprint length will be defined based on the `vector` shape.
- ****kwargs** (*dict, optional*) – Extra arguments to *Fingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *CountFingerprint*

Examples

```
>>> from luna.interaction.fp.fingerprint import CountFingerprint
>>> import numpy as np
>>> np.random.seed(0)
>>> fp_length = 32
>>> vector = np.random.choice(5, size=(fp_length,), p=[0.76, 0.1, 0.1, 0.02, 0.
↪02])
>>> print(vector)
[0 0 0 0 2 3 0 1 0 0 2 0 0 0 1 1 2 3 1 0 1 0 0 0 2 0 0 0 1 0 0 0]
>>> fp = CountFingerprint.from_vector(vector)
>>> print(fp.indices)
[ 4  5  7 10 14 15 16 17 18 20 24 28]
>>> print(fp.counts)
{4: 2, 5: 3, 7: 1, 10: 2, 14: 1, 15: 1, 16: 2, 17: 3, 18: 1, 20: 1, 24: 2, 28:␣
↪1}
```

get_count(*index*)

Get the count value at index `index`. Return 0 if index is not in `counts`.

class Fingerprint(*indices*, *fp_length=4294967296*, *unfolded_fp=None*, *unfolding_map=None*, *props=None*)

Bases: object

A fingerprint that stores indices of "on" bits.

Parameters

- **indices** (*array_like of int*) – Indices of "on" bits.
- **fp_length** (*int*) – The fingerprint length (total number of bits). The default value is 2^{32}.
- **unfolded_fp** (*Fingerprint* or None) – The unfolded version of this fingerprint. If None, this fingerprint may have not been folded yet.
- **unfolding_map** (*dict, optional*) – A mapping between current indices and indices from the unfolded version of this fingerprint what makes it possible to trace folded bits back to the original shells (features).
- **props** (*dict, optional*) – Custom properties of the fingerprint, consisting of a string keyword and some value. It can be used, for instance, to save the ligand name and parameters used to generate shells (IFP features).

property bit_count

Number of "on" bits.

Type int, read-only

calc_similarity(*other*)

Calculates the Tanimoto similarity between this fingeprint and `other`.

Return type float

Examples

```
>>> from luna.interaction.fp.fingerprint import Fingerprint
>>> fp1 = Fingerprint.from_bit_string("0010101110000010")
>>> fp2 = Fingerprint.from_bit_string("1010100110010010")
>>> print(fp1.calc_similarity(fp2))
0.625
```

property counts

Mapping between each index in `indices` to the number of counts, which is always 1 for bit fingerprints.

Type dict, read-only

property density

Proportion of "on" bits in fingerprint.

Type float, read-only

difference(*other*)

Return indices in this fingerprint but not in `other`.

Return type numpy.ndarray

Raises

- **InvalidFingerprintType** – If the informed fingerprint is not an instance of *Fingerprint*.
- **BitsValueError** – If the fingerprints have different lengths.

fold(*new_length=4096*)

Fold this fingerprint to size `new_length`.

Parameters **new_length** (*int*) – Length of the new fingerprint, ideally multiple of 2. The default value is 4096.

Returns Folded *Fingerprint*.

Return type *Fingerprint*

Raises **BitsValueError** – If the new fingerprint length is not a multiple of 2 or is greater than the existing fingerprint length.

Examples

```
>>> from luna.interaction.fp.fingerprint import Fingerprint
>>> import numpy as np
>>> np.random.seed(0)
>>> on_bits = 8
>>> fp_length = 32
>>> indices = np.random.randint(0, fp_length, on_bits)
>>> print(indices)
[12 15 21  0  3 27  3  7]
>>> fp = Fingerprint.from_indices(indices, fp_length=fp_length)
>>> print(fp.indices)
[ 0  3  7 12 15 21 27]
>>> print(fp.to_vector(compressed=False))
[1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0]
>>> folded_fp = fp.fold(8)
>>> print(folded_fp.indices)
[0 3 4 5 7]
>>> print(folded_fp.to_vector(compressed=False))
[1 0 0 1 1 1 0 1]
```

property fp_length

The fingerprint length (total number of bits).

Type int, read-only

classmethod from_bit_string(*bit_string*, *fp_length=None*, ***kwargs*)

Initialize from a bit string (e.g. '0010100110').

Parameters

- **bit_string** (*str*) – String of 0s and 1s.
- **fp_length** (*int, optional*) – The fingerprint length (total number of bits). If not provided, the fingerprint length will be defined based on the string length.
- ****kwargs** (*dict, optional*) – Extra arguments to *Fingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *Fingerprint*

Examples

```
>>> from luna.interaction.fp.fingerprint import Fingerprint
>>> fp = Fingerprint.from_bit_string("0010100110000010")
>>> print(fp.indices)
[ 2  4  7  8 14]
>>> print(fp.fp_length)
16
```

classmethod from_fingerprint(*fp*, ***kwargs*)
Initialize from an existing fingerprint.

Parameters

- **fp** (*Fingerprint*) – An existing fingerprint.
- ****kwargs** (*dict, optional*) – Extra arguments to *Fingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *Fingerprint*

classmethod from_indices(*indices*, *fp_length=4294967296*, ***kwargs*)
Initialize from an array of indices.

Parameters

- **indices** (*array_like of int*) – Indices of "on" bits.
- **fp_length** (*int*) – The fingerprint length (total number of bits). The default value is 2^{32}.
- ****kwargs** (*dict, optional*) – Extra arguments to *Fingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *Fingerprint*

Examples

```
>>> from luna.interaction.fp.fingerprint import Fingerprint
>>> import numpy as np
>>> np.random.seed(0)
>>> on_bits = 8
>>> fp_length = 32
>>> indices = np.random.randint(0, fp_length, on_bits)
>>> print(indices)
[12 15 21  0  3 27  3  7]
>>> fp = Fingerprint.from_indices(indices, fp_length=fp_length)
>>> print(fp.indices)
[ 0  3  7 12 15 21 27]
>>> print(fp.to_vector(compressed=False))
[1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0]
```

classmethod from_rdkit(*rdkit_fp*, ***kwargs*)
Initialize from an RDKit fingerprint.

Parameters

- **rdkit_fp** (ExplicitBitVect or SparseBitVect) – An existing RDKit fingerprint.

- ****kwargs** (*dict, optional*) – Extra arguments to *Fingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *Fingerprint*

classmethod from_vector(*vector*, *fp_length=None*, ***kwargs*)
Initialize from a vector.

Parameters

- **vector** (numpy.ndarray or scipy.sparse.csr_matrix) – Array of bits.
- **fp_length** (*int, optional*) – The fingerprint length (total number of bits). If not provided, the fingerprint length will be defined based on the `vector` shape.
- ****kwargs** (*dict, optional*) – Extra arguments to *Fingerprint*. Refer to the documentation for a list of all possible arguments.

Return type *Fingerprint*

Examples

```
>>> from luna.interaction.fp.fingerprint import Fingerprint
>>> import numpy as np
>>> np.random.seed(0)
>>> fp_length = 32
>>> vector = np.random.choice([0, 1], size=(fp_length,), p=[0.8, 0.2])
>>> print(vector)
[0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0]
>>> fp = Fingerprint.from_vector(vector)
>>> print(fp.indices)
[ 7  8 13 17 19 20 27]
>>> print(fp.fp_length)
32
```

get_bit(*index*)
Get the bit/count value at index `index`.

Raises **BitsValueError** – If the provided index is in a different bit scale.

get_num_bits()
Get the fingerprint length (total number of bits).

get_num_off_bits()
Get the number of "off" bits.

get_num_on_bits()
Get the number of "on" bits.

get_on_bits()
Get "on" bits.

Return type numpy.ndarray

get_prop(*key*)
Get value of the property `key`. If not set, raise KeyError.

property indices
Indices of "on" bits.

Type array_like of int, read-only

intersection(*other*)
Return the intersection between indices of two fingerprints.

Return type numpy.ndarray

Raises

- **InvalidFingerprintType** – If the informed fingerprint is not an instance of *Fingerprint*.
- **BitsValueError** – If the fingerprints have different lengths.

property name
The property 'name'. If it was not provided, then return an empty string.

Type str

property num_levels
The property 'num_levels' used to generate this fingerprint (see *ShellGenerator*). If it was not provided, then return None.

Type int

property num_shells
The property 'num_shells' (see *ShellGenerator*). If it was not provided, then return None.

Type int

property props
The custom properties of the fingerprint.

Type dict, read-only

property radius_step
The property 'radius_step' used to generate this fingerprint (see *ShellGenerator*). If it was not provided, then return None.

Type float

set_prop(*key*, *value*)
Set value to the property `key`.

symmetric_difference(*other*)
Return indices in either this fingerprint or `other` but not both.

Return type numpy.ndarray

Raises

- **InvalidFingerprintType** – If the informed fingerprint is not an instance of *Fingerprint*.
- **BitsValueError** – If the fingerprints have different lengths.

to_bit_string()
Convert this fingerprint to a string of bits.

> **Warning:** This function may raise a *MemoryError* exception when using huge indices vectors. If you found this issue, you may want to try a different data type or apply a folding operation before calling *to_bit_string*.

Return type str

Raises `MemoryError` – If the operation ran out of memory.

`to_bit_vector`(*compressed=True*)
Convert this fingerprint to a vector of bits.

> **Warning:** This function may raise a *MemoryError* exception when using huge indices vectors. If you found this issue, you may want to try a different data type or apply a folding operation before calling *to_bit_vector*.

Parameters **compressed** (*bool*) – If True, build a compressed sparse matrix (scipy.sparse.csr_matrix).

Returns Vector of bits/counts. Return a compressed sparse matrix (`scipy.sparse.csr_matrix`) if `compressed` is True. Otherwise, return a Numpy array (`numpy.ndarray`)

Return type `numpy.ndarray` or `scipy.sparse.csr_matrix`

Raises

- **`BitsValueError`** – If some of the fingerprint indices are greater than the fingerprint length.
- `MemoryError` – If the operation ran out of memory.

`to_rdkit`(*rdkit_fp_cls=None*)
Convert this fingerprint to an RDKit fingerprint.

Note: If the fingerprint length exceeds the maximum RDKit fingerprint length ($2^{31} - 1$), this fingerprint will be folded to length $2^{31} - 1$ before conversion.

Returns If `fp_length` is less than $1e5$, `ExplicitBitVect` is used. Otherwise, `SparseBitVect` is used.

Return type `ExplicitBitVect` or `SparseBitVect`

`to_vector`(*compressed=True*, *dtype=<class 'numpy.int32'>*)
Convert this fingerprint to a vector of bits/counts.

> **Warning:** This function may raise a *MemoryError* exception when using huge indices vectors. If you found this issue, you may want to try a different data type or apply a folding operation before calling *to_vector*.

Parameters

- **compressed** (*bool*) – If True, build a compressed sparse matrix (scipy.sparse.csr_matrix).
- **dtype** (*data-type*) – The default value is np.int32.

Returns Vector of bits/counts. Return a compressed sparse matrix (`scipy.sparse.csr_matrix`) if `compressed` is True. Otherwise, return a Numpy array (`numpy.ndarray`)

Return type `numpy.ndarray` or `scipy.sparse.csr_matrix`

Raises

- **BitsValueError** – If some of the fingerprint indices are greater than the fingerprint length.
- **MemoryError** – If the operation ran out of memory.

unfold()
Unfold this fingerprint and return its parent fingerprint.

Return type *Fingerprint*

property unfolded_fp
The unfolded version of this fingerprint. If None, this fingerprint may have not been folded yet.

Type *Fingerprint* or None, read-only

property unfolded_indices
Indices of "on" bits in the unfolded fingerprint.

Type array_like of int, read-only

property unfolding_map
The mapping between current indices and indices from the unfolded version of this fingerprint what makes it possible to trace folded bits back to the original shells (features).

Type dict, read-only

union(*other*)
Return the union of indices of two fingerprints.

Return type numpy.ndarray

Raises

- **InvalidFingerprintType** – If the informed fingerprint is not an instance of *Fingerprint*.
- **BitsValueError** – If the fingerprints have different lengths.

luna.interaction.fp.shell module

class CompoundClassIds(*value*)
Bases: enum.Enum

An enumeration of compound classes.

HETATM = 1

NUCLEOTIDE = 3

RESIDUE = 2

UNKNOWN = 5

WATER = 4

class Shell(*central_atm_grp*, *level*, *radius*, *neighborhood=None*, *inter_tuples=None*, *diff_comp_classes=True*, *dtype=<class 'numpy.int64'>*, *seed=0*, *manager=None*, *valid=True*, *feature_mapper=None*)
Bases: object

A container to store substructural information, which is the base for LUNA fingerprints.

Shells are centered on an atom or atom group (*AtomGroup* objects) and represent all atoms and interactions explicitly within it.

Parameters

- **central_atm_grp** (*AtomGroup*) – The shell center.
- **level** (*int*) – The level (iteration) at which the shell was generated.
- **radius** (*float*) – The shell radius.
- **neighborhood** (iterable of *AtomGroup*) – All atoms and atom groups within a shell of radius `radius` centered on `central_atm_grp`.
- **inter_tuples** (iterable of (*InteractionType*, *AtomGroup*)) – All interactions within a shell of radius `radius` centered on `central_atm_grp`. Each tuple contains an *InteractionType* object and one of the *AtomGroup* objects participating to the interaction.

Note: As an interaction involves two participants, it would be expected that each interaction produces two tuples. However, by default, ShellGenerator sorts atom groups and considers only the first tuple that appears, which guarantees that only one of the possible tuples is added to avoid information duplication.

- **diff_comp_classes** (*bool*) – If True (the default), include differentiation between compound classes. That means structural information originated from *AtomGroup* objects belonging to residues, nucleotides, ligands, or water molecules will be considered different even if their structural information are the same. This is useful for example to differentiate protein-ligand interactions from residue-residue ones.
- **dtype** (*data-type*) – Use arrays of type `dtype` to store information. The default value is np.int64.
- **seed** (*int*) – A seed to generate shell identifiers through the MurmurHash3 hash function. The default value is 0.
- **manager** (*ShellManager*) – The *ShellManager* object that stores and controls this *Shell* object.
- **valid** (*bool*) – If the shell is valid or not. By default, all shells are considered valid.
- **feature_mapper** (*dict, optional*) – A dict that maps atoms and interactions to unique values. If not provided, `feature_mapper` will inherit from the default mappings `CHEMICAL_FEATURE_IDS` and `INTERACTION_IDS`.

Variables

- **~Shell.central_atm_grp** (*AtomGroup*) –
- **~Shell.level** (*int*) –
- **~Shell.radius** (*float*) –
- **~Shell.diff_comp_classes** (*bool*) –
- **~Shell.dtype** (*data-type*) –
- **~Shell.seed** (*int*) –
- **~Shell.valid** (*bool*) –
- **~Shell.feature_mapper** (*dict*) –

property encoded_data

The data encoded in this shell.

Type iterable of tuple, read-only

hash_shell()
Hash this shells' substructural information into a 32-bit integer using MurmurHash3.

Returns A 32-bit integer representing this shell's substructural information.

Return type int

property identifier
This shell identifier, which is generated by hashing its encoded data with a hash function. By default, LUNA uses MurmurHash3 as the hash function.

Type int, read-only

property inter_tuples
Each tuple contains an *InteractionType* object and one of the *AtomGroup* objects participating to the interaction.

Type iterable of tuple, read-only

property interactions
All interactions within this shell.

Type iterable of *InteractionType*, read-only

is_similar(*shell*)
If this shell is similar to `shell`.

Two shells are similar if they represent the same substructural information.

Parameters **shell** (*Shell*)

Return type bool

is_valid()
If the shell is valid or not.

Return type bool

property manager
The *ShellManager* object that stores and controls this *Shell* object.

Type *ShellManager*, read-only

property neighborhood
All atoms and atom groups within this shell.

Type iterable of *AtomGroup*, read-only

property previous_shell
The previous shell, i.e., a shell centered on the same central *AtomGroup* object from a previous level. For example, if this shell is in level 5, return a shell from level 4 having the same center.

Type *Shell*, read-only

class ShellGenerator(*num_levels*, *radius_step*, *fp_length=4294967296*, *ifp_type=IFPType.EIFP*, *diff_comp_classes=True*, *dtype=<class 'numpy.int64'>*, *seed=0*, *bucket_size=10*)
Bases: object

Generate shells, the base information of LUNA fingerprints.

Parameters

- **num_levels** (*int*) – The maximum number of iterations for fingerprint generation.

- **radius_step** (*float*) – The multiplier used to increase shell size at each iteration. At iteration 0, shell radius is 0 * `radius_step`, at iteration 1, radius is 1 * `radius_step`, etc.
- **fp_length** (*int*) – The fingerprint length (total number of bits). The default value is 2^{32}.
- **ifp_type** (*IFPType*) – The fingerprint type (EIFP, FIFP, or HIFP). The default value is EIFP.
- **diff_comp_classes** (*bool*) – If True (the default), include differentiation between compound classes. That means structural information originated from *AtomGroup* objects belonging to residues, nucleotides, ligands, or water molecules will be considered different even if their structural information are the same. This is useful for example to differentiate protein-ligand interactions from residue-residue ones.
- **dtype** (*data-type*) – Use arrays of type `dtype` to store information. The default value is np.int64.
- **seed** (*int*) – A seed to generate shell identifiers through the MurmurHash3 hash function. The default value is 0.
- **bucket_size** (*int*) – Bucket size of KD tree. You can play around with this to optimize speed if you feel like it. The default value is 10.

Variables

- **~ShellGenerator.num_levels** (*int*) –
- **~ShellGenerator.radius_step** (*float*) –
- **~ShellGenerator.fp_length** (*int*) –
- **~ShellGenerator.ifp_type** (*IFPType*) –
- **~ShellGenerator.diff_comp_classes** (*bool*) –
- **~ShellGenerator.dtype** (*data-type*) –
- **~ShellGenerator.seed** (*int*) –
- **~ShellGenerator.bucket_size** (*int*) –

Examples

In the below example, we will assume a LUNA project object named `proj_obj` already exists.

First, let's define a *ShellGenerator* object that will create shells over 2 iterations (levels). At each iteration, the shell radius will be increased by 3 and substructural information will be encoded following EIFP definition.

```
>>> from luna.interaction.fp.shell import ShellGenerator
>>> from luna.interaction.fp.type import IFPType
>>> num_levels, radius_step = 2, 3
>>> sg = ShellGenerator(num_levels, radius_step, ifp_type=IFPType.EIFP)
```

After defining the generator, we can create shells by calling *create_shells()*, which expects an *AtomGroupsManager* object. In this example, we will the first *AtomGroupsManager* object from an existing LUNA project (`proj_obj`).

```
>>> atm_grps_mngr = list(proj_obj.atm_grps_mngrs)[0]
>>> sm = sg.create_shells(atm_grps_mngr)
>>> print(sm.num_shells)
528
```

Now, with shells stored in the *ShellManager* object you can, for instance:

- Generate fingerprints:

```
>>> fp = sm.to_fingerprint(fold_to_length=1024)
>>> print(fp.indices)
[   2   19   22   23   34   37   39   45   54   67   71   75   83   84
   93  109  138  140  157  162  181  182  186  187  191  194  206  209
  211  237  246  251  263  271  281  296  304  315  323  358  370  374
  388  392  399  400  419  439  476  481  487  509  519  527  532  578
  587  592  604  605  629  635  645  661  668  698  711  713  732  736
  740  753  764  795  813  815  820  824  825  831  836  855  873  882
  911  926  967  975  976  984  990  996 1020]
```

- Visualize substructural information in Pymol:

```
>>> from luna.interaction.fp.view import ShellViewer
>>> shell_tuples = [(atm_grps_mngr.entry, sm.unique_shells, proj_obj.pdb_path)]
>>> sv = ShellViewer()
>>> sv.new_session(shell_tuples, "example.pse")
```

create_shells(*atm_grps_mngr*)

Perceive substructural information from *AtomGroup* objects and their interactions, and represent such information as shells.

Parameters **atm_grps_mngr** (*AtomGroupsManager*) – Container of *AtomGroup* objects and their interactions.

Return type *ShellManager*

Raises **ShellCenterNotFound** – If it fails to recover a shell having a given center.

class ShellManager(*num_levels*, *radius_step*, *fp_length*, *ifp_type*, *shells=None*, *verbose=False*)

Bases: object

Store and manage *Shell* objects.

