OpenFF Toolkit Documentation

Release 0.10.1+119.g8acf6adb.dirty

Open Force Field Consortium

Jul 01, 2022

GETTING STARTED

A modern, extensible library for molecular mechanics force field science from the Open Force Field Initiative

CHAPTER

ONE

INSTALLATION

1.1 Installing via conda

The simplest way to install the Open Force Field Toolkit is via the conda package manager. We publish packages via `conda-forge`.

If you are using the Anaconda scientific Python distribution, you already have the `conda` package manager installed. If not, the quickest way to get started is to install the Miniconda distribution, a lightweight, minimal installation of Python and the Conda package manager. See the conda documentation for detailed installation instructions. We recommend Miniforge, a drop-in replacement for Miniconda that uses the community-run `conda-forge` channel by default.

Once Conda is installed, use it to install the OpenFF Toolkit:

```
$ conda install -c conda-forge openff-toolkit
```

Note: Installation via the Conda package manager is the preferred method since all dependencies are automatically fetched and installed for you.

1.1.1 OS support

The OpenFF Toolkit is pure Python, and we expect it to work on any platform that supports its dependencies. Our automated testing takes place on both MacOS and Ubuntu Linux. For Windows support, we recommend using the Windows Subsystem for Linux (WSL) to run a Linux system integrated into Windows. We strongly suggest using WSL2, if your hardware supports it, for a smoother experience. WSL2 requires virtualization support in hardware. This is available on most modern CPUs, but may require activation in the BIOS.

Once WSL is configured, installing and using the Toolkit is done exactly as it would be for Linux. Note that by default, Jupyter Notebook will not be able to open a browser window and will log an error on startup; just ignore the error and open the link it provides in your ordinary Windows web browser.

Note: WSL2 does support GPU compute, at least with nvidia cards, but setting it up takes some work.

1.1.2 Conda environments

Conda environments that mix packages from the default channels and `conda-forge` can become inconsistent; to prevent this mixing, we recommend using `conda-forge` for all packages. The easiest way to do this is to install Conda with Miniforge.

If you already have a complex environment, or you wish to install a version of the Toolkit that is incompatible with other software you have installed, you can install the Toolkit into a new environment:

```
$ conda create -c conda-forge --name offtk openff-toolkit
```

An environment must be activated in any new shell session to use the software installed in it:

```
$ conda activate offtk
```

1.1.3 Upgrading your installation

To update an earlier `conda` installation of `openff-toolkit` to the latest release version, you can use `conda update`:

```
$ conda update -c conda-forge openff-toolkit
```

Note that this may update other packages or install new packages if the most recent release of the Toolkit requires it.

1.2 Installing from source

The OpenFF Toolkit has a lot of dependencies, so we strongly encourage installation with a package manager. The *developer's guide* describes setting up a development environment. If you're sure you want to install from source, check the `conda-forge` recipe for current dependencies, install them, download and extract the source distribution from GitHub, and then run `setup.py`:

```
$ cd openff-toolkit
$ python setup.py install
```

1.3 Single-file installer

As of release 0.4.1, single-file installers are available for each Open Force Field Toolkit release. These are provided primarily for users who do not have access to the Anaconda cloud for installing packages. These installers have few requirements beyond a Linux or MacOS operating system and will, in one command, produce a functional Python executable containing the Open Force Field Toolkit, as well as all required dependencies. The installers are very similar to the widely-used Miniconda `*.sh` files. Accordingly, installation using the "single-file installer" does not require root access.

The installers are between 200 and 300 MB each, and can be downloaded from the "Assets" section of the Toolkit's GitHub Releases page. They are generated using a workflow leveraging the Conda Constructor utility.

Please report any installer difficulties to the OFF Toolkit issue tracker, as we hope to make this a major distribution channel for the toolkit moving forward.

1.3.1 Installation

Download the appropriate installer (`openff-toolkit-<X.Y.Z>-<py3x>-<your platform>-x86_64.sh.zip`) from the "Assets" section at the bottom of the desired release on GitHub. Then, install the toolkit with the following command:

```
$ bash openff-toolkit-<X.Y.Z>-py37-<your platform>-x86_64.sh
```

and follow the prompts.

Note: You must have write access to the installation directory. This is generally somewhere in the user's home directory. When prompted, we recommend NOT making modifications to your `bash_profile`.

Warning: We recommend that you do not install this package as root. Conda is intended to support on-the-fly creation of several independent environments, and managing a multi-user Conda installation is complicated.

1.3.2 Usage

Any time you want to use this Conda environment in a terminal, run

```
$ source <install_directory>/etc/profile.d/conda.sh
$ conda activate base
```

Once the `base` environment is activated, your system will default to use Python (and other executables) from the newly installed Conda environment. For more information about Conda environments, see *Conda environments*

1.4 Optional dependencies (toolkits)

The OpenFF Toolkit outsources many common computational chemistry algorithms to other toolkits. Only one such toolkit is needed to gain access to all of the OpenFF Toolkit's features. If more than one is available, the Toolkit allows the user to specify their preference with the `toolkit_registry` argument to most functions and methods.

The `openff-toolkit` package installs everything needed to run the toolkit, including the optional dependencies RDKit and AmberTools. To install only the hard dependencies and provide your own optional dependencies, install the `openff-toolkit-base` package.

The OpenFF Toolkit requires an external toolkit for most functions. Though a `builtin` toolkit is provided, it implements only a small number of functions and is intended primarily for testing.

There are certain differences in toolkit behavior between RDKit/AmberTools and OpenEye when reading a small fraction of molecules, and we encourage you to report any unexpected behavior that may be caused by toolkit differences to our issue tracker.

1.4.1 RDKit

RDKit is a free and open source chemistry toolkit installed by default with the `openff-toolkit` package. It provides most of the functionality that the OpenFF Toolkit relies on.

1.4.2 AmberTools

AmberTools is a collection of free tools provided with the Amber MD software and installed by default with the `openff-toolkit` package. It provides a free implementation of functionality required by OpenFF Toolkit and not provided by RDKit.

1.4.3 OpenEye

The OpenFF Toolkit can optionally make use of the OpenEye toolkit if the user has a license key installed. Academic laboratories intending to release results into the public domain can obtain a free license key, while other users (including academics intending to use the software for purposes of generating protected intellectual property) must pay to obtain a license.

To install the OpenEye toolkits:

```
$ conda install -c openeye -c conda-forge openeye-toolkits
```

Though OpenEye can be installed for free, using it requires a license file. No essential `openff-toolkit` release capabilities *require* the OpenEye toolkit, but the Open Force Field developers make use of it in parameterizing new open source force fields.

CHAPTER

TWO

EXAMPLES USING SMIRNOFF WITH THE TOOLKIT

The following examples are available in the OpenFF toolkit repository. Each can be run interactively in the browser with binder, without installing anything on your computer.

2.1 Index of provided examples

- toolkit_showcase - parameterize a protein-ligand system with an OpenFF force field, simulate the resulting system, and visualize the result in the notebook
- forcefield_modification - modify force field parameters and evaluate how system energy changes
- conformer_energies - compute conformer energies of one or more small molecules using a SMIRNOFF force field
- SMIRNOFF_simulation - simulation of a molecule in the gas phase with the SMIRNOFF force field format
- forcefield_modification - modify force field parameters and evaluate how system energy changes
- using_smirnoff_in_amber_or_gromacs - convert a System generated with the Open Force Field Toolkit, which can be simulated natively with OpenMM, into AMBER prmtop/inpcrd and GROMACS top/gro input files through the ParmEd library.
- swap_amber_parameters - take a prepared AMBER protein-ligand system (prmtop and crd) along with a structure file of the ligand, and replace ligand parameters with OpenFF parameters.
- inspect_assigned_parameters - check which parameters are used in which molecules and generate parameter usage statistics.
- using_smirnoff_with_amber_protein_forcefield - use SMIRNOFF parameters for small molecules in combination with more conventional force fields for proteins and other components of your system (using ParmEd to combine parameterized structures)
- check_dataset_parameter_coverage - shows how to use the Open Force Field Toolkit to ingest a dataset of molecules, and generate a report summarizing any chemistry that can not be parameterized.
- visualization - shows how rich representation of `Molecule` objects work in the context of Jupyter Notebooks.

CHAPTER

THREE

RELEASE HISTORY

Releases follow the `major.minor.micro` scheme recommended by PEP440, where

- `major` increments denote a change that may break API compatibility with previous `major` releases
- `minor` increments add features but do not break API compatibility
- `micro` increments represent bugfix releases or improvements in documentation

3.1 Migration guide

3.1.1 Units

Import the unit registry provided by `openff-units`:

```
from openff.units import unit
```

Create a `unit.Quantity` object:

```
value = unit.Quantity(1.0, unit.nanometer)  # or 1.0 * unit.nanometer
```

Inspect the value and unit of this quantity:

```
print(value.magnitude)  # or value.m
# 1.0
print(value.units)
# <Unit('nanometer')>
```

Convert to compatible units:

```
converted = value.to(unit.angstrom)
print(converted)
# 10.0 <Unit('angstrom')>
```

Report the value in compatible units:

```
print(value.magnitude_as(unit.angstrom))  $ or .m_as()
# 10.0 <Unit('angstrom')>
```

Convert to and from OpenMM quantities:

```
from openff.units.openmm import to_openmm, from_openmm
value_openmm = to_openmm(value)
print(value_openmm)
# Quantity(value=1.0, unit=nanometer)
print(type(value_openmm))
# 1.0 <Unit('nanometer')>
value_roundtrip = from_openmm(value_openmm)
print(value_roundtrip)
# 1.0 <Unit('nanometer')>
```

3.2 Current Development

- PR #1276: Removes the `use_interchange` argument to `create_openmm_system`. Deletes the `create_force` and `postprocess_system` methods of `ParameterHandler` `ParameterHandler.create_force`, `ParameterHandler.postprocess_system` and other methods related to creating OpenMM systems and forces. This is now handled in Interchange.
- PR #1303: Deprecates `Topology.particles`, `Topology.n_particles`, `Topology.particle_index` as `Molecule` objects do not store virtual sites, only atoms.
- PR #1297: Drops support for Python 3.7, following NEP-29.
- PR #1194: Adds `Topology.__add__`, allowing `Topology` objects to be added together, including added in-place, using the `+` operator.
- PR #1277: Adds support for version 0.4 of the `<Electrostatics>` section of the SMIRNOFF specification.
- PR #1279: `ParameterHandler.version` and the `.version` attribute of its subclasses is now a `Version` object. Previously it was a string, which is not safe for PEP440-style versioning.
- PR #1250: Adds support for `return_topology` in the Interchange code path in `create_openmm_system`.
- PR #964: Adds initial implementation of atom metadata dictionaries.
- PR #1097: Deprecates TopologyMolecule.
- PR #1097: Topology.from_openmm is no longer guaranteed to maintain the ordering of bonds, but now explicitly guarantees that it maintains the order of atoms (Neither of these ordering guarantees were explicitly documented before, but this may be a change from the previous behavior).
- PR #1165: Adds the boolean argument `use_interchange` to `create_openmm_system` with a default value of False. Setting it to True routes `openmm.System` creation through Interchange.
- PR #1192: Add re-exports for core classes to the new `openff.toolkit.app` module and re-exports for parameter types to the new `openff.toolkit.topology.parametertypes` module. This does not affect existing paths and gives some new, easier to remember paths to core objects.
- PR #1198: Ensure the vdW switch width is correctly set and enabled.
- PR #1213: Removes `Topology.charge_model` and `Topology.fractional_bond_order_model`.

3.2.1 Critical bugfixes

- PR #1200: Fixes a bug (Issue #1199) in which library charges were ignored in some force fields, including `openff-2.0.0` code name "Sage." This resulted in the TIP3P partial charges included Sage not being applied correctly in versions 0.10.1 and 0.10.2 of the OpenFF Toolkit. This regression was not present in version 0.10.0 and earlier and therefore is not believed to have any impact on the fitting or benchmarking of the first release of Sage (version 2.0.0). The change causing regression only affected library charges and therefore no other parameter types are believed to be affected.

3.2.2 Behaviors changed and bugfixes

- PR #1277: Version 0.3 `<Electrostatics>` sections of OFFXML files will automatically be up-converted (in memory) to version 0.4 according to the recomendations provided in OFF-EP 0005. Note this means the `method` attribute is replaced by `periodic_potential`, `nonperiodic_potential`, and `exception_potential`.
- PR #1277: Fixes a bug in which attempting to convert `ElectrostaticsHandler.switch_width` did nothing.
- PR #1130: Running unit tests will no longer generate force field files in the local directory.
- PR #1182: Removes `Atom.element`, thereby also removing `Atom.element.symbol`, `Atom.element.mass` and `Atom.element.atomic_number`. These are replaced with corresponding properties directly on the `Atom` class: `Atom.symbol`, `Atom.mass`, and `Atom.atomic_number`.
- PR #1209: Fixes Issue #1073, where the `fractional_bondorder_method` kwarg to the `BondHandler` initializer was being ignored.
- PR #1214: A long overdue fix for Issue #837! If OpenEye is available, the `ToolkitAM1BCCHandler` will use the ELF10 method to select conformers for AM1BCC charge assignment.
- PR #1160: Fixes the bug identified in Issue #1159, in which the order of atoms defining a `BondChargeVirtualSite` (and possibly other virtual sites types too) might be reversed if the `match` attribute of the virtual site has a value of `"once"`.
- PR #1231: Fixes Issue #1181 and Issue #1190, where in rare cases double bond stereo would cause `to_rdkit` to raise an error. The transfer of double bond stereochemistry from OpenFF's E/Z representation to RDKit's local representation is now handled as a constraint satisfaction problem.

3.2.3 Examples added

- PR #1113: Updates the Amber/GROMACS example to use Interchange.

3.2.4 Tests updated

- PR #1188: Add an `<Electrostatics>` section to the TIP3P force field file used in testing (`test_forcefields/tip3p.offxml`)

3.2.5 Minor bugfixes

- PR #1290: Fixes Issue #1216 by adding internal logic to handle the possibility that multiple vsites share the same parent atom, and makes the return value of `VirtualSiteHandler.find_matches` be closer to the base class.

3.3 0.10.5 Bugfix release

- PR #1252: Refactors virtual site support, resolving Issue #1235, Issue #1233, Issue #1222, Issue #1221, and Issue #1206.
  - Attempts to make virtual site handler more resilient through code simplification.
  - Virtual sites are now associated with a particular 'parent' atom, rather than with a set of atoms. In particular, when checking if a v-site has been assigned we now only check the main 'parent' atom associated with the v-site, rather than all additional orientation atoms. As an example, if a force field contained a bond-charge v-site that matches [O:1]=[C:2] and a monovalent lone pair that matches [O:1]=[C:2]-[*:3] in that order, then only the monovalent lone pair will be assigned to formaldehyde as the oxygen is the main atom that would be associated with both v-sites, and the monovalent lone pair appears later in the hierarchy. This constitutes a behaviour change over previous versions.
  - All v-site exclusion policies have been removed except for 'parents' which has been updated to match OFF-EP 0006.
  - checks have been added to enforce that the 'match' keyword complies with the SMIRNOFF spec.
  - Molecule virtual site classes no longer store FF data such as epsilon and sigma.
  - Sanity checks have been added when matching chemical environments for v-sites that ensure the environment looks like one of our expected test cases.
  - Fixes di- and trivalent lone pairs mixing the `:1` and `:2` indices.
  - Fixes trivalent v-site positioning.
  - Correctly splits `TopologyVirtualSite` and `TopologyVirtualParticle` so that virtual particles no longer have attributes such as `particles`, and ensure that indexing methods now work correctly.

3.4 0.10.4 Bugfix release

3.4.1 Critical bugfixes

- PR #1242: Fixes Issue #837. If OpenEye Toolkits are available, `ToolkitAM1BCCHandler` will use the ELF10 method to select conformers for AM1-BCC charge assignment.
- PR #1184: Fixes Issue #1181 and Issue #1190, where in rare cases double bond stereochemistry would cause `Molecule.to_rdkit` to raise an error. The transfer of double bond stereochemistry from OpenFF's E/Z representation to RDKit's local representation is now handled as a constraint satisfaction problem.

3.5 0.10.3 Bugfix release

3.5.1 Critical bugfixes

- PR #1200: Fixes a bug (Issue #1199) in which library charges were ignored in some force fields, including `openff-2.0.0` code name "Sage." This resulted in the TIP3P partial charges included Sage not being applied correctly in versions 0.10.1 and 0.10.2 of the OpenFF Toolkit. This regression was not present in version 0.10.0 and earlier and therefore is not believed to have any impact on the fitting or benchmarking of the first release of Sage (version 2.0.0). The change causing the regression only affected library charges and therefore no other parameter types are believed to be affected.

3.5.2 API breaking changes

- PR #855: In earlier versions of the toolkit, we had mistakenly made the assumption that cheminformatics toolkits agreed on the number and membership of rings. However we later learned that this was not true. This PR removes `Molecule.rings` and `Molecule.n_rings`. To find rings in a molecule, directly use a cheminformatics toolkit after using `Molecule.to_rdkit` or `Molecule.to_openeye`. `Atom.is_in_ring` and `Bond.is_in_ring` are now methods, not properties.

3.5.3 Behaviors changed and bugfixes

- PR #1171: Failure of `Molecule.apply_elf_conformer_selection()` due to excluding all available conformations (Issue #428) now provides a better error. The `make_carboxylic_acids_cis` argument (`False` by default) has been added to `Molecule.generate_conformers()` to mitigate a common cause of this error. By setting this argument to `True` in internal use of this method, trans carboxylic acids are no longer generated in `Molecule.assign_partial_charges()` and `Molecule.assign_fractional_bond_orders()` methods (though users may still pass trans conformers in, they'll just be pruned by ELF methods). This should work around most instances of the OpenEye Omega bug where trans carboxylic acids are more common than they should be.

3.5.4 Improved documentation and warnings

- PR #1172: Adding discussion about constraints to the FAQ
- PR #1173: Expand on the SMIRNOFF section of the toolkit docs
- PR #855: Refactors `Atom.is_in_ring` and `Bond.is_in_ring` to use corresponding functionality in OpenEye and RDKit wrappers.

3.5.5 API breaking changes

- PR #855: Removes `Molecule.rings` and `Molecule.n_rings`. To find rings in a molecule, directly use a cheminformatics toolkit after using `Molecule.to_rdkit` or `Molecule.to_openeye`. `Atom.is_in_ring` and `Bond.is_in_ring` are now methods, not properties.

3.6 0.10.2 Bugfix release

3.6.1 API-breaking changes

- PR #1118: `Molecule.to_hill_formula` is now a class method and no longer accepts input of NetworkX graphs.

3.6.2 Behaviors changed and bugfixes

- PR #1160: Fixes a major bug identified in Issue #1159, in which the order of atoms defining a `BondChargeVirtualSite` (and possibly other virtual sites types too) might be reversed if the `match` attribute of the virtual site has a value of `"once"`.
- PR #1130: Running unit tests will no longer generate force field files in the local directory.
- PR #1148: Adds a new exception `UnsupportedFileTypeError` and descriptive error message when attempting to use `Molecule.from_file` to parse XYZ/`.xyz` files.
- PR #1153: Fixes Issue #1152 in which running `Molecule.generate_conformers` using the OpenEye backend would use the stereochemistry from an existing conformer instead of the stereochemistry from the molecular graph, leading to undefined behavior if the molecule had a 2D conformer.
- PR #1158: Fixes the default representation of `Molecule` failing in Jupyter notebooks when NGLview is not installed.
- PR #1151: Fixes Issue #1150, in which calling `Molecule.assign_fractional_bond_orders` with all default arguments would lead to an error as a result of trying to lowercase `None`.
- PR #1149: `TopologyAtom`, `TopologyBond`, and `TopologyVirtualSite` now properly reference their reference molecule from their `.molecule` attribute.
- PR #1155: Ensures big-endian byte order of NumPy arrays when serialized to dictionaries or files formats except JSON.
- PR #1163: Fixes the bug identified in Issue #1161, which was caused by the use of the deprecated `pkg_resources` package. Now the recommended `importlib_metadata` package is used instead.

