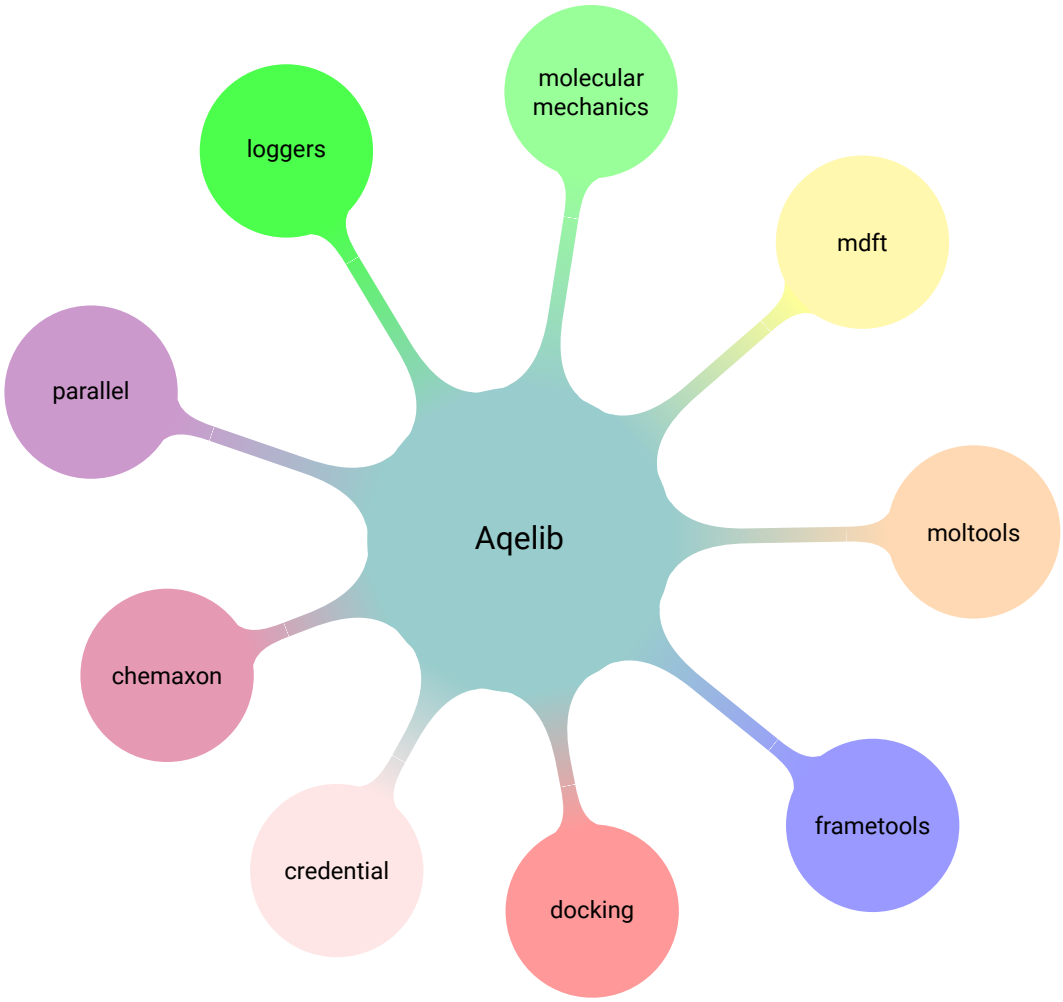
Molecular screening library documentation

BARRE kevin

Contents



Module `aqelib`

Sub-modules

- aqelib.chemaxon
- aqelib.credentials
- aqelib.docking
- aqelib.frametools
- aqelib.functional
- aqelib.loggers
- aqelib.mdft
- aqelib.molecular_mechanics
- aqelib.moltools
- aqelib.parallel