Parameters

- **num_levels** (*int*) – The maximum number of iterations for fingerprint generation.
- **radius_step** (*float*) – The multiplier used to increase shell size at each iteration. At iteration 0, shell radius is 0 * `radius_step`, at iteration 1, radius is 1 * `radius_step`, etc.
- **fp_length** (*int*) – The fingerprint length (total number of bits).
- **ifp_type** (*IFPType*) – The fingerprint type (EIFP, FIFP, or HIFP).
- **shells** (iterable of *Shell*, optional) – An initial sequence of *Shell* objects (fingerprint features).
- **verbose** (*bool*) – If True, warnings issued during the usage of this *ShellManager* will be displayed. The default value is False.

Variables

- **~ShellManager.num_levels** (*int*) – The maximum number of iterations for fingerprint generation.
- **~ShellManager.radius_step** (*float*) – The multiplier used to increase shell size at each iteration.
- **~ShellManager.fp_length** (*int*) – The fingerprint length (total number of bits).
- **~ShellManager.ifp_type** (*IFPType*) – The fingerprint type (EIFP, FIFP, or HIFP).

- **~ShellManager.shells** (iterable of *Shell*) – The sequence of shells (fingerprint features).
- **~ShellManager.verbose** (*bool*) – The verbosity state.
- **~ShellManager.version** (*str*) – The LUNA's version with which shells were generated.
- **~ShellManager.levels** (dict of {int: list of *Shell*}) – Register shells by level, where keys are levels and values are lists of *Shell* objects.

  Note: Levels are 0-indexed. So, the first level is 0, second is 1, etc. That means if `num_levels` is 5, the last level will be 4.

- **~ShellManager.centers** (dict of dict of {int: *Shell*}) – Register shells by center, where keys are *AtomGroup* objects and values are dict that store all shells generated for that center at each iteration (level).

add_shell(*shell*)
Add a new shell to `shells`.

Parameters **shell** (*Shell*)

find_similar_shell(*shell*)
Find a shell in `shells` similar to `shell`.

Two shells are similar if they represent the same substructural information.

Parameters **shell** (*Shell*)

Returns Return a similar shell or None if it does not find any.

Return type *Shell* or None

get_identifiers(*level=None*, *unique_shells=False*)
Get all shells' identifier.

Parameters

- **level** (*int, optional*) – If provided, only return identifiers of shells at level `level`.
- **unique_shells** (*bool*) – If True, ignore identifiers of non-valid shells. The default value is False.

Return type list of int

get_last_shell(*center*, *unique_shells=False*)
Get the last shell generated for center `center`.

Parameters

- **center** (*AtomGroup*) – The center of a shell, which consists of an *AtomGroup* object.
- **unique_shells** (*bool*) – If True, ignore non-valid shells. That means shells generated at superior levels may be ignored if they are not valid. The default value is False.

Returns The last shell generated for center `center` or None if no valid shell was found.

Return type *Shell* or None

get_previous_shell(*center*, *curr_level*, *unique_shells=False*)
Get the last shell having center `center` that was generated before level `curr_level`. For instance, if the current level (iteration) is 5 and the last valid shell generated for center C was at level 4, then *get_previous_shell()* would return that shell at level 4.

Parameters

- **center** (*AtomGroup*) – The center of a shell, which consists of an *AtomGroup* object.
- **curr_level** (*int*) – The current level (iteration).
- **unique_shells** (*bool*) – If True, ignore non-valid shells and go down to inferior levels until a valid shell is found. If level 0 was reached and no valid shell was found, then return None. The default value is False.

Returns The first previous valid shell or None if no valid shell was found.

Return type *Shell* or None

get_shell_by_center_and_level(*center*, *level*, *unique_shells=False*)
Get the shell generated for center `center` (*AtomGroup* object) at level (iteration) `level`.

Parameters

- **center** (*AtomGroup*) – The center of a shell, which consists of an *AtomGroup* object.
- **level** (*int*) – The target level (iteration).
- **unique_shells** (*bool*) – If True, return the *Shell* object if it is unique and None otherwise. The default value is False.

Returns The shell generated for center `center` at level `level`. If the *Shell* object is not unique, return None.

Return type *Shell* or None

get_shells_by_center(*center*, *unique_shells=False*)
Get shells by center (*AtomGroup* object).

Parameters

- **center** (*AtomGroup*) – The center of a shell, which consists of an *AtomGroup* object.
- **unique_shells** (*bool*) – If True, return only unique shells. Otherwise, return all shells generated for center `center` (the default).

Returns All shells generated for center `center` at each iteration (key).

Return type dict of {int: *Shell*}

get_shells_by_identifier(*identifier*, *unique_shells=False*)
Get shells by identifier.

Parameters

- **identifier** (*int*) – The shell identifier.
- **unique_shells** (*bool*) – If True, return only unique shells. Otherwise, return all shells having the identifier `identifier` (the default).

Return type list of *Shell*

get_shells_by_level(*level*, *unique_shells=False*)
Get shells by level (iteration number).

Parameters

- **level** (*int*)
- **unique_shells** (*bool*) – If True, return only unique shells. Otherwise, return all shells at level `level` (the default).

Return type list of *Shell*

`get_valid_shells()`
Return only valid shells.

A shell is considered invalid if, by the time it is added in `shells`, there is another shell representing the same substructural information. That means this shell is not unique and does not contributes to any new information.

On the other hand, if the shell contributes by adding new information to `shells`, then it will be considered valid and unique. So, the first shell of a series of shells containing the same information is considered valid and the others invalid.

Return type list of *Shell*

`property num_shells`
Total number of shells in `shells`.

Type int, read-only

`property num_unique_shells`
Total number of unique shells in `shells`.

Type int, read-only

`to_fingerprint`(*fold_to_length=None*, *count_fp=False*, *unique_shells=False*)
Encode shells into an interaction fingerprint.

Parameters

- **fold_to_length** (*int, optional*) – If provided, fold the fingerprint to length `fold_to_length`.
- **count_fp** (*bool*) – If True, create a count fingerprint (*CountFingerprint*). Otherwise, return a bit fingerprint (*Fingerprint*).
- **unique_shells** (*bool*) – If True, only unique shells are used to create the fingerprint. The default value is False.

Return type *CountFingerprint* or *Fingerprint*

`trace_back_feature`(*feature_id*, *ifp*, *unique_shells=False*)
Trace a feature from a fingerprint back to the shells that originated that feature.

Note: Due to fingerprint folding, multiple substructures may end up encoded in the same bit, the so-called collision problem. So, if the provided feature contains collisions, shells representing different substructures may be returned by *trace_back_feature()*.

Parameters

- **feature_id** (*int*) – The target feature id.
- **ifp** (`Fingerprint`) – The fingerprint containing the feature `feature_id`.
- **unique_shells** (*bool*) – If True, ignore identifiers of non-valid shells. The default value is False.

Yields *Shell*

Examples

In the below example, we will assume a LUNA project object named `proj_obj` already exists. Then, we will generate an EIFP fingerprint for the first *AtomGroupsManager* object at `proj_obj`.

```
>>> from luna.interaction.fp.shell import ShellGenerator
>>> from luna.interaction.fp.type import IFPType
>>> atm_grps_mngr = list(proj_obj.atm_grps_mngrs)[0]
>>> num_levels, radius_step = 2, 3
>>> sg = ShellGenerator(num_levels, radius_step, ifp_type=IFPType.EIFP)
>>> sm = sg.create_shells(atm_grps_mngr)
>>> fp = sm.to_fingerprint(fold_to_length=1024, count_fp=True)
>>> print(fp.indices)
[   2   19   22   23   34   37   39   45   54   67   71   75   83   84
   93  109  138  140  157  162  181  182  186  187  191  194  206  209
  211  237  246  251  263  271  281  296  304  315  323  358  370  374
  388  392  399  400  419  439  476  481  487  509  519  527  532  578
  587  592  604  605  629  635  645  661  668  698  711  713  732  736
  740  753  764  795  813  815  820  824  825  831  836  855  873  882
  911  926  967  975  976  984  990  996 1020]
```

Now, we can trace features back to original identifiers and investigate its substructural information.

```
>>> ori_indices = list(sm.trace_back_feature(34, fp, unique_shells=True))
>>> print(ori_indices)
[(494318626, [<Shell: level=0, radius=0.000000, center=<AtomGroup: [
↪<ExtendedAtom: 3QQK/0/A/GLN/85/CD>, <ExtendedAtom: 3QQK/0/A/GLN/85/NE2>,
↪<ExtendedAtom: 3QQK/0/A/GLN/85/OE1>]>, interactions=0>])]
```

property unique_shells

Unique shells. Return the same as *get_valid_shells()*.

Type iterable of *Shell*, read-only

luna.interaction.fp.type module

class IFPType(*value*)

Bases: enum.Enum

An enumeration of Interaction FingerPrints (IFPs) available in LUNA.

EIFP = 1

FIFP = 3

HIFP = 2

luna.interaction.fp.view module

class ShellViewer(*show_cartoon=False*, *bg_color='white'*, *add_directional_arrows=True*, *show_res_labels=True*, *inter_color=<luna.util.ColorPallete object>*, *pse_export_version='1.8'*)

Bases: *luna.wrappers.pymol.PymolSessionManager*

Class that inherits from *PymolSessionManager* and implements *set_view()* to depict shells (IFP features) and interactions in Pymol and save the view as a Pymol session.

This class can be used to visualize multiple complexes into the same Pymol session, for instance, to compare binding modes and analyze similar shells. To do so, it is recommended that the protein structures are in the same coordinate system, i.e., they should be aligned first. This can be achieved with `luna.align.tmalign.align_2struct()` or any other tool of your preference.

Examples

In the below example, we will assume a LUNA project object named `proj_obj` already exists.

To visualize shells, we first need to generate them. So, let's define a *ShellGenerator* object that will create shells over 2 iterations (levels). At each iteration, the shell radius will be increased by 3 and substructural information will be encoded following EIFP definition. Here, as an example, we will generate shells for the first *AtomGroupsManager* object at `proj_obj`.

```
>>> from luna.interaction.fp.shell import ShellGenerator
>>> from luna.interaction.fp.type import IFPType
>>> num_levels, radius_step = 2, 3
>>> sg = ShellGenerator(num_levels, radius_step, ifp_type=IFPType.EIFP)
>>> atm_grps_mngr = list(proj_obj.atm_grps_mngrs)[0]
>>> sm = sg.create_shells(atm_grps_mngr)
```

The function *create_shells()* returns a *ShellManager* object, which provides built-in methods to access the created shells. Therefore, you can interact with it to select the shells you want to visualize in Pymol. As an example, let's select all unique shells at the last level:

```
>>> shells = sm.get_shells_by_level(num_levels - 1, unique_shells=True)
```

Note: Levels are 0-indexed. So, the first level is 0, second is 1, etc. That means if `num_levels` is 5, the last level will be 4.

As *ShellViewer* expects a list of tuples, we will define it now. The first item of the tuple is the *Entry* instance, which represents a ligand. This can be obtained directly from the *AtomGroupsManager* object. The second item is the list of shells you want to visualize. Finally, the third item can be either a PDB file or a directory. In this example, we will use the PDB directory defined during the LUNA project initialization.

```
>>> shell_tuples = [(atm_grps_mngr.entry, shells, proj_obj.pdb_path)]
```

To finish, we now create a new *ShellViewer* object and call *new_session()*, which will initialize the session, depict shells and interactions, and save the session to an output PSE file.

```
>>> from luna.interaction.fp.view import ShellViewer
>>> sv = ShellViewer()
>>> sv.new_session(shell_tuples, "example.pse")
```

set_view(*shell_tuples*)
Depict shells (IFP features) and interactions into the current Pymol session.

Parameters **shell_tuples** (*iterable of tuple*) – Each tuple must contain three items: an *Entry* instance, an iterable of *Shell*, and a PDB file or the directory where the PDB file is located.

Module contents

Module contents

luna.mol package

Submodules

luna.mol.atom module

class AtomData(*atomic_num*, *coord*, *bond_type*, *serial_number=None*)
Bases: object

Store atomic data (atomic number, coordinates, bond type, and serial number).

Parameters

- **atomic_num** (*int*) – Atomic number.
- **coord** (*array_like of float (size 3)*) – Atomic coordinates (x, y, z).
- **bond_type** (*int*) – Bond type.
- **serial_number** (*int, optional*) – Atom serial number.

Variables

- **~AtomData.atomic_num** (*int*) – The atomic number.
- **~AtomData.bond_type** (*int*) – The bond type.
- **~AtomData.serial_number** (*int or None*) – The atom serial number.

property coord
The atomic coordinates (x, y, z).

Type array_like of float (size 3)

property x
The orthogonal coordinates for x in Angstroms.

Type float

property y
The orthogonal coordinates for y in Angstroms.

Type float

property z
The orthogonal coordinates for z in Angstroms.

Type float

class ExtendedAtom(*atom*, *nb_info=None*, *atm_grps=None*, *invariants=None*)

Bases: object

Extend Atom with additional properties and methods.

Parameters

- **atom** (Atom) – An atom.
- **nb_info** (iterable of *AtomData*, optional) – A sequence of *AtomData* containing information about atoms covalently bound to atom.
- **atm_grps** (iterable of AtomGroup, optional) – A sequence of atom groups that contain atom.
- **invariants** (*list or tuple, optional*) – Atomic invariants.

add_atm_grps(*atm_grps*)

Add AtomGroup objects to atm_grps.

add_nb_info(*nb_info*)

Add *AtomData* objects to neighbors_info.

as_json()

Represent the atom as a dict containing the structure id, model id, chain id, residue name, residue id, and atom name.

The dict is defined as follows:

- pdb_id (str): structure id;
- model (str): model id;
- chain (str): chain id;
- res_name (str): residue name;
- res_id (tuple): residue id (hetflag, sequence identifier, insertion code);
- name (tuple): atom name (atom name, alternate location).

property atm_grps

The list of atom groups that contain atom.

To add or remove atom groups from atm_grps use *add_atm_grps()* or *remove_atm_grps()*, respectively.

Type list of AtomGroup, read-only

property atom

Atom, read-only.

property electronegativity

The Pauling electronegativity for this atom. This information is obtained from Open Babel.

Type float, read-only

property full_atom_name

The full name of an atom is composed by the structure id, model id, chain id, residue name, residue id, atom name, and alternate location if available. Fields are slash-separated.

Type str, read-only

property full_id

The full id of an atom is the tuple (structure id, model id, chain id, residue id, atom name, alternate location).

Type tuple, read-only

get_neighbor_info(*atom*)
Get information from a covalently bound atom.

property invariants
The list of atomic invariants.

Type list

is_neighbor(*atom*)
Check if a given atom is covalently bound to it.

property neighbors_info
The list of *AtomData* containing information about atoms covalently bound to `atom`.

To add or remove neighbors information from `neighbors_info` use *add_nb_info()* or *remove_nb_info()*, respectively.

Type list of *AtomData*, read-only

remove_atm_grps(*atm_grps*)
Remove `AtomGroup` objects from `atm_grps`.

remove_nb_info(*nb_info*)
Remove *AtomData* objects from `neighbors_info`.

luna.mol.charge_model module

class ChargeModel
Bases: `object`

Implementation of a charge model.

get_charge()

class OpenEyeModel
Bases: *luna.mol.charge_model.ChargeModel*

Implementation of OpenEye charge model.

get_charge(*atm_obj*)
Get the formal charge for atom `atom_obj`.

Currently, only formal charges for the elements Hydrogen, Carbon, Nitrogen, Oxygen, Phosphorus, Sulfur, Chlorine, Fluorine, Bromine, Iodine, Magnesium, Calcium, Zinc, Lithium, Sodium, Potassium, and Boron can be recovered.

Parameters **atm_obj** (*AtomWrapper*) – The target atom.

Examples

```
>>> from luna.mol.charge_model import OpenEyeModel
>>> from luna.wrappers.base import MolWrapper
>>> cm = OpenEyeModel()
>>> mol_obj = MolWrapper.from_smiles("[NH3+]CC([O-])=O")
>>> for atm_obj in mol_obj.get_atoms():
>>>     formal_charge = cm.get_charge(atm_obj)
>>>     print("Charge for atom #%d (%s): %d" % (atm_obj.get_idx(),
...                                            atm_obj.get_symbol(),
...                                            formal_charge))
```

(continues on next page)

(continued from previous page)

```
Charge for atom #0 (N): 1.
Charge for atom #1 (C): 0.
Charge for atom #2 (C): 0.
Charge for atom #3 (O): -1.
Charge for atom #4 (O): 0.
```

luna.mol.clustering module

available_similarity_functions()
Return a list of all similarity metrics available at RDKit.

calc_distance_matrix(*fps*, *similarity_func='BulkTanimotoSimilarity'*)

Calculate the pairwise distance (dissimilarity) between fingerprints in fps using the similarity metric similarity_func.

Parameters

- **fps** (iterable of RDKit ExplicitBitVect or SparseBitVect) – A sequence of fingerprints.
- **similarity_func** (*str*) – A similarity metric to calculate the distance between the provided fingerprints. The default value is 'BulkTanimotoSimilarity'.

  To check out the list of available similarity metrics, call the function *available_similarity_functions()*.

Examples

First, let's define a set of molecules.

```
>>> from luna.wrappers.base import MolWrapper
>>> mols = [MolWrapper.from_smiles("CCCCCC").unwrap(),
...         MolWrapper.from_smiles("CCCCCCCC").unwrap(),
...         MolWrapper.from_smiles("CCCCCCCCO").unwrap()]
```

Now, we generate fingerprints for those molecules.

```
>>> from luna.mol.fingerprint import generate_fp_for_mols
>>> fps = [d["fp"] for d in generate_fp_for_mols(mols, "morgan_fp")]
```

Finally, calculate the distance between the molecules based on their fingerprints.

```
>>> from luna.mol.clustering import calc_distance_matrix
>>> print(calc_distance_matrix(fps))
[0.125, 0.46153846153846156, 0.3846153846153846]
```

Returns **distances** – Flattened diagonal matrix.

Return type list of float

cluster_fps(*fps*, *cutoff=0.2*, *similarity_func='BulkTanimotoSimilarity'*)
Clusterize molecules based on fingerprints using the Butina clustering algorithm.

Parameters

- **fps** (iterable of RDKit `ExplicitBitVect` or `SparseBitVect`) – A sequence of fingerprints.
- **cutoff** (*float*) – Elements within this range of each other are considered to be neighbors.
- **similarity_func** (*str*) – A similarity metric to calculate the distance between the provided fingerprints. The default value is 'BulkTanimotoSimilarity'.

  To check out the list of available similarity metrics, call the function *available_similarity_functions()*.

Examples

First, let's define a set of molecules.

```
>>> from luna.wrappers.base import MolWrapper
>>> mols = [MolWrapper.from_smiles("CCCCCC").unwrap(),
...         MolWrapper.from_smiles("CCCCCCCC").unwrap(),
...         MolWrapper.from_smiles("CCCCCCCCO").unwrap()]
```

Now, we generate fingerprints for those molecules.

```
>>> from luna.mol.fingerprint import generate_fp_for_mols
>>> fps = [d["fp"] for d in generate_fp_for_mols(mols, "morgan_fp")]
```

Finally, clusterize the molecules based on their fingerprints.

```
>>> from luna.mol.clustering import cluster_fps
>>> print(cluster_fps(fps, cutoff=0.2))
((1, 0), (2,))
```

Returns **clusters** – Each cluster is defined as a tuple of tuples, where the first element for each cluster is its centroid.

Return type tuple of tuples

luna.mol.depiction module

class PharmacophoreDepiction(*feature_extractor=None*, *colors=<luna.util.ColorPallete object>*, *format='png'*, *fig_size=(800, 800)*, *font_size=0.5*, *circle_dist=0.2*, *circle_radius=0.3*, *use_bw_atom_palette=True*)

Bases: `object`

Draw molecules and depict pharmacophoric properties as colored circles.

Parameters

- **feature_extractor** (`FeatureExtractor`) – Perceive pharmacophoric properties from molecules.
- **colors** (*ColorPallete*) – Color scheme for pharmacophoric properties perceived by `feature_extractor`. The default value is `ATOM_TYPES_COLOR`.
- **format** (*{'png', 'svg'}*) – The output file format. The default value is 'png'.
- **figsize** (*tuple of (float, float)*) – Width and height in inches. The default value is (800, 800).
- **font_size** (*float*) – The font size. The units are, roughly, pixels. The default value is 0.5.

- **circle_dist** (*float*) – Distance between circles (pharmacophoric properties). The default value is 0.2.
- **circle_radius** – Circles' radius size of pharmacophoric properties. The default value is 0.3.
- **use_bw_atom_palette** (*bool*) – Use a black & white palette for atoms and bonds.