3.6.3 Breaking changes

- PR #1118: `Molecule.to_hill_formula` is now a class method and no longer accepts input of NetworkX graphs.
- PR #1156: Removes `ParseError` and `MessageException`, which has been deprecated since version 0.10.0.

3.6.4 Examples added

- PR #1113: Updates the Amber/GROMACS example to use Interchange.

3.7 0.10.1 Minor feature and bugfix release

3.7.1 Behaviors changed and bugfixes

- PR #1096: Atom names generated by `Molecule.generate_unique_atom_names` are now appended with an `"x"`. See the linked issue for more details.
- PR #1050: In `Molecule.generate_conformers`, a single toolkit wrapper failing to generate conformers is no longer fatal, but if all wrappers in a registry fail, then a `ValueError` will be raised. This mirrors the behavior of `Molecule.assign_partial_charges`.
- PR #1050: Conformer generation failures in `OpenEyeToolkitWrapper.generate_conformers`, and `RDKitToolkitWrapper.generate_conformers` now each raise `openff.toolkit.utils.exceptions.ConformerGenerationError` if conformer generation fails. The same behavior occurs in `Molecule.generate_conformers`, but only when the `toolkit_registry` argument is a `ToolkitWrapper`, not when it is a `ToolkitRegistry`.
- PR #1046: Changes OFFXML output to replace tabs with 4 spaces to standardize representation in different text viewers.
- PR #1001: RDKit `Mol` objects created through the `Molecule.to_rdkit()` method have the `NoImplicit` property set to `True` on all atoms. This prevents RDKit from incorrectly adding hydrogen atoms to to molecule.
- PR #1058: Removes the unimplemented methods `ForceField.create_parmed_structure`, `Topology.to_parmed`, and `Topology.from_parmed`.
- PR #1065: The example `conformer_energies.py` script now uses the Sage 2.0.0 force field.
- PR #1036: SMARTS matching logic for library charges was updated to use only one unique match instead of enumerating all possible matches. This results in faster matching, particularly with larger molecules. No adverse side effects were found in testing, but bad behavior may possibly exist in some unknown cases. Note that the default behavior for other parameter handlers was not updated.
- PR #1001: Revamped the `Molecule.visualize()` method's `rdkit` backend for more pleasing and idiomatic 2D visualization by default.
- PR #1087: Fixes Issue #1073 in which `Molecule.__repr__` fails if the molecule can not be represented as a SMILES pattern. Now, if SMILES generation fails, the molecule will be described by its Hill formula.
- PR #1052: Fixes Issue #986 by raising a subclass of `AttributeError` in `_ParameterAttributeHandler.__getattr__`
- PR #1030: Fixes a bug in which the expectations for capitalization for values of `bond_order_model` attributes and keywords are inconsistent.
- PR #1101: Fixes a bug in which calling `to_qcschema` on a molecule with no connectivity feeds `QCElemental.Molecule` an empty list for the `connectivity` field; now feeds `None`.

3.7.2 Tests updated

- PR #1017: Ensures that OpenEye-only CI builds really do lack both AmberTools and RDKit.

3.7.3 Improved documentation and warnings

- PR #1065: Example notebooks were updated to use the Sage Open Force Field
- PR #1062: Rewrote installation guide for clarity and comprehensiveness.

3.8 0.10.0 Improvements for force field fitting

3.8.1 Behaviors changed

- PR #1021: Renames `openff.toolkit.utils.exceptions.ParseError` to `openff.toolkit.utils.exceptions.SMILESParseError` to avoid a conflict with an identically-named exception in the SMIRNOFF XML parsing code.
- PR #1021: Renames and moves `openff.toolkit.typing.engines.smirnoff.forcefield.ParseError` to `openff.toolkit.utils.exceptions.SMIRNOFFParseError`. This `ParseError` is deprecated and will be removed in a future release.

3.8.2 New features and behaviors changed

- PR #1027: Corrects interconversion of Molecule objects with OEMol objects by ensuring atom names are correctly accessible via the `OEAtomBase.GetName()` and `OEAtomBase.SetName()` methods, rather that the non-standard `OEAtomBase.GetData("name")` and `OEAtomBase.SetData("name", name)`.
- PR #1007: Resolves Issue #456 by adding the `normalize_partial_charges` (default is `True`) keyword argument to `Molecule.assign_partial_charges`, `AmberToolsToolkitWrapper.assign_partial_charges`, `OpenEyeToolkitWrapper.assign_partial_charges`, `RDKitToolkitWrapper.assign_partial_charges`, and `BuiltInToolkitWrapper.assign_partial_charges`. This adds an offset to each atom's partial charge to ensure that their sum is equal to the net charge on the molecule (to the limit of a python float's precision, generally less than 1e-6 electron charge). **Note that, because this new behavior is ON by default, it may slightly affect the partial charges and energies of systems generated by running `create_openmm_system`.**
- PR #954: Adds `LibraryChargeType.from_molecule` which returns a `LibraryChargeType` object that will match the full molecule being parameterized, and assign it the same partial charges as are set on the input molecule.
- PR #923: Adds `Molecule.nth_degree_neighbors`, `Topology.nth_degree_neighbors`, `TopologyMolecule.nth_degree_neighbors`, which returns pairs of atoms that are separated in a molecule or topology by *exactly* N atoms.
- PR #917: `ForceField.create_openmm_system` now ensures that the cutoff of the `NonbondedForce` is set to the cutoff of the `vdWHandler` when it and a `Electrostatics` handler are present in the force field.
- PR #850: `OpenEyeToolkitWrapper.is_available` now returns `True` if *any* OpenEye tools are licensed (and installed). This allows, i.e, use of functionality that requires `OEChem` without having an `OEOmega` license.

- PR #909: Virtual site positions can now be computed directly in the toolkit. This functionality is accessed through
  - `FrozenMolecule.compute_virtual_site_positions_from_conformer`
  - `VirtualSite.compute_positions_from_conformer`
  - `VirtualParticle.compute_position_from_conformer`
  - `FrozenMolecule.compute_virtual_site_positions_from_atom_positions`
  - `VirtualSite.compute_positions_from_atom_positions`
  - `VirtualParticle.compute_position_from_atom_positions` where the positions can be computed from a stored conformer, or an input vector of atom positions.
  - Tests have been added (`TestMolecule.test_*_virtual_site_position`) to check for sane behavior. The tests do not directly compare OpenMM position equivalence, but offline tests show that they are equivalent.
  - The helper method `VirtualSiteHandler.create_openff_virtual_sites` is now public, which returns a modified topology with virtual sites added.
  - Virtual sites now expose the parameters used to create its local frame via the read-only properties
    - `VirtualSite.local_frame_weights`
    - `VirtualSite.local_frame_position`
  - Adding virtual sites via the `Molecule` API now have defaults for `sigma`, `epsilon`, and `charge_increment` set to 0 with appropriate units, rather than `None`
- PR #956: Added `ForceField.get_partial_charges()` to more easily compute the partial charges assigned by a force field for a molecule.
- PR #1006: Two behavior changes in the SMILES output for `to_file()` and `to_file_obj()`:
  - The RDKit and OpenEye wrappers now output the same SMILES as `to_smiles()`. This uses explicit hydrogens rather than the toolkit's default of implicit hydrogens.
  - The RDKit wrapper no longer includes a header line. This improves the consistency between the OpenEye and RDKit outputs.

3.8.3 Bugfixes

- PR #1024: Small changes for compatibility with OpenMM 7.6.
- PR #1003: Fixes Issue #1000, where a stereochemistry warning is sometimes erroneously emitted when loading a stereogenic molecule using `Molecule.from_pdb_and_smiles`
- PR #1002: Fixes a bug in which OFFXML files could inadvertently be loaded from subdirectories.
- PR #969: Fixes a bug in which the cutoff distance of the `NonbondedForce` generated by `ForceField.create_openmm_system` was not set to the value specified by the vdW and Electrostatics handlers.
- PR #909: Fixed several bugs related to creating an OpenMM system with virtual sites created via the `Molecule` virtual site API
- PR #1006: Many small fixes to the toolkit wrapper I/O for better error handling, improved consistency between reading from a file vs. file object, and improved consistency between the RDKit and OEChem toolkit wrappers. For the full list see Issue #1005. Some of the more significant fixes are:
  - `RDKitToolkitWrapper.from_file_obj()` now uses the same structure normaliation as `from_file()`.

- `from_smiles()` now raises an `openff.toolkit.utils.exceptions.SMILESParsingError` if the SMILES could not be parsed.
- OEChem input and output files now raise an OSError if the file could not be opened.
- All input file object readers now support file objects open in binary mode.

3.8.4 Examples added

- PR #763: Adds an introductory example showcasing the toolkit parameterizing a protein-ligand simulation.
- PR #955: Refreshed the force field modification example
- PR #934 and conda-forge/openff-toolkit-feedstock#9: Added `openff-toolkit-examples` Conda package for easy installation of examples and their dependencies. Simply `conda install -c conda-forge openff-toolkit-examples` and then run the `openff-toolkit-examples` script to copy the examples suite to a convenient place to run them!

3.8.5 Tests updated

- PR #963: Several tests modules used functions from test_forcefield.py that created an OpenFF Molecule without a toolkit. These functions are now in their own module so they can be imported directly, without the overhead of going through test_forcefield.
- PR #997: Several XML snippets in `test_forcefield.py` that were scattered around inside of classes and functions are now moved to the module level.

3.9 0.9.2 Minor feature and bugfix release

3.9.1 New features and behaviors changed

- PR #762: `Molecule.from_rdkit` now converts implicit hydrogens into explicit hydrogens by default. This change may affect `RDKitToolkitWrapper/Molecule.from_smiles`, `from_mapped_smiles`, `from_file`, `from_file_obj`, `from_inchi`, and `from_qcschema`. This new behavior can be disabled using the `hydrogens_are_explicit=True` keyword argument to `from_smiles`, or loading the molecule into the desired explicit protonation state in RDKit, then calling `from_rdkit` on the RDKit molecule with `hydrogens_are_explicit=True`.
- PR #894: Calls to `Molecule.from_openeye`, `Molecule.from_rdkit`, `Molecule.from_smiles`, `OpenEyeToolkitWrapper.from_smiles`, and `RDKitToolkitWrapper.from_smiles` will now load atom maps into the the resulting `Molecule's offmol.properties['atom_map']` field, even if not all atoms have map indices assigned.
- PR #904: `TopologyAtom` objects now have an element getter `TopologyAtom.element`.

3.9.2 Bugfixes

- PR #891: Calls to `Molecule/OpenEyeToolkitWrapper.from_openeye` no longer mutate the input OE molecule.
- PR #897: Fixes enumeration of stereoisomers for molecules with already defined stereochemistry using `RDKitToolkitWrapper.enumerate_stereoisomers`.
- PR #859: Makes `RDKitToolkitWrapper.enumerate_tautomers` actually use the `max_states` keyword argument during tautomer generation, which will reduce resource use in some cases.

3.9.3 Improved documentation and warnings

- PR #862: Clarify that `System` objects produced by the toolkit are OpenMM `Systems` in anticipation of forthcoming OpenFF `Systems`. Fixes Issue #618.
- PR #863: Documented how to build the docs in the developers guide.
- PR #870: Reorganised documentation to improve discoverability and allow future additions.
- PR #871: Changed Markdown parser from m2r2 to MyST for improved documentation rendering.
- PR #880: Cleanup and partial rewrite of the developer's guide.
- PR #906: Cleaner instructions on how to setup development environment.

3.10 Earlier releases

3.10.1 0.9.1 - Minor feature and bugfix release

New features

- PR #839: Add support for computing WBOs from multiple conformers using the AmberTools and OpenEye toolkits, and from ELF10 conformers using the OpenEye toolkit wrapper.
- PR #832: Expose ELF conformer selection through the `Molecule` API via a new `apply_elf_conformer_selection` function.
- PR #831: Expose ELF conformer selection through the OpenEye wrapper.
- PR #790: Fixes Issue #720 where qcschema roundtrip to/from results in an error due to missing cmiles entry in attributes.
- PR #793: Add an initial ELF conformer selection implementation which uses RDKit.
- PR #799: Closes Issue #746 by adding `Molecule.smirnoff_impropers`, `Molecule.amber_impropers`, `TopologyMolecule.smirnoff_impropers`, `TopologyMolecule.amber_impropers`, `Topology.smirnoff_impropers`, and `Topology.amber_impropers`.
- PR #847: Instances of `ParameterAttribute` documentation can now specify their docstrings with the optional `docstring` argument to the `__init__()` method.
- PR #827: The setter for `Topology.box_vectors` now infers box vectors when box lengths are pass as a list of length 3.

Behavior changed

- PR #802: Fixes Issue #408. The 1-4 scaling factor for electrostatic interactions is now properly set by the value specified in the force field. Previously it fell back to a default value of 0.83333. The toolkit may now produce slightly different energies as a result of this change.
- PR #839: The average WBO will now be returned when multiple conformers are provided to `assign_fractional_bond_orders` using `use_conformers`.
- PR #816: Force field file paths are now loaded in a case-insensitive manner.

Bugfixes

- PR #849: Changes `create_openmm_system` so that it no longer uses the conformers on existing reference molecules (if present) to calculate Wiberg bond orders. Instead, new conformers are always generated during parameterization.

Improved documentation and warnings

- PR #838: Corrects spacing of "forcefield" to "force field" throughout documentation. Fixes Issue #112.
- PR #846: Corrects dead links throughout release history. Fixes Issue #835.
- PR #847: Documentation now compiles with far fewer warnings, and in many cases more correctly. Additionally, `ParameterAttribute` documentation no longer appears incorrectly in classes where it is used. Fixes Issue #397.

3.10.2 0.9.0 - Namespace Migration

This release marks the transition from the old `openforcefield` branding over to its new identity as `openff-toolkit`. This change has been made to better represent the role of the toolkit, and highlight its place in the larger Open Force Field (OpenFF) ecosystem.

From version `0.9.0` onwards the toolkit will need to be imported as `import openff.toolkit.XXX` and `from openff.toolkit import XXX`.

API-breaking changes

- PR #803: Migrates `openforcefield` imports to `openff.toolkit`.

3.10.3 0.8.4 - Minor feature and bugfix release

This release is intended to be functionally identical to 0.9.1. The only difference is that it uses the "openforcefield" namespace.

This release is a final patch for the `0.8.X` series of releases of the toolkit, and also marks the last version of the toolkit which will be imported as `import openforcefield.XXX` / `from openforcefield import XXX`. From version `0.9.0` onwards the toolkit will be importable only as `import openff.toolkit.XXX` / `from openff.toolkit import XXX`.

Note This change will also be accompanied by a renaming of the package from `openforcefield` to `openff-toolkit`, so users need not worry about accidentally pulling in a version with changed imports. Users will have to explicitly choose to install the `openff-toolkit` package once released which will contain the breaking import changes.

3.10.4 0.8.3 - Major bugfix release

This release fixes a critical bug in van der Waals parameter assignment.

This release is also a final patch for the `0.8.X` series of releases of the toolkit, and also marks the last version of the toolkit which will be imported as `import openforcefield.XXX` / `from openforcefield import XXX`. From version `0.9.0` onwards the toolkit will be importable only as `import openff.toolkit.XXX` / `from openff.toolkit import XXX`.

Note This change will also be accompanied by a renaming of the package from `openforcefield` to `openff-toolkit`, so users need not worry about accidentally pulling in a version with changed imports. Users will have to explicitly choose to install the `openff-toolkit` package once released which will contain the breaking import changes.

Bugfixes

- PR #808: Fixes Issue #807, which tracks a major bug in the interconversion between a vdW `sigma` and `rmin_half` parameter.

New features

- PR #794: Adds a decorator `@requires_package` that denotes a function requires an optional dependency.
- PR #805: Adds a deprecation warning for the up-coming release of the `openff-toolkit` package and its import breaking changes.

3.10.5 0.8.2 - Bugfix release

WARNING: This release was later found to contain a major bug, Issue #807, **and produces incorrect energies.**

Bugfixes

- PR #786: Fixes Issue #785 where RDKitToolkitWrapper would sometimes expect stereochemistry to be defined for non-stereogenic bonds when loading from SDF.
- PR #786: Fixes an issue where using the `Molecule` copy constructor (`newmol = Molecule(oldmol)`) would result in the copy sharing the same `.properties` dict as the original (as in, changes to the `.properties` dict of the copy would be reflected in the original).
- PR #789: Fixes a regression noted in Issue #788 where creating `vdWHandler.vdWType` or setting `sigma` or `rmin_half` using Quantities represented as strings resulted in an error.

3.10.6 0.8.1 - Bugfix and minor feature release

WARNING: This release was later found to contain a major bug, Issue #807, and produces incorrect energies.

API-breaking changes

- PR #757: Renames `test_forcefields/smirnoff99Frosst.offxml` to `test_forcefields/test_forcefield.offxml` to avoid confusion with any of the ACTUAL released FFs in the smirnoff99Frosst line
- PR #751: Removes the optional `oetools=("oechem", "oequacpac", "oeiupac", "oeomega")` keyword argument from `OpenEyeToolkitWrapper.is_available`, as there are no special behaviors that are accessed in the case of partially-licensed OpenEye backends. The new behavior of this method is the same as if the default value above is always provided.

Behavior Changed

- PR #583: Methods such as `Molecule.from_rdkit` and `Molecule.from_openeye`, which delegate their internal logic to `ToolkitRegistry` functions, now guarantee that they will return an object of the correct type when being called on `Molecule`-derived classes. Previously, running these constructors using subclasses of `FrozenMolecule` would not return an instance of that subclass, but rather just an instance of a `Molecule`.
- PR #753: `ParameterLookupError` is now raised when passing to `ParameterList.index` a SMIRKS pattern not found in the parameter list.

New features

- PR #751: Adds `LicenseError`, a subclass of `ToolkitUnavailableException` which is raised when attempting to add a cheminformatics `ToolkitWrapper` for a toolkit that is installed but unlicensed.
- PR #678: Adds `ForceField.deregister_parameter_handler`.
- PR #730: Adds `Topology.is_periodic`.
- PR #753: Adds `ParameterHandler.__getitem__` to look up individual `ParameterType` objects.

Bugfixes

- PR #745: Fixes bug when serializing molecule with conformers to JSON.
- PR #750: Fixes a bug causing either `sigma` or `rmin_half` to sometimes be missing on `vdWHandler.vdWType` objects.
- PR #756: Fixes bug when running `vdWHandler.create_force` using a `vdWHandler` that was initialized using the API.
- PR #776: Fixes a bug in which the `Topology.from_openmm` and `Topology.from_mdtraj` methods would dangerously allow `unique_molecules=None`.
- PR #777: `RDKitToolkitWrapper` now outputs the full warning message when `allow_undefined_stereo=True` (previously the description of which stereo was undefined was squelched)

3.10.7 0.8.0 - Virtual Sites

Major Feature: Support for the SMIRNOFF VirtualSite tag

This release implements the SMIRNOFF virtual site specification. The implementation enables support for models using off-site charges, including 4- and 5-point water models, in addition to lone pair modeling on various functional groups. The primary focus was on the ability to parameterize a system using virtual sites, and generating an OpenMM system with all virtual sites present and ready for evaluation. Support for formats other than OpenMM has not be implemented in this release, but may come with the appearance of the OpenFF system object. In addition to implementing the specification, the toolkit `Molecule` objects now allow the creation and manipulation of virtual sites.

This change is documented in the Virtual sites page of the user guide.

Minor Feature: Support for the 0.4 ChargeIncrementModel tag

To allow for more convenient fitting of `ChargeIncrement` parameters, it is now possible to specify one less `charge_increment` value than there are tagged atoms in a `ChargeIncrement`'s `smirks`. The missing `charge_increment` value will be calculated at parameterization-time to make the sum of the charge contributions from a `ChargeIncrement` parameter equal to zero. Since this change allows for force fields that are incompatible with the previous specification, this new style of `ChargeIncrement` must specify a `ChargeIncrementModel` section version of `0.4`. All `0.3`-compatible `ChargeIncrement` parameters are compatible with the `0.4` `ChargeIncrementModel` specification.