Module `aqelib.chemaxon`

Sub-modules

- aqelib.chemaxon.chemaxon

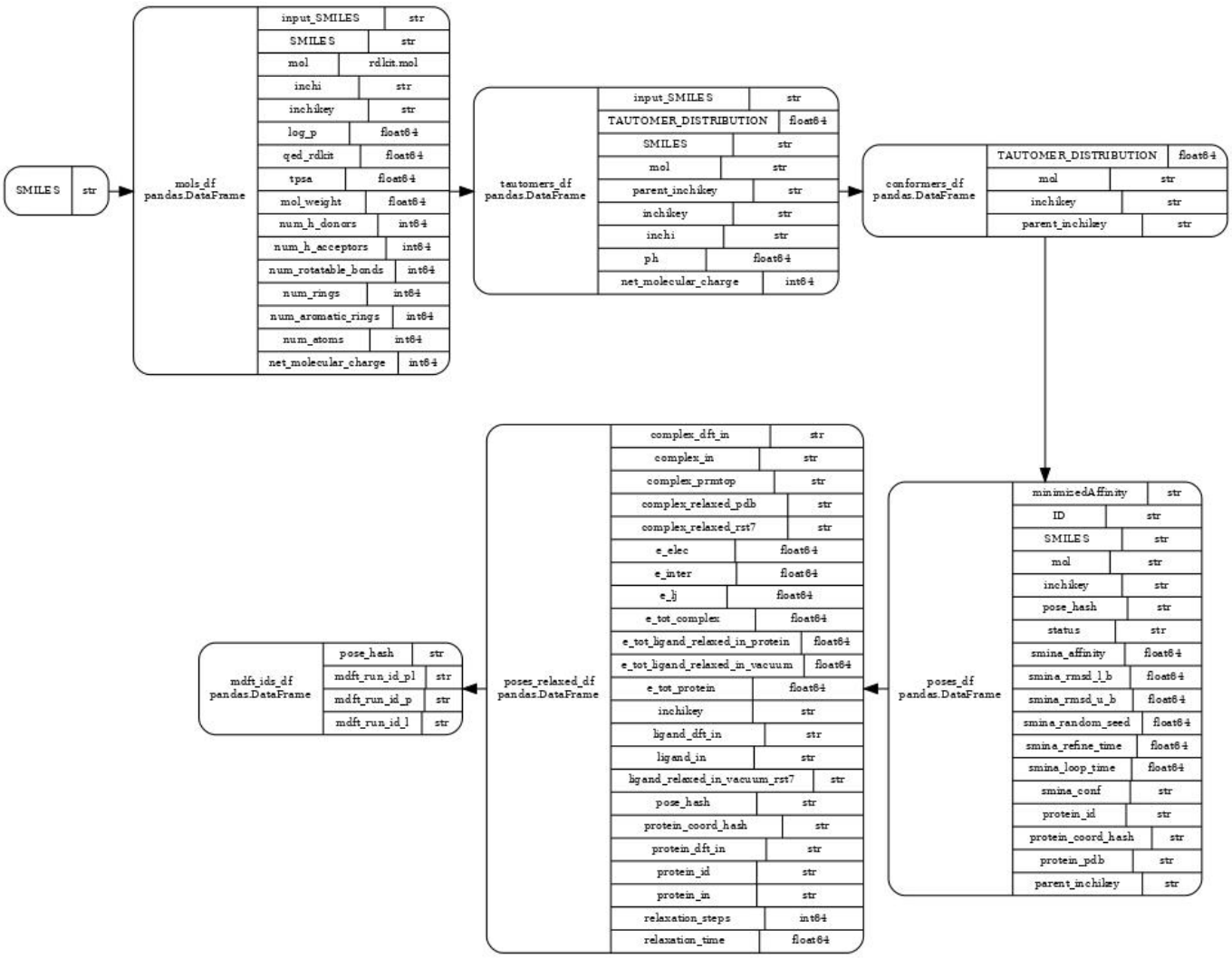


Figure 1: Data Screening Architecture

Module `aqelib.chemaxon.chemaxon`

Functions

Function `cxcalc_dominant_tautomer_distribution`

```
def cxcalc_dominant_tautomer_distribution(
    mol,
    ph: float = 7.4
) -> pandas.core.frame.DataFrame
```

cxcalc_dominant_tautomer_distribution get predominance weight

Use Chemaxon cxcalc binary:

```
cxcalc dominanttautomerdistribution -p 2 --pH $ph, ${Chem.MolToSmiles(mol)}, -f sdf
```

mol must be ***str*** a json Chem.Mol or ***rdkit.Chem.Mol***

Parameters

`mol` : **str or rdkit.Chem.Mol**
- str: ***json Chem.Mol***
- obj: ***rdkit.Chem.Mol***

rdkit mol or rdkit mol json

`ph` : **float, optional** ph is 7.4 by default

Returns

pandas.DataFrame pandas.DataFrame if successful, pandas.DataFrame.empty otherwise.