Examples

First, let's read a molecule (glutamine).

```
>>> from luna.wrappers.base import MolWrapper
>>> mol = MolWrapper.from_smiles("N[C@@H](CCC(N)=O)C(O)=O")
```

Now, create a feature factory and instantiate a new FeatureExtractor object.

```
>>> from luna.util.default_values import ATOM_PROP_FILE
>>> from rdkit.Chem import ChemicalFeatures
>>> from luna.mol.features import FeatureExtractor
>>> feature_factory = ChemicalFeatures.BuildFeatureFactory(ATOM_PROP_FILE)
>>> feature_extractor = FeatureExtractor(feature_factory)
```

Instantiate a new PharmacophoreDepiction object with the desired configuration. For example, you can provide a color scheme for pharmacophoric properties, the image size, and its format.

```
>>> from luna.util.default_values import ATOM_TYPES_COLOR
>>> from luna.mol.depiction import PharmacophoreDepiction
pd = PharmacophoreDepiction(feature_extractor=feature_extractor, colors=ATOM_TYPES_
↪COLOR,
                            fig_size=(500, 500), format="svg")
```

Finally, you can draw the molecule with annotated pharmacophoric properties.

```
>>> pd.plot_fig(mol, "output.svg")
```

plot_fig(*mol_obj*, *output=None*, *atm_types=None*, *legend=None*)

Draw the molecule `mol_obj` and depict its pharmacophoric properties.

Parameters

- **mol_obj** (*MolWrapper*, `rdkit.Chem.rdchem.Mol`, or `openbabel.pybel.Molecule`) – The molecule.
- **output** (*str*) – The output file where the molecule will be drawn. If None, returns a drawing object (`MolDraw2DCairo` or `MolDraw2DSVG`).
- **atm_types** (*dict or None*) – A pre-annotated dictionary for mapping atoms and pharmacophoric properties. If None, try to perceive properties with `feature_extractor`.
- **legend** (*str*) – A title for the figure.

Returns drawer

Return type None or a drawing object (`MolDraw2DCairo` or `MolDraw2DSVG`)

luna.mol.entry module

class ChainEntry(*pdb_id*, *chain_id*, *sep=':'*, *parser=None*)

Bases: *luna.mol.entry.Entry*

Define a chain.

Parameters

- **pdb_id** (*str*) – A 4-symbols structure id from PDB or a local PDB filename. Example: '3QL8' or 'file1'.
- **chain_id** (*str*) – A 1-symbol chain id. Example: 'A'.
- **sep** (*str*) – A separator character to format the entry string. The default value is ':'.

Raises **InvalidEntry** – If the provided information does not match the PDB format.

Examples

```
>>> from luna.mol.entry import ChainEntry
>>> e = ChainEntry(pdb_id="3QL8", chain_id="A")
>>> print(e)
<ChainEntry: 3QL8:A>
```

classmethod from_string(*entry_str*, *sep=':'*)

Initialize from a string.

Parameters

- **entry_str** (*str*) – A string representing the entry. Example: '3QL8:A'.
- **sep** (*str*) – The separator character used in `entry_str`. The default value is ':'. For example: if `entry_str` is set to '3QL8|A', then `sep` should be defined as '|'.

Return type *Entry*

Raises **IllegalArgumentError** – If the fields in `entry_str` do not match the format expected to define a chain.

Examples

```
>>> from luna.mol.entry import ChainEntry
>>> e = ChainEntry.from_string("3QL8:A", sep=":")
>>> print(e)
<ChainEntry: 3QL8:A>
```

property full_id

The full id of the entry is the tuple (PDB id or filename, chain id).

Type tuple, read-only

class Entry(*pdb_id*, *chain_id*, *comp_name=None*, *comp_num=None*, *comp_icode=None*, *is_hetatm=True*, *sep=':'*, *parser=None*)

Bases: `object`

Entries determine the target molecule to which interactions and other properties will be calculated. They can be ligands, chains, etc, and can be defined in a number of ways. Each entry has an associated PDB file that

may contain macromolecules (protein, RNA, DNA) and other small molecules, water, and ions. The PDB file provides the context to where the interactions with the target molecule will be calculated.

Parameters

- **pdb_id** (*str*) – A 4-symbols structure id from PDB or a local PDB filename. Example: '3QL8' or 'file1'.
- **chain_id** (*str*) – A 1-symbol chain id. Example: 'A'.
- **comp_name** (*str, optional*) – A 1 to 3-symbols compound name (residue name in the PDB format). Obligatory if `is_hetatm` is True. Example: 'X01'.
- **comp_num** (*int, optional*) – A valid 4-digits integer (residue sequence number in the PDB format). Obligatory if `is_hetatm` is True. Example: 300 or -1.
- **comp_icode** (*str, optional*) – A 1-character compound insertion code (residue insertion code in the PDB format). Example: 'A'.
- **is_hetatm** (*bool*) – If the compound is a ligand or not. The default value is True.
- **sep** (*str*) – A separator character to format the entry string. The default value is ':'.
- **parser** (`PDBParser` or `FTMapParser`, optional) – Define a PDB parser object. If not provided, the default parser will be used.

Raises

- **`IllegalArgumentError`** – If `is_hetatm` is True, but the compound name and number are not provided. If the compound number is provided but it is not an integer. If `comp_icode` is provided but it is not a valid character.
- **`InvalidEntry`** – If the provided information does not match the PDB format.

Examples

Chain entry: can be used to calculate interactions with a given chain.

```
>>> from luna.mol.entry import Entry
>>> e = Entry(pdb_id="3QL8", chain_id="A")
>>> print(e)
<Entry: 3QL8:A>
```

Compound entry: can be used to calculate interactions with a given compound (residue or nucleotide).

```
>>> from luna.mol.entry import Entry
>>> e = Entry(pdb_id="3QL8", chain_id="A", comp_name="HIS", comp_num=125, is_
↪hetatm=False)
>>> print(e)
<Entry: 3QL8:A:HIS:125>
```

Ligand entry: can be used to calculate interactions with a given ligand.

```
>>> from luna.mol.entry import Entry
>>> e = Entry(pdb_id="3QL8", chain_id="A", comp_name="X01", comp_num=300, is_
↪hetatm=True)
>>> print(e)
<Entry: 3QL8:A:X01:300>
```

You can use a different character separator for the entries. For example:

```
>>> from luna.mol.entry import Entry
>>> e = Entry(pdb_id="3QL8", chain_id="A", comp_name="X01", comp_num=300, is_
↪hetatm=True, sep="/")
>>> print(e)
<Entry: 3QL8/A/X01/300>
```

property chain_id
the chain id.

Type str, read-only

property comp_icode
the compound insertion code.

Type str, read-only

property comp_name
the compound name.

Type str, read-only

property comp_num
the compound number.

Type int, read-only

classmethod from_string(*entry_str*, *is_hetatm=True*, *sep=':'*)
Initialize from a string.

Parameters

- **entry_str** (*str*) – A string representing the entry. Example: '3QL8:A:X01:300'.
- **is_hetatm** (*bool*) – Defines if the compound is a ligand or not. The default value is True.
- **sep** (*str*) – The separator character used in `entry_str`. The default value is ':'. For example: if `entry_str` is set to '3QL8|A|X01|300', then `sep` should be defined as '|'.

Return type *Entry*

Raises **IllegalArgumentError** – If the fields in `entry_str` do not match the format expected to define a chain (ChainEntry) or a compound (MolEntry).

Examples

Chain entry: can be used to calculate interactions with a given chain.

```
>>> from luna.mol.entry import Entry
>>> e = Entry.from_string("3QL8:A", sep=":")
>>> print(e)
<Entry: 3QL8:A>
```

Compound entry: can be used to calculate interactions with a given compound (residue or nucleotide).

```
>>> from luna.mol.entry import Entry
>>> e = Entry.from_string("3QL8:A:HIS:125", sep=":")
>>> print(e)
<Entry: 3QL8:A:HIS:125>
```

Ligand entry: can be used to calculate interactions with a given ligand.

```
>>> from luna.mol.entry import Entry
>>> e = Entry.from_string("3QL8:A:X01:300", sep=":")
>>> print(e)
<Entry: 3QL8:A:X01:300>
```

property full_id

The full id of the entry is the tuple (PDB id or filename, chain id) for entries representing chains and (PDB id or filename, chain id, compound name, compound number, insertion code) for entries representing compounds.

Type tuple, read-only

get_biopython_key(*full_id=False*)

Represent the entry as a key to select chains or compounds from Biopython Entity objects.

Parameters full_id : bool

If True, return the full id of a chain or ligand. For chains, it consists of a tuple containing the PDB and the chain id. For ligands, it consists of a tuple containing the PDB, the chain, and the ligand id. The default value is False.

Returns Return str if the entry represents a chain and if `full_id` is False. Otherwise, return a tuple.

Return type str or tuple

Examples

```
>>> from luna.mol.entry import Entry
>>> e = Entry(pdb_id="3QL8", chain_id="A", comp_name="X01", comp_num=300, is_
↪hetatm=True, sep=":")
>>> print(e.get_biopython_key())
('H_X01', 300, ' ')
```

is_valid()

Check if the entry matches the expected format for protein-protein or protein-compound complexes.

Return type bool

property pdb_id

the pdb id.

Type str, read-only

to_string(*sep=None*)

Convert the entry to a string using `sep` as a separator character.

Parameters **sep** (*str or None*) – If None (the default), use the separator character defined during the entry object creation. Otherwise, uses `sep` as the separator character.

Examples

```
>>> from luna.mol.entry import Entry
>>> e = Entry(pdb_id="3QL8", chain_id="A", comp_name="X01", comp_num=300, is_
↪hetatm=True, sep=":")
>>> print(e.to_string("/"))
3QL8/A/X01/300
```

class MolEntry(*pdb_id*, *chain_id*, *comp_name*, *comp_num*, *comp_icode=None*, *sep=':'*, *parser=None*)

Bases: *luna.mol.entry.Entry*

Define a compound from a PDB file, which can be a residue, nucleotide, or ligand.

Parameters

- **pdb_id** (*str*) – A 4-symbols structure id from PDB or a local PDB filename. Example: '3QL8' or 'file1'.
- **chain_id** (*str*) – A 1-symbol chain id. Example: 'A'.
- **comp_name** (*str*) – A 1 to 3-symbols compound name (residue name in the PDB format). Example: 'X01'.
- **comp_num** (*int*) – A valid 4-digits integer (residue sequence number in the PDB format). Example: 300 or -1.
- **comp_icode** (*str, optional*) – A 1-character compound insertion code (residue insertion code in the PDB format). Example: 'A'.
- **sep** (*str*) – A separator character to format the entry string. The default value is ':'.

Raises **InvalidEntry** – If the provided information does not match the PDB format.

Examples

Compound entry: can be used to calculate interactions with a given compound (residue or nucleotide).

```
>>> from luna.mol.entry import MolEntry
>>> e = MolEntry(pdb_id="3QL8", chain_id="A", comp_name="HIS", comp_num=125, is_
↪hetatm=False)
>>> print(e)
<MolEntry: 3QL8:A:HIS:125>
```

Ligand entry: can be used to calculate interactions with a given ligand.

```
>>> from luna.mol.entry import MolEntry
>>> e = MolEntry(pdb_id="3QL8", chain_id="A", comp_name="X01", comp_num=300, is_
↪hetatm=True)
>>> print(e)
<MolEntry: 3QL8:A:X01:300>
```

classmethod from_file(*input_file*, *sep=':'*)

Initialize from a list of strings representing compounds.

Parameters

- **input_file** (*str*) – The file from where the list of strings (one per line) will be read from.

- **sep** (*str*) – The separator character used in `input_file`. The default value is ':'. For example: if entries from `input_file` use '|' as the separator, then `sep` should be defined as '|'.

Yields *MolEntry* – An entry recovered from `input_file`.

classmethod from_string(*entry_str*, *sep=':'*)

Initialize from a string.

Parameters

- **entry_str** (*str*) – A string representing the entry. Example: '3QL8:A:X01:300'.
- **is_hetatm** (*bool*) – Defines if the compound is a ligand or not. The default value is True.
- **sep** (*str*) – The separator character used in `entry_str`. The default value is ':'. For example: if `entry_str` is set to '3QL8|A|X01|300', then `sep` should be defined as '|'.

Return type *Entry*

Raises **IllegalArgumentError** – If the fields in `entry_str` do not match the format expected to define a chain (ChainEntry) or a compound (MolEntry).

Examples

Chain entry: can be used to calculate interactions with a given chain.

```
>>> from luna.mol.entry import Entry
>>> e = Entry.from_string("3QL8:A", sep=":")
>>> print(e)
<Entry: 3QL8:A>
```

Compound entry: can be used to calculate interactions with a given compound (residue or nucleotide).

```
>>> from luna.mol.entry import Entry
>>> e = Entry.from_string("3QL8:A:HIS:125", sep=":")
>>> print(e)
<Entry: 3QL8:A:HIS:125>
```

Ligand entry: can be used to calculate interactions with a given ligand.

```
>>> from luna.mol.entry import Entry
>>> e = Entry.from_string("3QL8:A:X01:300", sep=":")
>>> print(e)
<Entry: 3QL8:A:X01:300>
```

class MolFileEntry(*pdb_id*, *mol_id*, *sep=':'*)

Bases: *luna.mol.entry.Entry*

Define a ligand from a molecular file. This class should be used for docking and molecular dynamics campaigns where usually one has the protein structure in the PDB format and the ligand structure in a separate molecular file.

Parameters

- **pdb_id** (*str*) – A 4-symbols structure id from PDB or a local PDB filename. Example: '3QL8' or 'file1'.
- **mol_id** (*str*) – The ligand id in the molecular file.

- **sep** (*str*) – A separator character to format the entry string. The default value is ':'.

Variables

- **~MolFileEntry.mol_id** (*str*) – The ligand id.
- **~MolFileEntry.mol_file** (*str*) – Pathname of the molecular file.
- **~MolFileEntry.mol_file_ext** (*str*) – The molecular file format. If not provided, try to recover the molecular file extension directly from `mol_file`.
- **~MolFileEntry.mol_obj_type** (*{'rdkit', 'openbabel'}*) – Define which library (RDKit or Open Babel) to use to parse the molecular file.
- **~MolFileEntry.overwrite_mol_name** (*bool*) – If True, substitute the ligand name in the parsed molecular object with `mol_id`. Only works for single-molecule files (`is_multimol_file` = False) as in these cases `mol_id` does not need to match the ligand name in the molecular file.
- **~MolFileEntry.is_multimol_file** (*bool*) – If `mol_file` contains multiple molecules or not. If True, `mol_id` should match some ligand name in `mol_file`.

classmethod from_file(*input_file*, *pdb_id*, *mol_file*, ***kwargs*)

Initialize from a list of ligand names.

Parameters

- **input_file** (*str*) – The file from where the list of ligand names (one per line) will be read from.
- **pdb_id** (*str*) – A 4-symbols structure id from PDB or a local PDB filename. Example: '3QL8' or 'file1'.
- **mol_file** (*str*) – Pathname of a multi-molecular file.
- **mol_file_ext** (*str, optional*) – The molecular file format. If not provided, try to recover the molecular file extension directly from `mol_file`.
- **mol_obj_type** (*{'rdkit', 'openbabel'}*) – If "rdkit", parse the converted molecule with RDKit and return an instance of `rdkit.Chem.rdchem.Mol`. If "openbabel", parse the converted molecule with Open Babel and return an instance of `openbabel.pybel.Molecule`. The default value is 'rdkit'.
- **autoload** (*bool*) – If True, parse the ligand from the molecular file during the entry initialization. Otherwise, only load the ligand when first used.
- **sep** (*str*) – A separator character to format the entry string. The default value is ':'.

Yields *MolFileEntry* – An entry recovered from `input_file`.

Raises

- **FileNotFoundError** – If `mol_file` does not exist.
- **IllegalArgumentError** – If `mol_obj_type` is not either 'rdkit' nor 'openbabel'.
- **MoleculeObjectError** – If any errors occur while parsing the molecular file. Detailed information about the errors can be found in the logging outputs.
- **MoleculeNotFoundError** – If some ligand from `input_file` was not found in `mol_file`.

Examples

```
>>> from luna.mol.entry import MolFileEntry
>>> entries = MolFileEntry.from_file(input_file="tutorial/inputs/MolEntries.txt
↪",
...                                   pdb_id="D4", mol_file="tutorial/inputs/
↪ligands.mol2",
...                                   mol_obj_type="openbabel", autoload=True)
>>> for e in entries:
>>>     print(e)
<MolFileEntry: D4:ZINC000012442563>
<MolFileEntry: D4:ZINC000065293174>
<MolFileEntry: D4:ZINC000096459890>
<MolFileEntry: D4:ZINC000343043015>
<MolFileEntry: D4:ZINC000575033470>
```

classmethod from_mol_file(*pdb_id*, *mol_id*, *mol_file*, *is_multimol_file*, *mol_file_ext=None*, *mol_obj_type='rdkit'*, *autoload=False*, *overwrite_mol_name=False*, *sep=':'*)

Initialize from a molecular file.

Parameters

- **pdb_id** (*str*) – A 4-symbols structure id from PDB or a local PDB filename. Example: '3QL8' or 'file1'.
- **mol_id** (*str*) – The ligand id in the molecular file.
- **mol_file** (*str*) – Pathname of the molecular file.
- **is_multimol_file** (*bool*) – If `mol_file` contains multiple molecules or not. If True, `mol_id` should match some ligand name in `mol_file`.
- **mol_file_ext** (*str, optional*) – The molecular file format. If not provided, try to recover the molecular file extension directly from `mol_file`.
- **mol_obj_type** (*{'rdkit', 'openbabel'}*) – If "rdkit", parse the converted molecule with RDKit and return an instance of `rdkit.Chem.rdchem.Mol`. If "openbabel", parse the converted molecule with Open Babel and return an instance of `openbabel.pybel.Molecule`. The default value is 'rdkit'.
- **autoload** (*bool*) – If True, parse the ligand from the molecular file during the entry initialization. Otherwise, only load the ligand when first used.
- **overwrite_mol_name** (*bool*) – If True, substitute the ligand name in the parsed molecular object with `mol_id`. Only works for single-molecule files (`is_multimol_file` = False) as in these cases `mol_id` does not need to match the ligand name in the molecular file.
- **sep** (*str*) – A separator character to format the entry string. The default value is ':'.

Return type *MolFileEntry*

Raises

- **FileNotFoundError** – If `mol_file` does not exist.
- **IllegalArgumentError** – If `mol_obj_type` is not either 'rdkit' nor 'openbabel'.
- **MoleculeObjectError** – If any errors occur while parsing the molecular file. Detailed information about the errors can be found in the logging outputs.

- **MoleculeNotFoundError** – If the ligand `mol_id` was not found in the input file and `is_multimol_file` is True.

Examples

In this first example, we will read the ligand 'ZINC000007786517' from a single-molecule file. As we are working with a single-molecule file, `mol_id` can be any value you prefer.

```
>>> from luna.mol.entry import MolFileEntry
>>> e = MolFileEntry.from_mol_file(pdb_id="D4", mol_id="Ligand", mol_file=
↪"tutorial/inputs/ZINC000007786517.mol",
...                                mol_obj_type='rdkit', is_multimol_file=False)
>>> print(e)
D4:Ligand
>>> print(e.mol_obj.to_smiles())
Cc1cccc(NC(=O)C[N@@H+](C)C2CCCCC2)c1C
```

Now, let's say we need to read the ligand 'ZINC000096459890' from a multi-molecular file and that we want to use Open Babel to parse the molecule. To do so, remember that it should exist a ligand with the name `mol_id` in `mol_file`. Otherwise, it will raise the exception MoleculeNotFoundError.

```
>>> from luna.mol.entry import MolFileEntry
>>> e = MolFileEntry.from_mol_file(pdb_id="D4", mol_id="ZINC000096459890", mol_
↪file="tutorial/inputs/ligands.mol2",
...                                mol_obj_type='openbabel', is_multimol_
↪file=True)
>>> print(e)
<MolFileEntry: D4:ZINC000096459890>
>>> print(e.mol_obj.to_smiles())
O=C(OCCCN1C=CC=CC1=O)c1ccc2ccc(Cl)cc2n1
```

Below, we show what happens if `mol_id` does not exist in `mol_file`. Observe we set `autoload` to True to parse the molecule right away.

```
>>> from luna.mol.entry import MolFileEntry
>>> e = MolFileEntry.from_mol_file(pdb_id="D4", mol_id="Ligand", mol_file=
↪"tutorial/inputs/ligands.mol2",
...                                mol_obj_type='openbabel', is_multimol_
↪file=True, autoload=True)
luna.util.exceptions.MoleculeNotFoundError: "The ligand 'Ligand' was not found␣
↪in the input file         or generated errors while parsing it with Open␣
↪Babel."
```

classmethod from_mol_obj(*pdb_id*, *mol_id*, *mol_obj*, *sep=':'*)

Initialize from an already loaded molecular object.