More details and examples of this change are available in The ChargeIncrementModel tag in the SMIRNOFF specification

New features

- PR #726: Adds support for the 0.4 ChargeIncrementModel spec, allowing for the specification of one fewer `charge_increment` values than there are tagged atoms in the `smirks`, and automatically assigning the final atom an offsetting charge.
- PR #548: Adds support for the `VirtualSites` tag in the SMIRNOFF specification
- PR #548: Adds `replace` and `all_permutations` kwarg to
  - `Molecule.add_bond_charge_virtual_site`
  - `Molecule.add_monovalent_lone_pair_virtual_site`
  - `Molecule.add_divalent_lone_pair_virtual_site`
  - `Molecule.add_trivalent_lone_pair_virtual_site`
- PR #548: Adds `orientations` to
  - `BondChargeVirtualSite`
  - `MonovalentLonePairVirtualSite`
  - `DivalentLonePairVirtualSite`
  - `TrivalentLonePairVirtualSite`
- PR #548: Adds
  - `VirtualParticle`
  - `TopologyVirtualParticle`
  - `BondChargeVirtualSite.get_openmm_virtual_site`
  - `MonovalentLonePairVirtualSite.get_openmm_virtual_site`

  - DivalentLonePairVirtualSite.get_openmm_virtual_site
  - TrivalentLonePairVirtualSite.get_openmm_virtual_site
  - ValenceDict.key_transform
  - ValenceDict.index_of
  - ImproperDict.key_transform
  - ImproperDict.index_of
- PR #705: Adds interpolation based on fractional bond orders for harmonic bonds. This includes interpolation for both the force constant k and/or equilibrium bond distance length. This is accompanied by a bump in the <Bonds> section of the SMIRNOFF spec (but not the entire spec).
- PR #718: Adds .rings and .n_rings to Molecule and .is_in_ring to Atom and Bond

Bugfixes

- PR #682: Catches failures in Molecule.from_iupac instead of silently failing.
- PR #743: Prevents the non-bonded (vdW) cutoff from silently falling back to the OpenMM default of 1 nm in Forcefield.create_openmm_system and instead sets its to the value specified by the force field.
- PR #737: Prevents OpenEye from incidentally being used in the conformer generation step of AmberToolsToolkitWrapper.assign_fractional_bond_orders.

Behavior changed

- PR #705: Changes the default values in the <Bonds> section of the SMIRNOFF spec to fractional_bondorder_method="AM1-Wiberg" and potential="(k/2)*(r-length)^2", which is backwards-compatible with and equivalent to potential="harmonic".

Examples added

- PR #548: Adds a virtual site example notebook to run an OpenMM simulation with virtual sites, and compares positions and potential energy of TIP5P water between OpenFF and OpenMM force fields.

API-breaking changes

- PR #548: Methods
  - Molecule.add_bond_charge_virtual_site
  - Molecule.add_monovalent_lone_pair_virtual_site
  - Molecule.add_divalent_lone_pair_virtual_site
  - Molecule.add_trivalent_lone_pair_virtual_site now only accept a list of atoms, not a list of integers, to define to parent atoms
- PR #548: Removes VirtualParticle.molecule_particle_index
- PR #548: Removes outOfPlaneAngle from
  - DivalentLonePairVirtualSite
  - TrivalentLonePairVirtualSite

- PR #548: Removes `inPlaneAngle` from `TrivalentLonePairVirtualSite`
- PR #548: Removes `weights` from
  - `BondChargeVirtualSite`
  - `MonovalentLonePairVirtualSite`
  - `DivalentLonePairVirtualSite`
  - `TrivalentLonePairVirtualSite`

Tests added

- PR #548: Adds test for
  - The virtual site parameter handler
  - TIP5P water dimer energy and positions
  - Adds tests to for virtual site/particle indexing/counting

3.10.8 0.7.2 - Bugfix and minor feature release

New features

- PR #662: Adds `.aromaticity_model` of `ForceField` and `.TAGNAME` of `ParameterHandler` as public attributes.
- PR #667 and PR #681 linted the codebase with `black` and `isort`, respectively.
- PR #675 adds `.toolkit_version` to `ToolkitWrapper` and `.registered_toolkit_versions` to `ToolkitRegistry`.
- PR #696 Exposes a setter for `ForceField.aromaticity_model`
- PR #685 Adds a custom `__hash__` function to `ForceField`

Behavior changed

- PR #684: Changes `ToolkitRegistry` to return an empty registry when initialized with no arguments, i.e. `ToolkitRegistry()` and makes the `register_imported_toolkit_wrappers` argument private.
- PR #711: The setter for `Topology.box_vectors` now infers box vectors (a 3x3 matrix) when box lengths (a 3x1 array) are passed, assuming an orthogonal box.
- PR #649: Makes SMARTS searches stereochemistry-specific (if stereo is specified in the SMARTS) for both OpenEye and RDKit backends. Also ensures molecule aromaticity is re-perceived according to the ForceField's specified aromaticity model, which may overwrite user-specified aromaticity on the `Molecule`
- PR #648: Removes the `utils.structure` module, which was deprecated in 0.2.0.
- PR #670: Makes the `Topology` returned by `create_openmm_system` contain the partial charges and partial bond orders (if any) assigned during parameterization.
- PR #675 changes the exception raised when no `antechamber` executable is found from `IOError` to `AntechamberNotFoundError`
- PR #696 Adds an `aromaticity_model` keyword argument to the `ForceField` constructor, which defaults to `DEFAULT_AROMATICITY_MODEL`.

Bugfixes

- PR #715: Closes issue Issue #475 writing a "PDB" file using OE backend rearranges the order of the atoms by pushing the hydrogens to the bottom.
- PR #649: Prevents 2020 OE toolkit from issuing a warning caused by doing stereo-specific smarts searches on certain structures.
- PR #724: Closes issue Issue #502 Adding a utility function Topology.to_file() to write topology and positions to a "PDB" file using openmm backend for pdb file write.

Tests added

- PR #694: Adds automated testing to code snippets in docs.
- PR #715: Adds tests for pdb file writes using OE backend.
- PR #724: Adds tests for the utility function Topology.to_file().

3.10.9 0.7.1 - OETK2020 Compatibility and Minor Update

This is the first of our patch releases on our new planned monthly release schedule.

Detailed release notes are below, but the major new features of this release are updates for compatibility with the new 2020 OpenEye Toolkits release, the `get_available_force_fields` function, and the disregarding of pyrimidal nitrogen stereochemistry in molecule isomorphism checks.

Behavior changed

- PR #646: Checking for `Molecule` equality using the `==` operator now disregards all pyrimidal nitrogen stereochemistry by default. To re-enable, use `Molecule.{is|are}_isomorphic` with the `strip_pyrimidal_n_atom_stereo=False` keyword argument.
- PR #646: Adds an optional `toolkit_registry` keyword argument to `Molecule.are_isomorphic`, which identifies the toolkit that should be used to search for pyrimidal nitrogens.

Bugfixes

- PR #647: Updates `OpenEyeToolkitWrapper` for 2020.0.4 OpenEye Toolkit behavior/API changes.
- PR #646: Fixes a bug where `Molecule.chemical_environment_matches` was not able to accept a `ChemicalEnvironment` object as a query.
- PR #634: Fixes a bug in which calling `RDKitToolkitWrapper.from_file` directly would not load files correctly if passed lowercase `file_format`. Note that this bug did not occur when calling `Molecule.from_file`.
- PR #631: Fixes a bug in which calling `unit_to_string` returned `None` when the unit is dimensionless. Now `"dimensionless"` is returned.
- PR #630: Closes issue Issue #629 in which the wrong exception is raised when attempting to instantiate a `ForceField` from an unparsable string.

New features

- PR #632: Adds `ForceField.registered_parameter_handlers`
- PR #614: Adds `ToolkitRegistry.deregister_toolkit` to de-register registered toolkits, which can include toolkit wrappers loaded into `GLOBAL_TOOLKIT_REGISTRY` by default.
- PR #656: Adds a new allowed `am1elf10` option to the OpenEye implementation of `assign_partial_charges` which calculates the average partial charges at the AM1 level of theory using conformers selected using the ELF10 method.
- PR #643: Adds `openforcefield.typing.engines.smirnoff.forcefield.get_available_force_fields`, which returns paths to the files of force fields available through entry point plugins.

3.10.10 0.7.0 - Charge Increment Model, Proper Torsion interpolation, and new Molecule methods

This is a relatively large release, motivated by the idea that changing existing functionality is bad so we shouldn't do it too often, but when we do change things we should do it all at once.

Here's a brief rundown of what changed, migration tips, and how to find more details in the full release notes below:

- To provide more consistent partial charges for a given molecule, existing conformers are now disregarded by default by `Molecule.assign_partial_charges`. Instead, new conformers are generated for use in semiempirical calculations. Search for `use_conformers`.
- Formal charges are now always returned as `simtk.unit.Quantity` objects, with units of elementary charge. To convert them to integers, use `from simtk import unit` and `atom.formal_charge.value_in_unit(unit.elementary_charge)` or `mol.total_charge.value_in_unit(unit.elementary_charge)`. Search `atom.formal_charge`.
- The OpenFF Toolkit now automatically reads and writes partial charges in SDF files. Search for `atom.dprop.PartialCharges`.
- The OpenFF Toolkit now has different behavior for handling multi-molecule and multi-conformer SDF files. Search `multi-conformer`.
- The OpenFF Toolkit now distinguishes between partial charges that are all-zero and partial charges that are unknown. Search `partial_charges = None`.
- `Topology.to_openmm` now assigns unique atoms names by default. Search `ensure_unique_atom_names`.
- Molecule equality checks are now done by graph comparison instead of SMILES comparison. Search `Molecule.are_isomorphic`.
- The `ChemicalEnvironment` module was almost entirely removed, as it is an outdated duplicate of some Chemper functionality. Search `ChemicalEnvironment`.
- `TopologyMolecule.topology_particle_start_index` has been removed from the `TopologyMolecule` API, since atoms and virtualsites are no longer contiguous in the `Topology` particle indexing system. Search `topology_particle_start_index`.
- `compute_wiberg_bond_orders` has been renamed to `assign_fractional_bond_orders`.

There are also a number of new features, such as:

- Support for `ChargeIncrementModel` sections in force fields.
- Support for `ProperTorsion` `k` interpolation in force fields using fractional bond orders.

- Support for AM1-Mulliken, Gasteiger, and other charge methods using the new `assign_partial_charges` methods.
- Support for AM1-Wiberg bond order calculation using either the OpenEye or RDKit/AmberTools backends and the `assign_fractional_bond_orders` methods.
- Initial (limited) interoperability with QCArchive, via `Molecule.to_qcschema` and `from_qcschema`.
- A `Molecule.visualize` method.
- Several additional `Molecule` methods, including state enumeration and mapped SMILES creation.

Major Feature: Support for the SMIRNOFF ChargeIncrementModel tag

The ChargeIncrementModel tag in the SMIRNOFF specification provides analagous functionality to AM1-BCC, except that instead of AM1-Mulliken charges, a number of different charge methods can be called, and instead of a fixed library of two-atom charge corrections, an arbitrary number of SMIRKS-based, N-atom charge corrections can be defined in the SMIRNOFF format.

The initial implementation of the SMIRNOFF `ChargeIncrementModel` tag accepts keywords for `version`, `partial_charge_method`, and `number_of_conformers`. `partial_charge_method` can be any string, and it is up to the `ToolkitWrapper`'s `compute_partial_charges` methods to understand what they mean. For geometry-independent `partial_charge_method` choices, `number_of_conformers` should be set to zero.

SMIRKS-based parameter application for `ChargeIncrement` parameters is different than other SMIRNOFF sections. The initial implementation of `ChargeIncrementModelHandler` follows these rules:

- an atom can be subject to many `ChargeIncrement` parameters, which combine additively.
- a `ChargeIncrement` that matches a set of atoms is overwritten only if another `ChargeIncrement` matches the same group of atoms, regardless of order. This overriding follows the normal SMIRNOFF hierarchy.

To give a concise example, what if a molecule `A-B(-C)-D` were being parametrized, and the force field defined `ChargeIncrement` SMIRKS in the following order?

1) `[A:1]-[B:2]`
2) `[B:1]-[A:2]`
3) `[A:1]-[B:2]-[C:3]`
4) `[*:1]-[B:2](-[*:3])-[*:4]`
5) `[D:1]-[B:2](-[*:3])-[*:4]`

In the case above, the ChargeIncrement from parameters 1 and 4 would NOT be applied to the molecule, since another parameter matching the same set of atoms is specified further down in the parameter hierarchy (despite those subsequent matches being in a different order).

Ultimately, the ChargeIncrement contributions from parameters 2, 3, and 5 would be summed and applied.

It's also important to identify a behavior that these rules were written to *avoid*: if not for the "regardless of order" clause in the second rule, parameters 4 and 5 could actually have been applied six and two times, respectively (due to symmetry in the SMIRKS and the use of wildcards). This situation could also arise as a result of molecular symmetry. For example, a methyl group could match the SMIRKS `[C:1]([H:2])([H:3])([H:4])` six ways (with different orderings of the three hydrogen atoms), but the user would almost certainly not intend for the charge increments to be applied six times. The "regardless of order" clause was added specifically to address this.

In short, the first time a group of atoms becomes involved in a `ChargeIncrement` together, the OpenMM `System` gains a new parameter "slot". Only another `ChargeIncrement` which applies to the exact same group of atoms (in any order) can take over the "slot", pushing the original `ChargeIncrement` out.

Major Feature: Support for ProperTorsion k value interpolation

Chaya Stern's work showed that we may be able to produce higher-quality proper torsion parameters by taking into account the "partial bond order" of the torsion's central bond. We now have the machinery to compute AM1-Wiberg partial bond orders for entire molecules using the `assign_fractional_bond_orders` methods of either `OpenEyeToolkitWrapper` or `AmberToolsToolkitWrapper`. The thought is that, if some simple electron population analysis shows that a certain aromatic bond's order is 1.53, maybe rotations about that bond can be described well by interpolating 53% of the way between the single and double bond k values.

Full details of how to define a torsion-interpolating SMIRNOFF force fields are available in the ProperTorsions section of the SMIRNOFF specification.

Behavior changed

- PR #508: In order to provide the same results for the same chemical species, regardless of input conformation, `Molecule` `assign_partial_charges`, `compute_partial_charges_am1bcc`, and `assign_fractional_bond_orders` methods now default to ignore input conformers and generate new conformer(s) of the molecule before running semiempirical calculations. Users can override this behavior by specifying the keyword argument `use_conformers=molecule.conformers`.
- PR #281: Closes Issue #250 by adding support for partial charge I/O in SDF. The partial charges are stored as a property in the SDF molecule block under the tag `<atom.dprop.PartialCharge>`.
- PR #281: If a `Molecule`'s `partial_charges` attribute is set to `None` (the default value), calling `to_openeye` will now produce a OE molecule with partial charges set to `nan`. This would previously produce an OE molecule with partial charges of 0.0, which was a loss of information, since it wouldn't be clear whether the original OFFMol's partial charges were REALLY all-zero as opposed to `None`. OpenEye toolkit wrapper methods such as `from_smiles` and `from_file` now produce OFFMols with `partial_charges = None` when appropriate (previously these would produce OFFMols with all-zero charges, for the same reasoning as above).
- PR #281: `Molecule` `to_rdkit` now sets partial charges on the RDAtom's `PartialCharges` property (this was previously set on the `partial_charges` property). If the `Molecule`'s partial_charges attribute is `None`, this property will not be defined on the RDAtoms.
- PR #281: Enforce the behavior during SDF I/O that a SDF may contain multiple *molecules*, but that the OFF Toolkit does not assume that it contains multiple *conformers of the same molecule*. This is an important distinction, since otherwise there is ambiguity around whether properties of one entry in a SDF are shared among several molecule blocks or not, or how to resolve conflicts if properties are defined differently for several "conformers" of chemically-identical species (More info here). If the user requests the OFF Toolkit to write a multi-conformer `Molecule` to SDF, only the first conformer will be written. For more fine-grained control of writing properties, conformers, and partial charges, consider using `Molecule.to_rdkit` or `Molecule.to_openeye` and using the functionality offered by those packages.
- PR #281: Due to different constraints placed on the data types allowed by external toolkits, we make our best effort to preserve `Molecule` `properties` when converting molecules to other packages, but users should be aware that no guarantee of data integrity is made. The only data format for keys and values in the property dict that we will try to support through a roundtrip to another toolkit's Molecule object is `string`.
- PR #574: Removed check that all partial charges are zero after assignment by `quacpac` when AM1BCC used for charge assignment. This check fails erroneously for cases in which the partial charge assignments are correctly all zero, such as for `N#N`. It is also an unnecessary check given that `quacpac` will reliably indicate when it has failed to assign charges.
- PR #597: Energy-minimized sample systems with Parsley 1.1.0.
- PR #558: The `Topology` particle indexing system now orders `TopologyVirtualSites` after all atoms.

- PR #469: When running `Topology.to_openmm`, unique atom names are generated if the provided atom names are not unique (overriding any existing atom names). This uniqueness extends only to atoms in the same molecule. To disable this behavior, set the kwarg `ensure_unique_atom_names=False`.
- PR #472: `Molecule.__eq__` now uses the new `Molecule.are_isomorphic` to perform the similarity checking.
- PR #472: The `Topology.from_openmm` and `Topology.add_molecule` methods now use the `Molecule.are_isomorphic` method to match molecules.
- PR #551: Implemented the `ParameterHandler.get_parameter` function (would previously return `None`).

API-breaking changes

- PR #471: Closes Issue #465. `atom.formal_charge` and `molecule.total_charge` now return `simtk.unit.Quantity` objects instead of integers. To preserve backward compatibility, the setter for `atom.formal_charge` can accept either a `simtk.unit.Quantity` or an integer.
- PR #601: Removes almost all of the previous `ChemicalEnvironment` API, since this entire module was simply copied from Chemper several years ago and has fallen behind on updates. Currently only `ChemicalEnvironment.get_type`, `ChemicalEnvironment.validate`, and an equivalent classmethod `ChemicalEnvironment.validate_smirks` remain. Also, please comment on this GitHub issue if you HAVE been using the previous extra functionality in this module and would like us to prioritize creation of a Chemper conda package.
- PR #558: Removes `TopologyMolecule.topology_particle_start_index`, since the `Topology` particle indexing system now orders `TopologyVirtualSites` after all atoms. `TopologyMolecule.atom_start_topology_index` and `TopologyMolecule.virtual_particle_start_topology_index` are still available to access the appropriate values in the respective topology indexing systems.
- PR #508: `OpenEyeToolkitWrapper.compute_wiberg_bond_orders` is now `OpenEyeToolkitWrapper.assign_fractional_bond_orders`. The `charge_model` keyword is now `bond_order_model`. The allowed values of this keyword have changed from `am1` and `pm3` to `am1-wiberg` and `pm3-wiberg`, respectively.
- PR #508: `Molecule.compute_wiberg_bond_orders` is now `Molecule.assign_fractional_bond_orders`.
- PR #595: Removed functions `openforcefield.utils.utils.temporary_directory` and `openforcefield.utils.utils.temporary_cd` and replaced their behavior with `tempfile.TemporaryDirectory()`.

New features

- PR #471: Closes Issue #208 by implementing support for the `ChargeIncrementModel` tag in the SMIRNOFF specification.
- PR #471: Implements `Molecule.assign_partial_charges`, which calls one of the newly-implemented `OpenEyeToolkitWrapper.assign_partial_charges`, and `AmberToolsToolkitWrapper.assign_partial_charges`. `strict_n_conformers` is a optional boolean keyword argument indicating whether an `IncorrectNumConformersError` should be raised if an invalid number of conformers is supplied during partial charge calculation. For example, if two conformers are supplied, but `partial_charge_method="AM1BCC"` is also set, then there is no clear use for the second conformer. The previous behavior in this case was to raise a warning, and to preserve that behavior, `strict_n_conformers` defaults to a value of `False`.
- PR #471: Adds keyword argument `raise_exception_types` (default: `[Exception]`) to `ToolkitRegistry.call`. The default value will provide the previous OpenFF Toolkit behavior, which is that the first ToolkitWrapper that can provide the requested method is called, and it either returns

on success or raises an exception. This new keyword argument allows the ToolkitRegistry to *ignore* certain exceptions, but treat others as fatal. If `raise_exception_types = []`, the ToolkitRegistry will attempt to call each ToolkitWrapper that provides the requested method and if none succeeds, a single `ValueError` will be raised, with text listing the errors that were raised by each ToolkitWrapper.

