CAT Documentation

Release 0.4.6

B. F. van Beek

Jul 12, 2019

Contents

Contents:

CHAPTER 1

Compound Attachment/Analysis Tool 0.4.6

CAT is a collection of tools designed for the construction, and subsequent analysis, of various chemical compounds. Further information is provided in the documentation.

1.1 Installation

- Download miniconda for python3: miniconda (also you can install the complete anaconda version).
- Install according to: installConda.
- Create a new virtual environment, for python 3.7, using the following commands:
  - `conda create --name CAT python`
- The virtual environment can be enabled and disabled by, respectively, typing:
  - Enable: `conda activate CAT`
  - Disable: `conda deactivate`

1.1.1 Dependencies installation

Using the conda environment the following packages should be installed:

- rdkit & HDF5: `conda install -y --name CAT --channel conda-forge rdkit h5py`

1.1.2 Package installation

Finally, install **CAT** using pip:

- **CAT**: `pip install git+https://github.com/nlesc-nano/CAT@master --upgrade`

Now you are ready to use **CAT**.

1.2 Input files

Running **CAT** and can be done with the following command: `init_cat my_settings.yaml`. The user merely has to provide a yaml file with the job settings, settings which can be tweaked and altered to suit ones purposes (see example1). Alternatively, **CAT** can be run like a regular python script, bypassing the command-line interface (*i.e.* `python input.py`, see example2).

An extensive description of the various available settings is available in the documentation.

CHAPTER 2

CAT Documentation

For a more detailed description of the **CAT** compound builder read the documentation. The documentation is divided into three parts: The basics, further details about the input cores & ligands and finally a more detailed look into the customization of the various jobs.

2.1 General Overview & Getting Started

A basic recipe for running **CAT**:

1. Create two directories named 'core' and 'ligand'. The 'core' directory should contain the input cores & the 'ligand' should contain the input ligands. The quantum dots will be exported to the 'QD' directory.

2. Customize the job settings to your liking, see CAT/examples/input_settings.yaml for an example. Note: everything under the `optional` section does **not** have to be included in the input settings. As is implied by the name, everything in `optional` is completely optional.

3. Run **CAT** with the following command: `init_cat input_settings.yaml`

4. Congratulations, you just ran **CAT**!

The default **CAT** settings, at various levels of verbosity, are provided below.

2.1.1 Default Settings

```
path: None

input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
    - OC(C)=O
    - OC(CC)=O
```

2.1.2 Verbose default Settings

```
path: None

input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
    - OC(C)=O
    - OC(CC)=O

optional:
    database:
        dirname: database
        read: True
        write: True
        overwrite: False
        mol_format: [pdb, xyz]
        mongodb: False

    core:
        dirname: core
        dummy: Cl

    ligand:
        dirname: ligand
        optimize: True
        split: True
        cosmo-rs: False

    qd:
        dirname: QD
        optimize: False
        activation_strain: False
        dissociate: False
```

2.1.3 Maximum verbose default Settings

```
path: None

input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
    - OC(C)=O
    - OC(CC)=O

optional:
    database:
        dirname: database
        read: True
        write: True
        overwrite: False
```

(continues on next page)

(continued from previous page)

```
        mol_format: [pdb, xyz]
        mongodb: False

    core:
        dirname: core
        dummy: Cl

    ligand:
        dirname: ligand
        optimize: True
        split: True
        cosmo-rs: False

    qd:
        dirname: QD
        optimize: False
        activation_strain: False
        dissociate:
            core_atom: Cd
            lig_count: 2
            core_core_dist: 5.0
            lig_core_dist: 5.0
            topology:
                6: vertice
                7: edge
                9: face

            job1: False
            s1: False
            job2: False
            s2: False
```

2.2 path

2.2.1 Default Settings

```
path: None
```

2.2.2 Arguments

path `None` or `str` = *None*

The path were all working directories are/will be stored. To use the current working directory, use one of the following values:

- *None*
- .
- *cwd*
- *$PWD*
- /path/to/my/current/working/directory

2.3 input_cores & input_ligands

Thia section related relates the importing and processing of cores and ligands. Ligand & cores can be imported from a wide range of different files and files types, which can roughly be divided into three categories:

1. Files containing coordinates of a single molecule: .xyz, .pdb & .mol files
2. Python objects: `plams.Molecule`, `rdkit.Chem.Mol` & (SMILES) `str`
3. Containers with one or multiple input molecules: directories & .txt files

In the later case, the container can consist of multiple SMILES strings or paths to .xyz, .pdb and/or .mol files. If necessary, containers are searched recursively. Both absolute and relative paths are explored.