This function is useful in cases where a molecular object is parsed and pre-processed using a different protocol defined by the user.

Parameters

- **pdb_id** (*str*) – A 4-symbols structure id from PDB or a local PDB filename. Example: '3QL8' or 'file1'.
- **mol_id** (*str*) – The ligand id. As the molecular object is already provided, the ligand id does not need to match the ligand name in the molecular object.

- **mol_obj** (*MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule) – The molecular object.
- **sep** (*str*) – A separator character to format the entry string. The default value is ':'.

Return type *MolFileEntry*

Raises

- **MoleculeObjectTypeError** – If the molecular object is not an instance of *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule.
- **IllegalArgumentError** – If entity is not a valid Biopython object.

Examples

In this example, we will initialize a new MolFileEntry with the ligand 'ZINC000007786517' and the structure located in a PDB file of name 'D4', which is the structure used for docking the molecule.

First, let's parse the molecular file.

```
>>> from luna.wrappers.rdkit import read_mol_from_file
>>> mol_obj = read_mol_from_file("tutorial/inputs/ZINC000007786517.mol", mol_
↪format="mol")
```

Now, we create the new MolFileEntry object as follows:

```
>>> e = MolFileEntry.from_mol_obj("D4", "ZINC000007786517", mol_obj, sep=ENTRY_
↪SEPARATOR)
>>> print(e)
<MolFileEntry: D4:ZINC000007786517>
>>> print(e.mol_obj.to_smiles())
Cc1cccc(NC(=O)C[N@@H+](C)C2CCCCC2)c1C
```

property full_id

The full id of the entry is the tuple (PDB id or filename, ligand id).

Type tuple, read-only

get_biopython_structure(*entity=None*, *parser=None*)

Transform the molecular object into a Biopython Entity object.

If entity is provided, the molecular object is appended to it, i.e., this function can be used to join a ligand and the structure used during docking or molecular dynamics.

By default, the ligand is added to a chain of id *z*.

Parameters

- **entity** (Entity, optional) – Append the molecular object to entity. If not provided, a new Entity is created.
- **parser** (PDBParser, optional) – Define a PDB parser object. If not provided, the default parser will be used.

Return type Entity

Raises **IllegalArgumentError** – If entity is not a valid Biopython object.

Examples

In this example, we will demonstrate how to join a protein structure and a ligand docked against it.

First, let's parse the PDB file.

```
>>> from luna.MyBio.PDB.PDBParser import PDBParser
>>> pdb_parser = PDBParser(PERMISSIVE=True, QUIET=True)
>>> structure = pdb_parser.get_structure("Protein", "tutorial/inputs/D4.pdb")
```

Observe that the list of chains in the parsed structure contains only one element.

```
>>> print(structure[0].child_list)
[<Chain id=A>]
```

Now, we will read the ligand and append it to the existing protein structure.

```
>>> from luna.mol.entry import MolFileEntry
>>> e = MolFileEntry.from_mol_file("D4", "ZINC000007786517", "tutorial/inputs/
↪ZINC000007786517.mol",
...                                 mol_obj_type='rdkit', is_multimol_file=False)
>>> joined_structure = e.get_biopython_structure(structure)
```

Observe that now the list of chains contains chains 'A' and 'z', which is the default chain where ligands are added.

```
>>> print(joined_structure[0].child_list)
[<Chain id=A>, <Chain id=z>]
```

If we loop over the residues in chain 'z', we will find our ligand.

```
>>> for r in structure[0]["z"]:
>>>     print(r)
<Residue LIG het=H_LIG resseq=9999 icode= >
```

is_mol_obj_loaded()
Check if the molecular object has already been loaded.

Return type bool

is_valid()
Check if the entry represents a valid protein-ligand complex.

Return type bool

property mol_obj
The molecule.

Type *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule

recover_entries_from_entity(*entity*, *get_small_molecules=True*, *get_chains=False*, *ignore_artifacts=True*, *by_cluster=False*, *sep=':'*)
Search for chains and small molecules in entity and return them as strings.

Parameters

- **entity** (Entity) – An entity from where chains or small molecules will be recovered.
- **get_small_molecules** (*bool*) – If True, identify small molecules and return them as *MolEntry* objects. The default value is True.

- **get_chains** (*bool*) – If True, identify chains and return them as *ChainEntry* objects. The default value is False.
- **ignore_artifacts** (*bool*) – If True, ignore the following crystallography artifacts: ACE, ACT, BME, CSD, CSW, EDO, FMT, GOL, MSE, NAG, NO3, PO4, SGM, SO4, or TPO. The default value is True.
- **by_cluster** (*bool*) – If True, aggregate entries by cluster. Cluster ids are exclusive to `Residue` instances and are automatically set by `FTMapParser`, a parser for FTMap results. By default, the cluster id of `Residue` instances are set to None, therefore, if the cluster id is not explicitly defined, all entries will be aggregated to the same key `None`.
- **sep** (*str*) – A separator character to format the entry string. The default value is ':'.

Returns If `by_cluster` is set to False, a list of *ChainEntry* or *MolEntry* objects is returned. Otherwise, a dict is returned, in which keys are clusters and values are lists of *ChainEntry* or *MolEntry* objects. When no cluster information is available, all entries are aggregated in a key of value `None`. Cluster ids are exclusive to `Residue` instances, therefore, *ChainEntry* objects are always placed in a key of value `None`.

Return type list or dict

Examples

First, let's parse a PDB file.

```
>>> from luna.MyBio.PDB.PDBParser import PDBParser
>>> pdb_parser = PDBParser(PERMISSIVE=True, QUIET=True)
>>> structure = pdb_parser.get_structure("Protein", "tutorial/inputs/3QQK.pdb")
```

Now, we can recover entries from the parsed PDB file:

```
>>> from luna.mol.entry import recover_entries_from_entity
>>> entries = recover_entries_from_entity(structure, get_chains=True)
>>> for e in entries:
>>>     print(e)
<MolEntry: Protein:A:X02:497>
<ChainEntry: Protein:A>
```

luna.mol.features module

class ChemicalFeature(*name*)

Bases: object

Define chemical features as for example pharmacophore properties.

Parameters **name** (*str*) – The chemical feature name.

format_name(*case_func='sentencecase'*)

Convert chemical feature names to another string case.

Parameters **name** (*str*) – The name of a string case function from *luna.util.stringcase*.

class FeatureExtractor(*feature_factory*)

Bases: object

Perceive chemical features from molecules.

Parameters **feature_factory** (MolChemicalFeatureFactory) – An RDKit feature factory.

Examples

First, let's read a molecule (glutamine).

```
>>> from luna.wrappers.base import MolWrapper
>>> mol = MolWrapper.from_smiles("N[C@@H](CCC(N)=O)C(O)=O").unwrap()
```

Now, create a feature factory and instantiate a new `FeatureExtractor` object.

```
>>> from luna.util.default_values import ATOM_PROP_FILE
>>> from rdkit.Chem import ChemicalFeatures
>>> from luna.mol.features import FeatureExtractor
>>> feature_factory = ChemicalFeatures.BuildFeatureFactory(ATOM_PROP_FILE)
>>> feature_extractor = FeatureExtractor(feature_factory)
```

Finally, you can extract features by group or atom.

```
>>> features = feature_extractor.get_features_by_atoms(mol)
>>> features = feature_extractor.get_features_by_groups(mol)
```

get_features_by_atoms(*mol_obj*, *atm_map=None*)

Perceive chemical features from the molecule `mol_obj` by atom.

Parameters

- **mol_obj** (*MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule) – The molecule.
- **atm_map** (*dict*) – A dictionary to map an atom's index to a different value.

Returns **atm_features** – Chemical features by atoms that are represented by their index or by a value from `atm_map`.

Return type dict of {int : list of `ChemicalFeature`}

get_features_by_groups(*mol_obj*, *atm_map=None*)

Perceive chemical features from the molecule `mol_obj` by atom groups.

Parameters

- **mol_obj** (*MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule) – The molecule.
- **atm_map** (*dict*) – A dictionary to map an atom's index to a different value.

Returns

grp_features – Chemical features by groups. Each dictionary value is defined as follows:

- `atm_ids` (list): list of atoms represented by their index or by a value from `atm_map` ;
- `features` (list of `ChemicalFeature`): list of chemical features.

Return type dict of {str : dict}

class OBMolChemicalFeature(*family*, *atom_ids*)

Bases: object

Mimic rdkit.Chem.rdMolChemicalFeatures.MolChemicalFeature for Open Babel.

Parameters

- **family** (*str*) – The family name, which is the term used by RDKit for chemical features.
- **atom_ids** (*list*) – List of atom identifiers.

`GetAtomIds()`
Get the IDs of the atoms that participate in the feature.

`GetFamily()`
Get the family to which the feature belongs (e.g., donor, acceptor, etc.)

luna.mol.fingerprint module

`class FingerprintGenerator(mol_obj=None)`
Bases: `object`

Generate molecular fingerprints for the molecule `mol`.

Parameters **mol** (*MolWrapper*, `rdkit.Chem.rdchem.Mol`, or `openbabel.pybel.Molecule`, optional) – The molecule.

Examples

First, create a new *FingerprintGenerator* object.

```
>>> from luna.mol.fingerprint import FingerprintGenerator
>>> fg = FingerprintGenerator()
```

Now, let's read a molecule (glutamine) and set it to the *FingerprintGenerator* object.

```
>>> from luna.wrappers.base import MolWrapper
>>> fg.mol = MolWrapper.from_smiles("N[C@@H](CCC(N)=O)C(O)=O")
```

Finally, you can call any available function to generate the desired fingerprint type. In the below example, a count ECFP4 fingerprint of size 1,024 is created.

```
>>> fp = fg.morgan_fp(radius=2, length=1024, type=2)
>>> print(fp.GetNonzeroElements())
{1: 1, 80: 2, 140: 1, 147: 2, 389: 1, 403: 1, 540: 1, 545: 1, 650: 2, 728: 1, 739:␣
↪1, 767: 1, 786: 1, 807: 3, 820: 1, 825: 1, 874: 1, 893: 2, 900: 1}
```

You can then continue using the *FingerprintGenerator* object to create other fingerprint types. For example, let's create a 2D pharmacophore fingerprint.

```
>>> fp = fg.pharm2d_fp()
>>> print(fp.GetNumOnBits())
90
```

`atom_pairs_fp()`
Generate an atom pairs fingerprint for the molecule `mol`.

Raises **FingerprintNotCreated** – If the fingerprint could not be created.

`maccs_keys_fp()`
Generate a MACCS keys fingerprint for the molecule `mol`.

Raises **FingerprintNotCreated** – If the fingerprint could not be created.

property mol
The molecule.

Type *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule

morgan_fp(*radius=2*, *length=2048*, *features=False*, *type=2*)
Generate a Morgan fingerprint for the molecule mol.

Parameters

- **radius** (*int*) – Define the maximum radius of the circular neighborhoods considered for each atom. The default value is 2, which is roughly equivalent to ECFP4 and FCFP4.
- **length** (*int*) – The length of the fingerprint. The default value is 2,048.
- **features** (*bool*) – If True, use pharmacophoric properties (FCFP) instead of atomic invariants (ECFP). The default value is False.
- **type** (*{1, 2, 3}*) – Define the type of the Morgan fingerprint function to be used, where:
  - 1 means **GetMorganFingerprintAsBitVect()**. It returns an explicit bit vector of size length (hashed fingerprint), where 0s and 1s represent the presence or absence of a given feature, respectively.
  - 2 means **GetHashedMorganFingerprint()**. It returns a sparse int vector length elements long (hashed fingerprint) containing the occurrence number of each feature.
  - 3 means **GetMorganFingerprint()**. It returns a sparse int vector 2^32 elements long containing the occurrence number of each feature.

  The default value is 2.

Raises

- **FingerprintNotCreated** – If the fingerprint could not be created.
- **IllegalArgumentError** – If type is a value other than 1, 2, or 3.

pharm2d_fp(*sig_factory=None*)
Generate a 2D pharmacophore fingerprint for the molecule mol.

Parameters **sig_factory** (RDKit SigFactory, optional) – Factory object for producing signatures. The default signature factory is defined as shown below:

```
>>> feat_factory = ChemicalFeatures.BuildFeatureFactory(MIN_FDEF_FILE)
>>> sig_factory = SigFactory(feat_factory, minPointCount=2,
...                          maxPointCount=3, trianglePruneBins=False)
>>> sig_factory.SetBins([(0, 2), (2, 5), (5, 8)])
>>> sig_factory.Init()
```

Raises **FingerprintNotCreated** – If the fingerprint could not be created.

rdk_fp()
Generate an RDKit topological fingerprint for the molecule mol.

Raises **FingerprintNotCreated** – If the fingerprint could not be created.

torsion_fp()
Generate a topological torsion fingerprint for the molecule mol.

Raises **FingerprintNotCreated** – If the fingerprint could not be created.

available_fp_functions()
Return a list of all fingerprints available at *FingerprintGenerator*.

generate_fp_for_mols(*mols*, *fp_function=None*, *fp_opt=None*, *critical=False*)

Generate molecular fingerprints for a sequence of molecules.

Parameters

- **mols** (iterable of *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule) – A sequence of molecules.
- **fp_function** (*str*) – The fingerprint function to use. The default value is 'pharm2d_fp'.

  To check out the list of available functions, call the function *available_fp_functions()*.
- **fp_opt** (*dict, optional*) – A set of parameters to pass to fp_function.
- **critical** (*bool*) – If True, raises any exceptions caught during the generation of fingerprints. Otherwise, ignores all exceptions (the default). The error messages are always printed to the logging output.

Returns

A list of dictionaries where each item contains the molecule name and its fingerprint.

The dict is defined as follows:

- mol (str): the molecule name;
- fp (RDKit ExplicitBitVect or SparseBitVect): the fingerprint;

Return type list of dict

Raises **IllegalArgumentError** – If fp_function is not a function available in *FingerprintGenerator*.

Examples

First, let's define a set of molecules.

```
>>> from luna.wrappers.base import MolWrapper
>>> mols = [MolWrapper.from_smiles("N[C@@H](CCC(N)=O)C(O)=O"),
...         MolWrapper.from_smiles("C[C@@H](C(=O)O)N"),
...         MolWrapper.from_smiles("C1=CC(=CC=C1CC(C(=O)O)N)O")]
```

Now, you can generate fingerprints for these molecules using the function *generate_fp_for_mols()*. For example, let's create count ECFP4 fingerprints of size 1,024 for the above molecules.

```
>>> from luna.mol.fingerprint import generate_fp_for_mols
>>> fps = generate_fp_for_mols(mols, fp_function="morgan_fp", fp_opt={"length":␣
↪1024})
```

Then, you can loop through the results as shown below:

```
>>> for d in fps:
>>>     print(f"{d['mol'].ljust(25)} - {len(d['fp'].GetNonzeroElements())}")
N[C@@H](CCC(N)=O)C(O)=O   - 19
C[C@@H](C(=O)O)N          - 12
C1=CC(=CC=C1CC(C(=O)O)N)O - 24
```

luna.mol.groups module

class AtomGroup(*atoms*, *features=None*, *interactions=None*, *recursive=True*, *manager=None*)
Bases: object

Represent single atoms, chemical functional groups, or simply an arrangement of atoms as in hydrophobes.

Parameters

- **atoms** (iterable of *ExtendedAtom*) – A sequence of atoms.
- **features** (iterable of ChemicalFeature, optional) – A sequence of chemical features.
- **interactions** (iterable of *InteractionType*, optional) – A sequence of interactions established by an atom group.
- **recursive** (*bool*) – If True, add the new atom group to the list of atom groups of each atom in atoms.
- **manager** (*AtomGroupsManager*, optional) – The *AtomGroupsManager* object that contains this *AtomGroup* object.

add_features(*features*)
Add ChemicalFeature objects to features.

add_interactions(*interactions*)
Add *InteractionType* objects to interactions.

as_json()
Represent the atom group as a dict containing the atoms, compounds, features, and compound classes (water, hetero group, residue, or nucleotide).

The dict is defined as follows:

- atoms (iterable of *ExtendedAtom*): the list of atoms comprising the atom group;
- compounds (iterable of Residue): the list of unique compounds that contain the atoms;
- classes (iterable of str): the list of compound classes;
- features (iterable of ChemicalFeature): the atom group's list of chemical features.

property atoms
The sequence of atoms that belong to an atom group.

Type iterable of *ExtendedAtom*, read-only

property centroid
The centroid (x, y, z) of the atom group.

If atoms contains only one atom, then centroid returns the same as coords.

Type array-like of floats, read-only

clear_refs()
References to this *AtomGroup* instance will be removed from the list of atom groups of each atom in atoms.

property compounds
The set of unique compounds that contain the atoms in atoms.

As an atom group can be formed by the union of two or more compounds (e.g., amide of peptide bonds), it may return more than one compound.

Type set of Residue, read-only

contain_group(*atm_grp*)
Check if the atom group `atm_grp` is a subset of this atom group.

For example, consider the benzene molecule. Its aromatic ring itself forms an *AtomGroup* object composed of all of its six atoms. Consider now any subset of carbons in the benzene molecule. This subset forms an *AtomGroup* object that is part of the group formed by the aromatic ring. Therefore, in this example, *contain_group()* will return True because the aromatic ring contains the subset of hydrophobic atoms.

Parameters **atm_grp** (*AtomGroup*)

Returns If one atom group contains another atom group.

Return type bool

property coords
Atomic coordinates (x, y, z) of each atom in `atoms`.

Type array-like of floats

property feature_names
The name of each chemical feature in `features`.

Type iterable of str

property features
A sequence of chemical features.

To add or remove a feature use *add_features()* or *remove_features()*, respectively.

Type iterable of `ChemicalFeature`

get_chains()
Get all unique chains in an atom group.

get_interactions_with(*atm_grp*)
Get all interactions that an atom group establishes with another atom group `atm_grp`.

Returns All interactions established with the atom group `atm_grp`.

Return type iterable of `InteractionType`

get_serial_numbers()
Get the serial number of each atom in an atom group.

get_shortest_path_length(*trgt_grp*, *cutoff=None*)
Compute the shortest path length between this atom group to another atom group `trgt_grp`.

The shortest path between two atom groups is defined as the shortest path between any of their atoms, which are calculated using Dijkstra's algorithm.

If `manager` is not provided, None is returned.

If there is not any path between `src_grp` and `trgt_grp`, infinite is returned.

Parameters

- **trgt_grp** (*AtomGroup*) – The target atom group to calculate the shortest path.
- **cutoff** (*int, optional*) – Only paths of length <= `cutoff` are returned. If None, all path lengths are considered.

Returns The shortest path.

Return type int, float('inf'), or None:

has_atom(*atom*)
Check if an atom group contains a given atom `atom`.

Parameters **atom** (*ExtendedAtom*)

Returns If the atom group contains or not `atom`.

Return type bool

has_hetatm()
Return True if at least one atom in the atom group belongs to a hetero group, i.e., non-standard residues of proteins, DNAs, or RNAs, as well as atoms in other kinds of groups, such as carbohydrates, substrates, ligands, solvent, and metal ions.

has_nucleotide()
Return True if at least one atom in the atom group belongs to a nucleotide.

has_residue()
Return True if at least one atom in the atom group belongs to a standard residue of proteins.

has_target()
Return True if at least one compound is the target of LUNA's analysis

has_water()
Return True if at least one atom in the atom group belongs to a water molecule.

property interactions
The sequence of interactions established by an atom group.

To add or remove an interaction use *add_interactions()* or *remove_interactions()*, respectively.

Type iterable of *InteractionType*

is_hetatm()
Return True if all atoms in the atom group belong to hetero group, i.e., non-standard residues of proteins, DNAs, or RNAs, as well as atoms in other kinds of groups, such as carbohydrates, substrates, ligands, solvent, and metal ions.

Hetero groups are designated by the flag HETATM in the PDB format.

is_mixed()
Return True if the atoms in the atom group belong to different compound classes (water, hetero group, residue, or nucleotide).

is_nucleotide()
Return True if all atoms in the atom group belong to nucleotides.

is_residue()
Return True if all atoms in the atom group belong to standard residues of proteins.

is_water()
Return True if all atoms in the atom group belong to water molecules.

property manager
The *AtomGroupsManager* object that contains an *AtomGroup* object.

Type *AtomGroupsManager*

property normal
The normal vector (x, y, z) of the points given by `coords`.