- PR #601: Adds `RDKitToolkitWrapper.get_tagged_smarts_connectivity` and `OpenEyeToolkitWrapper.get_tagged_smarts_connectivity`, which allow the use of either toolkit for smirks/tagged smarts validation.
- PR #600: Adds `ForceField.__getitem__` to look up `ParameterHandler` objects based on their string names.
- PR #508: Adds `AmberToolsToolkitWrapper.assign_fractional_bond_orders`.
- PR #469: The `Molecule` class adds `Molecule.has_unique_atom_names` and `Molecule.has_unique_atom_names`.
- PR #472: Adds to the `Molecule` class `Molecule.are_isomorphic` and `Molecule.is_isomorphic_with` and `Molecule.hill_formula` and `Molecule.to_hill_formula` and `Molecule.to_qcschema` and `Molecule.from_qcschema` and `Molecule.from_mapped_smiles` and `Molecule.from_pdb_and_smiles` and `Molecule.canonical_order_atoms` and `Molecule.remap`

  Note: The to_qcschema method accepts an extras dictionary which is passed into the validated qcelemental.models.Molecule object.

- PR #506: The `Molecule` class adds `Molecule.find_rotatable_bonds`
- PR #521: Adds `Molecule.to_inchi` and `Molecule.to_inchikey` and `Molecule.from_inchi`

  Warning: InChI was not designed as an molecule interchange format and using it as one is not recommended. Many round trip tests will fail when using this format due to a loss of information. We have also added support for fixed hydrogen layer nonstandard InChI which can help in the case of tautomers, but overall creating molecules from InChI should be avoided.

- PR #529: Adds the ability to write out to XYZ files via `Molecule.to_file` Both single frame and multiframe XYZ files are supported. Note reading from XYZ files will not be supported due to the lack of connectivity information.
- PR #535: Extends the the API for the `Molecule.to_smiles` to allow for the creation of cmiles identifiers through combinations of isomeric, explicit hydrogen and mapped smiles, the default settings will return isomeric explicit hydrogen smiles as expected.

  Warning: Atom maps can be supplied to the properties dictionary to modify which atoms have their map index included, if no map is supplied all atoms will be mapped in the order they appear in the `Molecule`.

- PR #563: Adds `test_forcefields/ion_charges.offxml`, giving `LibraryCharges` for monatomic ions.
- PR #543: Adds 3 new methods to the `Molecule` class which allow the enumeration of molecule states. These are `Molecule.enumerate_tautomers`, `Molecule.enumerate_stereoisomers`, `Molecule.enumerate_protomers`

> **Warning:** Enumerate protomers is currently only available through the OpenEye toolkit.

- PR #573: Adds quacpac error output to quacpac failure in `Molecule.compute_partial_charges_am1bcc`.
- PR #560: Added visualization method to the the Molecule class.
- PR #620: Added the ability to register parameter handlers via entry point plugins. This functionality is accessible by initializing a `ForceField` with the `load_plugins=True` keyword argument.
- PR #582: Added fractional bond order interpolation Adds *return_topology* kwarg to `Forcefield.create_openmm_system`, which returns the processed topology along with the OpenMM `System` when `True` (default `False`).

Tests added

- PR #558: Adds tests ensuring that the new Topology particle indexing system are properly implemented, and that TopologyVirtualSites reference the correct TopologyAtoms.
- PR #469: Added round-trip SMILES test to add coverage for `Molecule.from_smiles`.
- PR #469: Added tests for unique atom naming behavior in `Topology.to_openmm`, as well as tests of the `ensure_unique_atom_names=False` kwarg disabling this behavior.
- PR #472: Added tests for `Molecule.hill_formula` and `Molecule.to_hill_formula` for the various supported input types.
- PR #472: Added round-trip test for `Molecule.from_qcschema` and `Molecule.to_qcschema`.
- PR #472: Added tests for `Molecule.is_isomorphic_with` and `Molecule.are_isomorphic` with various levels of isomorphic graph matching.
- PR #472: Added toolkit dependent tests for `Molecule.canonical_order_atoms` due to differences in the algorithms used.
- PR #472: Added a test for `Molecule.from_mapped_smiles` using the molecule from issue #412 to ensure it is now fixed.
- PR #472: Added a test for `Molecule.remap`, this also checks for expected error when the mapping is not complete.
- PR #472: Added tests for `Molecule.from_pdb_and_smiles` to check for a correct combination of smiles and PDB and incorrect combinations.
- PR #509: Added test for `Molecule.chemical_environment_matches` to check that the complete set of matches is returned.
- PR #509: Added test for `Forcefield.create_openmm_system` to check that a protein system can be created.
- PR #506: Added a test for the molecule identified in issue #513 as losing aromaticity when converted to rdkit.
- PR #506: Added a verity of toolkit dependent tests for identifying rotatable bonds while ignoring the user requested types.
- PR #521: Added toolkit independent round-trip InChI tests which add coverage for `Molecule.to_inchi` and `Molecule.from_inchi`. Also added coverage for bad inputs and `Molecule.to_inchikey`.
- PR #529: Added to XYZ file coverage tests.

- PR #563: Added *LibraryCharges* parameterization test for monatomic ions in `test_forcefields/ion_charges.offxml`.
- PR #543: Added tests to assure that state enumeration can correctly find molecules tautomers, stereoisomers and protomers when possible.
- PR #573: Added test for `quacpac` error output for `quacpac` failure in `Molecule.compute_partial_charges_am1bcc`.
- PR #579: Adds regression tests to ensure RDKit can be be used to write multi-model PDB files.
- PR #582: Added fractional bond order interpolation tests, tests for `ValidatedDict`.

Bugfixes

- PR #558: Fixes a bug where `TopologyVirtualSite.atoms` would not correctly apply `TopologyMolecule` atom ordering on top of the reference molecule ordering, in cases where the same molecule appears multiple times, but in a different order, in the same Topology.
- Issue #460: Creates unique atom names in `Topology.to_openmm` if the existing ones are not unique. The lack of unique atom names had been causing problems in workflows involving downstream tools that expect unique atom names.
- Issue #448: We can now make molecules from mapped smiles using `Molecule.from_mapped_smiles` where the order will correspond to the indeing used in the smiles. Molecules can also be re-indexed at any time using the `Molecule.remap`.
- Issue #462: We can now instance the `Molecule` from a QCArchive entry record instance or dictionary representation.
- Issue #412: We can now instance the `Molecule` using `Molecule.from_mapped_smiles`. This resolves an issue caused by RDKit considering atom map indices to be a distinguishing feature of an atom, which led to erroneous definition of chirality (as otherwise symmetric substituents would be seen as different). We anticipate that this will reduce the number of times you need to type `allow_undefined_stereo=True` when processing molecules that do not actually contain stereochemistrty.
- Issue #513: The `Molecule.to_rdkit` now re-sets the aromaticity model after sanitizing the molecule.
- Issue #500: The `Molecule.find_rotatable_bonds` has been added which returns a list of rotatable `Bond` instances for the molecule.
- Issue #491: We can now parse large molecules without hitting a match limit cap.
- Issue #474: We can now convert molecules to InChI and InChIKey and from InChI.
- Issue #523: The `Molecule.to_file` method can now correctly write to `MOL` files, in line with the supported file type list.
- Issue #568: The `Molecule.to_file` can now correctly write multi-model PDB files when using the RDKit backend toolkit.

Examples added

- PR #591 and PR #533: Adds an example notebook and utility to compute conformer energies. This example is made to be reverse-compatible with the 0.6.0 OpenFF Toolkit release.
- PR #472: Adds an example notebook QCarchive_interface.ipynb which shows users how to instance the `Molecule` from a QCArchive entry level record and calculate the energy using RDKit through QCEngine.

3.10.11 0.6.0 - Library Charges

This release adds support for a new SMIRKS-based charge assignment method, Library Charges. The addition of more charge assignment methods opens the door for new types of experimentation, but also introduces several complex behaviors and failure modes. Accordingly, we have made changes to the charge assignment infrastructure to check for cases when partial charges do not sum to the formal charge of the molecule, or when no charge assignment method is able to generate charges for a molecule. More detailed explanation of the new errors that may be raised and keywords for overriding them are in the "Behavior Changed" section below.

With this release, we update `test_forcefields/tip3p.offxml` to be a working example of assigning LibraryCharges. However, we do not provide any force field files to assign protein residue `LibraryCharges`. If you are interested in translating an existing protein FF to SMIRNOFF format or developing a new one, please feel free to contact us on the Issue tracker or open a Pull Request.

New features

- PR #433: Closes Issue #25 by adding initial support for the LibraryCharges tag in the SMIRNOFF specification using `LibraryChargeHandler`. For a molecule to have charges assigned using Library Charges, all of its atoms must be covered by at least one `LibraryCharge`. If an atom is covered by multiple `LibraryCharge` s, then the last `LibraryCharge` matched will be applied (per the hierarchy rules in the SMIRNOFF format).

  This functionality is thus able to apply per-residue charges similar to those in traditional protein force fields. At this time, there is no concept of "residues" or "fragments" during parametrization, so it is not possible to assign charges to *some* atoms in a molecule using `LibraryCharge` s, but calculate charges for other atoms in the same molecule using a different method. To assign charges to a protein, LibraryCharges SMARTS must be provided for the residues and protonation states in the molecule, as well as for any capping groups and post-translational modifications that are present.

  It is valid for `LibraryCharge` SMARTS to *partially* overlap one another. For example, a molecule consisting of atoms `A-B-C` connected by single bonds could be matched by a SMIRNOFF `LibraryCharges` section containing two `LibraryCharge` SMARTS: `A-B` and `B-C`. If listed in that order, the molecule would be assigned the `A` charge from the `A-B` `LibraryCharge` element and the `B` and `C` charges from the `B-C` element. In testing, these types of partial overlaps were found to frequently be sources of undesired behavior, so it is recommended that users define whole-molecule `LibraryCharge` SMARTS whenever possible.
- PR #455: Addresses Issue #393 by adding `ParameterHandler.attribute_is_cosmetic` and `ParameterType.attribute_is_cosmetic`, which return True if the provided attribute name is defined for the queried object but does not correspond to an allowed value in the SMIRNOFF spec.

Behavior changed

- PR #433: If a molecule can not be assigned charges by any charge-assignment method, an `openforcefield.typing.engines.smirnoff.parameters.UnassignedMoleculeChargeException` will be raised. Previously, creating a system without either `ToolkitAM1BCCHandler` or the `charge_from_molecules` keyword argument to `ForceField.create_openmm_system` would produce an OpenMM `System` where the molecule has zero charge on all atoms. However, given that we will soon be adding more options for charge assignment, it is important that failures not be silent. Molecules with zero charge can still be produced by setting the `Molecule.partial_charges` array to be all zeroes, and including the molecule in the `charge_from_molecules` keyword argument to `create_openmm_system`.
- PR #433: Due to risks introduced by permitting charge assignment using partially-overlapping `LibraryCharge` s, the toolkit will now raise a `openforcefield.typing.engines.smirnoff.parameters.NonIntegralMoleculeChargeException` if the sum of partial charges on a molecule are found to be more than 0.01 elementary charge units different than the molecule's formal charge. This exception can be overridden by providing the `allow_nonintegral_charges=True` keyword argument to `ForceField.create_openmm_system`.

Tests added

- PR #430: Added test for Wiberg Bond Order implemented in OpenEye Toolkits. Molecules taken from DOI:10.5281/zenodo.3405489 . Added by Sukanya Sasmal.
- PR #569: Added round-trip tests for more serialization formats (dict, YAML, TOML, JSON, BSON, messagepack, pickle). Note that some are unsupported, but the tests raise the appropriate error.

Bugfixes

- PR #431: Fixes an issue where `ToolkitWrapper` objects would improperly search for functionality in the `GLOBAL_TOOLKIT_REGISTRY`, even though a specific `ToolkitRegistry` was requested for an operation.
- PR #439: Fixes Issue #438, by replacing call to NetworkX `Graph.node` with call to `Graph.nodes`, per 2.4 migration guide.

Files modified

- PR #433: Updates the previously-nonfunctional `test_forcefields/tip3p.offxml` to a functional state by updating it to the SMIRNOFF 0.3 specification, and specifying atomic charges using the `LibraryCharges` tag.

3.10.12 0.5.1 - Adding the parameter coverage example notebook

This release contains a new notebook example, check_parameter_coverage.ipynb, which loads sets of molecules, checks whether they are parameterizable, and generates reports of chemical motifs that are not. It also fixes several simple issues, improves warnings and docstring text, and removes unused files.

The parameter coverage example notebook goes hand-in-hand with the release candidate of our initial force field, openff-1.0.0-RC1.offxml , which will be temporarily available until the official force field release is made in October. Our goal in publishing this notebook alongside our first major refitting is to allow interested users to check whether there is parameter coverage for their molecules of interest. If the force field is unable to parameterize a molecule, this notebook will generate reports of the specific chemistry that is not covered. We understand that many organizations in our field have restrictions about sharing specific molecules, and

the outputs from this notebook can easily be cropped to communicate unparameterizable chemistry without revealing the full structure.

The force field release candidate is in our new refit force field package, openforcefields. This package is now a part of the Open Force Field Toolkit conda recipe, along with the original smirnoff99Frosst line of force fields.

Once the `openforcefields` conda package is installed, you can load the release candidate using:

```
ff = ForceField('openff-1.0.0-RC1.offxml')
```

The release candidate will be removed when the official force field, `openff-1.0.0.offxml`, is released in early October.

Complete details about this release are below.

Example added

- PR #419: Adds an example notebook check_parameter_coverage.ipynb which shows how to use the toolkit to check a molecule dataset for missing parameter coverage, and provides functionality to output tagged SMILES and 2D drawings of the unparameterizable chemistry.

New features

- PR #419: Unassigned valence parameter exceptions now include a list of tuples of `TopologyAtom` which were unable to be parameterized (`exception.unassigned_topology_atom_tuples`) and the class of the `ParameterHandler` that raised the exception (`exception.handler_class`).
- PR #425: Implements Trevor Gokey's suggestion from Issue #411, which enables pickling of `ForceFields` and `ParameterHandlers`. Note that, while XML representations of `ForceFields` are stable and conform to the SMIRNOFF specification, the pickled `ForceFields` that this functionality enables are not guaranteed to be compatible with future toolkit versions.

Improved documentation and warnings

- PR #425: Addresses Issue #410, by explicitly having toolkit warnings print `Warning:` at the beginning of each warning, and adding clearer language to the warning produced when the OpenEye Toolkits can not be loaded.
- PR #425: Addresses Issue #421 by adding type/shape information to all Molecule partial charge and conformer docstrings.
- PR #425: Addresses Issue #407 by providing a more extensive explanation of why we don't use RDKit's mol2 parser for molecule input.

Bugfixes

- PR #419: Fixes Issue #417 and Issue #418, where `RDKitToolkitWrapper.from_file` would disregard the `allow_undefined_stereo` kwarg and skip the first molecule when reading a SMILES file.

Files removed

- PR #425: Addresses Issue #424 by deleting the unused files `openforcefield/typing/engines/smirnoff/gbsaforces.py` and `openforcefield/tests/test_smirnoff.py`. `gbsaforces.py` was only used internally and `test_smirnoff.py` tested unsupported functionality from before the 0.2.0 release.

3.10.13 0.5.0 - GBSA support and quality-of-life improvements

This release adds support for the GBSA tag in the SMIRNOFF specification. Currently, the `HCT`, `OBC1`, and `OBC2` models (corresponding to AMBER keywords `igb=1`, `2`, and `5`, respectively) are supported, with the `OBC2` implementation being the most flexible. Unfortunately, systems produced using these keywords are not yet transferable to other simulation packages via ParmEd, so users are restricted to using OpenMM to simulate systems with GBSA.

OFFXML files containing GBSA parameter definitions are available, and can be loaded in addition to existing parameter sets (for example, with the command `ForceField('test_forcefields/smirnoff99Frosst.offxml', 'test_forcefields/GBSA_OBC1-1.0.offxml')`). A manifest of new SMIRNOFF-format GBSA files is below.

Several other user-facing improvements have been added, including easier access to indexed attributes, which are now accessible as `torsion.k1`, `torsion.k2`, etc. (the previous access method `torsion.k` still works as well). More details of the new features and several bugfixes are listed below.

New features

- PR #363: Implements `GBSAHandler`, which supports the GBSA tag in the SMIRNOFF specification. Currently, only GBSAHandlers with `gb_model="OBC2"` support setting non-default values for the `surface_area_penalty` term (default `5.4*calories/mole/angstroms**2`), though users can zero the SA term for `OBC1` and `HCT` models by setting `sa_model="None"`. No model currently supports setting `solvent_radius` to any value other than `1.4*angstroms`. Files containing experimental SMIRNOFF-format implementations of `HCT`, `OBC1`, and `OBC2` are included with this release (see below). Additional details of these models, including literature references, are available on the SMIRNOFF specification page.

  > **Warning:** The current release of ParmEd can not transfer GBSA models produced by the Open Force Field Toolkit to other simulation packages. These GBSA forces are currently only computable using OpenMM.

- PR #363: When using `Topology.to_openmm()`, periodic box vectors are now transferred from the Open Force Field Toolkit Topology into the newly-created OpenMM Topology.
- PR #377: Single indexed parameters in `ParameterHandler` and `ParameterType` can now be get/set through normal attribute syntax in addition to the list syntax.
- PR #394: Include element and atom name in error output when there are missing valence parameters during molecule parameterization.

Bugfixes

- PR #385: Fixes Issue #346 by having `OpenEyeToolkitWrapper.compute_partial_charges_am1bcc` fall back to using standard AM1-BCC if AM1-BCC ELF10 charge generation raises an error about "trans COOH conformers"
- PR #399: Fixes issue where `ForceField` constructor would ignore `parameter_handler_classes` kwarg.
- PR #400: Makes link-checking tests retry three times before failing.

Files added

- PR #363: Adds `test_forcefields/GBSA_HCT-1.0.offxml`, `test_forcefields/GBSA_OBC1-1.0.offxml`, and `test_forcefields/GBSA_OBC2-1.0.offxml`, which are experimental implementations of GBSA models. These are primarily used in validation tests against OpenMM's models, and their version numbers will increment if bugfixes are necessary.

3.10.14 0.4.1 - Bugfix Release

This update fixes several toolkit bugs that have been reported by the community. Details of these bugfixes are provided below.

It also refactors how `ParameterType` and `ParameterHandler` store their attributes, by introducing `ParameterAttribute` and `IndexedParameterAttribute`. These new attribute-handling classes provide a consistent backend which should simplify manipulation of parameters and implementation of new handlers.

Bug fixes

- PR #329: Fixed a bug where the two `BondType` parameter attributes `k` and `length` were treated as indexed attributes. (`k` and `length` values that correspond to specific bond orders will be indexed under `k_bondorder1`, `k_bondorder2`, etc when implemented in the future)
- PR #329: Fixed a bug that allowed setting indexed attributes to single values instead of strictly lists.
- PR #370: Fixed a bug in the API where `BondHandler`, `ProperTorsionHandler` , and `ImproperTorsionHandler` exposed non-functional indexed parameters.
- PR #351: Fixes Issue #344, in which the main `FrozenMolecule` constructor and several other Molecule-construction functions ignored or did not expose the `allow_undefined_stereo` keyword argument.
- PR #351: Fixes a bug where a molecule which previously generated a SMILES using one cheminformatics toolkit returns the same SMILES, even though a different toolkit (which would generate a different SMILES for the molecule) is explicitly called.
- PR #354: Fixes the error message that is printed if an unexpected parameter attribute is found while loading data into a `ForceField` (now instructs users to specify `allow_cosmetic_attributes` instead of `permit_cosmetic_attributes`)
- PR #364: Fixes Issue #362 by modifying `OpenEyeToolkitWrapper.from_smiles` and `RDKitToolkitWrapper.from_smiles` to make implicit hydrogens explicit before molecule creation. These functions also now raise an error if the optional keyword `hydrogens_are_explicit=True` but the SMILES are interpreted by the backend cheminformatic toolkit as having implicit hydrogens.
- PR #371: Fixes error when reading early SMIRNOFF 0.1 spec files enclosed by a top-level `SMIRFF` tag.