2.3.1 Default Settings

```
input_cores:
    - Cd68Se55.xyz:
        guess_bonds: False

input_ligands:
    - OC(C)=O
    - OC(CC)=O
    - OC(CCC)=O
    - OC(CCCC)=O
```

2.3.2 Optional arguments

guess_bonds `bool` = *False*

Try to guess bonds and bond orders in a molecule based on the types atoms and the relative of atoms. Is set to False by default, with the exception of .xyz files.

column `int` = *0*

The column containing the to be imported molecules. Relevant when importing structures from .txt and .xlsx files with multiple columns. Numbering starts from 0.

row `int` = *0*

The first row in a column which contains a molecule. Useful for when, for example, the very first row contains the title of aforementioned row, in which case row = 1 would be a sensible choice. Relevant for .txt and .xlsx files. Numbering starts from 0.

indices `tuple` [`int`] = ()

For cores: Manually specify the atomic index of one ore more atom(s) in the core that will be replaced with ligands. If left empty, all atoms of a user-specified element (see `optional.cores.dummy = str or int`) will be replaced with ligands.

For ligands: Manually specify the atomic index of the ligand atom that will be attached to core (implying argument_dict: `optional.ligand.split = False`).If two atomic indices are rovided, the bond between `tuple` [`0`] and `tuple` [`1`] will be broken and the molecule containing `tuple` [`0`] is attached to the core, (implying argument_dict: `optional.ligand.split = True`). Serves as an alternative to the functional group based `CAT.attachment.ligand_anchoring.find_substructure()` function, which identifies the to be attached atom based on connectivity patterns (*i.e.* functional groups).

In both cases the numbering of atoms starts from 1, following the PLAMS [1, 2] convention.

2.4 Optional

There are a number of arguments which can be used to modify the functionality and behaviour of the quantum dot builder. Herein an overview is provided.

Note: Inclusion of this section in the input file is not required, assuming one is content with the default settings.

2.4.1 Default Settings

```
optional:
    database:
        dirname: database
        read: True
        write: True
        overwrite: False
        mol_format: [pdb, xyz]
        mongodb: False

    core:
        dirname: core
        dummy: Cl

    ligand:
        dirname: ligand
        optimize: True
        split: True
        cosmo-rs: False

    qd:
        dirname: QD
        optimize: False
        activation_strain: False
        dissociate: False
```

2.4.2 Arguments

Database

```
optional:
    database:
        dirname: database
        read: True
        write: True
        overwrite: False
        mol_format: [pdb, xyz]
        mongodb: False
```

database.dirname `str` = *database*

The name of the directory where the database will be stored. The database directory will be created (if it does not yet exist) at the path specified in *path*.

database.read `bool`, `str` or `list` [`str`] = *True*

Before optimizing a structure, check if a geometry is available from previous calculations. If a match is found, use that structure and avoid a geometry reoptimizations. If one wants more control then the boolean can be substituted for a list of strings (*i.e. core*, *ligand* and/or *QD*), meaning that structures will be read only for a specific subset.

For example:

```
optional:
    database:
        read: [core, ligand, QD]  # is equivalent to read: True
```

```
optional:
    database:
        read: ligand
```

database.write `bool`, `str` or `list` [`str`] = *True*

Export the optimized structures to the database of results. Previous results will **not** be overwritten unless `optional.database.overwrite = True`. If one wants more control then the boolean can be substituted for a list of strings (*i.e. core*, *ligand* and/or *QD*), meaning that structures written for for a specific subset.

See **database.read** for a similar relevant example.

database.overwrite `bool`, `str` or `list` [`str`] = *False*

Allows previous results in the database to be overwritten. Only apllicable if `optional.database.write = True`. If one wants more control then the boolean can be substituted for a list of strings (*i.e. core*, *ligand* and/or *QD*), meaning that structures written for for a specific subset.