Type array-like of floats, read-only

remove_features(*features*)
Remove `ChemicalFeature` objects from `features`.

remove_interactions(*interactions*)
Remove *InteractionType* objects from `interactions`.

property size
The number of atoms comprising an atom group.

Type int

class AtomGroupNeighborhood(*atm_grps*, *bucket_size=10*)
Bases: object

Class for fast neighbor atom groups searching.

AtomGroupNeighborhood makes use of a KD Tree implemented in C, so it's fast.

Parameters

- **atm_grps** (iterable of *AtomGroup*, optional) – A sequence of *AtomGroup* objects, which is used in the queries. It can contain atom groups from different molecules.
- **bucket_size** (*int*) – Bucket size of KD tree. You can play around with this to optimize speed if you feel like it. The default value is 10.

search(*center*, *radius*)
Return all atom groups in atm_grps that is up to a maximum of radius away (measured in Å) of center.

For atom groups with more than one atom, their centroid is used as a reference.

class AtomGroupPerceiver(*feature_extractor*, *add_h=False*, *ph=None*, *amend_mol=True*, *mol_obj_type='rdkit'*, *charge_model=<luna.mol.charge_model.OpenEyeModel object>*, *tmp_path=None*, *expand_selection=True*, *radius=2.2*, *critical=True*)
Bases: object

Perceive and create atom groups for molecules.

Parameters

- **feature_extractor** (FeatureExtractor) – Perceive pharmacophoric properties from molecules.
- **add_h** (*bool*) – If True, add hydrogen to the molecules.
- **ph** (*float, optional*) – If not None, add hydrogens appropriate for pH ph.
- **amend_mol** (*bool*) – If True, apply validation and standardization of molecules read from a PDB file.
- **mol_obj_type** (*{"rdkit", "openbabel"}*) – If "rdkit", parse the converted molecule with RDKit and return an instance of rdkit.Chem.rdchem.Mol. If "openbabel", parse the converted molecule with Open Babel and return an instance of openbabel.pybel.Molecule.
- **charge_model** (class:*ChargeModel*) – A charge model object. By default, the implementation of OpenEye charge model is used.
- **tmp_path** (*str, optional*) – A temporary directory to where temporary files will be saved. If not provided, the system's default temporary directory will be used instead.
- **expand_selection** (*bool*) – If True (the default), perceive features for a given molecule considering all nearby molecules. The goal is to identify any covalently bonded molecules that may alter the pharmacophoric properties or chemical functional groups.

  For instance, consider an amide of a peptide bond. If expand_selection is False, the residues forming the peptide bond will be analyzed separately, which will make the oxygen and the nitrogen of the amide to be perceived as carbonyl oxygen and amine, respectively. On the other hand, if expand_selection is True, the covalent bond between the residues will be identified and the amide will be correctly perceived.

- **radius** (*float*) – If `expand_selection` is True, select all molecules up to a maximum of `radius` away (measured in Å). The default value is 2.2, which comprises covalent bond distances.
- **critical** (*bool*) – If False, ignore any errors during the processing a molecule and continue to the next one. The default value is True, which implies that any errors will raise an exception.

Raises `IllegalArgumentError` – If `mol_obj_type` is not either 'rdkit' nor 'openbabel'.

`perceive_atom_groups`(*compounds*, *mol_objs_dict=None*)

Perceive and create atom groups for each molecule in `compounds`.

Parameters

- **compounds** (iterable of `Residue`) – A sequence of molecules.
- **mol_objs_dict** (*dict*) – Map a compound, represented by its id, to a molecular object (*MolWrapper*, `rdkit.Chem.rdchem.Mol`, or `openbabel.pybel.Molecule`).

  This parameter can be used in cases where the ligand is read from a molecular file and no standardization or validation is required.

Returns An *AtomGroupsManager* object containing all atom groups perceived for the molecules in `compounds`.

Return type *AtomGroupsManager*

`class AtomGroupsManager`(*atm_grps=None*, *entry=None*)

Bases: `object`

Store and manage *AtomGroup* objects.

Parameters

- **atm_grps** (iterable of *AtomGroup*, optional) – An initial sequence of *AtomGroup* objects.
- **entry** (*Entry*, optional) – The chain or molecule from where the atom groups were perceived.

Variables

- **~AtomGroupsManager.entry** (*Entry*) – The chain or molecule from where the atom groups were perceived.
- **~AtomGroupsManager.graph** (`networkx.Graph`) – Represent `entry` as a graph and its vicinity.
- **~AtomGroupsManager.version** (*str*) – The LUNA version when the object was created.

`add_atm_grps`(*atm_grps*)

Add one or more *AtomGroup* objects to `atm_grps` and automatically update `child_dict`.

`apply_filter`(*func*)

Apply a filtering function over the atom groups in ``atm_grps`.

Parameters **func** (*callable*) – A filtering function that returns True case an *AtomGroup* object is valid and False otherwise.

Yields *AtomGroup* – A valid *AtomGroup* object.

`property atm_grps`

The sequence of *AtomGroup* objects. Additional objects should be added using the method *add_atm_grps()*.

Type iterable of *AtomGroup*, read-only

property child_dict
Mapping between atoms (ExtendedAtom) and atom groups (*AtomGroup*).

The mapping is a dict of {tuple of ExtendedAtom instances : *AtomGroup*} and is automatically updated when *add_atm_grps()* is called.

Type dict, read-only

filter_by_types(*types*, *must_contain_all=True*)
Filter *AtomGroup* objects by their physicochemical features.

Parameters

- **types** (*iterable of str*) – A sequence of physicochemical features.
- **must_contain_all** (*bool*) – If True, an *AtomGroup* object should contain all physicochemical features in types to be accepted. Otherwise, it will be filtered out.

Yields *AtomGroup* – A valid *AtomGroup* object.

find_atm_grp(*atoms*)
Find the atom group that contains the sequence of atoms atoms.

Returns An atom group object or None if atoms is not in the child_dict mapping.

Return type *AtomGroup* or None

get_all_interactions()
Return all interactions established by the atom groups in atm_grps.

Returns All interactions.

Return type set of InteractionType

get_shortest_path_length(*src_grp*, *trgt_grp*, *cutoff=None*)
Compute the shortest path length between two atom groups src_grp and trgt_grp.

The shortest path between two atom groups is defined as the shortest path between any of their atoms, which are calculated using Dijkstra's algorithm and the graph graph.

If there is not any path between src_grp and trgt_grp, infinite is returned.

Parameters

- **src_grp, trgt_grp** (*AtomGroup*) – Two atom groups to calculate the shortest path.
- **cutoff** (*int*) – Only paths of length <= cutoff are returned. If None, all path lengths are considered.

Returns The shortest path.

Return type int or float('inf'):

static load(*input_file*)
Load the pickled representation of an *AtomGroupsManager* object saved at the file input_file.

Returns The reconstituted *AtomGroupsManager* object, including its set of atom groups and interactions.

Return type *AtomGroupsManager*

Raises **PKLNotReadError** – If the file could not be loaded.

merge_hydrophobic_atoms(*interactions_mngr*)
Create hydrophobic islands by merging covalently bonded hydrophobic atoms in atm_grps. Hydrophobic islands are atom groups having the feature *Hydrophobe*.

Atom-atom hydrophobic interactions in `interactions_mngr` are also converted to island-island interactions.

Parameters **interactions_mngr** (*InteractionsManager*) – An *InteractionsManager* object from where hydrophobic interactions are selected and convert from atom-atom to island-island interactions.

new_atm_grp(*atoms*, *features=None*, *interactions=None*)

Create a new *AtomGroup* object for `atoms` if one does not exist yet. Otherwise, return the existing *AtomGroup* object.

Parameters

- **atoms** (iterable of *ExtendedAtom*) – A sequence of atoms.
- **features** (iterable of `ChemicalFeature`, optional) – If provided, add `features` to a new or an already existing *AtomGroup* object.
- **interactions** (iterable of *InteractionType*, optional) – If provided, add `interactions` to a new or an already existing *AtomGroup* object.

Returns A new or an already existing *AtomGroup* object.

Return type *AtomGroup*

remove_atm_grps(*atm_grps*)

Remove one or more *AtomGroup* objects from `atm_grps` and automatically update `child_dict`.

Any recursive references to the removed objects will also be cleared.

save(*output_file*, *compressed=True*)

Write the pickled representation of the *AtomGroupsManager* object to the file `output_file`.

Parameters

- **output_file** (*str*) – The output file.
- **compressed** (*bool, optional*) – If True (the default), compress the pickled representation as a gzip file (.gz).

Raises **FileNotCreated** – If the file could not be created.

property size

The number of atom groups in `atm_grps`.

Type int, read-only

property summary

The number of physicochemical features in `atm_grps`.

Type dict, read-only

class PseudoAtomGroup(*parent_grp*, *atoms*, *features=None*, *interactions=None*)

Bases: *luna.mol.groups.AtomGroup*

Represent only the atoms from an *AtomGroup* object that are involved in an interaction.

Currently, this class is only used during the generation of LUNA's fingerprints.

Parameters

- **parent_grp** (*AtomGroup*) – The atom group that contains the subset of atoms `atoms`.
- **atoms** (iterable of *ExtendedAtom*) – A sequence of atoms.
- **features** (iterable of `ChemicalFeature`, optional) – A sequence of chemical features.

- **interactions** (iterable of *InteractionType*, optional) – A sequence of interactions established by an atom group.

luna.mol.standardiser module

class ResidueStandard(*value*)

Bases: enum.Enum

An enumeration of protonation states for standard residues.

CYM = 5

CYS = 4

HID = 1

HIE = 2

HIP = 3

class ResiduesStandardiser(*break_metal_bonds=False*, *his_type=ResidueStandard.HIE*)

Bases: object

Standardize residues.

Parameters

- **break_metal_bonds** (*bool*) – If True, break covalent bonds with metals and correct the topology of the involved atoms.
- **his_type** ({*HID*, *HIE*, *HIP*}) – Define which histidine protonation state to use. Currently, this option is still not been used.

standardise(*atom_pairs*)

Standardize residues.

Parameters **atom_pairs** (iterable of tuple of (*MolWrapper*, Atom)) – The atoms to standardise.

luna.mol.templates module

class LigandExpoTemplate(*lig_expo_file='/home/docs/checkouts/readthedocs.org/user_builds/luna-toolkit/checkouts/latest/luna/data/ligand_expo.tsv'*)

Bases: *luna.mol.templates.Template*

Standardize small molecules based on templates (SMILES) from LigandExpo.

Parameters **lig_expo_file** (*str*) – The LigandExpo file containing the SMILES and ligand ids.

assign_bond_order(*mol_obj*, *lig_id*)

Assign bond order to a molecular object based on its LigandExpo SMILES.

Parameters

- **mol_obj** (*MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule) – A molecule to standardise.
- **lig_id** (*str*) – The ligand identifier (PDB id) in LigandExpo.

Returns **new_mol** – A standardized molecular object of the same type as mol_obj.

Return type *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule

property data
The LigandExpo data.

Type pandas.DataFrame

get_ligand_smiles(*lig_id*)
Get SMILES for the ligand lig_id.

Parameters **lig_id** (*str*) – The ligand identifier (PDB id) in LigandExpo.

Returns **smiles** – The ligand SMILES.

Return type str or None

class Template
Bases: object

Standardize small molecules based on templates.

assign_bond_order()
Assign bond order to a molecular object. However, this method is not implemented by default. Instead, you should use a class that inherits from *Template* and implements *assign_bond_order()*. An example is the class *LigandExpoTemplate* that assigns bonds order based on ligands SMILES from LigandExpo. Therefore, you should define your own logic beyond *assign_bond_order()* that meets your goals.

luna.mol.validator module

class MolValidator(*charge_model=<luna.mol.charge_model.OpenEyeModel object>*, *fix_nitro=True*, *fix_amidine_and_guanidine=True*, *fix_valence=True*, *fix_charges=True*)
Bases: object

Validate and fix molecules with the errors most commonly found when parsing PDB files with Open Babel.

Parameters

- **charge_model** (*ChargeModel*) – A charge model object. By default, the implementation of OpenEye charge model is used.
- **fix_nitro** (*bool*) – If True, fix nitro groups whose nitrogens are perceived as having a valence of 5 and that are double-bonded to the oxygens. SMILES representation of the error: '[$([NX3v5]([!#8])(=O)=O)]'.
- **fix_amidine_and_guanidine** (*bool*) – If True, fix amidine and guanidine-like groups. When the molecule is ionized, it may happen that the charge is incorrectly assigned to the central carbon, which ends up with a +1 charge and the nitrogen double-bonded to it ends up with a +0 charge.
- **fix_valence** (*bool*) – If True, fix the valence of atoms. Currently, we only detect and fix errors of quaternary ammonium nitrogens. These charged atoms are sometimes perceived as having a valence equal to 5 and no charge, i.e., Open Babel considers those nitrogens as hypervalent.
- **fix_charges** (*bool*) – If True, fix charges on basis of the charge model charge_model.

validate_mol(*mol_obj*)
Validate molecule mol_obj.

It will first try to fix the provided molecule according to the initialization flags. If any errors are detected and if it fails to fix them, then this function may return False, i.e., the molecule is invalid.

Parameters **mol_obj** (*MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule) – The molecule.

Returns **is_valid** – If the molecule is valid or not.

Return type bool

class RDKitValidator(*sanitize_opts=rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_ALL*)
Bases: object

Check if RDKit molecular objects are valid or not.

Parameters **sanitize_opts** (rdkit.Chem.rdchem.SanitizeFlags) – Sanitization operations to be carried out.

is_valid(*rdk_mol*)
Try to sanitize the molecule rdk_mol. If it succeeds, returns True. Otherwise, returns False.

Parameters **rdk_mol** (rdkit.Chem.rdchem.Mol)

Returns **is_valid** – If the molecule is valid or not.

Return type bool

Module contents

luna.MyBio package

Submodules

luna.MyBio.extractor module

class Extractor(*entity*)
Bases: object

Extract chains or residues from entity.

Parameters **entity** (Model or Chain) – A model or chain object from where chains and residues will be extracted, respectively.

property entity

extract_chains(*chains*, *output_file*)
Extract chains from entity and save it to output_file.

Parameters

- **chains** (*iterable of str*) – A sequence of chains to extract from model entity.
- **output_file** (*str*) – Save extracted chains to this file.

extract_residues(*residues*, *output_file*)
Extract residues from entity and save it to output_file.

Parameters

- **residues** (*iterable*) – A sequence of residues to extract from chain entity.
- **output_file** (*str*) – Save extracted residues to to this file.

luna.MyBio.selector module

class AtomSelector(*entries*, ***kwargs*)
Bases: *luna.MyBio.selector.Selector*

Select atoms defined at `entries`.

Parameters **entries** (iterable of `Atom`) – Sequence of atom objects.

accept_atom(*atom*)
Decide if the atom is valid or not.

class ChainSelector(*entries*, ***kwargs*)
Bases: *luna.MyBio.selector.Selector*

Select atoms from chains defined at `entries`.

Parameters **entries** (iterable of `Chain` instances) – Sequence of chain objects.

accept_atom(*atom*)
Decide if the atom is valid or not.

accept_chain(*chain*)
Decide if the chain is valid or not.

accept_residue(*res*)
Decide if the residue is valid or not.

class ResidueSelector(*entries*, ***kwargs*)
Bases: *luna.MyBio.selector.Selector*

Select atoms from residues and other molecules defined at `entries`.

Parameters **entries** (iterable of `Residue`) – Sequence of residue objects.

accept_atom(*atom*)
Decide if the atom is valid or not.

accept_residue(*res*)
Decide if the residue is valid or not.

class ResidueSelectorBySeqNum(*entries*, ***kwargs*)
Bases: *luna.MyBio.selector.Selector*

Select atoms from compounds defined at `entries`.

Parameters **entries** (*iterable of int*) – Sequence of residue sequence numbers.

accept_atom(*atom*)
Decide if the atom is valid or not.

accept_residue(*res*)
Decide if the residue is valid or not.

class Selector(*keep_hydrog=True*, *keep_altloc=True*, *altloc=('A', '1')*)
Bases: `luna.MyBio.PDB.PDBIO.Select`

Select atoms for PDB output.

Parameters

- **keep_hydrog** (*bool*) – If True (default), keeps all hydrogens. Otherwise, filter them out. Hydrogens are atoms whose element is either "H" or "D" (deuterium).
- **keep_altloc** (*bool*) – If True (default), keeps all atoms. Otherwise, keeps only atoms whose alternate location is in the list `altloc`.

- **altloc** (*iterable*) – List of valid alternate location identifiers. The default valid values are "A" and "1".

`accept_atom`(*atom*)
Decide if the atom is valid or not.

luna.MyBio.util module

`biopython_entity_to_mol`(*entity*, *select=<Select all>*, *validate_mol=True*, *standardize_mol=True*, *template=None*, *add_h=False*, *ph=None*, *break_metal_bonds=False*, *mol_obj_type='rdkit'*, *wrapped=True*, *openbabel='obabel'*, *tmp_path=None*, *keep_tmp_files=False*)
Convert an object `Entity` to a molecular object (*MolWrapper*, `rdkit.Chem.rdchem.Mol`, or `openbabel.pybel.Molecule`).

Parameters

- **entity** (`Entity`) – The entity to be converted.
- **select** (`Select`) – Decide which atoms will be consired. By default, all atoms are accepted.
- **validate_mol** (*bool*) – If True, validate the converted molecule with *MolValidator*.
- **standardize_mol** (*bool*) – If True, standardize the converted molecule. Currently, only residues are standardized as it uses *luna.mol.standardiser.ResiduesStandardiser* by default.
- **template** (`luna.mol.template.Template`, optional) – Fix the converted molecule's bond order based on the bond orders in a template molecule. The `template` should implement assign_bond_order().
- **add_h** (*bool*) – If True, add hydrogen to the converted molecule.
- **ph** (*float, optional*) – Add hydrogens considering pH `ph`.
- **break_metal_bonds** (*bool*) – If True, remove covalent bonds between residues and metals and fix the residue bond order.
- **mol_obj_type** (*{"rdkit", "openbabel"}*) – If "rdkit", parse the converted molecule with RDKit and return an instance of `rdkit.Chem.rdchem.Mol`. If "openbabel", parse the converted molecule with Open Babel and return an instance of `openbabel.pybel.Molecule`. If `wrapped` is True, the molecular object will be wrapped with *MolWrapper*.
- **wrapped** (*bool*) – If True, wrap the molecular object with *MolWrapper*.
- **openbabel** (*str*) – Pathname to Open Babel. Even if `mol_obj_type` is set to 'rdkit', this function uses Open Babel to properly parse PDB files and to add hydrogens to the molecule considering different pHs.
- **tmp_path** (*str*) – A temporary directory to where temporary files will be saved. If not provided, the system's default temporary directory will be used instead.
- **keep_tmp_files** (*bool*) – If True, keep all temporary files. Otherwise, removes them in the end.

Returns

- **mol_obj** (*MolWrapper*, `rdkit.Chem.rdchem.Mol`, or `openbabel.pybel.Molecule`) – The converted molecule.
- **ignored_atoms** (*list*) – List of ignored atoms. Currently, ignored atoms may contain only metals.

download_pdb(*pdb_id*, *output_path='.'*, *overwrite=False*)
Download a PDB file from RCSB.org.

Parameters

- **pdb_id** (*str*) – 4-symbols structure Id from PDB (e.g. 3J92).
- **output_path** (*str*) – Put the PDB file in this directory.
- **overwrite** (*bool*) – If True, overwrite any existing PDB files.

entity_to_string(*entity*, *select=<Select all>*, *write_conects=True*, *write_end=True*, *preserve_atom_numbering=True*)
Convert a Structure object (or a subset of a `Structure` object) to string.

This function works on a structural level. That means if `entity` is not a `Structure` object, the structure will be recovered directly from `entity`. Therefore, use `select` to select specific chains, residues, and atoms from the structure object.

Parameters

- **entity** (`Entity`) – The PDB object to be converted.
- **select** (`Select`) – Decides which atoms will be saved at the output. By default, all atoms are accepted.
- **write_conects** (*bool*) – If True, writes CONECT records.
- **write_end** (*bool*) – If True, writes the END record.
- **preserve_atom_numbering** (*bool*) – If True, preserve the atom numbering. Otherwise, re-enumerate the atom serial numbers.

Returns The converted `entity`.

Return type str

get_entity_from_entry(*entity*, *entry*, *model=0*)
Get a `Chain` or `Residue` instance based on the provided entry `entry`.

Parameters

- **entity** (`Entity`) – The PDB object to recover the target entry from.
- **entry** (`Entry`) – The target entry.
- **model** (*int*) – The PDB model where the entry could be found. The default value is 0.