Note: The enclosing `SMIRFF` tag is present only in legacy files. Since developing a formal specification, the only acceptable top-level tag value in a SMIRNOFF data structure is `SMIRNOFF`.

Code enhancements

- PR #329: `ParameterType` was refactored to improve its extensibility. It is now possible to create new parameter types by using the new descriptors `ParameterAttribute` and `IndexedParameterAttribute`.
- PR #357: Addresses Issue #356 by raising an informative error message if a user attempts to load an OpenMM topology which is probably missing connectivity information.

Force fields added

- PR #368: Temporarily adds `test_forcefields/smirnoff99frosst_experimental.offxml` to address hierarchy problems, redundancies, SMIRKS pattern typos etc., as documented in issue #367. Will ultimately be propagated to an updated force field in the `openforcefield/smirnoff99frosst` repo.
- PR #371: Adds `test_forcefields/smirff99Frosst_reference_0_1_spec.offxml`, a SMIRNOFF 0.1 spec file enclosed by the legacy `SMIRFF` tag. This file is used in backwards-compatibility testing.

3.10.15 0.4.0 - Performance optimizations and support for SMIRNOFF 0.3 specification

This update contains performance enhancements that significantly reduce the time to create OpenMM systems for topologies containing many molecules via `ForceField.create_openmm_system`.

This update also introduces the SMIRNOFF 0.3 specification. The spec update is the result of discussions about how to handle the evolution of data and parameter types as further functional forms are added to the SMIRNOFF spec.

We provide methods to convert SMIRNOFF 0.1 and 0.2 force fields written with the XML serialization (`.offxml`) to the SMIRNOFF 0.3 specification. These methods are called automatically when loading a serialized SMIRNOFF data representation written in the 0.1 or 0.2 specification. This functionality allows the toolkit to continue to read files containing SMIRNOFF 0.2 spec force fields, and also implements backwards-compatibility for SMIRNOFF 0.1 spec force fields.

Warning: The SMIRNOFF 0.1 spec did not contain fields for several energy-determining parameters that are exposed in later SMIRNOFF specs. Thus, when reading SMIRNOFF 0.1 spec data, the toolkit must make assumptions about the values that should be added for the newly-required fields. The values that are added include 1-2, 1-3 and 1-5 scaling factors, cutoffs, and long-range treatments for nonbonded interactions. Each assumption is printed as a warning during the conversion process. Please carefully review the warning messages to ensure that the conversion is providing your desired behavior.

SMIRNOFF 0.3 specification updates

- The SMIRNOFF 0.3 spec introduces versioning for each individual parameter section, allowing asynchronous updates to the features of each parameter class. The top-level `SMIRNOFF` tag, containing information like `aromaticity_model`, `Author`, and `Date`, still has a version (currently 0.3). But, to allow for independent development of individual parameter types, each section (such as `Bonds`, `Angles`, etc) now has its own version as well (currently all 0.3).
- All units are now stored in expressions with their corresponding values. For example, distances are now stored as `1.526*angstrom`, instead of storing the unit separately in the section header.
- The current allowed value of the `potential` field for `ProperTorsions` and `ImproperTorsions` tags is no longer `charmm`, but is rather `k*(1+cos(periodicity*theta-phase))`. It was pointed out to us that CHARMM-style torsions deviate from this formula when the periodicity of a torsion term is 0, and we do not intend to reproduce that behavior.
- SMIRNOFF spec documentation has been updated with tables of keywords and their defaults for each parameter section and parameter type. These tables will track the allowed keywords and default behavior as updated versions of individual parameter sections are released.

Performance improvements and bugfixes

- PR #329: Performance improvements when creating systems for topologies with many atoms.
- PR #347: Fixes bug in charge assignment that occurs when charges are read from file, and reference and charge molecules have different atom orderings.

New features

- PR #311: Several new experimental functions.
  - Adds `convert_0_2_smirnoff_to_0_3`, which takes a SMIRNOFF 0.2-spec data dict, and updates it to 0.3. This function is called automatically when creating a `ForceField` from a SMIRNOFF 0.2 spec OFFXML file.
  - Adds `convert_0_1_smirnoff_to_0_2`, which takes a SMIRNOFF 0.1-spec data dict, and updates it to 0.2. This function is called automatically when creating a `ForceField` from a SMIRNOFF 0.1 spec OFFXML file.
  - NOTE: The format of the "SMIRNOFF data dict" above is likely to change significantly in the future. Users that require a stable serialized ForceField object should use the output of `ForceField.to_string('XML')` instead.
  - Adds `ParameterHandler` and `ParameterType` `add_cosmetic_attribute` and `delete_cosmetic_attribute` functions. Once created, cosmetic attributes can be accessed and modified as attributes of the underlying object (eg. `ParameterType.my_cosmetic_attrib = 'blue'`) These functions are experimental, and we are interested in feedback on how cosmetic attribute handling could be improved. (See Issue #338) Note that if a new cosmetic attribute is added to an object without using these functions, it will not be recognized by the toolkit and will not be written out during serialization.
  - Values for the top-level `Author` and `Date` tags are now kept during SMIRNOFF data I/O. If multiple data sources containing these fields are read, the values are concatenated using "AND" as a separator.

API-breaking changes

- `ForceField.to_string` and `ForceField.to_file` have had the default value of their `discard_cosmetic_attributes` kwarg set to False.
- `ParameterHandler` and `ParameterType` constructors now expect the `version` kwarg (per the SMIRNOFF spec change above) This requirement can be skipped by providing the kwarg `skip_version_check=True`
- `ParameterHandler` and `ParameterType` functions no longer handle `X_unit` attributes in SMIRNOFF data (per the SMIRNOFF spec change above).
- The scripts in `utilities/convert_frosst` are now deprecated. This functionality is important for provenance and will be migrated to the `openforcefield/smirnoff99Frosst` repository in the coming weeks.
- `ParameterType._SMIRNOFF_ATTRIBS` is now `ParameterType._REQUIRED_SPEC_ATTRIBS`, to better parallel the structure of the `ParameterHandler` class.
- `ParameterType._OPTIONAL_ATTRIBS` is now `ParameterType._OPTIONAL_SPEC_ATTRIBS`, to better parallel the structure of the `ParameterHandler` class.
- Added class-level dictionaries `ParameterHandler._DEFAULT_SPEC_ATTRIBS` and `ParameterType._DEFAULT_SPEC_ATTRIBS`.

3.10.16 0.3.0 - API Improvements

Several improvements and changes to public API.

New features

- PR #292: Implement `Topology.to_openmm` and remove `ToolkitRegistry.toolkit_is_available`
- PR #322: Install directories for the lookup of OFFXML files through the entry point group `openforcefield.smirnoff_forcefield_directory`. The `ForceField` class doesn't search in the `data/forcefield/` folder anymore (now renamed `data/test_forcefields/`), but only in `data/`.

API-breaking Changes

- PR #278: Standardize variable/method names
- PR #291: Remove `ForceField.load/to_smirnoff_data`, add `ForceField.to_file/string` and `ParameterHandler.add_parameters`. Change behavior of `ForceField.register_X_handler` functions.

Bugfixes

- PR #327: Fix units in tip3p.offxml (note that this file is still not loadable by current toolkit)
- PR #325: Fix solvent box for provided test system to resolve periodic clashes.
- PR #325: Add informative message containing Hill formula when a molecule can't be matched in `Topology.from_openmm`.
- PR #325: Provide warning or error message as appropriate when a molecule is missing stereochemistry.
- PR #316: Fix formatting issues in GBSA section of SMIRNOFF spec
- PR #308: Cache molecule SMILES to improve system creation speed

- PR #306: Allow single-atom molecules with all zero coordinates to be converted to OE/RDK mols
- PR #313: Fix issue where constraints are applied twice to constrained bonds

3.10.17 0.2.2 - Bugfix release

This release modifies an example to show how to parameterize a solvated system, cleans up backend code, and makes several improvements to the README.

Bugfixes

- PR #279: Cleanup of unused code/warnings in main package `__init__`
- PR #259: Update T4 Lysozyme + toluene example to show how to set up solvated systems
- PR #256 and PR #274: Add functionality to ensure that links in READMEs resolve successfully

3.10.18 0.2.1 - Bugfix release

This release features various documentation fixes, minor bugfixes, and code cleanup.

Bugfixes

- PR #267: Add neglected `<ToolkitAM1BCC>` documentation to the SMIRNOFF 0.2 spec
- PR #258: General cleanup and removal of unused/inaccessible code.
- PR #244: Improvements and typo fixes for BRD4:inhibitor benchmark

3.10.19 0.2.0 - Initial RDKit support

This version of the toolkit introduces many new features on the way to a 1.0.0 release.

New features

- Major overhaul, resulting in the creation of the SMIRNOFF 0.2 specification and its XML representation
- Updated API and infrastructure for reference SMIRNOFF `ForceField` implementation
- Implementation of modular `ParameterHandler` classes which process the topology to add all necessary forces to the system.
- Implementation of modular `ParameterIOHandler` classes for reading/writing different serialized SMIRNOFF force field representations
- Introduction of `Molecule` and `Topology` classes for representing molecules and biomolecular systems
- New `ToolkitWrapper` interface to RDKit, OpenEye, and AmberTools toolkits, managed by `ToolkitRegistry`
- API improvements to more closely follow PEP8 guidelines
- Improved documentation and examples

3.10.20 0.1.0

This is an early preview release of the toolkit that matches the functionality described in the preprint describing the SMIRNOFF v0.1 force field format: [DOI].

New features

This release features additional documentation, code comments, and support for automated testing.

Bugfixes

Treatment of improper torsions

A significant (though currently unused) problem in handling of improper torsions was corrected. Previously, non-planar impropers did not behave correctly, as six-fold impropers have two potential chiralities. To remedy this, SMIRNOFF impropers are now implemented as three-fold impropers with consistent chirality. However, current force fields in the SMIRNOFF format had no non-planar impropers, so this change is mainly aimed at future work.

CHAPTER

FOUR

FREQUENTLY ASKED QUESTIONS (FAQ)

4.1 What kinds of input files can I apply SMIRNOFF parameters to?

SMIRNOFF force fields use direct chemical perception meaning that, unlike many molecular mechanics (MM) force fields, they apply parameters based on substructure searches acting directly on molecules. This creates unique opportunities and allows them to encode a great deal of chemistry quite simply, but it also means that the *starting point* for parameter assignment must be well-defined chemically, giving not just the elements and connectivity for all of the atoms of all of the components of your system, but also providing the formal charges and bond orders.

Specifically, to apply SMIRNOFF to a system, you must either:

1. Provide Open Force Field Toolkit `Molecule` objects corresponding to the components of your system, or
2. Provide an OpenMM `Topology` which includes bond orders and thus can be converted to molecules corresponding to the components of your system

Without this information, our direct chemical perception cannot be applied to your molecule, as it requires the chemical identity of the molecules in your system – that is, bond order and formal charge as well as atoms and connectivity. Unless you provide the full chemical identity in this sense, we must attempt to guess or infer the chemical identity of your molecules, which is a recipe for trouble. Different molecules can have the same chemical graph but differ in bond order and formal charge, or different resonance structures may be treated rather differently by some force fields (e.g. `c1cc(ccc1c2cc[nH+]cc2)[O-]` vs `C1=CC(C=CC1=C2C=CNC=C2)=O`, where the central bond is rotatable in one resonance structure but not in the other) even though they have identical formal charge and connectivity (chemical graph). A force field which uses the chemical identity of molecules to assign parameters needs to know the exact chemical identity of the molecule you are intending to parameterize.

4.2 Can I use an AMBER (or GROMACS) topology/coordinate file as a starting point for applying a SMIRNOFF force field?

In a word, “no”.

Parameter files used by typical molecular dynamics simulation packages do not currently encode enough information to identify the molecules chemically present, or at least not without drawing inferences. For example, one could take a structure file and infer bond orders based on bond lengths, or attempt to infer bond orders from force constants in a parameter file. Such inference work is outside the scope of SMIRNOFF.

4.3 What about starting from a PDB file?

PDB files do not in general provide the chemical identity of small molecules contained therein, and thus do not provide suitable starting points for applying SMIRNOFF to small molecules. This is especially problematic for PDB files from X-ray crystallography which typically do not include proteins, making the problem even worse. For our purposes here, however, we assume you begin with the coordinates of all atoms present and the full topology of your system.

Given a PDB file of a hypothetical biomolecular system of interest containing a small molecule, there are several routes available to you for treating the small molecule present:

- Use a cheminformatics toolkit (see below) to infer bond orders
- Identify your ligand from a database; e.g. if it is in the Protein Data Bank (PDB), it will be present in the Ligand Expo meaning that it has a database entry and code you can use to look up its putative chemical identity
- Identify your ligand by name or SMILES string (or similar) from the literature or your collaborators

4.4 What about starting from an XYZ file?

XYZ files generally only contain elements and positions, and are therefore similar in content to PDB files. See the above section "What about starting from a PDB file?" for more information.

4.5 What do you recommend as a starting point?

For application of SMIRNOFF force fields, we recommend that you begin your work with formats which provide the chemical identity of your small molecule (including formal charge and bond order). This means we recommend one of the following or equivalent:

- A `.sdf`, `.mol`, or `.mol2` file or files for the molecules comprising your system, with correct bond orders and formal charges. (Note: Do NOT generate this from a simulation package or tool which does not have access to bond order information; you may end up with a correct-seeming file, but the bond orders will be incorrect)
- Isomeric SMILES strings for the components of your system
- InChi strings for the components of your system
- Chemical Identity Registry numbers for the components of your system
- IUPAC names for the components of your system

Essentially, anything which provides the full identity of what you want to simulate (including stereochemistry) should work, though it may require more or less work to get it into an acceptable format.

4.6 I understand the risks and want to perform bond and formal charge inference anyway

If you are unable to provide a molecule in the formats recommended above and want to attempt to infer the bond orders and atomic formal charges, there are tools available elsewhere that can provide guesses for this problem. These tools are not perfect, and the inference problem itself is poorly defined, so you should review each output closely (see our *Core Concepts* for an explanation of what information is needed to construct an OpenFF Molecule). Some tools we know of include:

- the OpenEye Toolkit's `OEPerceiveBondOrders` functionality
- MDAnalysis' RDKit converter, with an example here
- the Jensen group's xyz2mol program

4.7 I'm getting stereochemistry errors when loading a molecule from a SMILES string.

By default, the OpenFF Toolkit throws an error if a molecule with undefined stereochemistry is loaded. This is because the stereochemistry of a molecule may affect its partial charges, and assigning parameters using direct chemical perception may require knowing the stereochemistry of chiral centers. In addition, coordinates generated by the Toolkit for undefined chiral centers may have any combination of stereochemistries; the toolkit makes no guarantees about consistency, uniformity, or randomness. Note that the main-line OpenFF force fields currently use a stereochemistry-dependent charge generation method, but do not include any other stereospecific parameters.

This behavior is in line with OpenFF's general attitude of requiring users to explicitly acknowledge actions that may cause silent errors later on. If you're confident a `Molecule` with unassigned stereochemistry is acceptable, pass `allow_undefined_stereo=True` to molecule loading methods like *Molecule.from_smiles* to downgrade the exception to a warning. For an example, see the "SMILES without stereochemistry" section in the *Molecule cookbook*. Where possible, our parameter assignment infrastructure will gracefully handle molecules with undefined stereochemistry that are loaded this way, though they will be missing any stereospecific parameters.

4.8 My conda installation of the toolkit doesn't appear to work. What should I try next?

We recommend that you install the toolkit in a fresh conda environment, explicitly passing the channels to be used, in-order:

```
conda create -n <my_new_env> -c conda-forge openff-toolkit
conda activate <my_new_env>
```

Installing into a new environment avoids forcing conda to satisfy the dependencies of both the toolkit and all existing packages in that environment. Taking the approach that conda environments are generally disposable, even ephemeral, minimizes the chances for hard-to-diagnose dependency issues.

4.9 My conda installation of the toolkit STILL doesn't appear to work.

Many of our users encounter issues that are ultimately due to their terminal finding a different `conda` at higher priority in their `PATH` than the `conda` deployment where OpenFF is installed. To fix this, find the conda deployment where OpenFF is installed. Then, if that folder is something like `~/miniconda3`, run in the terminal:

```
source ~/miniconda3/etc/profile.d/conda.sh
```

and then try rerunning and/or reinstalling the Toolkit.

4.10 The partial charges generated by the toolkit don't seem to depend on the molecule's conformation! Is this a bug?

No! This is the intended behavior. The force field parameters of a molecule should be independent of both their chemical environment and conformation so that they can be used and compared across different contexts. When applying AM1BCC partial charges, the toolkit achieves a deterministic output by ignoring the input conformation and producing several new conformations for the same molecule. Partial charges are then computed based on these conformations. This behavior can be controlled with the `use_conformers` argument to *Molecule.assign_partial_charges()*.

4.11 Parameterizing my system, which contains a large molecule, is taking forever. What's wrong?

The mainline OpenFF force fields use AM1-BCC to assign partial charges (via the `<ToolkitAM1BCCHandler>` tag in the OFFXML file). This method unfortunately scales poorly with the size of a molecule and ligands roughly 100 atoms (about 40 heavy atoms) or larger may take so long (i.e. 10 minutes or more) that it seems like your code is simply hanging indefinitely. If you have an OpenEye license and OpenEye Toolkits *installed*, the OpenFF Toolkit will instead use `quacpac`, which can offer better performance on large molecules. Otherwise, it uses AmberTools' `sqm`, which is free to use.

In the future, the use of AM1-BCC in OpenFF force fields may be replaced with method(s) that perform better and scale better with molecule size, but (as of April 2022) these are still in an experimental phase.

4.12 How can I distribute my own force fields in SMIRNOFF format?

We support conda data packages for distribution of force fields in `.offxml` format! Just add the relevant entry point to `setup.py` and distribute on conda Forge:

```
entry_points={
    'openforcefield.smirnoff_forcefield_directory' : [
        'my_new_force_field_paths = my_package:get_my_new_force_field_paths',
    ],
}
```

Where `get_my_new_force_field_paths` is a function in the `my_package` module providing a list of strings holding the paths to the directories to search. You should also rename `my_new_force_field_paths` to suit your force field. See `openff-forcefields` for an example.

4.13 What does "unconstrained" mean in a force field name?

Each release of an OpenFF force field has two associated `.offxml` files: one unadorned (for example, `openff-2.0.0.offxml`) and one labeled "unconstrained" (`openff_unconstrained-2.0.0.offxml`). This reflects the presence or absence of holonomic constraints on hydrogen-involving bonds in the force field specification.

Typically, OpenFF force fields treat bonds with a harmonic potential according to Hooke's law. With this treatment, bonds involving hydrogen atoms have a much higher vibration frequency than any other part of a typical biochemical system. By constraining these bonds to a fixed length, MD time steps can be increased past 1 fs, improving simulation performance. These bond vibrations are not structurally important to proteins so can usually be ignored.

While we recommend hydrogen-involving bond constraints and a time step of 2 fs for ordinary use, some other specialist uses require a harmonic treatment. The unconstrained force fields are provided for these uses.

Use the constrained force field:

- When running MD with a time step greater than 1 fs

Use the unconstrained force field:

- When computing single point energy calculations or energy minimization
- When running MD with a time step of 1 fs (or less)
- When bond lengths may deviate from equilibrium
- When fitting a force field, both because many fitting techniques require continuity and because deviations from equilibrium bond length may be important
- Any other circumstance when forces or energies must be defined or continuous for any possible position of a hydrogen atom

Starting with v2.0.0 (Sage), TIP3P water is included in OpenFF force fields. The geometry of TIP3P water is always constrained, even in the unconstrained force fields.