See **database.read** for a similar relevant example.

database.mol_format `bool`, `str` or `list` [`str`] = [*pdb*, *xyz*]

The file format(s) for storing moleculair structures. By default all structures are stored in the .hdf5 format as (partially) de-serialized .pdb files. Additional formats can be requisted with this keyword. Accepted values: *pdb* and/or *xyz*.

database.mongodb `bool` = *False*

Handles convertion of the database to the mongoDB format. Not implemented as of yet, this keyword is a placeholder.

Core

```
optional:
    core:
        dirname: core
        dummy: Cl
```

core.dirname `str` = *core*

The name of the directory where all cores will be stored. The core directory will be created (if it does not yet exist) at the path specified in *path*.

core.dummy `str` or `int` = *Cl*

The atomic number or atomic symbol of the atoms in the core which are to be replaced with ligands. Alternatively, dummy atoms can be manually specified with the core_indices variable.

Ligand

```
optional:
    ligand:
        dirname: ligand
        optimize: True
        split: True
        cosmo-rs: False
```

ligand.dirname `str` = *ligand*

The name of the directory where all ligands will be stored. The ligand directory will be created (if it does not yet exist) at the path specified in *path*.

ligand.optimize `bool` = *True*

Optimize the geometry of the to be attached ligands. The ligand is split into one or multiple (more or less) linear fragments, which are subsequently optimized (RDKit UFF [1, 2, 3]) and reassembled while checking for the optimal dihedral angle. The ligand fragments are biased towards more linear conformations to minimize inter-ligand repulsion once the ligands are attached to the core.

ligand.split `bool` = *True*

If *False*: The ligand in its entirety is to be attached to the core.

- N^+R_4 -> N^+R_4
- O_2CR -> O_2CR
- HO_2CR -> HO_2CR
- H_3CO_2CR -> H_3CO_2CR

If *True*: A proton, counterion or functional group is to be removed from the ligand before attachment to the core.

- $X^-.N^+R_4$ -> N^+R_4
- HO_2CR -> $O^-{}_2CR$
- $Na^+.O^-{}_2CR$ -> $O^-{}_2CR$
- H_3CO_2CR -> $O^-{}_2CR$

ligand.cosmo-rs `bool` = *False*

Perform a property calculation with COSMO-RS [4, 5, 6, 7]; the COSMO surfaces are constructed using ADF MOPAC [8, 9, 10].

The solvation energy of the ligand and its activity coefficient are calculated in the following solvents: acetone, acetonitrile, dimethyl formamide (DMF), dimethyl sulfoxide (DMSO), ethyl acetate, ethanol, *n*-hexane, toluene and water.

QD

```
optional:
    qd:
        dirname: QD
        optimize: False
        activation_strain: False
        dissociate: False
```

qd.dirname `str` = *QD*

The name of the directory where all quantum dots will be stored. The quantum dot directory will be created (if it does not yet exist) at the path specified in *path*.

qd.optimize `bool` = *False*

Optimize the quantum dot (i.e. core + all ligands) with ADF UFF [3, 11]. The geometry of the core and ligand atoms directly attached to the core are frozen during this optimization.

qd.activation_strain `bool` = *False*

Perform an activation strain analyses [12, 13, 14] (kcal mol^{-1}) on the ligands attached to the quantum dot surface with RDKit UFF [1, 2, 3].

The core is removed during this process; the analyses is thus exclusively focused on ligand deformation and inter-ligand interaction. Yields three terms:

1. dE_{strain} : The energy required to deform the ligand from their equilibrium geometry to the geometry they adopt on the quantum dot surface. This term is, by definition, destabilizing. Also known as the preperation energy (dE_{prep}).

2. dE_{int} : The mutual interaction between all deformed ligands. This term is characterized by the non-covalent interaction between ligands (UFF Lennard-Jones potential) and, depending on the inter-ligand distances, can be either stabilizing or destabilizing.