Returns The target entry.

Return type `Entity`

Raises

- **MoleculeNotFoundError** – If the entry's molecule was not found in `entity`.
- **ChainNotFoundError** – If the entry's chain was not found in `entity`.

get_residue_cov_bonds(*residue*, *select=<Select all>*)
Get covalently bound atoms from residues or other molecules.

Parameters

- **residue** (`Residue`) – The residue or other molecule from which covalently bound atoms will be recovered.
- **select** (`Select`) – Decides which atoms will be consired. By default, all atoms are accepted.

Returns List of pairs of covalently bound atoms.

Return type list of tuple(`Atom`, `Atom`)

`get_residue_neighbors`(*residue*, *select=<Select all>*)
Get all neighbors from a residue.

In the case of an amino acid that is part of a peptide bond, its neighbors are any predecessor or successor residues. The same idea applies to nucleic acids.

Parameters

- **residue** (`Residue`) – The residue or other molecule from which covalently bound molecules will be recovered.
- **select** (`Select`) – Decides which atoms will be consired. By default, all atoms are accepted.

Returns **neighbors** – A dictionary containing the predecessor (`previous`) or successor (`next`) molecules.

Return type dict of {str : `Residue`}

`is_covalently_bound`(*atm1*, *atm2*)
Verifies if atoms `atm1` and `atm2` are covalently bound.

`parse_from_file`(*pdb_id*, *file*)
Read a PDB file and return a `Structure` object.

Parameters

- **pdb_id** (*str*) – The structure identifier.
- **file** (*str*) – Pathname of the PDB file.

Returns **structure** – The parsed PDB file as a `Structure` object.

Return type `Structure`

Raises `PDBNotReadError` – If the PDB file could not be parsed.

`save_to_file`(*entity*, *output_file*, *select=<Select all>*, *write_conects=True*, *write_end=True*, *preserve_atom_numbering=True*)
Write a Structure object (or a subset of a `Structure` object) into a file.

Parameters

- **entity** (`Entity`) – The PDB object to be saved.
- **output_file** (*str*) – Save the selected atoms to this file.
- **select** (`Select`) – Decide which atoms will be saved at the PDB output. By default, all atoms are accepted.
- **write_conects** (*bool*) – If True, write CONECT records.
- **write_end** (*bool*) – If True, write the END record.
- **preserve_atom_numbering** (*bool*) – If True, preserve the atom numbering. Otherwise, re-enumerate the atom serial numbers.

Module contents

luna.util package

Submodules

luna.util.config module

class Config(*conf_file*)
Bases: object

Parser for configuration files.

Parameters **conf_file** (*str*) – The pathname for the configuration file.

Variables **~Config.config** (configparser.ConfigParser) – The parsed configuration file.

get_section_map(*section*)
Try to access the section section from the parsed configuration file.

luna.util.file module

create_directory(*path*, *clear=False*)
Create the directory pathaname path.

Parameters

- **path** (*str*) – The directory pathname to be created.
- **clear** (*bool*) – If True, clear the directory if already exists. The default value is False.

detect_compression_format(*filename*)
Attempts to detect compression format from the filename extension. Returns None if no format could be detected.

generate_json_file(*json_data*, *output_file*, *indent=4*, *sort_keys=True*)
Serialize json_data to a JSON formatted string and save it at output_file.

Parameters

- **json_data** (*object*) – The data to be serialized.
- **output_file** (*str*) – The output file where the serialized data will be saved.
- **indent** (*int*) – If indent is a non-negative integer or string, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0, negative, or "" will only insert newlines. None selects the most compact representation. Using a positive integer indent indents that many spaces per level. If indent is a string (such as " "), that string is used to indent each level. The default value is 4.
- **sort_keys** (*bool*) – If sort_keys is True, the output of dictionaries will be sorted by key.

generic_splitext(*path*, *max_split=None*)
Split the pathname path into a pair (filename, ext).

Parameters

- **path** (*str*) – The pathname.
- **max_split** (*int or None*) – Specifies how many splits to do. The default value is None, which is "all occurrences".

Returns

- **filename** (*str*) – The filename.
- **ext** (*str*) – The file extension.

`get_file_format`(*path*, *max_split=None*, *ignore_compression=False*)
Detect the file format from pathname `path`.

Parameters

- **path** (*str*) – The pathname.
- **max_split** (*int or None*) – Specifies how many splits to do. The default value is None, which is "all occurrences".
- **ignore_compression** (*bool, optional*) – Ignore compression format. The default value is False, which does not ignore compression format.

Returns **filename** – The file format.

Return type str

`get_filename`(*path*, *max_split=None*)
Detect the filename from pathname `path`.

Parameters

- **path** (*str*) – The pathname.
- **max_split** (*int or None*) – Specifies how many splits to do. The default value is None, which is "all occurrences".

Returns **filename** – The filename.

Return type str

`is_directory_valid`(*path*)
Check if `path` exists and if it is in fact a directory.

`is_file_valid`(*path*)
Check if `path` exists and if it is in fact a file.

`new_unique_filename`(*path*, *size=32*, *chars='ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'*, *retries=5*)
Generate a new unique random pathname.

Parameters

- **path** (*str*) – The target pathname.
- **size** (*int, optional*) – The size of the new filename. The default value is 32.
- **chars** (*iterable, optional*) – A sequence of characters to choose from. The default value is 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'.
- **retries** (*int, optional*) – The function will keep trying to generate a unique filename (not exist in the pathname `path`) until the maximum number of retries `retries` is reached.

Returns **unique_pathname** – A new random unique pathname (path + random filename).

Return type str

`parse_json_file`(*json_file*)
Deserialize the JSON file `json_file`.

Parameters **json_file** (*str*) – The input JSON file.

Returns **json_data** – The deserialized data.

Return type object

`pickle_data`(*data*, *output_file*, *compressed=True*)
Write the pickled representation of the object `data` to the file `output_file`.

Parameters

- **data** (*object*) – The object to be pickled.
- **output_file** (*str*) – The output file where the pickled representation will be saved.
- **compressed** (*bool, optional*) – If True (the default), compress the pickled representation as a gzip file (.gz).

Raises `FileNotCreated` – If the file could not be created.

`remove_directory`(*path*, *only_empty_paths=False*)
Remove the directory given by the pathname `path`.

Parameters

- **path** (*str*) – The pathname.
- **only_empty_paths** (*bool, optional*) – If True, do not remove non-empty directories. The default value is False, which removes all directories.

`remove_files`(*files*)
Remove the provided files.

Parameters **files** (*iterable*) – An iterable object containing a sequence of files to be removed.

`unpickle_data`(*input_file*)
Read the pickled representation of an object from the file `input_file` and return the reconstituted object hierarchy specified therein. `input_file` can be a gzip-compressed file.

Raises `PKLNotReadError` – If the file could not be loaded.

`validate_directory`(*path*)
Validate `path` as a directory.

`validate_file`(*path*)
Validate `path` as a file.

`validate_filesystem`(*path*, *type*)
Validate if a file or directory exists and if it has the expected type.

Parameters

- **path** (*str*) – The pathname of the file or directory to be validated.
- **type** (*{'file', 'directory'}*) – The filesystem type.

Raises

- `FileNotFoundError` – If the file or directory given by the pathname `path` does not exist.
- `IsADirectoryError` – If a file is provided but a directory is found instead at the pathname `path`.
- `NotADirectoryError` – If the filesystem given by the pathname `path` is not a directory.

luna.util.jobs module

class ArgsGenerator(*generator*, *nargs*)

Bases: object

Custom generator that implements __len__(). This class can be used in conjunction with *ProgressTracker* in cases where the tasks are obtained from generators. Note that *ProgressTracker* requires a pre-defined number of tasks to calculate the progress, therefore a standard generator cannot be used directly as it does not implement __len__(). Then, with *ArgsGenerator*, one may take advantage of generators and *ProgressTracker* by explicitly providing the number of tasks that will be generated.

Parameters

- **generator** (*generator*) – The tasks generator.
- **nargs** (*int*) – The number of tasks that will be generated.

class ParallelJobs(*nproc=1*)

Bases: object

Executes a set of tasks in parallel (JoinableQueue) or sequentially.

Parameters **nproc** (*int or None*) – The number of CPUs to use. The default value is the `maximum number of CPUs - 1`. If `nproc` is None, 0, or 1, run the jobs sequentially. Otherwise, use the `maximum number of CPUs - 1`.

Variables

- **~ParallelJobs.nproc** (*int*) – The number of CPUs to use.
- **~ParallelJobs.progress_tracker** (ProgressTracker) – A *ProgressTracker* object to track the tasks' progress.

run_jobs(*args*, *consumer_func*, *output_file=None*, *proc_output_func=None*, *output_header=None*, *job_name=None*)

Run a set of tasks in parallel or sequentially according to the `nproc`.

Parameters

- **args** (iterable of iterables, *ArgsGenerator*) – A sequence of arguments to be provided to the consumer function `consumer_func`.
- **consumer_func** (*function*) – The function that will be executed for each set of arguments in `args`.
- **output_file** (*str, optional*) – Save outputs to this file. If `proc_output_func` is not provided, it tries to save a stringified version of each output data. Otherwise, it executes `proc_output_func` first and its output will be printed to the output file instead.

  Note: if `proc_output_func` is provided but not `output_file`, a new random unique filename will be generated and the file will be saved in the current directory.
- **proc_output_func** (*function, optional*) – Post-processing function that is executed for each output data produced by `consumer_func`.
- **output_header** (*str, optional*) – A header for the output file.
- **job_name** (*str, optional*) – A name to identify the job.

Return type *ProgressResult*

class Sentinel

Bases: object

Custom sentinel to stop workers

luna.util.progress_tracker module

class ProgressData(*input_data*, *proc_time*, *output_data=None*, *exception=None*, *func=None*)
Bases: object

Custom data structure to store data generated during the execution of a single task.

Parameters

- **input_data** (*any*) – The input data of the executed task.
- **proc_time** (*float*) – How long a task took to be executed.
- **output_data** (*any, optional*) – The output data produced by a given task. The data type is the same as the executed function's return.
- **exception** (Exception, optional) – If an exception was raised, then `exception` stores an Exception object. Otherwise, `exception` will be set to None.
- **func** (*function, optional*) – The executed function for reference.

class ProgressResult(*results=None*)
Bases: object

Custom iterable class to store *ProgressData* objects as they are produced during the execution of a set of tasks.

This class implements append() and __iter__() by default.

Parameters **results** (*list, optional*) – A pre-populated list of *ProgressData* objects.

Variables **~ProgressResult.results** (*list*) – The list of *ProgressData* objects.

append(*r*)
Add a new *ProgressData* object to `results`

property errors
Errors from each *ProgressData* object stored in `results`. Each tuple contains the input and the exception raised during the execution of a task with that input.

Type list of tuple

property inputs
Inputs from each *ProgressData* object stored in `results`.

Type list

property outputs
Outputs from each *ProgressData* object stored in `results`. Each tuple contains the input and the output produced for that input.

Type list of tuple

class ProgressTracker(*ntasks*, *queue*, *task_name=None*)
Bases: object

A progress tracker for tasks queued in `queue`.

Parameters

- **ntasks** (*int*) – The number of tasks to be executed.
- **queue** (Queue) – A queue to track the tasks' progress.
- **task_name** (*str, optional*) – A name to identify the set of tasks.

Variables

- **~ProgressTracker.ntasks** (*int*) – The number of tasks to be executed.
- **~ProgressTracker.queue** (Queue) – The queue to track the tasks' progress from.
- **~ProgressTracker.task_name** (*str, optional*) – The name to identify the set of tasks.
- **~ProgressTracker.results** (ProgressResult) – Store results and any errors found during the tasks' processing.
- **~ProgressTracker.nerrors** (*int*) – Number of errors.

property avg_running_time
Average running time.

Type float

end()
Finish the progress tracker.

property progress
Current number of tasks executed.

Type int

property running_time
Total running time.

Type float

start()
Start the progress tracker.

luna.util.stringcase module

Functions to convert strings.

alphanumcase(*string*)
Cuts all non-alphanumeric symbols, i.e. cuts all expect except 0-9, a-z and A-Z.

Parameters **string** (*str*) – String to convert.

Returns **string** – String with cutted non-alphanumeric symbols.

Return type str

backslashcase(*string*)
Convert string into spinal case. Join punctuation with backslash.

Parameters **string** (*str*) – String to convert.

Returns **string** – Spinal cased string.

Return type str

camelcase(*string*)
Convert string into camel case.

Parameters **string** (*str*) – String to convert.

Returns **string** – Camel case string.

Return type str

capitalcase(*string*)
Convert string into capital case. First letters will be uppercase.

Parameters **string** (*str*) – String to convert.

Returns **string** – Capital case string.

Return type str

Examples

```
>>> import luna.util.stringcase as sc
>>> new_str = sc.camelcase("This is an interesting example.")
>>> print(new_str)
'this is an interesting example.'
```

constcase(*string*)

Convert string into upper snake case. Join punctuation with underscore and convert letters into uppercase.

Parameters **string** (*str*) – String to convert.

Returns **string** – Const cased string.

Return type str

dotcase(*string*)

Convert string into dot case. Join punctuation with dot.

Parameters **string** (*str*) – String to convert.

Returns **string** – Dot cased string.

Return type str

lowercase(*string*)

Convert string into lower case.

Parameters **string** (*str*) – String to convert.

Returns **string** – Lowercase case string.

Return type str

Examples

```
>>> import luna.util.stringcase as sc
>>> new_str = sc.camelcase("This is an Interesting EXAMPLE.")
>>> print(new_str)
'this is an interesting example.'
```

pascalcase(*string*)

Convert string into pascal case.

Parameters **string** (*str*) – String to convert.

Returns **string** – Pascal case string.

Return type str

pathcase(*string*)

Convert string into path case. Join punctuation with slash.

Parameters **string** (*str*) – String to convert.

Returns **string** – Path cased string.

Return type str

sentencecase(*string*)
Convert string into sentence case. First letter capped and each punctuations are joined with space.

Parameters **string** (*str*) – String to convert.

Returns **string** – Sentence cased string.

Return type str

snakecase(*string*)
Convert string into snake case. Join punctuation with underscore.

Parameters **string** (*str*) – String to convert.

Returns **string** – Snake cased string.

Return type str

spinalcase(*string*)
Convert string into spinal case. Join punctuation with hyphen.

Parameters **string** (*str*) – String to convert.

Returns **string** – Spinal cased string.

Return type str

titlecase(*string*)
Convert string into sentence case. First letter capped while each punctuations is capitalsed and joined with space.

Parameters **string** (*str*) – String to convert.

Returns **string** – Title cased string.

Return type str

trimcase(*string*)
Convert string into trimmed string.

Parameters **string** (*str*) – String to convert.

Returns **string** – Trimmed case string.

Return type str

Examples

```
>>> import luna.util.stringcase as sc
>>> new_str = sc.camelcase("      This is an Interesting EXAMPLE.     ")
>>> print(new_str)
'This is an Interesting EXAMPLE.'
```

uppercase(*string*)
Convert string into upper case.

Parameters **string** (*str*) – String to convert.

Returns **string** – Uppercase case string.

Return type str

Examples

```
>>> import luna.util.stringcase as sc
>>> new_str = sc.camelcase("This is an Interesting EXAMPLE.")
>>> print(new_str)
'THIS IS AN INTERESTING EXAMPLE.'
```

Module contents

class ColorPallete(*color_map=None*, *default_color=(255, 255, 255)*)

Bases: object

add_color(*key*, *color*)

get_color(*key*)

get_normalized_color(*key*)

get_unnormalized_color(*key*)

func_call_to_str(*func*, **args*, ***kwargs*)

hex2rgb(*hexcode*)

iter_to_chunks(*l*, *n*)

new_random_string(*size=32*, *chars='ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'*)

Generate a new random string of size `size` containing only the characters provided in `chars`.

Parameters

- **size** (*int, optional*) – The size of the new string. The default value is 32.
- **chars** (*iterable, optional*) – A sequence of characters to choose from. The default value is 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'.

Returns **random_string** – The new random string.

Return type str

rgb2hex(*r*, *g*, *b*)

luna.wrappers package

Submodules

luna.wrappers.base module

class AtomWrapper(*atm_obj*, *mol_obj=None*)

Bases: object

This class provides util functions to access atomic properties and other information from RDKit and Open Babel objects.

Parameters

- **atm_obj** (*AtomWrapper*, `rdkit.Chem.rdchem.Atom`, or `openbabel.OBAtom`) – An atom to wrap.

- **mol_obj** (*MolWrapper*, rdkit.Chem.rdchem.Mol, openbabel.pybel.Molecule, or None) – The molecule that contains the atom `atom`. If None, the molecule is recovered directly from the atom object.

Raises **AtomObjectTypeError** – If the atom object is not an instance of *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule.

property atm_obj
The wrapped atom object.

Type rdkit.Chem.rdchem.Atom or openbabel.OBAtom

get_atomic_invariants()
Get the atomic invariants of this atom.

Atomic invariants are derived from ECFP[1] and E3FP[2] and consists of seven fields:

- Number of heavy atoms;
- Valence - Number of hydrogens;
- Atomic number;
- Isotope number;
- Formal charge;
- Number of hydrogens;
- If the atom belongs to a ring or not.

Return type list

get_atomic_mass()
Get this atom's atomic mass given by standard IUPAC average molar mass.

Return type float

get_atomic_num()
Get this atom's atomic number.

Return type int

get_bonds(*wrapped=True*)
Get this atom's bonds.

Parameters **wrapped** (*bool*) – If True, wrap each bond with *BondWrapper*.

Return type iterable of *BondWrapper*, rdkit.Chem.rdchem.Bond, or openbabel.OBBond

get_charge()
Get this atom's formal charge.

Return type int

get_degree()
Get this atom's total degree.

Return type int

[1] Rogers D & Hahn M. Extended-connectivity fingerprints. *J. Chem. Inf. Model.* **50**: 742-54 (2010). doi 10.1021/ci100050t

[2] Axen SD, Huang XP, Caceres EL, Gendelev L, Roth BL, Keiser MJ. A Simple Representation Of Three-Dimensional Molecular Structure. *J. Med. Chem.* **60** (17): 7393–7409 (2017). doi 10.1021/acs.jmedchem.7b00696 bioRxiv 136705 F1000Prime RECOMMENDED

get_h_count()
Get the total number of hydrogens (implicit and explicit) bound to this atom.

Return type int

get_id()
Get this atom's unique id.

When using RDKit, *get_id()* and *get_idx()* returns the same value.

Return type int

get_idx()
Get this atom's internal index within a molecule.

Return type int

get_isotope()
Get this atom's isotope number.

Return type int

get_mass()
Get this atom's exact atomic mass, which can vary given the isotope.

Return type float

get_neighbors(*wrapped=True*)
Get all atoms that are bound to this atom.

Parameters **wrapped** (*bool*) – If True, wrap each atom with *AtomWrapper*.

Return type iterable of *AtomWrapper*, rdkit.Chem.rdchem.Atom, or openbabel.OBAtom

get_neighbors_number(*only_heavy_atoms=False*)
Get the number of atoms bound to this atom.

Parameters **only_heavy_atoms** (*bool*) – If True, count only heavy atoms.

Return type int

get_parent(*wrapped=True*)
Get the molecule that contains this atom.

Parameters **wrapped** (*bool*) – If True, wrap the molecule with *MolWrapper*.

Return type *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule

get_symbol()
Get the element symbol of this atom.

Return type str

get_valence()
Get this atom's total valence (implicit and explicit).

has_bond_type(*bond_type*)
Check if this atom has a bond of type bond_type.

Parameters **bond_type** (*BondType*)

Raises **IllegalArgumentError** – If the informed bond type is not an instance of *BondType*.

has_only_bond_type(*bond_type*)
Check if this atom has only bonds of type bond_type.

Parameters **bond_type** (*BondType*)

Raises **IllegalArgumentError** – If the informed bond type is not an instance of *BondType*.

is_aromatic()
Check if this atom is aromatic.

is_in_ring()
Check if this atom is in a ring.

is_openbabel_obj()
Check if this atom is an Open Babel object.

is_rdkit_obj()
Check if this atom is an RDKit object.

matches_smarts(*smarts*)
Check if this atom matches the substructure through a SMARTS substructure search.

Note: currently, this function only works with molecules read with Open Babel.

Parameters **smarts** (*str*) – A substructure defined as SMARTS.

Returns Whether matches occurred. Return None if the molecule was read with RDKit.