4.14 How do I add or remove constraints from my own force field?

To make applying or removing bond constraints easy, constrained force fields released by OpenFF always include full bond parameters. Constraints on Hydrogen-involving bonds inherit their lengths from the harmonic parameters also included in the force field. To restore the harmonic treatment, simply remove the appropriate constraint entry from the force field.

Hydrogen-involving bonds are constrained with a single constraint entry in a `.offxml` file:

```
<Constraints version="0.3">
    <!-- constrain all bonds to hydrogen to their equilibrium bond length -->
    <Constraint smirks="[#1:1]-[*:2]" id="c1"></Constraint>
</Constraints>
```

Adding or removing the inner `<Constraint...` line will convert a force field between being constrained and unconstrained. A `ForceField` object can constrain its bonds involving hydrogen by adding the relevant parameter to its `'Constraints'` parameter handler:

```
ch = force_field.get_parameter_handler('Constraints')
ch.add_parameter(smirks="[#1:1]-[*:2]")
```

Constraints can be removed from bonds involving hydrogen by removing the corresponding parameter:

```
del forcefield['Constraints']["[#1:1]-[*:2]"]
```

CHAPTER

FIVE

CORE CONCEPTS

OpenFF `Molecule` A graph representation of a molecule containing enough information to unambiguously parametrize it. Required data fields for a `Molecule` are:

- `atoms`: element (integer), formal_charge (integer), is_aromatic (boolean), stereochemistry (`'R'/'S'/None`)
- `bonds`: order (integer), is_aromatic (boolean), stereochemistry (`'E'/'Z'/None`)

There are several other optional attributes such as `conformers` and `partial_charges` that may be populated in the `Molecule` data structure. These are considered "optional" because they are not required for system creation, however if those fields are populated, the user MAY use them to override values that would otherwise be generated during system creation.

A dictionary, `Molecule.properties` is exposed, which is a Python dict that can be populated with arbitrary data. This data should be considered cosmetic and should not affect system creation. Whenever possible, molecule serialization or format conversion should preserve this data.

OpenFF `Topology` A collection of molecules, as well as optional box vector, positions, and other information. A `Topology` describes the chemistry of a system, but has no force field parameters.

OpenFF `ForceField` An object generated from an OFFXML file (or other source of SMIRNOFF data). Most information from the SMIRNOFF data source is stored in this object's several `ParameterHandlers`, however some top-level SMIRNOFF data is stored in the `ForceField` object itself.

OpenFF `Interchange` A `Topology` that has been parametrized by a `ForceField`. An `Interchange` contains all the information needed to calculate an energy or start a simulation. `Interchange` also provides methods to export this information to MD simulation engines.

CHAPTER

SIX

COOKBOOK: EVERY WAY TO MAKE A MOLECULE

Every pathway through the OpenFF Toolkit boils down to four steps:

1. Using other tools, assemble a graph of a molecule, including all of its atoms, bonds, bond orders, formal charges, and stereochemistry[1]
2. Use that information to construct a `Molecule`
3. Combine a number of `Molecule` objects to construct a `Topology`
4. Call `ForceField.create_openmm_system(topology)` to create an OpenMM `System` (or, in the near future, an OpenFF `Interchange` for painless conversion to all sorts of MD formats)

So let's take a look at every way there is to construct a molecule! We'll use zwitterionic L-alanine as an example biomolecule with all the tricky bits - a stereocenter, non-zero formal charges, and bonds of different orders.

6.1 From SMILES

SMILES is the classic way to create a `Molecule`. SMILES is a widely-used compact textual representation of arbitrary molecules. This lets us specify an exact molecule, including stereochemistry and bond orders, very easily — though they may not be the most human-readable format.

The `Molecule.from_smiles()` method is used to create a `Molecule` from a SMILES code.

6.1.1 Implicit hydrogens SMILES

```
zw_l_alanine = Molecule.from_smiles("C[C@H]([NH3+])C(=O)[O-]")

zw_l_alanine.visualize()
```

```
<IPython.core.display.SVG object>
```

[1] Note that this stereochemistry must be defined on the *graph* of the molecule. It's not good enough to just co-ordinates with the correct stereochemistry. But if you have the co-ordinates, you can try getting the stereochemistry automatically with `rdkit` or `openeye` — If you dare!

6.1.2 Explicit hydrogens SMILES

```
smiles_explicit_h = Molecule.from_smiles(
    "[H][C]([H])([H])[C@@]([H])([C](=[O])[O-])[N+]([H])([H])[H]",
    hydrogens_are_explicit=True
)

assert zw_l_alanine.is_isomorphic_with(smiles_explicit_h)

smiles_explicit_h.visualize()
```

```
<IPython.core.display.SVG object>
```

6.1.3 Mapped SMILES

By default, no guarantees are made about the indexing of atoms from a SMILES string. If the indexing is important, a mapped SMILES string may be used. In this case, Hydrogens must be explicit. Note that though mapped SMILES strings must start at index 1, Python lists start at index 0.

```
mapped_smiles = Molecule.from_mapped_smiles(
    "[H:10][C:2]([H:7])([H:8])[C@@:4]([H:9])([C:3](=[O:5])[O-:6])[N+:1]([H:11])([H:12])[H:13]
↪"
)

assert zw_l_alanine.is_isomorphic_with(mapped_smiles)

assert mapped_smiles.atoms[0].atomic_number == 7 # First index is the Nitrogen
assert all([a.atomic_number==1 for a in mapped_smiles.atoms[6:]]) # Final indices are all H

mapped_smiles.visualize()
```

```
<IPython.core.display.SVG object>
```

6.1.4 SMILES without stereochemistry

The Toolkit won't accept an ambiguous SMILES. This SMILES could be L- or D- alanine; rather than guess, the Toolkit throws an error:

```
smiles_non_isomeric = Molecule.from_smiles(
    "CC([NH3+])C(=O)[O-]"
)
```

```
UndefinedStereochemistryError: Unable to make OFFMol from RDMol: Unable to make OFFMol from␣
↪SMILES: RDMol has unspecified stereochemistry. Undefined chiral centers are:
 - Atom C (index 1)
```

We can downgrade this error to a warning with the `allow_undefined_stereo` argument. This will create a molecule with undefined stereochemistry, which might lead to incorrect parametrization or surprising conformer generation. See the *FAQ* for more details.

```
smiles_non_isomeric = Molecule.from_smiles(
    "CC([NH3+])C(=O)[O-]",
    allow_undefined_stereo=True
)

assert not zw_l_alanine.is_isomorphic_with(smiles_non_isomeric)

smiles_non_isomeric.visualize()
```

```
<IPython.core.display.SVG object>
```

6.2 By hand

You can always construct a `Molecule` by building it up from individual atoms and bonds. Other methods are generally easier, but it's a useful fallback for when you need to write your own constructor for an unsupported source format.

The `Molecule()` constructor and the `add_atom()` and `add_bond()` methods are used to construct a `Molecule` by hand.

```
by_hand = Molecule()
by_hand.name = "Zwitterionic l-Alanine"

by_hand.add_atom(
    atomic_number = 8, # Atomic number 8 is Oxygen
    formal_charge = -1, # Formal negative charge
    is_aromatic = False, # Atom is not part of an aromatic system
    stereochemistry = None, # Optional argument; "R" or "S" stereochemistry
    name = "O-" # Optional argument; descriptive name for the atom
)
by_hand.add_atom(6,  0, False, name="C")
by_hand.add_atom(8,  0, False, name="O")
by_hand.add_atom(6,  0, False, stereochemistry="S", name="CA")
by_hand.add_atom(1,  0, False, name="CAH")
by_hand.add_atom(6,  0, False, name="CB")
by_hand.add_atom(1,  0, False, name="HB1")
by_hand.add_atom(1,  0, False, name="HB2")
by_hand.add_atom(1,  0, False, name="HB3")
by_hand.add_atom(7, +1, False, name="N+")
by_hand.add_atom(1,  0, False, name="HN1")
by_hand.add_atom(1,  0, False, name="HN2")
by_hand.add_atom(1,  0, False, name="HN3")


by_hand.add_bond(
    atom1 = 0, # First (zero-indexed) atom specified above ("O-")
    atom2 = 1, # Second atom specified above ("C")
    bond_order = 1, # Single bond
    is_aromatic = False, # Bond is not aromatic
    stereochemistry = None, # Optional argument; "E" or "Z" stereochemistry
    fractional_bond_order = None # Optional argument; Wiberg (or similar) bond order
```

(continues on next page)

(continued from previous page)

```
)
by_hand.add_bond( 1,  2, 2, False) # C = O
by_hand.add_bond( 1,  3, 1, False) # C - CA
by_hand.add_bond( 3,  4, 1, False) # CA - CAH
by_hand.add_bond( 3,  5, 1, False) # CA - CB
by_hand.add_bond( 5,  6, 1, False) # CB - HB1
by_hand.add_bond( 5,  7, 1, False) # CB - HB2
by_hand.add_bond( 5,  8, 1, False) # CB - HB3
by_hand.add_bond( 3,  9, 1, False) # CB - N+
by_hand.add_bond( 9, 10, 1, False) # N+ - HN1
by_hand.add_bond( 9, 11, 1, False) # N+ - HN2
by_hand.add_bond( 9, 12, 1, False) # N+ - HN3

assert zw_l_alanine.is_isomorphic_with(by_hand)

by_hand.visualize()
```

```
<IPython.core.display.SVG object>
```

6.2.1 From a dictionary

Rather than build up the `Molecule` one method at a time, the `Molecule.from_dict()` method can construct a `Molecule` in one shot from a Python `dict` that describes the molecule in question. This allows `Molecule` objects to be written to and read from disk in any format that can be interpreted as a `dict`; this mechanism underlies the `from_bson()`, `from_json()`, `from_messagepack()`, `from_pickle()`, `from_toml()`, `from_xml()`, and `from_yaml()` methods.

This format can get very verbose, as it is intended for serialization, so this example uses hydrogen cyanide rather than alanine.

```
molecule_dict = {
    "name": "",
    "atoms": [
        {
            "atomic_number": 1,
            "formal_charge": 0,
            "is_aromatic": False,
            "stereochemistry": None,
            "name": "H",
        },
        {
            "atomic_number": 6,
            "formal_charge": 0,
            "is_aromatic": False,
            "stereochemistry": None,
            "name": "C",
        },
        {
            "atomic_number": 7,
            "formal_charge": 0,
            "is_aromatic": False,
```

(continues on next page)

(continued from previous page)

```
            "stereochemistry": None,
            "name": "N",
        },
    ],
    "virtual_sites": [],
    "bonds": [
        {
            "atom1": 0,
            "atom2": 1,
            "bond_order": 1,
            "is_aromatic": False,
            "stereochemistry": None,
            "fractional_bond_order": None,
        },
        {
            "atom1": 1,
            "atom2": 2,
            "bond_order": 3,
            "is_aromatic": False,
            "stereochemistry": None,
            "fractional_bond_order": None,
        },
    ],
    "properties": {},
    "conformers": None,
    "partial_charges": None,
    "partial_charges_unit": None,
    "hierarchy_schemes": {},
}

from_dictionary = Molecule.from_dict(molecule_dict)

from_dictionary.visualize()
```

```
<IPython.core.display.SVG object>
```

6.3 From a file

We can construct a `Molecule` from a file or file-like object with the `from_file()` method. We're a bit constrained in what file formats we can accept, because they need to provide all the information needed to construct the molecular graph; not just coordinates, but also elements, formal charges, bond orders, and stereochemistry.

6.3.1 From SDF file

We generally recommend the SDF format. The SDF file used here can be found on GitHub

```
sdf_path = Molecule.from_file("zw_l_alanine.sdf")
assert zw_l_alanine.is_isomorphic_with(sdf_path)
sdf_path.visualize()
```

```
<IPython.core.display.SVG object>
```

6.3.2 From SDF file object

`from_file()` can also take a file object, rather than a path. Note that the object must be in binary mode!

```
with open("zw_l_alanine.sdf", mode="rb") as file:
    sdf_object = Molecule.from_file(file, file_format="SDF")

assert zw_l_alanine.is_isomorphic_with(sdf_object)
sdf_object.visualize()
```

```
<IPython.core.display.SVG object>
```

6.3.3 From protein PDB file

The `from_polymer_pdb()` method can interpret a protein from a PDB file as a molecule. It can infer the full chemical graph of the canonical amino acids (26 including protonation states) in capped and uncapped forms. It does this according to the RCSB chemical component dictionary — the information is not explicitly in the PDB. For the moment, it's quite slow, but this is the easiest way to get a full protein into the OpenFF ecosystem:

```
from openff.toolkit.utils import get_data_file_path

path = get_data_file_path('proteins/T4-protein.pdb')
protein = Molecule.from_polymer_pdb(path)
protein.visualize("nglview")
```

```

```

```
NGLWidget()
```

6.3.4 From small molecule PDB file

Using PDB files for small molecules is not recommended, even if they have CONECT records, as they do not provide stereoisomeric information or bond orders. The RDKit backend assumes that bond orders are 1, so the toolkit refuses to use it:

```
from openff.toolkit.utils.toolkits import RDKitToolkitWrapper

pdb = Molecule.from_file("zw_l_alanine.pdb", "pdb", toolkit_registry=RDKitToolkitWrapper())
```

```
NotImplementedError: Toolkit The RDKit can not read file zw_l_alanine.pdb (format PDB).␣
↪Supported formats for this toolkit are ['SDF', 'MOL', 'SMI'].RDKit can however read PDBs␣
↪with a valid smiles string using the Molecule.from_pdb_and_smiles(file_path, smiles)␣
↪constructor
```

OpenEye can infer bond orders and stereochemistry from the structure. This is not recommended, as it can make mistakes that may be difficult to catch. Note also that this requires a license for OpenEye, as this is proprietary software.

If we instead provide a SMILES code, a PDB file can be used to populate the `Molecule` object's `conformers` attribute and provide atom ordering, as well as check that the SMILES code matches the PDB file. This method is the recommended way to create a `Molecule` from a PDB file. The PDB file used here can be found on GitHub

---

Important: Note that the Toolkit doesn't guarantee that the coordinates in the PDB are correctly assigned to atoms. It makes an effort, but you should check that the results are reasonable.

```
pdb_with_smiles = Molecule.from_pdb_and_smiles(
    "zw_l_alanine.pdb",
    "C[C@H]([NH3+])C(=O)[O-]"
)

assert zw_l_alanine.is_isomorphic_with(pdb_with_smiles)

pdb_with_smiles.visualize()
```

```
<IPython.core.display.SVG object>
```

6.4 Other string identification formats

The OpenFF Toolkit supports a few text based molecular identity formats other than SMILES (*see above*)

6.4.1 From InChI

The `Molecule.from_inchi()` method constructs a `Molecule` from an IUPAC InChI string. Note that InChI cannot distinguish the zwitterionic form of alanine from the neutral form (see section 13.2 of the InChI Technical FAQ), so the toolkit defaults to the neutral form.

Warning: The OpenFF Toolkit makes no guarantees about the atomic ordering produced by the `from_inchi` method. InChI is not intended to be an interchange format.

```
inchi = Molecule.from_inchi("InChI=1S/C3H7NO2/c1-2(4)3(5)6/h2H,4H2,1H3,(H,5,6)/t2-/m0/s1")

inchi.visualize()
```

```
<IPython.core.display.SVG object>
```

6.4.2 From IUPAC name

The `Molecule.from_iupac()` method constructs a `Molecule` from an IUPAC name.

Important: This code requires the OpenEye toolkit.

```
iupac = Molecule.from_iupac("(2S)-2-azaniumylpropanoate")

assert zw_l_alanine.is_isomorphic_with(iupac)

iupac.visualize()
```

```
ValueError: No registered toolkits can provide the capability "from_iupac" for args "('(2S)-
↪2-azaniumylpropanoate',)" and kwargs "{'allow_undefined_stereo': False, '_cls': <class
↪'openff.toolkit.topology.molecule.Molecule'>}"
Available toolkits are: [ToolkitWrapper around The RDKit version 2022.03.3, ToolkitWrapper␣
↪around AmberTools version 21.0, ToolkitWrapper around Built-in Toolkit version None]
```

6.5 Re-ordering atoms in an existing `Molecule`

Most `Molecule` creation methods don't specify the ordering of atoms in the new `Molecule`. The `Molecule.remap()` method allows a new ordering to be applied to an existing `Molecule`.

See also *Mapped SMILES*.

Warning: The `Molecule.remap()` method is experimental and subject to change.

```
# Note that this mapping is off-by-one from the mapping taken
# by the remap method, as Python indexing is 0-based but SMILES
# is 1-based
print("Before remapping:", zw_l_alanine.to_smiles(mapped=True))

# Flip the order of the carbonyl carbon and oxygen
remapped = zw_l_alanine.remap({0: 0,1: 1, 2: 2, 3: 4, 4: 3, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9,␣
↪10: 10, 11: 11, 12: 12})
#                                                ^--------^

print("After remapping: ", remapped.to_smiles(mapped=True))

# Doesn't affect the identity of the molecule
assert zw_l_alanine.is_isomorphic_with(remapped)
remapped.visualize()
```

```
Before remapping: [C:1]([C@:2]([N+:3]([H:11])([H:12])[H:13])([C:4](=[O:5])[O-
↪:6])[H:10])([H:7])([H:8])[H:9]
After remapping:  [C:1]([C@:2]([N+:3]([H:11])([H:12])[H:13])([C:5](=[O:4])[O-
↪:6])[H:10])([H:7])([H:8])[H:9]
```

```
<IPython.core.display.SVG object>
```

6.6 Via Topology objects

The Topology class represents a biomolecular system; it is analogous to the similarly named objects in GROMACS, MDTraj or OpenMM. Notably, it does not include co-ordinates and may represent multiple copies of a particular molecular species or even more complex mixtures of molecules. Topology objects are usually built up one species at a time from Molecule objects.

Molecule objects can be retrieved from a Topology via the Topology.molecule() method by providing the index of the molecule within the topology. For a topology consisting of a single molecule, this is just topology.molecule(0).

Constructor methods that are available for Topology but not Molecule generally require a Molecule to be provided via the unique_molecules keyword argument. The provided Molecule is used to provide the identity of the molecule, including aromaticity, bond orders, formal charges, and so forth. These methods therefore don't provide a route to the graph of the molecule, but can be useful for reordering atoms to match another software package.

6.6.1 From an OpenMM Topology

The Topology.from_openmm() method constructs an OpenFF Topology from an OpenMM Topology. The method requires that all the unique molecules in the Topology are provided as OpenFF Molecule objects, as the structure of an OpenMM Topology doesn't include the concept of a molecule. When using this method to create a Molecule, this limitation means that the method really only offers a pathway to reorder the atoms of a Molecule to match that of the OpenMM Topology.

```
from openmm.app.pdbfile import PDBFile

openmm_topology = PDBFile('zw_l_alanine.pdb').getTopology()
openff_topology = Topology.from_openmm(openmm_topology, unique_molecules=[zw_l_alanine])

from_openmm_topology = openff_topology.molecule(0)

assert zw_l_alanine.is_isomorphic_with(from_openmm_topology)

from_openmm_topology.visualize()
```

```
<IPython.core.display.SVG object>
```

6.6.2 From an MDTraj Topology

The Topology.from_mdtraj() method constructs an OpenFF Topology from an MDTraj Topology. The method requires that all the unique molecules in the Topology are provided as OpenFF Molecule objects to ensure that the graph of the molecule is correct. When using this method to create a Molecule, this limitation means that the method really only offers a pathway to reorder the atoms of a Molecule to match that of the MDTraj Topology.

```
from mdtraj import load_pdb

mdtraj_topology = load_pdb('zw_l_alanine.pdb').topology
openff_topology = Topology.from_openmm(openmm_topology, unique_molecules=[zw_l_alanine])

from_mdtraj_topology = openff_topology.molecule(0)

assert zw_l_alanine.is_isomorphic_with(from_mdtraj_topology)

from_mdtraj_topology.visualize()
```

```
<IPython.core.display.SVG object>
```

6.7 From Toolkit objects

The OpenFF Toolkit calls out to other software to perform low-level tasks like reading SMILES or files. These external software packages are called toolkits, and presently include RDKit and the OpenEye Toolkit. OpenFF `Molecule` objects can be created from the equivalent objects in these toolkits.