Set LOGGING_LEVEL=DEBUG to see cxcalc full command for each mol if process get stderr, it will be logged as error

Raises

Chem.MolToSmiles(mol) is raises safe all exceptions will be logged as exception

Module `aqelib.credentials`

Sub-modules

- aqelib.credentials.aws

Module `aqelib.credentials.aws`

Functions

Function `get_aws_credentials`

```
def get_aws_credentials(
    items_name: str = 'default'
)
```

Module `aqelib.docking`

Sub-modules

- aqelib.docking.smina

Module `aqelib.docking.smina`

Functions

Function `generate_smina_conf`

```
def generate_smina_conf(
    smina_config: dict,
    config_dir: str = ''
) -> List[tuple]
```

Function `smina_dock_mol_to_protein`

```
def smina_dock_mol_to_protein(
    conf_mol: tuple,
    sminaconf: List[tuple],
    protein_id: str,
    aws_credentials: dict = {}
) -> pandas.core.frame.DataFrame
```

smina_dock_mol_to_protein

Use smina binary:

```
smina --receptor protein_id.pdb --ligand ${in_mol}.sdf --out ${out_mol}.sdf ${smina_config}
```

mol must be ***str*** a json Chem.Mol or ***rdkit.Chem.Mol***

Parameters

`conf_mol` : **Tuple[str or** `rdkit.Chem.Mol, str] | Tuple[mol, parent_inchikey]`

- str: ***json Chem.Mol***
- obj: ***rdkit.Chem.Mol***

rdkit mol or rdkit mol json - str: ***parent_inchikey***

```
conf_mol = (mol, "CNCCN(C)n1cc(c2ccccc2)c3ccccc13")
```

`sminaconf` : **List[tuple]** list of key:value:

```
smina = [
("size_x", 20),
("size_y", 20,)
("size_z", 20,)
("center_x", 6.554,)
("center_y", 43.184,)
("center_z", 51.151)
]
```

Will be converted

```
--size_x 20 --size_y 20 --size_z 20 --center_x 6.554 --center_y 43.184 --center_z 51.151
```

`protein_id` : **str** protein name

`aws_credentials` : **dict** use aqelib.credential.aws.get_aws_credentials aws_credentials = {} by default needed to get protein_id.pdb

Returns

pandas.DataFrame pandas.DataFrame with *DOCKING SUCCESS* if successful, pandas.DataFrame with *SMINA FAILED* if smina_command does not work pandas.DataFrame with *DOCKING add_explicit_hs ERROR* if obabel did not work

Set LOGGING_LEVEL=INFO to see smina_command full command for each mol if process get stderr, it will be logged as error

Raises

Smina_dock_mol_to_protein is raises safe. All exceptions will be logged as exception :

Info

Each protein are dowloaded in /tmp only one time. It will reuse same pdb if proteine_id is already in /tmp

You can put your protein.pdb in /tmp, it will use this one instead of downloading it

You can custom docking paramatere only with sminaconf input

Module `aqelib.frametools`

Sub-modules

- aqelib.frametools.frametools

Module `aqelib.frametools.frametools`

Functions

Function `bag_to_dataframe`

```
def bag_to_dataframe(
    bag,
    meta: dict = {},
    verify_meta=False,
    **concat_kwargs
)
```

Function `concat_dataframe`

```
def concat_dataframe(
    updf: pandas.core.frame.DataFrame,
    downdf: pandas.core.frame.DataFrame
) -> pandas.core.frame.DataFrame
```

concat two dataframe to one in up side down return pandas.DataFrame

Parameters

`updf` : **pandas.DataFrame** Up dataframe
`downdf` : **pandas.DataFrame** Down dataframe