3. dE : The sum of dE_{strain} and dE_{int}. Accounts for both the destabilizing ligand deformation and (de-)stabilizing interaction between all ligands in the absence of the core.

qd.dissociate `bool` = *False*

Calculate the bond dissociation energy (BDE) of ligands attached to the surface of the core. See *Bond Dissociation Energy* for more details. The calculation consists of five distinct steps:

1. Dissociate all combinations of *n* ligandsand an atom from the core within a radius *r* from aforementioned core atom. General structure: XY_n.

2. Optimize the geometry of XY_n at the first level of theory (lvl1): ADF MOPAC [1, 2, 3].

3. Calculate the "electronic" contribution to the BDE (d*E*) at the first level of theory (lvl1): ADF MOPAC [1, 2, 3]. This step consists of single point calculations of the complete quantum dot, XY_n and all XY_n-dissociated quantum dots.

4. Calculate the thermalchemical contribution to the BDE (dd*G*) at the second level of theory (lvl2): ADF UFF [4, 5]. This step consists of geometry optimizations and frequency analyses of the same compounds used for step 3.

5. $dG = dE_{lvl1} + ddG_{lvl2} = dE_{lvl1} + (dG_{lvl2} - dE_{lvl2})$.

2.5 Bond Dissociation Energy

Calculate the bond dissociation energy (BDE) of ligands attached to the surface of the core. The calculation consists of five distinct steps:

1. Dissociate all combinations of *n* ligands (Y, see **qd.dissociate.lig_count**) and an atom from the core (X, see **qd.dissociate.core_atom**) within a radius *r* from aforementioned core atom (see **qd.dissociate.lig_core_dist** and **qd.dissociate.core_core_dist**). The dissociated compound has the general structure of XY_n.

2. Optimize the geometry of XY_n at the first level of theory (lvl1). Default: ADF MOPAC [1, 2, 3].

3. Calculate the "electronic" contribution to the BDE (d*E*) at the first level of theory (lvl1): ADF MOPAC [1, 2, 3]. This step consists of single point calculations of the complete quantum dot, XY_n and all XY_n-dissociated quantum dots.

4. Calculate the thermalchemical contribution to the BDE (dd*G*) at the second level of theory (lvl2). Default: ADF UFF [4, 5]. This step consists of geometry optimizations and frequency analyses of the same compounds used for step 3.

5. $dG = dE_{lvl1} + ddG_{lvl2} = dE_{lvl1} + (dG_{lvl2} - dE_{lvl2})$.

2.5.1 Default Settings

```
optional:
    qd:
        dissociate:
            core_atom: Cd
            lig_count: 2
            core_core_dist: 5.0
            lig_core_dist: 5.0
            core_index: False
            topology:
```

(continues on next page)

(continued from previous page)

```
            7: vertice
            8: edge
            10: face

        job1: AMSJob
        s1: True
        job2: AMSJob
        s2: True
```

2.5.2 Arguments

qd.dissociate.core_atom `str` or `int` = *Cd*

The atomic number or atomic symbol of the core atoms (X) which are to be dissociated. The core atoms are dissociated in combination with *n* ligands (Y, see **qd.dissociate.lig_count**). Yields a compound with the general formula XY_n.

qd.dissociate.lig_count `int` = *2*

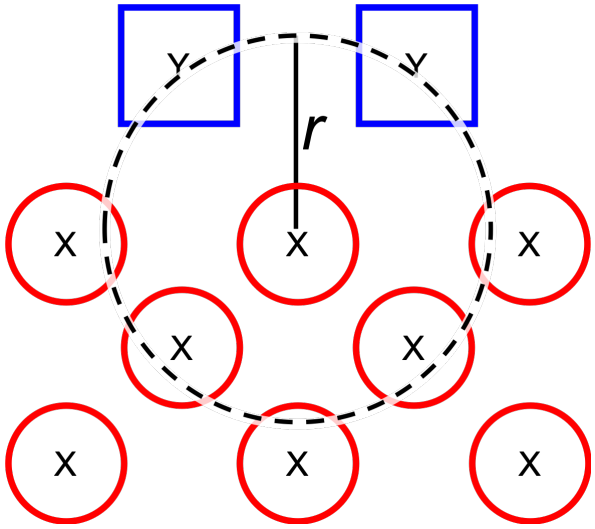
The number of ligands, *n*, which is to be dissociated in combination with a single core atom (X, see **qd.dissociate.core_atom**). Yields a compound with the general formula XY_n.