Return type bool or None

Examples

First, let's read a molecule (glutamine) using Open Babel.

```
>>> from luna.wrappers.base import MolWrapper
>>> mol_obj = MolWrapper.from_smiles("N[C@@H](CCC(N)=O)C(O)=O", mol_obj_type=
↪"openbabel")
```

Now, we'll loop over the list of atoms in the glutamine and check which atom is the amide's carbon. To do so, we can call the function *MolWrapper.get_atoms()*, which, by default, returns *AtomWrapper* objects and then call *matches_smarts()*.

```
>>> for atm in mol_obj.get_atoms():
>>>     print("%d       %s      %s" % (atm.get_idx(), atm.get_symbol(), atm.
↪matches_smarts("C(N)(C)=O")))
1   N   False
2   C   False
3   C   False
4   C   False
5   C   True
6   N   False
7   O   False
8   C   False
9   O   False
10  O   False
```

property parent
The molecule that contains this atom.

Type *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule

set_as_aromatic(*is_aromatic*)
Set whether this atom is aromatic or not.

Parameters **is_aromatic** (*bool*)

set_charge(*charge*)
Set the formal charge of this atom.

Parameters **charge** (*int*)

set_in_ring(*in_ring*)
Set whether this atom belongs to a ring or not.

Parameters **in_ring** (*bool*)

unwrap()
Return the original atomic object.

Return type rdkit.Chem.rdchem.Atom or openbabel.OBAtom

class BondType(*value*)
Bases: enum.Enum

An enumeration of bond types available at RDKit.

AROMATIC = 12

DATIVE = 17

DATIVEL = 18

DATIVEONE = 16

DATIVER = 19

DOUBLE = 2

FIVEANDAHALF = 11

FOURANDAHALF = 10

HEXTUPLE = 6

HYDROGEN = 14

IONIC = 13

ONEANDAHALF = 7

OTHER = 20

QUADRUPLE = 4

QUINTUPLE = 5

SINGLE = 1

THREEANDAHALF = 9

THREECENTER = 15

TRIPLE = 3

TWOANDAHALF = 8

UNSPECIFIED = 0

ZERO = 21

class BondWrapper(*bond_obj*)
Bases: object

This class provides util functions to access bond properties and other information from RDKit and Open Babel objects.

Parameters **bond_obj** (*BondWrapper*, rdkit.Chem.rdchem.Bond, or openbabel.OBBond) – A bond to wrap.

Raises **BondObjectTypeError** – If the bond object is not an instance of *BondWrapper*, rdkit.Chem.rdchem.Bond, or openbabel.pybel.OBBond.

property bond_obj
The wrapped bond object.

Type rdkit.Chem.rdchem.Bond or openbabel.OBBond

get_begin_atom(*wrapped=True*)
Return the bond's first atom.

Parameters **wrapped** (*bool*) – If True, wrap the atom with *AtomWrapper*.

Return type *AtomWrapper*, rdkit.Chem.rdchem.Atom, or openbabel.OBAtom

get_bond_type()
Get the bond type (e.g., single bond).

Return type *BondType*

get_end_atom(*wrapped=True*)
Return the bond's second atom.

Parameters **wrapped** (*bool*) – If True, wrap the atom with *AtomWrapper*.

Return type *AtomWrapper*, rdkit.Chem.rdchem.Atom, or openbabel.OBAtom

get_partner_atom(*atm*, *wrapped=True*)
Get the partner atom that forms this bond with atm.

Parameters

- **atm** (*AtomWrapper*, rdkit.Chem.rdchem.Atom, or openbabel.OBAtom) – Get the partner of this atom.
- **wrapped** (*bool*) – If True, wrap the partner atom with *AtomWrapper*.

Return type *AtomWrapper*, rdkit.Chem.rdchem.Atom, or openbabel.OBAtom

is_aromatic()
Check if this bond is aromatic or not.

is_openbabel_obj()
Check if this bond is an Open Babel object.

is_rdkit_obj()
Check if this bond is an RDKit object.

set_as_aromatic(*is_aromatic*)
Set if this bond is aromatic or not.

Parameters **is_aromatic** (*bool*)

set_bond_type(*bond_type*)
Set the type of the bond as a bond_type.

Parameters **bond_type** (*BondType*)

unwrap()
Return the original bond object.

Return type rdkit.Chem.rdchem.Bond or openbabel.OBBond

class MolWrapper(*mol_obj*)

Bases: object

This class provides util functions to access molecule properties and other information from RDKit and Open Babel objects.

Parameters **mol_obj** (*MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule) – A molecule to wrap.

Raises **MoleculeObjectTypeError** – If the molecular object is not an instance of *MolWrapper*, rdkit.Chem.rdchem.Mol, or openbabel.pybel.Molecule.

as_openbabel()

If the molecule is an RDKit object, convert it to an Open Babel object.

as_rdkit()

If the molecule is an Open Babel object, convert it to an RDKit object.

classmethod from_mol_block(*block*, *mol_format*, *mol_obj_type='rdkit'*)

Initialize a molecule from a string block.

Parameters

- **block** (*str*) – The molecular string block.
- **mol_format** (*str*) – Define the format in which the molecule is represented (e.g., 'mol2' or 'mol').
- **mol_obj_type** (*{'rdkit', 'openbabel'}*) – Define which library (RDKit or Open Babel) to use to parse the molecular block. The default value is RDKit.

Return type *MolWrapper*

classmethod from_smiles(*smiles*, *mol_obj_type='rdkit'*, *name=None*)

Initialize a molecule from a SMILES string.

Parameters

- **smiles** (*str*) – The SMILES string. Define the format in which the molecule is represented (e.g., 'mol2' or 'mol').
- **mol_obj_type** (*{'rdkit', 'openbabel'}*) – Define which library (RDKit or Open Babel) to use to parse the molecular block. The default value is RDKit.
- **name** (*str, optional*) – A name to identify the molecule.

Return type *MolWrapper*

Raises **MoleculeObjectError** – If it could not create a molecule from the provided SMILES.

Examples

```
>>> from luna.wrappers.base import MolWrapper
>>> mol_obj = MolWrapper.from_smiles("N[C@@H](CCC(N)=O)C(O)=O", mol_obj_type=
↪"openbabel", name="Glutamine")
```

get_atom_coord_by_id(*atm_id*)

Get the coordinates of an atom given by its id.

Returns Atomic coordinates (x, y, z).

Return type array_like of float (size 3)

get_atoms(*wrapped=True*)
Get all molecule's atoms.

Parameters **wrapped** (*bool*) – If True, wrap all atoms with *AtomWrapper*.

Return type iterable of *AtomWrapper*, rdkit.Chem.rdchem.Atom, or openbabel.OBAtom

get_bonds(*wrapped=True*)
Get all molecule's bonds.

Parameters **wrapped** (*bool*) – If True, wrap all bonds with *BondWrapper*.

Return type iterable of *BondWrapper*, rdkit.Chem.rdchem.Bond, or openbabel.OBBond

get_name()
Get the molecule name.

Return type str

get_num_heavy_atoms()
Get the number of heavy atoms in this molecule.

get_obj_type()
Get the object type ("rdkit" or "openbabel").

has_name()
Check if this molecule has a name.

is_openbabel_obj()
Check if this molecule is an Open Babel object.

is_pybel_obj()
Check if this molecule is a Pybel object.

is_rdkit_obj()
Check if this molecule is an RDKit object.

property mol_obj
The wrapped molecular object.

Type rdkit.Chem.rdchem.Mol or openbabel.pybel.Molecule

set_name(*name*)
Set a name to this molecule.

Parameters **name** (*str*)

to_mol_block()
Return the MOL string block for this molecule.

to_pdb_block()
Return the PDB string block for this molecule.

to_smiles()
Return the canonical SMILES string for this molecule.

unwrap()
Return the original molecular object.

Return type rdkit.Chem.rdchem.Mol or openbabel.pybel.Molecule.

class OBBondType(*value*)
Bases: enum.Enum

An enumeration of bond types available at Open Babel.

AROMATIC = 5

DOUBLE = 2

SINGLE = 1

TRIPLE = 3

References

luna.wrappers.cif module

get_atom_names_by_id(*cif_file*)

Read a single-molecule CIF file and return the molecule's atom names.

In the current version, if applied on multi-molecular CIF files, only the first molecule's atom names are returned.

Return type dict

luna.wrappers.obabel module

convert_molecule(*infile*, *output*, *infile_format=None*, *output_format=None*, *opts=None*, *openbabel='obabel'*)

Convert a molecular file to another format using Open Babel.

Parameters

- **infile** (*str*) – The pathname of the molecular file to be converted.
- **output** (*str*) – Save the converted molecule to this file.
- **infile_format** (*str, optional*) – The molecular format of `infile`. If not provided, the format will be defined by the file extension.
- **output_format** (*str, optional*) – The molecular format of `output`. If not provided, the format will be defined by the file extension.
- **opts** (*dict*) – A set of convertion options. Check Open Babel to discover which options are available.
- **openbabel** (*str, optional*) – The Open Babel binary location. If not provided, the default binary ('obabel') will be used.

Raises **InvalidFileFormat** – If the provided molecular formats are not accepted by Open Babel.

Examples

In this example, we will convert the molecule ZINC000007786517 from the format MOL to MOL2 and add hydrogens to it considering a pH of 7 (option "p").

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.wrappers.obabel import convert_molecule
>>> convert_molecule(infile=f"{LUNA_PATH}/tutorial/inputs/ZINC000007786517.mol",
...                  output="example.mol2",
...                  opts={"p": 7})
```

mol_to_svg(*infile*, *output*, *opts=None*)

Depict a molecule as SVG using Open Babel.

Parameters

- **infile** (*str*) – The pathname of a molecular file.
- **output** (*str*) – Save the SVG to this file.
- **opts** (*dict*) – A set of depiction options. Check Open Babel to discover which options are available.

Examples

In this example, we will depict the molecule ZINC000007786517 as SVG. The following options will be used:

- C: do not draw terminal C (and attached H) explicitly;
- d: do not display the molecule name;
- P: image size in pixels.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.wrappers.obabel import mol_to_svg
>>> mol_to_svg(infile=f"{LUNA_PATH}/tutorial/inputs/ZINC000007786517.mol",
...            output="example.svg",
...            opts={"C": None, "d": None, "P": 500})
```

luna.wrappers.pymol module

class PymolSessionManager(*show_cartoon=False*, *bg_color='white'*, *add_directional_arrows=True*, *show_res_labels=True*, *inter_color=<luna.util.ColorPallete object>*, *pse_export_version='1.8'*)

Bases: `object`

Class to start, manage, and save Pymol sessions. This class provides useful built-in functions to load PDB/Mol files and show interactions.

Note: This class is not intended to be used directly because *set_view()* is not implemented by default. Instead, you should use a class that inherits from *PymolSessionManager* and implements *set_view()*. An example is the class *InteractionViewer* that implements a custom *set_view()* to show interactions. Therefore, you should define your own logic beyond *set_view()* to save a Pymol session that meets your goals.

Parameters

- **show_cartoon** (*bool*) – If True, show the protein structure as cartoons.
- **bg_color** (*str*) – The background color. The default value is "white". Check Pymol to discover which colors are available.
- **add_directional_arrows** (*bool*) – If True, show arrows for directional interactions (e.g., hydrogen bonds and multipolar interactions).
- **show_res_labels** (*bool*) – If True (the default), show residue labels.
- **inter_color** (*ColorPallete*) – A Pymol-compatible color scheme for interactions. The default value is `PYMOL_INTERACTION_COLOR`.
- **pse_export_version** (*str*) – Define a legacy format for saving Pymol sessions (PSE files). The default value os '1.8'.

`finish_session`()
Clear all objects and resets all parameters to default.

`load_pdb`(*pdb_file*, *pdb_obj*, *mol_block=None*, *is_ftmap_output=False*)
Load molecules from PDB files to the current Pymol session.

Optionally, ligands can also be loaded from a separate molecular string block. This is especially useful when working with docked molecules in which the protein structure is in a PDB file and ligands are in a separate molecular file.

Parameters

- **pdb_file** (*str*) – The pathname of the PDB file to be loaded.
- **pdb_obj** (*str*) – Pymol object to store the loaded structure.
- **mol_block** (*str, optional*) – A molecular string block to load together with the PDB file.
- **is_ftmap_output** (*bool*) – If the PDB file is an FTMap output. If so, an additional processing step is performed to standardize the loaded Pymol objects.

`new_session`(*data*, *output_file*)
Start a new session, which includes the following steps:

- Start a new Pymol session (*start_session()*);
- Set the view (*set_view()*);
- Save the Pymol session to `output_file`;
- Finish the Pymol session.

Parameters

- **data** (*iterable*) – Data to be processed by *set_view()*.
- **output_file** (*str*) – The pathname to where the Pymol session will be saved.

`save_session`(*output_file*)
Save the Pymol session as a PSE file of name `output_file`.

`set_interactions_view`(*interactions*, *main_grp*, *secondary_grp=None*)
Display molecular interactions.

Parameters

- **interactions** (iterable of *InteractionType*) – A sequence of interactions to show.
- **main_grp** (*str*) – Main Pymol object to store atom groups.
- **secondary_grp** (*str, optional*) – Secondary Pymol object to store interactions. If not provided, `main_grp` will be used instead.

`set_last_details_to_view`()
This method can be called to apply final modifications to the Pymol session. In its default version, the following modifications are applied:

- Dash radius for interactions is set to 0.08;
- Labels are set to bold and their size is set to 20;
- Atomic spheres' scale is set to 0.3;
- Hydrogen atoms are hidden;
- The view is centered within the visible objects.

set_view(*data*)
Set the session view. However, this method is not implemented by default. Instead, you should use a class that inherits from *PymolSessionManager* and implements *set_view()*. An example is the class *InteractionViewer* that implements a custom *set_view()* to show interactions. Therefore, you should define your own logic beyond *set_view()* to save a Pymol session that meets your goals.

Parameters **data** (*iterable*) – The data that will be used to set the Pymol view.

start_session()
Start a new session and set Pymol settings, including the background color and the PSE export version.

class PymolWrapper
Bases: object

This class provides functions to provide easy access to common functions from Pymol.

add_pseudoatom(*name*, *opts=None*)
Create a molecular object with a pseudoatom or add a pseudoatom to a molecular object if the specified object already exists.

Parameters

- **name** (*str*) – The object name to create or modify.
- **opts** (*dict*) – A set of options to create the pseudoatom. Check Pymol to discover which options are available.

align(*mobile*, *target*, *opts=None*)
Align the structure `mobile` to the `target` structure.

Parameters

- **mobile** (*str*) – The structure to be aligned given by an atomic selection.
- **target** (*str*) – The target structure given by an atomic selection.
- **opts** (*dict*) – Alignment options. Check Pymol to discover which options are available.

alter(*selection*, *expression*)
Modify atomic properties.

Parameters

- **selection** (*str*) – The expression to select atoms.
- **expression** (*str*) – Expression in Python language to define which properties should be modified. This can be used, for instance, to rename an atom or chain.

Examples

Alter the name of a ligand carbon from 'C1' to 'CA'.

```
>>> pw_obj.alter("hetatm and name C1", "name='CA'")
```

Alter the chain A name to 'Z'.

```
>>> pw_obj.alter("chain A", "chain='Z'")
```

arrow(*name*, *atm_sel1*, *atm_sel2*, *opts=None*)
Draw an arrow object between two atoms given by their selection-expressions.

Parameters

- **name** (*str*) – Name of the arrow object to create.
- **sel1** (*str*) – The expression to select the first atom.
- **sel2** (*str*) – The expression to select the second atom.
- **opts** (*dict*) – A set of options to create the arrow. Check Pymol to discover which options are available.

`center`(*selection*)
Translate the window, the clipping slab, and the origin to a point centered within the selection.

`color`(*tuples*)
Color objects and atoms.

Parameters **tuples** (*iterable of tuple*) – Each tuple should contain a color (e.g., 'red') and a selection (e.g., 'hetatm').

`color_by_element`(*selections*, *c_color='green'*)
Color atoms by their default element color (e.g., oxygen in red).

Parameters

- **selections** (*iterable of str*) – A sequence of selections to define which atoms will be colored by element.
- **c_color** (*{'green', 'cyan', 'light magenta', 'yellow', 'salmon', 'white', 'slate', 'bright orange', 'purple', 'pink'}*) – The carbon color. The default value is 'green'.

`create`(*name*, *selection*)
Create a new molecular object from a selection.

Note that the selected atoms won't be extracted from the original object. Instead, a copy of them will be created in the new object.

Parameters

- **name** (*str*) – The object name to be created.
- **selection** (*str*) – The expression to select atoms.

`delete`(*selections*)
Delete the provided selections.

Parameters **selections** (*iterable of str*) – A sequence of selections to be deleted. Wildcards can be used to define object or selection names.

`distance`(*name*, *sel1*, *sel2*)
Create a new distance object between two atoms given by their selection-expressions.

Parameters

- **name** (*str*) – Name of the distance object to create.
- **sel1** (*str*) – The expression to select the first atom.
- **sel2** (*str*) – The expression to select the second atom.

`extract`(*tuples*)
Perform multiple extractions, i.e., extract atoms from an object to another object.

Parameters **tuples** (*iterable of tuple*) – Each tuple should contain the object name to where atoms will be added and the selection itself that defines which atoms will be extracted (e.g., 'hetatm').

`get_cmd`()
Expose the Pymol `cmd` object, so that one can call Pymol functions directly.

`get_coords`(*selection*)
Get atomic coordinates for a given atom selection.

Parameters **selection** (*str*) – The expression to select atoms.

Returns Atomic coordinates (x, y, z) of each atom selected.

Return type array_like of float (size 3)

`get_names`(*obj_type*)
Get names of objects, grouped objects, or selections.

Parameters **obj_type** (*{'objects', 'selections', 'all', 'public_objects', 'public_selections', 'public_nongroup_objects', 'public_group_objects', 'nongroup_objects', 'group_objects'}*) – The target object type.

Return type list of str

`group`(*name*, *members*, *action=None*)
Create or update a group object.

Parameters

- **name** (*str*) – The group name to create or update.
- **members** (*iterable of str*) – The objects to include in the group.
- **action** (*{'add', 'remove', 'open', 'close', 'toggle', 'auto', 'empty', 'purge', 'excise'}, optional*) – An action to take. If not provided, the default value 'auto' will be used instead. The description of the actions are described below (source: Pymol documentation):
  - add: add members to group.
  - remove: remove members from group (members will be ungrouped).
  - empty: remove all members from group.
  - purge: remove all members from group and delete them.
  - excise: remove all members from group and delete group.
  - open: expand group display in object menu panel.
  - close: collapse group display in object menu panel.
  - toggle: toggle group display in object menu panel.
  - auto: add or toggle.

`group_exists`(*name*)
Check if a group of objects exists given by its name `name`.

`hide`(*tuples*)
Hide atom and bond representations for certain selections.

Parameters **tuples** (*iterable of tuple*) – Each tuple should contain a Pymol representation (e.g., 'sticks') and a selection (e.g., 'hetatm').

`hide_all`()
Hide all representations.

`label`(*tuples*)
Draw text labels for PyMOL objects.

Parameters **tuples** (*iterable of tuple*) – Each tuple should contain a selection (e.g., 'hetatm') and some string to label the given selection.

`load`(*input_file*, *obj_name=None*)
Load a molecular file (e.g., PDB files).

Parameters

- **input_file** (*str*) – The pathname of the molecular file to be loaded.
- **obj_name** (*str, optional*) – Pymol object to store the loaded structure. If not provided, the filename will be used instead.

`load_mol_from_pdb_block`(*pdb_block*, *obj_name*)
Load a molecular file from a PDB block string.

Parameters

- **pdb_block** (*str*) – The PDB block string.
- **obj_name** (*str*) – Pymol object to store the loaded structure.

`obj_exists`(*name*)
Check if an object exists given by its name `name`.

`quit`()
Terminate Pymol.

`reinitialize`()
Clear all objects and resets all parameters to default.

`remove`(*selections*)
Remove a selection of atoms from models.

Parameters **selections** (*iterable of str*) – A sequence of selections to define which atoms will be removed.

`run`(*func_name*, *opts*)
Run a Pymol command `func_name` with parameters `opts`.

Parameters

- **func_name** (*str*) – The Pymol command name.
- **opts** (*dict*) – Parameters to pass to the command.

`run_cmds`(*commands*)
Run a set of Pymol commands.

Parameters **commands** (*iterable of tuple*) – Each tuple should contain a Pymol command and its parameters. See *run()* to more details.

`save_png`(*output_file*, *width=1200*, *height=1200*, *dpi=100*, *ray=1*)
Save the current Pymol session as a PNG format image file.