Returns

pandas.DataFrame Returning one pandas.DataFrame
updf and downdf must be empty pandas.DataFrame at least but not null

Function `dataframe_to_record_dict`

```
def dataframe_to_record_dict(
    df: pandas.core.frame.DataFrame
)
```

Module `aqelib.functional`

Sub-modules

- aqelib.functional.funcs

Module `aqelib.functional.funcs`

Functions

Function `compose`

```
def compose(
    *functions
)
```

Module `aqelib.loggers`

Sub-modules

- aqelib.loggers.chemlog

Module `aqelib.loggers.chemlog`

Functions

Function `disable_rdkit_logging`

```
def disable_rdkit_logging()
```

Disables RDKit whiny logging.

Module `aqelib.mdft`

Sub-modules

- aqelib.mdft.mdft
- aqelib.mdft.mdft_inputs

Module `aqelib.mdft.mdft`

Functions

Function `get_mdft_dg_sol`

```
def get_mdft_dg_sol(
    mdft_run_id: str,
    aws_credentials: dict,
    workdir: str = '/tmp'
)
```

Function `run_mdft`

```
def run_mdft(
    solute_in: str,
    solute_dft_in: str,
    pose_hash: str,
    system_name: str,
    aws_credentials: dict,
    workdir: str = '/tmp'
) -> float
```

Function `run_mdft_PL_P_L`

```
def run_mdft_PL_P_L(
    relax_df: pandas.core.frame.DataFrame,
    aws_credentials: dict
) -> pandas.core.frame.DataFrame
```

Function `zipdir`

```
def zipdir(
    path,
    ziph
)
```

Module `aqelib.mdft.mdft_inputs`

Functions

Function `set_box_size`

```
def set_box_size(
    system_in: str,
    buffer_layer=10
) -> (<class 'int'>, <class 'int'>, <class 'int'>)
```

Function `write_affinity_input`

```
def write_affinity_input(
    sytem_pmd
) -> str
```

Function `write_affinity_parameters_file`

```
def write_affinity_parameters_file(
    system_in,
    mdft_dxdydz,
    mmax=1,
    solvent_type='tip3p',
    buffer_layer=10
) -> str
```

Module `aqelib.molecular_mechanics`

Sub-modules

- aqelib.molecular_mechanics.relax
- aqelib.molecular_mechanics.run_gromacs
- aqelib.molecular_mechanics.utils

Module `aqelib.molecular_mechanics.relax`

Functions

Function `relax_pose`

```
def relax_pose(
    mol: str,
    protein_id: str,
    protein_coord_hash: str
) -> dict
```

Function `relax_posesdf`

```
def relax_posesdf(
    poses_df: pandas.core.frame.DataFrame,
    meta: dict = {}
) -> pandas.core.frame.DataFrame
```

Module aqelib.molecular_mechanics.run_gromacs

Functions

Function adapt_box_vector_to_coordinates

```
def adapt_box_vector_to_coordinates(
    system_pmd,
    buffer: float
)
```

Function get_max_distances_from_center

```
def get_max_distances_from_center(
    coordinates: str
) -> Tuple[int, int, int]
```

Function get_nsteps_from_gromacs_log

```
def get_nsteps_from_gromacs_log(
    logfile
) -> int
```

Function relax_with_gromacs

```
def relax_with_gromacs(
    run_dir: str,
    system_pmd,
    total_minimization_cycles: int
)
```

Module aqelib.molecular_mechanics.utils

Functions

Function calculate_epsilon_ij

```
def calculate_epsilon_ij(
    atoms_i,
    atoms_j,
    method='LB'
)
```

Returns epsilon_ij value by applying Lorentz-Berthelot (default) combining rule

Parameters

`atoms_a` : **dict** dictionary containing properties of atom I
`atoms_b` : **dict** dictionary containing properties of atom J
`method` : **str, optional** Method name for calculating epsilon_ij. Only “LB” is supported currently. The default is ‘LB’.