qd.dissociate.core_core_dist `float` = *5.0*

The maximum to be considered distance (Ångström) between atoms in **qd.dissociate.core_atom**. Used for determining the topology of the core atom (see **qd.dissociate.topology**) and whether it is exposed to the surface of the core or not. It is recommended to use a radius which encapsulates a single (complete) shell of neighbours.

qd.dissociate.lig_core_dist `float` = *5.0*

Dissociate all possible combinations of *n* ligands and a single core atom (see **qd.dissociate.core_atom**) within a given radius (Ångström) from aforementioned core atom. The number of ligands dissociated in combination with a single core atom is controlled by **qd.dissociate.lig_count**.



qd.dissociate.core_index `bool` or `list` [`int`]= *False*

Alternative to **qd.dissociate.lig_core_dist** and **qd.dissociate.core_atom**.

qd.dissociate.topology `dict` = {*7*: *vertice*, *8*: *edge*, *10*: *face*}

A dictionary which translates the number neighbouring core atoms (see **qd.dissociate.core_atom** and **qd.dissociate.core_core_dist**) into a topology. Keys represent the number of neighbours, values represent the matching topology.

Note: values can take on any user-specified value (*e.g.* Miller indices) and are thus not limited to *vertice*, *edge* and/or *face*.

2.5.3 Arguments - Job Customization

qd.dissociate.job1 `type`, `str` or `bool` = *AMSJob*

A `type` object of a `Job` subclass, used for calculating the "electronic" component (dE_{lvl1}) of the bond dissociation energy. Involves single point calculations.

Alternatively, an alias (`str`) can be provided for a specific job type (see *Type Aliases*).

Setting it to *True* (`bool`) will default to `type` (`AMSJob`), while *False* (`bool`) is equivalent to `optional.qd.dissociate = False`.

qd.dissociate.s1 `Settings`, `str` or `bool` =

```
s1:
    input:
        mopac:
            model: PM7
        ams:
            system:
                charge: 0
```

The job `Settings` used for calculating the "electronic" component (dE_{lvl1}) of the bond dissociation energy.

Alternatively, a path (`str`) can be provided to .json or .yaml file containing the job settings.

Setting it to *True* (`bool`) will default to the *MOPAC* block in CAT/data/templates/qd.yaml, while *False* (`bool`) is equivalent to `optional.qd.dissociate = False`.

qd.dissociate.job2 `type`, `str` or `bool` = *AMSJob*

A `type` object of a `Job` subclass, used for calculating the thermal component (ddG_{lvl2}) of the bond dissociation energy. Involves a geometry reoptimizations and frequency analyses.

Alternatively, an alias (`str`) can be provided for a specific job type (see *Type Aliases*).

Setting it to *True* (`bool`) will default to `type` (`AMSJob`), while *False* (`bool`) will skip the thermochemical analysis completely.

qd.dissociate.s2 `Settings`, `str` or `bool` =

```
s2:
    input:
        uff:
            library: uff
        ams:
            system:
                charge: 0
                bondorders:
                    _1: null
```

The job `Settings` used for calculating the thermal component (ddG_{lvl2}) of the bond dissociation energy.

Alternatively, a path (`str`) can be provided to .json or .yaml file containing the job settings.

Setting it to *True* (`bool`) will default to the the *MOPAC* block in CAT/data/templates/qd.yaml, while *False* (`bool`) will skip the thermochemical analysis completely.

2.6 Type Aliases

Aliases are available for a large number of job types, allowing one to pass a `str` instead of a `type` object, thus simplifying the input settings for **CAT**. Aliases are insensitive towards capitalization (or lack thereof).