Parameters

- **output_file** (*str*) – The output image pathname.
- **width** (*int*) – The width in pixels. The default value is 1,200.
- **height** (*int or str*) – The height in pixels. The default value is 1,200.
- **dpi** (*float*) – Dots-per-inch. The default value is 100.
- **ray** (*{0, 1}*) – If 1 (the default), run ray first to make high-resolution photos.

save_session(*output_file*)
Save the current PyMOL state to a PSE format file to later use.

Parameters **output_file** (*str*) – The output pathname.

sel_exists(*name*)
Check if a selection exists given by its name `name`.

select(*selection*, *name='sele'*, *enable=0*)
Create a named selection from an atom selection.

Parameters

- **selection** (*str*) – The expression to select atoms.
- **name** (*str*) – The selection name, which by default is 'sele'.
- **enable** (*{0, 1}*) – If 1, activate the selection, i.e., show selection indicators. The default value is 0, which implies the selection indicators won't be shown.

set(*name*, *value*, *opts=None*)
Modify global, object, object-state, or per-atom settings.

Parameters

- **name** (*str*) – The setting name to modify.
- **value** (*str*) – The new setting value.
- **opts** (*dict*) – A set of options. Check Pymol to discover which options are available.

show(*tuples*)
Display atom and bond representations for certain selections.

Parameters **tuples** (*iterable of tuple*) – Each tuple should contain a Pymol representation (e.g., 'sticks') and a selection (e.g., 'hetatm').

mybio_to_pymol_selection(*entity*)
Transform an `Entity` instance into a Pymol selection-expression, which can then be used to select atoms in a Pymol session.

Parameters **entity** (`Entity`) – An entity to be transformed into a Pymol selection-expression.

Returns The Pymol selection-expression.

Return type str

Examples

First, let's parse a PDB file to work with.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.MyBio.PDB.PDBParser import PDBParser
>>> pdb_parser = PDBParser(PERMISSIVE=True, QUIET=True)
>>> structure = pdb_parser.get_structure("Protein", f"{LUNA_PATH}/tutorial/inputs/
↪3QQK.pdb")
```

Now, let's get the Pymol selection-expression for the chain A.

```
>>> from luna.wrappers.pymol import mybio_to_pymol_selection
>>> print(mybio_to_pymol_selection(structure[0]['A']))
chain A
```

Finally, we can get the Pymol selection-expression for the ligand X02.

```
>>> from luna.wrappers.pymol import mybio_to_pymol_selection
>>> print(mybio_to_pymol_selection(structure[0]["A"][('H_X02', 497, ' ')]))
resn X02 AND res 497 AND chain A
```

luna.wrappers.rdkit module

new_mol_from_block(*block*, *mol_format*, *sanitize=True*, *removeHs=True*)

Read a molecule from a string block using RDKit.

Parameters

- **block** (*str*) – The molecular string block.
- **mol_format** (*str*) – The molecular format of `block`.
- **sanitize** (*bool*) – If True (the default), sanitize the molecule.
- **removeHs** (*bool*) – If True (the default), remove explict hydrogens from the molecule.

Returns The parsed molecule or None in case the sanitization process fails.

Return type `rdkit.Chem.rdchem.Mol` or None

Raises **IllegalArgumentError** – If the provided molecular format is not accepted by RDKit.

Examples

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.wrappers.rdkit import new_mol_from_block
>>> with open(f"{LUNA_PATH}/tutorial/inputs/GLY.sdf", "r") as IN:
...           mol_block = IN.read()
>>> rdk_mol = new_mol_from_block(block=mol_block, mol_format="sdf")
>>> print(rdk_mol.GetProp("_Name"))
GLY
```

read_mol_from_file(*mol_file*, *mol_format*, *sanitize=True*, *removeHs=True*)

Read a molecule from `mol_file` using RDKit.

Parameters

- **mol_file** (*str*) – The pathname of the molecular file to be read.
- **mol_format** (*str*) – The molecular format of `mol_file`.
- **sanitize** (*bool*) – If True (the default), sanitize the molecule.
- **removeHs** (*bool*) – If True (the default), remove explict hydrogens from the molecule.

Returns The parsed molecule or None in case the sanitization process fails.

Return type `rdkit.Chem.rdchem.Mol` or None

Raises **IllegalArgumentError** – If the provided molecular format is not accepted by RDKit.

Examples

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.wrappers.rdkit import read_mol_from_file
>>> rdk_mol = read_mol_from_file(mol_file=f"{LUNA_PATH}/tutorial/inputs/
↪ZINC000007786517.mol",
...                              mol_format="mol")
>>> print(rdk_mol.GetProp("_Name"))
 ZINC000007786517
```

read_multimol_file(*mol_file*, *targets=None*, *mol_format=None*, *sanitize=True*, *removeHs=True*)

Read molecules from a multimolecular file using RDKit.

Parameters

- **mol_file** (*str*) – The pathname of the molecular file to be read.
- **targets** (*iterable of str*) – Only parses molecules, given by their ids, defined in `targets`.
- **mol_format** (*str, optional*) – The molecular format of `mol_file`. If not provided, the format will be defined by the file extension.
- **sanitize** (*bool*) – If True (the default), sanitize the molecule.
- **removeHs** (*bool*) – If True (the default), remove explict hydrogens from the molecule.

Yields tuple of (`rdkit.Chem.rdchem.Mol`, int) – A tuple containing the parsed molecule and its id.

Raises `IllegalArgumentError` – If the provided or identified molecular format is not accepted by RDKit.

Examples

In this first example, we will read all molecules from a multimolecular file.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.wrappers.rdkit import read_multimol_file
>>> for mol_tuple in read_multimol_file(mol_file=f"{LUNA_PATH}/tutorial/inputs/
↪ligands.mol2"):
...     print(mol_tuple[0].GetProp("_Name"))
 ZINC000343043015
 ZINC000065293174
 ZINC000575033470
 ZINC000096459890
 ZINC000012442563
```

Now, we will only read two molecules (ZINC000065293174, ZINC000096459890) from the same multimolecular file.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.wrappers.rdkit import read_multimol_file
>>> for mol_tuple in read_multimol_file(mol_file=f"{LUNA_PATH}/tutorial/inputs/
↪ligands.mol2",
...                                     targets=["ZINC000065293174",
↪"ZINC000096459890"]):
...     print(mol_tuple[0].GetProp("_Name"))
```

(continues on next page)

(continued from previous page)

```
ZINC000065293174
ZINC000096459890
```

Module contents

Submodules

luna.projects module

class EntryResults(*entry*, *atm_grps_mngr*, *interactions_mngr*, *ifp=None*, *mfp=None*)

Bases: object

Store entry results.

Parameters

- **entry** (*Entry*) – An *Entry* object that represents a molecule or an entire chain.
- **atm_grps_mngr** (*AtomGroupsManager*) – An *AtomGroupsManager* object that stores the perceived atoms and atom groups in the vicinity given by `entry`.
- **interactions_mngr** (*InteractionsManager*) – An *InteractionsManager* object that interactions in the vicinity given by `entry`.
- **ifp** (*Fingerprint*, optional) – An interaction fingerprint (IFP) generated for `entry`.
- **mfp** (RDKit `ExplicitBitVect` or `SparseBitVect`, optional) – A molecular fingerprint generated for `entry`.

Variables

- **~EntryResults.entry** (*Entry*) –
- **~EntryResults.atm_grps_mngr** (*AtomGroupsManager*) –
- **~EntryResults.interactions_mngr** (*InteractionsManager*) –
- **~EntryResults.ifp** (*Fingerprint*) –
- **~EntryResults.mfp** (RDKit `ExplicitBitVect` or `SparseBitVect`) –
- **~EntryResults.version** (*str*) – The LUNA's version with which results were generated.

static load(*input_file*)

Read the pickled representation of an *EntryResults* object from the file `input_file` and return the reconstituted object hierarchy specified therein. `input_file` can be a gzip-compressed file.

Raises **PKLNotReadError** – If the file could not be loaded.

save(*output_file*, *compressed=True*)

Write the pickled representation of this object to the file `output_file`.

Parameters

- **output_file** (*str*) – The output file where the pickled representation will be saved.
- **compressed** (*bool, optional*) – If True (the default), compress the pickled representation as a gzip file (.gz).

Raises **FileNotCreated** – If the file could not be created.

class LocalProject(*entries*, *working_path*, ***kwargs*)

Bases: *luna.projects.Project*

Define a local LUNA project, i.e., results are saved locally and not to a database.

This class inherits from *Project* and implements *run()*.

Examples

In this minimum example, we will calculate protein-ligand interactions for dopamine D4 complexes.

First, we should define the ligand entries and initialize a new *InteractionCalculator* object.

```
>>> from luna.util.default_values import LUNA_PATH
>>> from luna.interaction.calc import InteractionCalculator
>>> entries = list(MolFileEntry.from_file(input_file=f"{LUNA_PATH}/tutorial/inputs/
↪MolEntries.txt",
...                                       pdb_id="D4", mol_file=f"{LUNA_PATH}/
↪tutorial/inputs/ligands.mol2"))
>>> ic = InteractionCalculator(inter_filter=InteractionFilter.new_pli_filter())
```

Finally, just create the new LUNA project with desired parameters and call *run()*. Here, we opted to define the parameters first as a dict, and then we pass it as an argument to *LocalProject*.

```
>>> from luna import LocalProject
>>> opts = {}
>>> opts["working_path"] = "%s/Results/Test3" % main_path
>>> opts["pdb_path"] = f"{LUNA_PATH}/tutorial/inputs/"
>>> opts["entries"] = entries
>>> opts["inter_calc"] = ic
>>> proj_obj = LocalProject(**opts)
>>> proj_obj.run()
```

generate_ifps()

Generate LUNA interaction fingerprints (IFPs).

This function can be used to generate new IFPs after a project is run. Thus, you can reload your project, vary IFP parameters (`ifp_num_levels`, `ifp_radius_step`, `ifp_length`, `ifp_count`, `ifp_diff_comp_classes`, `ifp_type`, `ifp_output`), and call *generate_ifps* to create new IFPs without having to run the project from the scratch.

Examples

In the below example, we will assume a LUNA project object named `proj_obj` already exists.

```
>>> from luna.interaction.fp.type import IFPType
>>> proj_obj.ifp_num_levels = 5
>>> proj_obj.ifp_radius_step = 1
>>> proj_obj.ifp_length = 4096
>>> proj_obj.ifp_type = IFPType.EIFP
>>> proj_obj.ifp_output = "EIFP-4096__length-5__radius-1.csv"
>>> proj_obj.generate_ifps()
```

class Project(*entries*, *working_path*, *pdb_path='/home/docs/checkouts/readthedocs.org/user_builds/luna-toolkit/checkouts/latest/output/public/pdb'*, *overwrite_path=False*, *add_h=True*, *ph=7.4*, *amend_mol=True*, *mol_obj_type='rdkit'*, *atom_prop_file='/home/docs/checkouts/readthedocs.org/user_builds/luna-toolkit/checkouts/latest/luna/data/LUNA.fdef'*, *inter_calc=None*, *binding_mode_filter=None*, *calc_mfp=False*, *mfp_output=None*, *calc_ifp=True*, *ifp_num_levels=2*, *ifp_radius_step=5.73171*, *ifp_length=4096*, *ifp_count=True*, *ifp_diff_comp_classes=True*, *ifp_type=IFPType.EIFP*, *ifp_output=None*, *ifp_sim_matrix_output=None*, *out_pse=False*, *append_mode=False*, *verbosity=3*, *logging_enabled=True*, *nproc=1*)

Bases: object

Define a LUNA project.

Note: This class is not intended to be used directly because *run()* is not implemented by default. Instead, you should use a class that inherits from *Project* and implements *run()*. An example is the class *LocalProject* that implements a custom *run()* that saves results as local files.

Parameters

- **entries** (iterable of *Entry*) – Entries determine the target molecule to which interactions and other properties will be calculated. They can be ligands, chains, etc, and can be defined in a number of ways. Each entry has an associated PDB file that may contain macromolecules (protein, RNA, DNA) and other small molecules, water, and ions. Refer to *Entry* for more information.
- **working_path** (*str*) – Where project results will be saved.
- **pdb_path** (*str*) – Path containing local PDB files or to where the PDB files will be downloaded. PDB filenames must match that defined for the entries. If not provided, the default PDB path will be used.
- **overwrite_path** (*bool*) – If True, allow LUNA to overwrite any existing directory, which may remove files from a previous project. The default value is False.
- **add_h** (*bool*) – Define if you need to add hydrogens or not. The default value is True.

  Note: To be cautious, it does not add hydrogens to NMR-solved structures and ligands initialized from molecular files (*MolFileEntry* objects) as they usually already contain hydrogens.

- **ph** (*float*) – Control the pH and how the hydrogens are going to be added. The default value is 7.4.

  Note: To be cautious, it does not modify the protonation of molecular files defined by a *MolFileEntry* object.

- **amend_mol** (*bool*) – If True (the default), try to fix atomic charges, valence, and bond types for small molecules and residues at PDB files. Only molecules at PDB files are validated because they do not contain charge, valence, and bond types, which may cause molecules to be incorrectly perceived. More information *here*.

  Note: Molecules from external files (*MolFileEntry* objects) will not be modified.

- **mol_obj_type** (*{'rdkit', 'openbabel'}*) – Define which library (RDKit or Open Babel) to use to parse molecules. The default value is 'rdkit'.
- **atom_prop_file** (*str*) – A feature definition file (FDef) containing all information needed to define a set of chemical or pharmacophoric features. The default value is 'LUNA.fdef', which contains default LUNA features definition.
- **inter_calc** (*InteractionCalculator*) – Define which and how interactions are calculated.
- **binding_mode_filter** (*BindingModeFilter*) – Define how to filter interactions based on binding modes.
- **calc_mfp** (*bool*) – If True, generate ECFP4 fingerprints for each entry in `entries`. The default value is False.
- **mfp_output** (*str*) – If `calc_mfp` is True, save ECFP4 fingerprints to file `mfp_output`. If not provided, fingerprints are saved at `<working_path>`/results/fingerprints/mfp.csv.
- **calc_ifp** (*bool*) – If True (the default), generate LUNA interaction fingerprints (IFPs) for each entry in `entries`.
- **ifp_num_levels** (*int*) – The maximum number of iterations for fingerprint generation. The default value is 2.
- **ifp_radius_step** (*float*) – The multiplier used to increase shell size at each iteration. At iteration 0, shell radius is 0 * `radius_step`, at iteration 1, radius is 1 * `radius_step`, etc. The default value is 5.73171.
- **ifp_length** (*int*) – The fingerprint length (total number of bits). The default value is 4096.
- **ifp_count** (*bool*) – If True (the default), create a count fingerprint (*CountFingerprint*). Otherwise, return a bit fingerprint (*Fingerprint*).
- **ifp_diff_comp_classes** – If True (the default), include differentiation between compound classes. That means structural information originated from *AtomGroup* objects belonging to residues, nucleotides, ligands, or water molecules will be considered different even if their structural information are the same. This is useful for example to differentiate protein-ligand interactions from residue-residue ones.
- **ifp_type** (*IFPType*) – The fingerprint type (EIFP, FIFP, or HIFP). The default value is EIFP.
- **ifp_output** (*str*) – If `calc_ifp` is True, save LUNA interaction fingerprints (IFPs) to file `ifp_output`. If not provided, fingerprints are saved at `<working_path>`/results/fingerprints/ifp.csv.
- **ifp_sim_matrix_output** (*str, optional*) – If provided, compute Tanimoto similarity between interaction fingerprints (IFPs) and save the similarity matrix to `ifp_sim_matrix_output`.
- **out_pse** (*bool*) – If True, depict interactions save them as Pymol sessions (PSE file). The default value is False. PSE files are saved at `<working_path>`/results/pse.
- **append_mode** (*bool*) – If True, skip entries from processing if a result for them already exists in `working_path`. This can save processing time in case additional entries are to be added to an existing project.
- **verbosity** (*int*) – Verbosity level. The higher the verbosity level the more information is displayed. Valid values are:
  - 4: DEBUG messages;
  - 3: INFO messages (the default);
  - 2: WARNING messages;

  - 1: ERROR messages;
  - 0: CRITICAL messages.
- **logging_enabled** (*bool*) – If True (the default), enable the logging system.
- **nproc** (*int*) – The number of CPUs to use. The default value is the `maximum number of CPUs - 1`. If `nproc` is smaller than 1 or greater than the maximum amount of available CPUs at your PC, then `nproc` is set to its default value. If you set it to None, LUNA will be run serially.

Variables

- **~Project.entries** (iterable of *Entry*) –
- **~Project.working_path** (*str*) –
- **~Project.pdb_path** (*str*) –
- **~Project.overwrite_path** (*bool*) –
- **~Project.add_h** (*bool*) –
- **~Project.ph** (*float*) –
- **~Project.amend_mol** (*bool*) –
- **~Project.mol_obj_type** (*{'rdkit', 'openbabel'}*) –
- **~Project.atom_prop_file** (*str*) –
- **~Project.inter_calc** (*InteractionCalculator*) –
- **~Project.binding_mode_filter** (*BindingModeFilter*) –
- **~Project.calc_mfp** (*bool*) –
- **~Project.mfp_output** (*str*) –
- **~Project.calc_ifp** (*bool*) –
- **~Project.ifp_num_levels** (*int*) –
- **~Project.ifp_radius_step** (*float*) –
- **~Project.ifp_length** (*int*) –
- **~Project.ifp_count** (*bool*) –
- **~Project.ifp_diff_comp_classes** (*bool*) –
- **~Project.ifp_type** (*IFPType*) –
- **~Project.ifp_output** (*str*) –
- **~Project.out_pse** (*bool*) –
- **~Project.out_ifp_sim_matrix** (*bool*) –
- **~Project.append_mode** (*bool*) –
- **~Project.logging_file** (*str*) – The file to where logging messages are saved.
- **~Project.version** (*str*) – The LUNA's version with which results were generated.
- **~Project.errors** (*list of tuple*) – Any errors found during the processing of an entry. Each tuple contains the input and the exception raised during the execution of a task with that input.

property atm_grps_mngrs
An *AtomGroupsManager* object for each entry.

Type iterable of *AtomGroupsManager*

get_entry_results(*entry*)
Get results for a given entry.

Parameters **entry** (*Entry*) – An entry from `entries`.

Return type *EntryResults*

property ifps
An interaction fingerprint (IFP) for each entry.

Type iterable of *Fingerprint*

property interactions_mngrs
An *InteractionsManager* object for each entry.

Type iterable of *InteractionsManager*

static load(*pathname*, *verbosity=3*, *logging_enabled=True*)
Read the pickled representation of a *Project* object from a file or project path and return the reconstituted object hierarchy specified therein. The `pathname` can be a gzip-compressed file.

Parameters

- **pathname** (*str*) – A file containing the pickled representation of a *Project* object or the project path (`working_path`) from where the pickled representation will be recovered.
- **verbosity** (*int*) – Verbosity level. The higher the verbosity level the more information is displayed. Valid values are:
  - 4: DEBUG messages;
  - 3: INFO messages (the default);
  - 2: WARNING messages;
  - 1: ERROR messages;
  - 0: CRITICAL messages.
- **logging_enabled** (*bool*) – If True (the default), enable the logging system.

Raises

- **CompatibilityError** – If the project version is not compatible with the current LUNA version.
- **PKLNotReadError** – If the file could not be loaded.
- **IllegalArgumentError** – If the provided pathname does not exist or is an invalid file/directory.

property logging_enabled
If the logging system is enable or not.

Type bool

property mfps
A molecular fingerprint for each entry.

Type iterable of RDKit `ExplicitBitVect` or `SparseBitVect`

property nproc
The number of CPUs to use.

Type int

property project_file
Where the pickled representation of the LUNA project is saved.

Type str

remove_duplicate_entries()
Search and remove duplicate entries from `entries`.

property results
LUNA results for each entry.

Type iterable of *EntryResults*

run()
Run LUNA. However, this method is not implemented by default. Instead, you should use a class that inherits from *Project* and implements *run()*. An example is the class *LocalProject* that implements a custom *run()* that saves results as local files.

save(*output_file*, *compressed=True*)
Write the pickled representation of this project to the file `output_file`.

Parameters

- **output_file** (*str*) – The output file where the pickled representation will be saved.
- **compressed** (*bool, optional*) – If True (the default), compress the pickled representation as a gzip file (.gz).

Raises **FileNotCreated** – If the file could not be created.

property verbosity
Verbosity level.

Type int

verify_pdb_files_existence()
Verify if a local PDB file exists for each entry in `entries`. If it does not find a given PDB file, then LUNA will try to download it from RCSB.

Module contents

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z