Returns

`epsilon_ij` : **float** value of epsilon_ij

Function calculate_sigma_ij

```
def calculate_sigma_ij(
    atoms_i,
    atoms_j,
    method='LB'
)
```

Returns sigma_ij value by applying Lorentz-Berthelot (default) or geometric combining rule

Parameters

`atoms_i` : **dict** dictionary containing properties of atom I
`atoms_j` : **dict** dictionary containing properties of atom J
`method` : **str, optional** Method name for calculating sigma_ij. Only "LB" and "geometric" are supported currently. The default is 'LB'.

Returns

`sigma_ij` : **float** value of sigma_ij

Function `compute_distance_matrix`

```
def compute_distance_matrix(
    molecule_a,
    molecule_b
)
```

Function `compute_eint_from_top_crd`

```
def compute_eint_from_top_crd(
    parmed_pose_l,
    parmed_pose_p
) -> (<class 'float'>, <class 'float'>, <class 'float'>)
```

Computes interaction energy between two molecules

Parameters

`parmed_pose_l` : **object** parmed object
`parmed_pose_p` : **object** parmed object

Returns

`compute_interaction_energy` : **tuple**
`The interaction energy between two molecules, in kcal/mol`

Function `compute_electrostatic_energy`

```
def compute_electrostatic_energy(
    molecule_a: list,
    molecule_b: list,
    relative_permittivity: float = 1
) -> float
```

Computes electrostatic energy between two molecules.

Parameters

`molecule_a` : **list** list of atoms dictionaries containing properties of atom i.
`molecule_b` : **list** list of atoms dictionaries containing properties of atom i.
`relative_permittivity` : **list** The default is 1.

Returns

`electrostatic_energy_kcal:`
`the electrostatic energy between two molecules, in kcal/mol`

Function `compute_interaction_energy`

```
def compute_interaction_energy(
    molecule_a: list,
    molecule_b: list
) -> (<class 'float'>, <class 'float'>, <class 'float'>)
```

Computes interaction energy between two molecules.

Parameters

`molecule_a` : **list** list of atoms dictionary containing properties of atom i.
`molecule_b` : **list** list of atoms dictionary containing properties of atom i.

Returns

```
interaction_energy, lennard_jones_energy, electrostatic_energy:
the interaction energies between two molecules in kcal/mol
```

Function `compute_lennard_jones_energy`

```
def compute_lennard_jones_energy(
    molecule_a: list,
    molecule_b: list
) -> float
```

Computes Lennard-Jones energy between two molecules.

Parameters

`molecule_a` : **list** list of atoms dictionary containing properties of atom i.
`molecule_b` : **list** list of atoms dictionary containing properties of atom i.

Returns

```
lennard_jones_energy_kcal:
the Lennard-Jones energy between two molecules, in kcal/mol
```

Function `create_dict_per_atom`

```
def create_dict_per_atom(
    parmed_pose: str
) -> list
```

Create a dictionary for each atom in a molecule from a topology file and a coordinate file

Parameters

`topology_file` : **str** Topology file of the molecule
`coordinate_file` : **str** Coordinate file of the molecule
`parmed_pose` : **parmed object**

Returns

`atoms` : **list**

list of `atom dictionaries containing :`

- charge
- atomic_number
- atom_reference
- sigma
- epsilon
- x_coord
- y_coord
- z_coord

Function `get_etot`

```
def get_etot(
    system_parmed
) -> float
```

Function `get_mol_charges`

```
def get_mol_charges(
    molecule: list
) -> <built-in function array>
```

Parameters

`molecule` : **list** list of atoms dictionary containing properties of atom i

Returns

`charges` : **np.array**

Function `get_mol_coords`

```
def get_mol_coords(
    molecule
)
```

Return list of atom coordinates:

- one list for x_coords,
- one y_coords,
- one for z_coords

Parameters

`molecule` : **list** list of atoms dictionary containing properties of atom i

Returns

`x_coords` : **list**
`y_coords` : **list**
`z_coords` : **list**

Function `get_mol_epsilon`

```
def get_mol_epsilon(
    molecule: list
) -> <built-in function array>
```

Parameters

`molecule` : **list of atoms dictionary containing properties of atom i**

Returns

epsilons

Function `get_mol_sigma`

```
def get_mol_sigma(
    molecule
)
```

Return list of atom sigmas

Parameters

`molecule` : **list** list of atoms dictionary containing properties of atom i

Returns

`sigmas : list`

Function `parametrize_ligand`

```
def parametrize_ligand(
    dockedmol: object,
    workdir: str
) -> object
```