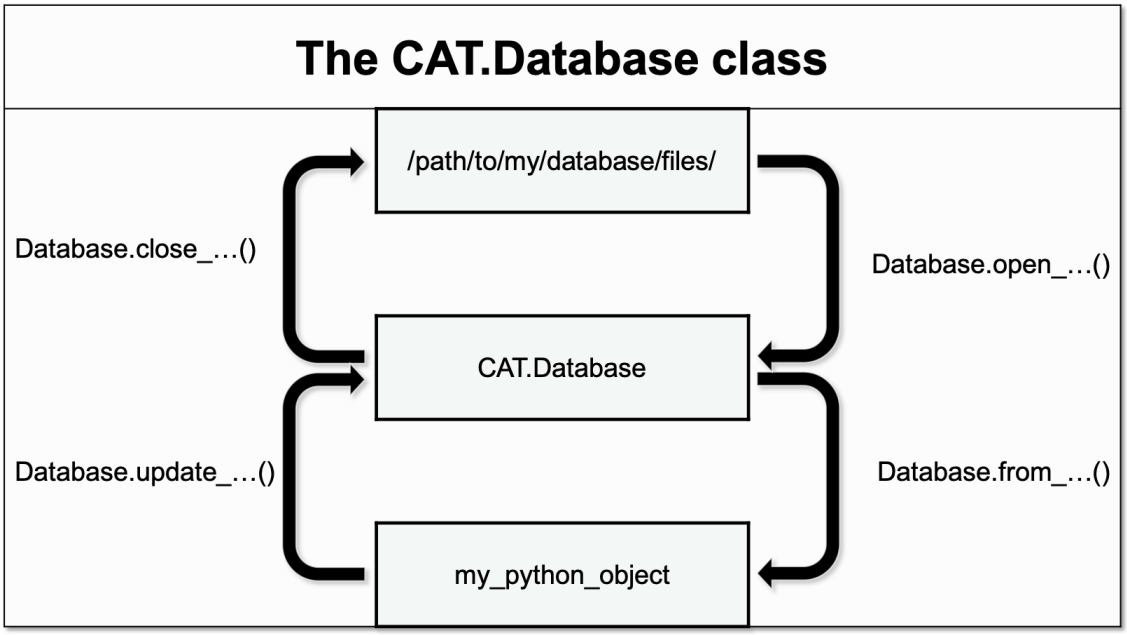
A comprehensive list of `Job` subclasses and their respective aliases (`str`) is presented below.

2.6.1 Aliases

- `ADFJob` = *adf* = *adfjob*
- `AMSJob` = *ams* = *amsjob*
- `UFFJob` = *uff* = *uffjob*
- `BANDJob` = *band* = *bandjob*
- `DFTBJob` = *dftb* = *dftbjob*
- `MOPACJob` = *mopac* = *mopacjob*
- `ReaxFFJob` = *reaxff* = *reaxffjob*
- `Cp2kJob` = *cp2k* = *cp2kjob*
- `ORCAJob` = *orca* = *orcajob*
- `DiracJob` = *dirac* = *diracjob*
- `GamessJob` = *gamess* = *gamessjob*
- `DFTBPlusJob` = *dftbplus* = *dftbplusjob*
- `CRSJob` = *crs* = *cosmo-rs* = *crsjob*

2.7 The Database Class

A Class designed for the storing, retrieval and updating of results.



The methods of the Database class can be divided into three categories accoring to their functionality:

- Opening & closing the database - these methods serve as context managers for loading and unloading parts of the database from the harddrive. These methods should be used in conjunction with `with` statements:

```
import CAT

database = CAT.Database()
with database.open_csv_lig(db.csv_lig) as db:
    print('my ligand database')
with database.open_yaml(db.yaml) as db:
    print('my job settings database')
with h5py.File(db.hdf5) as db:
    print('my structure database')
```

open_csv_lig	*open_csv_qd*	*open_yaml*	h5py.File

- Importing to the database - these methods handle the importing of new data from python objects to the Database class:

update_csv()	*update_yaml()*	*update_hdf5()*	*update_mongodb()*

- Exporting from the database - these methods handle the exporting of data from the Database class to other python objects or remote locations:

from_csv()	*from_hdf5()*

2.7.1 Index

open_yaml

Continued on next page

Table 1 – continued from previous page

open_csv_lig
open_csv_qd
DF
update_mongodb
update_csv
update_yaml
update_hdf5
from_csv
from_hdf5

mol_to_file
as_pdb_array
from_pdb_array
sanitize_yaml_settings

2.7.2 Class API

class CAT.data_handling.database.**Database**(*path=None*, *host: str = 'localhost'*, *port: int = 27017*, ***kwargs*)

The Database class.

Atributes

- **csv_lig** (*str*) – Path and filename of the .csv file containing all ligand related results.
- **csv_qd** (*str*) – Path and filename of the .csv file containing all quantum dot related results.
- **yaml** (*str*) – Path and filename of the .yaml file containing all job settings.
- **hdf5** (*str*) – Path and filename of the .hdf5 file containing all structures (as partiallize de-serialized .pdb files).
- **mongodb** (*None* or *dict*) – Optional: A dictionary with keyword

arguments for pymongo.MongoClient. # noqa

class open_yaml(*path=None*, *write=True*)

Context manager for opening and closing the job settings database.

Parameters

- **path** (*str*) – The path+filename to the database component.
- **write** (*bool*) – Whether or not the database file should be updated after closing **self**.

class open_csv_lig(*path=None*, *write=True*)

Context manager for opening and closing the ligand database.

Parameters

- **path** (*str*) – The path+filename to the database component.
- **write** (*bool*) – Whether or not the database file should be updated after closing **self**.

class open_csv_qd(*path=None*, *write=True*)

Context manager for opening and closing the quantum dot database.

Parameters

- **path** (*str*) – The path+filename to the database component.

- **write** (*bool*) – Whether or not the database file should be updated after closing **self**.

class DF(*df: pandas.core.frame.DataFrame*)
A mutable container for holding dataframes.

A subclass of `dict` containing a single key (`"df"`) and value (a Pandas DataFrame). Calling an item or attribute of *DF* will call said method on the underlaying DataFrame (`self["df"]`). An exception to this is the `"df"` key, which will get/set the DataFrame instead.

update_mongodb(*database: str = 'ligand'*, *overwrite: bool = False*) → None
Export ligand or qd results to the MongoDB database.

Parameters

- **database** (*str*) – The type of database; accepted values are `"ligand"` and `"QD"`.
- **overwrite** (*bool*) – Whether or not previous entries can be overwritten or not.

update_csv(*df*, *database='ligand'*, *columns=None*, *overwrite=False*, *job_recipe=None*, *opt=False*)
Update **self.csv_lig** or **self.csv_qd** with (potentially) new user provided settings.

Parameters

- **df** (*pd.DataFrame* (columns: *str*, index: *str*, values: *plams.Molecule*)) – A dataframe of new (potential) database entries.
- **database** (*str*) – The type of database; accepted values are *ligand* and *QD*.
- **columns** (*None* or *list* [*tuple* [*str*]]) – A list of column keys in **df** which (potentially) are to be added to **self**. If *None*: Add all columns.
- **overwrite** (*bool*) – Whether or not previous entries can be overwritten or not.
- **job_recipe** (*None* or *plams.Settings* (superclass: *dict*)) – A Settings object with settings specific to a job.

update_yaml(*job_recipe*)
Update **self.yaml** with (potentially) new user provided settings.

Parameters **job_recipe** (*plams.Settings* (superclass: *dict*)) – A settings object with one or more settings specific to a job.

Returns A dictionary with the column names as keys and the key for **self.yaml** as matching values.

Return type *dict* (keys: *str*, values: *str*)

update_hdf5(*df*, *database='ligand'*, *overwrite=False*, *opt=False*)
Export molecules (see the *mol* column in **df**) to the structure database. Returns a series with the **self.hdf5** indices of all new entries.

Parameters

- **df** (*pd.DataFrame* (columns: *str*, index: *str*, values: *plams.Molecule*)) – A dataframe of new (potential) database entries.
- **database** (*str*) – The type of database; accepted values are *ligand* and *QD*.
- **overwrite** (*bool*) – Whether or not previous entries can be overwritten or not.

Returns A series with the index of all new molecules in **self.hdf5**

Return type *pd.Series* (index: *str*, values: *np.int64*)

from_csv (*df*, *database='ligand'*, *get_mol=True*, *inplace=True*)

Pull results from **self.csv_lig** or **self.csv_qd**. Performs in inplace update of **df** if **inplace** = *True*, returing *None*.