Function `parametrize_protein`

```
def parametrize_protein(
    protein_pdb_path: str,
    workdir: str
) -> object
```

Module `aqelib.moltools`

Sub-modules

- aqelib.moltools.tools

Module `aqelib.moltools.tools`

Functions

Function `check_mol`

```
def check_mol(
    mol: rdkit.Chem.rdchem.Mol
)
```

Function `enumerate_stereoisomers`

```
def enumerate_stereoisomers(
    inputs: Tuple[str, rdkit.Chem.rdchem.Mol]
) -> List[Tuple[str, rdkit.Chem.rdchem.Mol]]
```

Enumerate stereoisomers from input mol return List of Chem.Mol List[Chem.Mol] list

- Parameters: mol (Chem.Mol): input molecule to find stereoisomers
- Returns: List[Chem.Mol]: Returning all stereoisomers in Chem.Mol

mol cannot be None Traceback (most recent call last): … TypeError: enumerate_stereoisomers() missing 1 required positional argument: 'mol'

Function `generate_conformers`

```
def generate_conformers(
    tauto_mol: tuple,
    numgenerate: int = 300,
    numconfs: int = 1
) -> pandas.core.frame.DataFrame
```

Function `hash_mol`

```
def hash_mol(
    mol: str,
    objtype: type = builtins.int
)
```

Function `hash_str`

```
def hash_str(
    inputfile: str,
    objtype: type = builtins.str
) -> str
```

Function `hash_to_int`

```
def hash_to_int(
    hash_val: str
)
```

Function `mol_to_dataframe`

```
def mol_to_dataframe(
    mol: rdkit.Chem.rdchem.Mol
) -> pandas.core.frame.DataFrame
```

Function `mols_to_dataframe`

```
def mols_to_dataframe(
    inputs: List[Tuple[str, rdkit.Chem.rdchem.Mol]]
) -> pandas.core.frame.DataFrame
```

Function `normalize_mol_to_dict`

```
def normalize_mol_to_dict(
    inputs: Tuple[str, rdkit.Chem.rdchem.Mol]
) -> dict
```

Function `round_jsonmol`

```
def round_jsonmol(
    jsonmol: str
) -> str
```

Function `smiles_to_mol`

```
def smiles_to_mol(
    smiles: str
) -> Tuple[str, rdkit.Chem.rdchem.Mol]
```

Return the Chem.Mol of smile string. Chem.Mol object

Parameters

`smile` : **str** input string

Returns:

Tuple[str, Chem.Mol]: Returning smiles in str and his associated mol in rdkit.Chem.Mol

Raises

`Smiles cannot be None`

Function `smiles_to_standard_mol`

```
def smiles_to_standard_mol(
    smile: str
) -> rdkit.Chem.rdchem.Mol
```

Function `standardize_mol`

```
def standardize_mol(
    inputs: Tuple[str, rdkit.Chem.rdchem.Mol]
) -> Tuple[str, rdkit.Chem.rdchem.Mol]
```

Function `standardize_mols`

```
def standardize_mols(
    mols: List[Tuple[str, rdkit.Chem.rdchem.Mol]]
) -> List[Tuple[str, rdkit.Chem.rdchem.Mol]]
```

Function `tautomer_distribution`

```
def tautomer_distribution(
    mols: List[rdkit.Chem.rdchem.Mol]
) -> pandas.core.frame.DataFrame
```

Compute tautomer distribution of mols list return pandas.DataFrame

- Parameters: mols (List[Chem.Mol]): Up dataframe
- Returns: pandas.DataFrame: Returning one pandas.DataFrame

Module `aqelib.parallel`

Sub-modules

- aqelib.parallel.thread

Module `aqelib.parallel.thread`

Functions

Function `map_parallel`

```
def map_parallel(
    f,
    iter,
    max_parallel=8
)
```

Just like map(f, iter) but each is done in a separate thread.

Generated by *pdoc* 0.9.2 (https://pdoc3.github.io).