Parameters

- **df** (*pd.DataFrame* (columns: *str*, index: *str*, values: *plams.Molecule*)) – A dataframe of new (potential) database entries.
- **database** (*str*) – The type of database; accepted values are *ligand* and *QD*.
- **columns** – A list of to be updated columns in **df**.
- **get_mol** (*bool*) – Attempt to pull preexisting molecules from the database. See **inplace** for more details.
- **inplace** (*bool*) – If *True* perform an inplace update of the *mol* column in **df**. Otherwise Return a new series of PLAMS molecules.

Returns If **inplace** = *False*: return a new series of PLAMS molecules pulled from **self**, else return *None*

Return type *None* or *pd.Series* (index: *str*, values: *plams.Molecule*)

from_hdf5 (*index*, *database='ligand'*, *rdmol=True*, *close=True*)

Import structures from the hdf5 database as RDKit or PLAMS molecules.

Parameters

- **index** (*list* [*int*]) – The indices of the to be retrieved structures.
- **database** (*str*) – The type of database; accepted values are *ligand* and *QD*.
- **rdmol** (*bool*) – If *True*, return an RDKit molecule instead of a PLAMS molecule.
- **close** (*bool*) – If the database component should be closed afterwards.

Returns A list of PLAMS or RDKit molecules.

Return type *list* [*plams.Molecule* or *rdkit.Chem.Mol*]

hdf5_availability (*timeout: float = 5.0*, *max_attempts: Optional[int] = None*) → None

Check if a .hdf5 file is opened by another process; return once it is not.

If two processes attempt to simultaneously open a single hdf5 file then h5py will raise an `OSError`. The purpose of this function is ensure that a .hdf5 is actually closed, thus allowing `to_hdf5()` to safely access **filename** without the risk of raising an `OSError`.

Parameters

- **filename** (*str*) – The path+filename of the hdf5 file.
- **timeout** (*float*) – Time timeout, in seconds, between subsequent attempts of opening **filename**.
- **max_attempts** (*int*) – Optional: The maximum number attempts for opening **filename**. If the maximum number of attempts is exceeded, raise an `OSError`.

Raises `OSError` – Raised if **max_attempts** is exceded.

2.7.3 Function API

CAT.data_handling.database_functions.**mol_to_file**(*mol_list, path=None, overwrite=False, mol_format=['xyz', 'pdb']*)

Export all molecules in **mol_list** to .pdb and/or .xyz files.

Parameters

- **mol_list** (*list* [*plams.Molecule*]) – A list of PLAMS molecules.
- **path** (*None* or *str*) – The path to the directory where the molecules will be stored. Defaults to the current working directory if *None*.
- **overwrite** (*bool*) – If previously generated structures can be overwritten or not.
- **mol_format** (*list* [*str*]) – A list of strings with the to-be exported file types. Accepted values are *xyz* and/or *pdb*.

CAT.data_handling.database_functions.**as_pdb_array**(*mol_list*, *min_size=0*)

Converts a list of PLAMS molecule into an array of strings representing (partially) de-serialized .pdb files.

Parameters

- **mol_list** (*list* [*plams.Molecule*]) – A list of PLAMS molecules.
- **min_size** (*int*) – The minimumum length of the pdb_array. The array is padded with empty strings if required.

Returns An array with *m* partially deserialized .pdb files with up to *n* lines each.

Return type *m*n np.ndarray* [*np.bytes* |*S80*]

CAT.data_handling.database_functions.**from_pdb_array**(*array*, *rdmol=True*)

Converts an array with a (partially) de-serialized .pdb file into an RDKit or PLAMS molecule.

Parameters

- **array** (*n np.ndarray* [*np.bytes* / S80]) – A (partially) de-serialized .pdb file with *n* lines.
- **rdmol** (*bool*) – If *True*, return an RDKit molecule instead of a PLAMS molecule.

Returns A PLAMS or RDKit molecule build from **array**.

Return type *plams.Molecule* or *rdkit.Chem.Mol*

CAT.data_handling.database_functions.**sanitize_yaml_settings**(*settings*, *job_type*)

Remove a predetermined set of unwanted keys and values from a settings object.

Parameters **settings** (*plams.Settings* (superclass: *dict*)) – A settings object with, potentially, undesired keys and values.

Returns A (nested) dictionary with unwanted keys and values removed.

Return type *dict*

Index