6.7.1 From RDKit `Mol`

The `Molecule.from_rdkit()` method converts an `rdkit.Chem.rdchem.Mol` object to an OpenFF `Molecule`.

```
from rdkit import Chem
rdmol = Chem.MolFromSmiles("C[C@H]([NH3+])C([O-])=O")

print("rdmol is of type", type(rdmol))

from_rdmol = Molecule.from_rdkit(rdmol)

assert zw_l_alanine.is_isomorphic_with(from_rdmol)
from_rdmol.visualize()
```

```
rdmol is of type <class 'rdkit.Chem.rdchem.Mol'>
```

```
<IPython.core.display.SVG object>
```

6.7.2 From OpenEye `OEMol`

The `Molecule.from_openeye()` method converts an object that inherits from `openeye.oechem.OEMolBase` to an OpenFF `Molecule`.

```
from openeye import oechem

oemol = oechem.OEGraphMol()
oechem.OESmilesToMol(oemol, "C[C@H]([NH3+])C([O-])=O")

assert isinstance(oemol, oechem.OEMolBase)
```

(continues on next page)

(continued from previous page)

```
from_oemol = Molecule.from_openeye(oemol)

assert zw_l_alanine.is_isomorphic_with(from_oemol)
from_oemol.visualize()
```

```
<IPython.core.display.SVG object>
```

6.8 From QCArchive

QCArchive is a repository of quantum chemical calculations on small molecules. The `Molecule.from_qcschema()` method creates a `Molecule` from a record from the archive. Because the identity of a molecule can change of the course of a QC calculation, the Toolkit accepts records only if they contain a hydrogen-mapped SMILES code.

Note: These examples use molecules other than l-Alanine because of their availability in QCArchive

6.8.1 From a QCArchive molecule record

The `Molecule.from_qcschema()` method can take a molecule record queried from the QCArchive and create a `Molecule` from it.

```
from qcportal import FractalClient

client = FractalClient()
query = client.query_molecules(molecular_formula="C16H20N3O5")

from_qcarchive = Molecule.from_qcschema(query[0])

from_qcarchive.visualize()
```

```
<IPython.core.display.SVG object>
```

6.8.2 From a QCArchive optimisation record

`Molecule.from_qcschema()` can also take an optimisation record and create the corresponding `Molecule`.

```
optimization_dataset = client.get_collection(
    "OptimizationDataset",
    "SMIRNOFF Coverage Set 1"
)
dimethoxymethanol_optimization = optimization_dataset.get_entry('coc(o)oc-0')

from_optimisation = Molecule.from_qcschema(dimethoxymethanol_optimization)

from_optimisation.visualize()
```

```
<IPython.core.display.SVG object>
```

CHAPTER

SEVEN

SMIRNOFF (SMIRKS NATIVE OPEN FORCE FIELD)

7.1 The SMIRNOFF specification

The SMIRNOFF specification can be found in the OpenFF standards repository.

7.2 SMIRNOFF and the Toolkit

OpenFF releases all its force fields in SMIRNOFF format. SMIRNOFF is a format developed by OpenFF; its specification can be found in our standards repository. SMIRNOFF-format force fields are distributed as XML files with the `.offxml` extension. Instead of using atom types like traditional force field formats, SMIRNOFF associates parameters directly with chemical groups using SMARTS and SMIRKS, which are extensions of the popular SMILES serialization format for molecules. SMIRNOFF goes to great lengths to ensure reproducibility of results generated from its force fields.

The OpenFF Toolkit is the reference implementation of the SMIRNOFF spec. The toolkit is responsible for reading and writing `.offxml` files, for facilitating their modification, and for applying them to a molecular system in order to produce an `Interchange` object. The OpenFF Interchange project then takes over and is responsible for producing input files and data for actual MD software. The toolkit strives to be backwards compatible with old versions of the spec, but owing to the vagaries of the arrow of time cannot be forward compatible. Trying to use an old version of the toolkit to load an `.OFFXML` file created with a new version of the spec will lead to an error.

A simplified `.offxml` file for TIP3P water might look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<SMIRNOFF version="0.3" aromaticity_model="OEAroModel_MDL">
    <Author>The Open Force Field Initiative</Author>
    <Date>2021-08-16</Date>
    <Constraints version="0.3">
        <Constraint smirks="[#1:1]-[#8X2H2+0:2]-[#1]" id="c-tip3p-H-O"
                    distance="0.9572 * angstrom"></Constraint>
        <Constraint smirks="[#1:1]-[#8X2H2+0]-[#1:2]" id="c-tip3p-H-O-H"
                    distance="1.5139006545247014 * angstrom"></Constraint>
    </Constraints>
    <vdW version="0.3" potential="Lennard-Jones-12-6" combining_rules="Lorentz-Berthelot"
         scale12="0.0" scale13="0.0" scale14="0.5" scale15="1.0" cutoff="9.0 * angstrom"
         switch_width="1.0 * angstrom" method="cutoff">
        <Atom smirks="[#1]-[#8X2H2+0:1]-[#1]" epsilon="0.1521 * mole**-1 * kilocalorie"
              id="n-tip3p-O" sigma="3.1507 * angstrom"></Atom>
        <Atom smirks="[#1:1]-[#8X2H2+0]-[#1]" epsilon="0 * mole**-1 * kilocalorie"
```

(continues on next page)

(continued from previous page)

```
            id="n-tip3p-H" sigma="1 * angstrom"></Atom>
    </vdW>
    <Electrostatics version="0.3" scale12="0.0" scale13="0.0" scale14="0.8333333333"
                    scale15="1.0" cutoff="9.0 * angstrom" switch_width="0.0 * angstrom"
                    method="PME"></Electrostatics>
    <LibraryCharges version="0.3">
        <LibraryCharge smirks="[#1]-[#8X2H2+0:1]-[#1]" charge1="-0.834 * elementary_charge"
                       id="q-tip3p-O"></LibraryCharge>
        <LibraryCharge smirks="[#1:1]-[#8X2H2+0]-[#1]" charge1="0.417 * elementary_charge"
                       id="q-tip3p-H"></LibraryCharge>
    </LibraryCharges>
</SMIRNOFF>
```

Note: TIP3P's geometry is specified entirely by constraints, but SMIRNOFF certainly supports a wide variety of bonded parameters and functional forms.

Note that this format specifies not just the individual parameters, but also their functional forms and units in very explicit terms. This both makes it easy to read and means that the correct implementation of each force is specifically defined, rather than being left up to the MD engine.

The complicated part is that each parameter is specified by a SMIRKS code. These codes are SMARTS codes with an optional numerical index on some atoms given after a colon. This indexing system comes from SMIRKS. Each parameter expects a certain number of indexed atoms, and applies the force accordingly. Unindexed atoms are used to match the chemistry, but forces are not applied to them. SMARTS/SMIRKS codes are less intimidating than they look; `[#1]` matches any Hydrogen atom (atomic number 1), while `[#8X2H2+0]` matches an oxygen atom (atomic number 8) with some additional constraints. Dashes represent bonds. So `[#1]-[#8X2H2+0:1]-[#1]` represents an oxygen atom indexed as 1 connected to two unindexed hydrogen atoms. This system allows individual parameters to be as general or as specific as needed.

Hint: This page is not the SMIRNOFF spec; it has been moved to the standards repository.

CHAPTER

EIGHT

VIRTUAL SITES

The Open Force Field Toolkit fully supports the SMIRNOFF virtual site specification for models using off-site charges, including 4- and 5-point water models, in addition to lone pair modelling on various functional groups. The primary focus is on the ability to add virtual sites to a system as part of system parameterization

Virtual sites are treated as massless particles whose positions are computed directly from the 3D coordinates of a set of atoms in the parent molecule. The number of atoms that are required to define the position will depend on the exact type of virtual site being used

Fig. 1: Examples of each type of virtual site with 'orientation' atoms colored blue and 'parent' atoms colored green.

Those atoms used to position the virtual site are referred to as 'orientation' atoms. Further, each type of virtual site will denote one of these orientation atoms to be the 'parent', which conceptually corresponds to the atom that the virtual site is 'attached to'.

8.1 Applying virtual site parameters

Virtual sites are incorporated into a force field by adding a [VirtualSites tag], which specifies both the parameters associated with the different virtual sites and how they should be applied to a molecule according to SMARTS-based rules.

As with all parameters in the SMIRNOFF specification, each virtual site parameter has an associated SMIRKS pattern with a number of atoms tagged with map indices. Each mapped atom corresponds to one of the atoms used to orientate the virtual site, and for all the currently supported types, the atom matched as atom `:1` is denoted the parent.

Fig. 2: The mappings between SMIRKS map indices to orientation particle

Virtual site parameters are applied by trying to match every associated SMIRKS pattern with the molecule of interest. In cases where multiple parameters *with the same* `name` would designate the same atom as a parent (i.e. an atom that a virtual site will be 'attached' to), then the last parameter to match will be assigned.

Fig. 3: The last parameter to match a particular parent atom wins. Here the monovalent lone parameter would be assigned rather than the bond charge parameter as it appears later in the parameter list.

In cases where the same parameter matches the same parent atom multiple times, as is the case in the above example of formaldehyde, the value of the `match` keyword will determine the outcome.

The "match" attribute accepts either "once" or "all_permutations", offering control for situations where a SMARTS pattern can possibly match the same group of atoms in different orders (either due to wildcards or local symmetry) and it is desired to either add just one or all of the possible virtual particles.

- once - only one of the possible matches will yield a virtual site being added to the system. This keyword is only valid for types virtual site whose coordinates are invariant to the ordering of the orientation atoms, e.g. the trivalent lone pair type, to avoid ambiguity as to which atom ordering to retain.
- all_permutations - all the possible matches will yield a virtual site being added to the system, such as in the monovalent lone pair example above.

If multiple parameters with different names would designate the same atom as a parent then the last matched parameter for each value of name would be assigned and yield a new virtual site being added.

Fig. 4: Multiple parameters can be used to create virtual sites on the same parent atom by giving them different names, e.g. in the case of a TIP6P model.

The following cases exemplify our reasoning in implementing this behavior, and should draw caution to complex issues that may arise when designing virtual site parameters. Let us consider 4-, 5-, and 6-point water models:

- A 4-point water model with a DivalentLonePair: This can be implemented by specifying match="once", outOfPlaneAngle="0*degree", and distance=-.15*angstrom". Since the SMIRKS pattern "[#1:1]-[#8X2:2]-[#1:3]" would match water twice and would create two particles in the exact same position if all_permutations was specified, we specify "once" to have only one particle generated. Although having two particles in the same position should not affect the physics if the proper exclusion policy is applied, it would effectively make the 4-point model just as expensive as 5-point models.
- A 5-point water model with a DivalentLonePair: This can be implemented by using match="all_permutations" (unlike the 4-point model), outOfPlaneAngle="56.26*degree, and distance=0.7*angstrom, for example. Here the permutations will cause particles to be placed at $\pm$56.26 degrees.
- A 6-point water model with both DivalentLonePair sites above. Since these two parameters look identical, it is unclear whether they should both be applied or if one should override the other. The toolkit never compares the physical numbers to determine equality as this can lead to instability during e.g. parameter fitting. To get this to work, we specify name="EP1" for the first parameter, and name="EP2" for the second parameter. This instructs the parameter handler keep them separate, and therefore both are applied. If both had the same name, then the typical SMIRNOFF hierarchy rules are used, and only the last matched parameter would be applied.

8.2 Ordering of atoms and virtual sites

The toolkit handles the orders the atoms and virtual sites in a topology in a specific manner for internal convenience.

Virtual sites are expected to be added *after all molecules in the topology are present*. This is because the Open Force Field Toolkit organizes a topology by placing all atoms first, then all virtual sites last. This differs from the OpenMM Modeller object, for example, which interleaves the order of atoms and virtual sites in such a way that all particles of a molecule are contiguous.

In addition, due to the fact that a virtual site may contain multiple particles coupled to single parameters, the toolkit makes a distinction between a virtual *site*, and a virtual *particle*. A virtual site may represent multiple virtual particles, so the total number of particles cannot be directly determined by simply summing

the number of atoms and virtual sites in a molecule. This is taken into account, however, and the `Molecule` and `Topology` classes both implement `particle` iterators.

Note: The distinction between a virtual site and virtual particle is due to be removed in a future version of the toolkit, and a 'virtual site' will simply refer to one massless particle placed on a parent atom rather than to a collection of massless particles.

Intramolecular interactions

The virtual site specification allows a virtual site section to define the policy that should be used to handle intramolecular interactions (exclusions). The toolkit currently only supports the `parents` policy as outline in the virtual site section of the SMIRNOFF specification, which states that each virtual site should inherit their 1-2, 1-3, 1-4, and 1-n exclusions directly from the parent atom.

CHAPTER

NINE

DEVELOPING FOR THE TOOLKIT

This guide is written with the understanding that our contributors are NOT professional software developers, but are instead computational chemistry trainees and professionals. With this in mind, we aim to use a minimum of bleeding-edge technology and alphabet soup, and we will define any potentially unfamiliar processes or technologies the first time they are mentioned. We enforce use of certain practices (tests, formatting, coverage analysis, documentation) primarily because they are worthwhile upfront investments in the long-term sustainability of this project. The resources allocated to this project will come and go, but we hope that following these practices will ensure that minimal developer time will maintain this software

far into the future.

The process of contributing to the OpenFF Toolkit is more than just writing code. Before contributing, it is a very good idea to start a discussion on the Issue tracker about the functionality you'd like to add. This Issue discussion will help us decide with you where in the codebase it should go, any overlapping efforts with other developers, and what the user experience should be. Please note that the OpenFF Toolkit is intended to be used primarily as one piece of larger workflows, and that simplicity and reliability are two of our primary goals. Often, the cost/benefit of new features must be discussed, as a complex codebase is harder to maintain. When new functionality is added to the OpenFF Toolkit, it becomes our responsibility to maintain it, so it's important that we understand contributed code and are in a position to keep it up to date.

9.1 Overview

9.1.1 Philosophy

- The *core functionality* of the OpenFF Toolkit is to combine an Open Force Field `ForceField` and `Topology` to create an OpenMM `System`.
- An OpenMM `System` contains *everything* needed to compute the potential energy of a system, except the coordinates and (optionally) box vectors.
- The OpenFF toolkit employs a modular "plugin" architecture wherever possible, providing a standard interface for contributed features.

9.1.2 Terminology

For high-level toolkit concepts and terminology important for both development and use of the Toolkit, see the *core concepts page*.

SMIRNOFF and the OpenFF Toolkit

SMIRNOFF data A hierarchical data structure that complies with the *SMIRNOFF specification*. This can be serialized in many formats, including XML (OFFXML). The subsections in a SMIRNOFF data source generally correspond to one energy term in the functional form of a force field.

Cosmetic attribute Data in a SMIRNOFF data source that does not correspond to a known attribute. These have no functional effect, but several programs use the extensibility of the OFFXML format to define additional attributes for their own use, and their workflows require the OFF toolkit to process the files while retaining these keywords.

Development Infrastructure

Continuous Integration (CI) Tests that run frequently while the code is undergoing changes, ensuring that the codebase still installs and has the intended behavior. Currently, we use a service called GitHub Actions for this. CI jobs run every time a commit is made to the `master` branch of the `openff-toolkit` GitHub repository or in a PR opened against it. These runs start by booting virtual machines that mimic brand new Linux and macOS computers. They then follow build instructions (see the `.github/workflows/CI.yml` file) to install the toolkit. After installing the OpenFF Toolkit and its dependencies, these virtual machines run our test suite. If the tests all pass, the build "passes" (returns a green check mark on GitHub).

If all the tests for a specific change to the `master` branch return green, then we know that the change has not broken the toolkit's existing functionality. When proposing code changes, we ask that contributors open a Pull Request (PR) on GitHub to merge their changes into the `master` branch. When a pull request is open, CI will run on the latest set of proposed changes and indicate whether they are safe to merge through status checks, summarized as a green check mark or red cross.

CodeCov An extension to our testing framework that reports the fraction of our source code lines that were run during the tests (our "code coverage"). This functionality is actually the combination of several components – GitHub Actions runners run the tests using `pytest` with the `pytest-cov` plugin, and then coverage reports are uploaded to CodeCov's website. This analysis is re-run alongside the rest of our CI, and a badge showing our coverage percentage is in the project README.

"Looks Good To Me" (LGTM) A service that analyzes the code in our repository for simple style and formatting issues. This service assigns a letter grade to the codebase, and a badge showing our LGTM report is in the project README.

ReadTheDocs (RTD) A service that compiles and renders the package's documentation (from the `docs/` folder). The documentation itself can be accessed from the ReadTheDocs badge in the README. It is compiled by RTD alongside the other CI checks, and the compiled documentation for a pull request can be viewed by clicking the "details" link after the status.

9.1.3 User Experience

One important aspect of how we make design decisions is by asking "who do we envision using this software, and what would they want it to do here?". There is a wide range of possible users, from non-chemists, to students/trainees, to expert computational medicinal chemists. We have decided to build functionality intended for use by *expert medicinal chemists*, and whenever possible, add fatal errors if the toolkit risks doing the wrong thing. So, for example, if a molecule is loaded with an odd ionization state, we assume that the user has input it this way intentionally.

This design philosophy inevitably has trade-offs — For example, the OpenFF Toolkit will give the user a hard time if they try to load a "dirty" molecule dataset, where some molecules have errors or are not described in enough detail for the toolkit to unambiguously parametrize them. If there is risk of misinterpreting the molecule (for example, bond orders being undefined or chiral centers without defined stereochemistry), the toolkit should raise an error that the user can override. In this regard we differ from RDKit, which is more permissive in the level of detail it requires when creating molecules. This makes sense for RDKit's use cases, as several of its analyses can operate with a lower level of detail about the molecules. Often, the same design decision is the best for all types of users, but when we do need to make trade-offs, we assume the user is an expert.

At the same time, we aim for "automagic" behavior whenever a decision will clearly go one way over another. System parameterization is an inherently complex topic, and the OFF toolkit would be nearly unusable if we required the user to explicitly approve every aspect of the process. For example, if a `Topology` has its `box_vectors` attribute defined, we assume that the resulting OpenMM `System` should be periodic.

9.2 Modular design features

There are a few areas where we've designed the toolkit with extensibility in mind. Adding functionality at these interfaces should be considerably easier than in other parts of the toolkit, and we encourage experimentation and contribution on these fronts.

These features have occasionally confusing names. "Parameter" here refers to a single value in a force field, as it is generally used in biophysics; it does not refer to an argument to a function. "Attribute" is used to refer to an XML attribute, which allows data to be defined for a particular tag; it does not refer to a member of a Python class or object. For example, in the following XML excerpt the `<SMIRNOFF>` tag has the attributes `version` and `aromaticity_model`:

```
<SMIRNOFF version="0.3" aromaticity_model="OEAroModel_MDL">
  ...
</SMIRNOFF>
```

"Member" is used here to describe Python attributes. This terminology is borrowed for the sake of clarity in this section from languages like C++ and Java.

9.2.1 `ParameterAttribute`

A `ParameterAttribute` is a single value that can be validated at runtime.

A `ParameterAttribute` can be instantiated as Python class or instance members to define the kinds of value that a particular parameter can take. They are used in the definitions of both `ParameterHandler` and `ParameterType`. The sorts of values a `ParameterAttribute` can take on are restricted by runtime validation. This validation is highly customizable, and may do things like allowing only certain values for a string or enforcing the correct units or array dimensions on the value; in fact, the validation can be defined using arbitrary code. The name of a `ParameterAttribute` should correspond exactly to the corresponding attribute in an OFFXML file.

`IndexedParameterAttribute`

An `IndexedParameterAttribute` is a `ParameterAttribute` with a sequence of values, rather than just one. Each value in the sequence is indexed by an integer.

The exact name of an `IndexedParameterAttribute` is NOT expected to appear verbatim in a OFFXML file, but instead should appear with a numerical integer suffix. For example the `IndexedParameterAttribute` `k` should only appear as `k1`, `k2`, `k3`, and so on in an OFFXML. The current implementation requires this indexing to start at 1 and subsequent values be contiguous (no skipping numbers), but does not enforce an upper limit on the integer.

For example, dihedral torsions are often parameterized as the sum of several sine wave terms. Each of the parameters of the sine wave `k`, `periodicity`, and `phase` is implemented as an `IndexedParameterAttribute`.

MappedParameterAttribute

A MappedParameterAttribute is a ParameterAttribute with several values, with some arbitrary mapping to access values.

IndexedMappedParameterAttribute

An IndexedMappedParameterAttribute is a ParameterAttribute with a sequence of maps of values.

9.2.2 ParameterHandler

ParameterHandler is a generic base class for objects that perform parameterization for one section in a SMIRNOFF data source. A ParameterHandler has the ability to produce one component of an OpenMM System. Extend this class to add a support for a new force or energy term to the toolkit.

Each ParameterHandler-derived class MUST implement the following methods and define the following attributes:

- Class members ParameterAttributes: These correspond to the header-level attributes in a SMIRNOFF data source. For example, the Bonds tag in the SMIRNOFF spec has an optional fractional_bondorder_method field, which corresponds to the line fractional_bondorder_method = ParameterAttribute(default=None) in the BondHandler class definition. The ParameterAttribute and IndexedParameterAttribute classes offer considerable flexibility for validating inputs. Defining these attributes at the class level implements the corresponding behavior in the default __init__ function.
- Class members _MIN_SUPPORTED_SECTION_VERSION and _MAX_SUPPORTED_SECTION_VERSION. ParameterHandler versions allow us to evolve ParameterHandler behavior in a controlled, recorded way. Force field development is experimental by nature, and it is unlikely that the initial choice of header attributes is suitable for all use cases. Recording the "versions" of a SMIRNOFF spec tag allows us to encode the default behavior and API of a specific generation of a ParameterHandler, while allowing the safe addition of new attributes and behaviors. If these attributes are not defined, defaults in the base class will apply and updates introducing new versions may break the existing code.

Each ParameterHandler-derived class MAY implement:

- _KWARGS: Keyword arguments passed to ForceField.create_openmm_system are validated against the _KWARGS lists of each ParameterHandler that the ForceField owns. If present, these keyword arguments and their values will be passed on to the ParameterHandler.
- _TAGNAME: The name of the SMIRNOFF OFFXML tag used to parameterize the class. This tag should appear in the top level within the <SMIRNOFF> tag; see the Parameter generators section of the SMIRNOFF specification.
- _INFOTYPE: The ParameterType subclass used to parse the elements in the ParameterHandler's parameter list.
- _DEPENDENCIES: A list of ParameterHandler subclasses that, when present, must run before this one. Note that this is *not* a list of ParameterHandler subclasses that are required by this one. Ideally, child classes of ParameterHandler are entirely independent, and energy components of a force field form distinct terms; when this is impossible, _DEPENDENCIES may be used to guarantee execution order.
- to_dict: converts the ParameterHandler to a hierarchical dict compliant with the SMIRNOFF specification. The default implementation of this function should suffice for most developers.
- check_handler_compatibility: Checks whether this ParameterHandler is "compatible" with another. This function is used when a ForceField is attempted to be constructed from *multiple* SMIRNOFF data sources, and it is necessary to check that two sections with the same tag name can be combined in

a sane way. For example, if the user instructed two `vdW` sections to be read, but the sections defined different vdW potentials, then this function should raise an Exception indicating that there is no safe way to combine the parameters. The default implementation of this function should suffice for most developers.

9.2.3 `ParameterType`

`ParameterType` is a base class for the SMIRKS-based parameters of a `ParameterHandler`. Extend this alongside `ParameterHandler` to define and validate the force field parameters of a new force. This is analogous to ParmEd's XType classes, like BondType. A `ParameterType` should correspond to a single SMARTS-based parameter.

For example, the Lennard-Jones potential can be parameterized through either the size `ParameterAttribute` `sigma` or `r_min`, alongside the energy `ParameterAttribute` `epsilon`. Both options are handled through the `vdWType` class, a subclass of `ParameterType`.

9.2.4 Non-bonded methods as implemented in OpenMM

The SMIRNOFF specification describes the contents of a force field, which can be implemented in a number of different ways in different molecular simulation engines. The OpenMM implementation provided by the OpenFF Toolkit either produces an `openmm.System` containing a `openmm.NonbondedForce` object or raises an exception depending on how the non-bonded parameters are specified. Exceptions are raised when parameters are incompatible with OpenMM (`IncompatibleParameterError`) or otherwise against spec (`SMIRNOFFSpecError`), and also when they are appropriate for the spec but not yet implemented in the toolkit (`SMIRNOFFSpecUnimplementedError`). This table describes which `NonbondedMethod` is used in the produced `NonbondedForce`, or else which exception is raised.

vdw_method	electrostatics_method	periodic	OpenMM Nonbonded method or exception	Common case
cutoff	Coulomb	True	raises `IncompatibleParameterError`	
cutoff	Coulomb	False	`openmm.NonbondedForce.NoCutoff`	
cutoff	reaction-field	True	raises `SMIRNOFFSpecUnimplementedError`	
cutoff	reaction-field	False	raises `SMIRNOFFSpecError`	
cutoff	PME	True	`openmm.NonbondedForce.PME`	*
cutoff	PME	False	`openmm.NonbondedForce.NoCutoff`	
LJPME	Coulomb	True	raises `IncompatibleParameterError`	
LJPME	Coulomb	False	`openmm.NonbondedForce.NoCutoff`	
LJPME	reaction-field	True	raises `IncompatibleParameterError`	
LJPME	reaction-field	False	raises `SMIRNOFFSpecError`	
LJPME	PME	True	`openmm.NonbondedForce.LJPME`	
LJPME	PME	False	`openmm.NonbondedForce.NoCutoff`	

Notes:

- The most commonly-used case (including the Parsley line) is in the fifth row (cutoff vdW, PME electrostatics, periodic topology) and marked with an asterisk.
- For all cases included a non-periodic topology, `openmm.NonbondedForce.NoCutoff` is currently used.
- Electrostatics method `reaction-field` can only apply to periodic systems, however it is not currently implemented.
- LJPME (particle mesh ewald for LJ/vdW interactions) is not yet fully described in the SMIRNOFF specification.
- In the future, the OpenFF Toolkit may create multiple `CustomNonbondedForce` objects in order to better de-couple vdW and electrostatic interactions.

9.3 Contributing

We always welcome GitHub pull requests. For bug fixes, major feature additions, or refactoring, please raise an issue on the GitHub issue tracker first to ensure the design will be amenable to current developer plans. Development of new toolkit features generally proceeds in the following stages:

- Begin a discussion on the GitHub issue tracker to determine big-picture "what should this feature do?" and "does it fit in the scope of the OpenFF Toolkit?"
  - "... typically, for existing water models, we want to assign library charges"
- Start identifying details of the implementation that will be clear from the outset
  - "Create a new "special section" in the SMIRNOFF format (kind of analogous to the BondChargeCorrections section) which allows SMIRKS patterns to specify use of library charges for specific groups
  - "Following #86, here's how library charges might work: ..."
- Create a branch or fork for development
  - The OpenFF Toolkit has one unusual aspect of its CI build process, which is that certain functionality requires the OpenEye toolkits, so the builds must contain a valid OpenEye license file. An OpenEye license is stored as an encrypted token within the `openforcefield` organization on GitHub. For security reasons, builds run from forks cannot access this key. Therefore, tests that depend on the OpenEye Toolkits will be skipped on forks. Contributions run on forks are still welcome, especially as features that do not interact directly with the OpenEye Toolktis are not likely affected by this limitation.

9.3.1 Setting up a development environment

1. Install the `conda` package manager as part of the Anaconda Distribution from here
2. Set up conda environment:

```
git clone https://github.com/openforcefield/openff-toolkit
cd openff-toolkit/
# Create a conda environment with dependencies from env/YAML file
conda env create -n openff-dev -f devtools/conda-envs/test_env.yaml
conda activate openff-dev
# Perform editable/dirty dev install
pip install -e .
```

3. Obtain and store Open Eye license somewhere like `~/.oe_license.txt`. Optionally store the path in environmental variable `OE_LICENSE`, i.e. using a command like `echo "export OE_LICENSE=/Users/yournamehere/.oe_license.txt" >> ~/.bashrc`

Building the Docs

The documentation is composed of two parts, a hand-written user guide and an auto-generated API reference. Both are compiled by Sphinx, and can be automatically served and regenerated on changes with `sphinx-autobuild`. Documentation for released versions is available at ReadTheDocs. ReadTheDocs also builds the documentation for each Pull Request opened on GitHub and keeps the output for 90 days.

To add the documentation dependencies to your existing `openff-dev` Conda environment:

```
# Add the documentation requirements to your Conda environment
conda env update --name openff-dev --file docs/environment.yml
conda install --name openff-dev -c conda-forge sphinx-autobuild
```

To build the documentation from scratch:

```
# Build the documentation
# From the openff-toolkit root directory
conda activate openff-dev
cd docs
make html
# Documentation can be found in docs/_build/html/index.html
```

To watch the source directory for changes and automatically rebuild the documentation and refresh your browser:

```
# Host the docs on a local HTTP server and rebuild when a source file is changed
# Works best when the docs have already been built
# From the openff-toolkit root directory
conda activate openff-dev
sphinx-autobuild docs docs/_build/html --watch openff
# Then navigate your web browser to http://localhost:8000
```

9.3.2 Style guide

Development for the `openff-toolkit` conforms to the recommendations given by the Software Development Best Practices for Computational Chemistry guide.

The naming conventions of classes, functions, and variables follows PEP8, consistently with the best practices guide. The naming conventions used in this library not covered by PEP8 are:

- Use `file_path`, `file_name`, and `file_stem` to indicate `path/to/stem.extension`, `stem.extension`, and `stem` respectively, consistent with the variables in the `pathlib` module of the standard library.
- Use `n_x` to abbreviate "number of x" (e.g. `n_atoms`, `n_molecules`).

We place a high priority on code cleanliness and readability, even if code could be written more compactly. For example, 15-character variable names are fine. Triply nested list comprehensions are not.

The `openff-toolkit` has adopted code formatting tools ("linters") to maintain consistent style and remove the burden of adhering to these standards by hand. Currently, two are employed:

1. Black, the uncompromising code formatter, automatically formats code with a consistent style.

2. isort, sorts imports

There is a step in CI that uses these tools to check for a consistent style (see the file `.github/workflows/lint.yml`). These checks will use the most recent versions of each linter. To ensure that changes follow these standards, you can install and run these tools locally:

```
conda install black isort -c conda-forge
black openff
isort openff
```

Anything not covered above is currently up to personal preference, but this may change as new linters are added.

9.3.3 Pre-commit

The `pre-commit` tool can *optionally* be used to automate some or all of the above style checks. It automatically runs other programs ("hooks") when you run `git commit`. It aborts the commit with an exit code if any of the hooks fail, i.e. if `black` reformats code. This project uses `pre-commit ci`, a free service that enforces style on GitHub using the `pre-commit` framework in CI.

To use `pre-commit` locally, first install it:

```
conda conda install pre-commit -c conda-forge  # also available via pip
```

Then, install the pre-commit hooks (note that it installs the linters into an isolated virtual environment, not the current conda environment):

```
pre-commit install
```

Hooks will now run automatically before commits. Once installed, they should run in a total of a few seconds.

If `pre-commit` is not used by the developer and style issues are found, a `pre-commit.ci` bot may commit directly to a PR to make these fixes. This bot should only ever alter styl and never make functional changes to code.

Note that tests (too slow) and type-checking (weird reasons) are not run by `pre-commit`. You should still manually run tests before commiting code.

9.4 Checking code coverage locally

Run something like

```
$ python -m pytest --cov=openff --cov-config=setup.cfg --cov-report html openff
$ open htmlcov/index.html
```

to see a code coverage report. This uses Coverage.py and also requires a pytest plugin (`pytest-cov`), both of which should be installed if you are using the provided conda environments. CodeCov provides this as an automated service with their own web frontend using a similar file generated by our CI. However, it can be useful to run this locally to check coverage changes (i.e. "is this function I added sufficiently coverged by tests?") without waiting for CI to complete.

9.5 Supported Python versions

The OpenFF Toolkit roughly follows NEP 29. As of version 0.11.0 (June 2022) this means Python 3.8-3.9 are officially supported. We develop, test, and distribute on macOS and Linux-based operating systems. We do not currently support Windows. Some CI builds run using only RDKit as a backend, some run using only OpenEye Toolkits, and some run using both installed at once.

The CI matrix is currently as follows:

	Linux			macOS		
	RDKit	OpenEye	RDKit + OE	RDKit	OpenEye	RDKit + OE
Python 3.7 and older	No support after 0.11.0 (March 2021)					
Python 3.8	**Test**	**Test**	**Test**	**Test**	**Test**	**Test**
Python 3.9	**Test**	Skip	Skip	**Test**	Skip	Skip
Python 3.10	**Test**	Skip	Skip	**Test**	Skip	Skip
Python 3.11 and newer	Pending official releases and/or upstream support					

CHAPTER

TEN

MOLECULE CONVERSION TO OTHER PACKAGES

Molecule conversion spec

10.1 Hierarchy data (chains and residues)

Note that the representations of hierarchy data (namely, chains and residues) in different software packages have different expectations. For example, in OpenMM, the atoms of a single residue must be contiguous. In RDKit, it is permissible to have an atom with no PDB residue information, whereas in OpenEye the fields must be populated. In most packages, it is expected that any atom with a residue name defined will also have a residue number.

The OpenFF toolkit does not have these restrictions, and records hierarchy metadata almost entirely for interoperability and user convenience reasons. No code paths in the OpenFF Toolkit consider hierarchy metadata during parameter assignment. While users should expect hierarchy metadata to be correctly handled in simple loading operations and export to other packages, *modifying* hierarchy metadata in the OpenFF Toolkit may lead to unexpected incompatibilities with other packages/representations.

Another consequence of this difference in representations is that hierarchy iterators (like `Molecule.residues` and `Topology.chains`) are not accessed during conversion to other packages. Only the underlying hierarchy metadata from the atoms is transferred, and the OpenFF Toolkit makes no attempt to match the behavior of iterators in other packages.

In cases where only *some* common metadata fields are set (but not others), the following calls happen during conversion TO other packages

- RDKit - We run `rdatom.SetPDBMetadata` if ANY of `residue_name`, `residue_number`, or `chain_id` are set on the OpenFF Atom. This means that, in cases where only one or two of those fields are filled, the others will be set to be the default values in RDKit
- OpenEye - We always run `oechem.OEAtomSetResidue(oe_atom, res)`. If the metadata values are not defined in OpenFF, we assign default values (`residue_name="UNL"`, `residue_number=1`, `chain_id=" "`)
- OpenMM - OpenMM *requires* identification of chains and residues when constructing a Topology, and residues must be entirely contained within their parent chain. Our Topology.to_openmm method creates at least one chain for each OpenFF Molecule. Contiguously-indexed atoms with the same `chain_id` value within a OpenFF Molecule will be assigned to a single OpenMM chain. Continuously indexed atoms with the same `residue_name`, `residue_number`, and `chain_id` will be assigned to the same OpenMM residue.

Toolkit	residue_name	residue_number	chain_id
PDB file ATOM/HETATM columns	Columns 18-20 (`resName`)	Columns 23-26 (`resSeq`)	Columns 22 (`chainID`)
PDBx/MMCIF fields	`label_comp_id`	`label_seq_id`	`label_asym_id`
OpenFF getter (defined)	`atom.metadata["residue_name"]`	`atom.metadata["residue_number"]`	`atom.metadata["chain_id"]`
OpenFF getter (undefined)	`"residue_name" not in atom.metadata`	`"residue_number" not in atom.metadata`	`"chain_id" not in atom.metadata`
OpenFF setter (defined)	`atom.metadata["residue_name"] = X`	`atom.metadata["residue_number"] = X`	`atom.metadata["chain_id"] = X`
OpenFF setter (undefined)	`del atom.metadata['residue_name']`	`del atom.metadata['residue_number']`	`del atom.metadata['chain_id']`
OpenMM getter (defined)	`omm_atom.residue.name`	`omm_atom.residue.id`	`omm_atom.residue.chain.id`
OpenMM getter (undefined)	All particles in an OpenMM Topology belong to a chain and residue	All particles in an OpenMM Topology belong to a chain and residue	All particles in an OpenMM Topology belong to a chain and residue
OpenMM setter (defined)	`omm_atom.residue.name = X`	`omm_atom.residue.id = X`	`omm_atom.residue.chain.id = X`
OpenMM setter (undefined)	`omm_atom.residue.name = "UNL"`	`omm_atom.residue.id = 0`	`omm_atom.residue.chain.id = "X"`
RDKit getter (defined)	`rda.GetPDBResidueInfo().GetResidueName()`	`rda.GetPDBResidueInfo().GetResidueNumber()`	`rda.GetPDBResidueInfo().GetChainId()`
RDKit getter (undefined)	`rda.GetPDBResidueInfo() is None`	`rda.GetPDBResidueInfo() is None`	`rda.GetPDBResidueInfo() is None`

RDKit setter (defined)	`res = rda.GetPDBResidueInfo()`
	`res.SetResidueName(X)`
	`rdatom.SetPDBResidueInfo(res)`
		`res = rda.GetPDBResidueInfo()`
		`res.SetResidueNumber(X)`
		`rdatom.SetPDBResidueInfo(res)`
			`.res = rda.GetPDBResidueInfo()`
			`res.SetSetChainId(X)`
			`rdatom.SetPDBResidueInfo(res)`

RDKit setter (if at least one field is defined, but not this one)	`res = rda.GetPDBResidueInfo()`	`res = rda.GetPDBResidueInfo()`	`res = rda.GetPDBResidueInfo()`

CHAPTER

ELEVEN

MOLECULAR TOPOLOGY REPRESENTATIONS

This module provides pure-Python classes for representing molecules and molecular systems. These classes offer several advantages over corresponding `Topology` objects in OpenMM and MDTraj, including offering serialization to a variety of standard formats (including XML, JSON, YAML, BSON, TOML, and MessagePack).

11.1 Primary objects

`FrozenMolecule`	Immutable chemical representation of a molecule, such as a small molecule or biopolymer.
`Molecule`	Mutable chemical representation of a molecule, such as a small molecule or biopolymer.
`Topology`	A Topology is a chemical representation of a system containing one or more molecules appearing in a specified order.

11.1.1 openff.toolkit.topology.FrozenMolecule

class openff.toolkit.topology.**FrozenMolecule**(*other=None*, *file_format=None*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *allow_undefined_stereo=False*)

Immutable chemical representation of a molecule, such as a small molecule or biopolymer.

Examples

Create a molecule from a sdf file

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = FrozenMolecule.from_file(sdf_filepath)
```

Convert to OpenEye OEMol object

```
>>> oemol = molecule.to_openeye()
```

Create a molecule from an OpenEye molecule

```
>>> molecule = FrozenMolecule.from_openeye(oemol)
```

Convert to RDKit Mol object

```
>>> rdmol = molecule.to_rdkit()
```

Create a molecule from an RDKit molecule

```
>>> molecule = FrozenMolecule.from_rdkit(rdmol)
```

Create a molecule from IUPAC name (requires the OpenEye toolkit)

```
>>> molecule = FrozenMolecule.from_iupac('imatinib')
```

Create a molecule from SMILES

```
>>> molecule = FrozenMolecule.from_smiles('Cc1ccccc1')
```

Warning: This API is experimental and subject to change.

__init__(*other=None*, *file_format=None*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *allow_undefined_stereo=False*)

Create a new FrozenMolecule object

Parameters

- **other** (optional, default=None) – If specified, attempt to construct a copy of the molecule from the specified object. This can be any one of the following:
  - a Molecule object
  - a file that can be used to construct a Molecule object
  - an openeye.oechem.OEMol
  - an rdkit.Chem.rdchem.Mol
  - a serialized Molecule object
- **file_format** (str, optional, default=None) – If providing a file-like object, you must specify the format of the data. If providing a file, the file format will attempt to be guessed from the suffix.
- **toolkit_registry** (a ToolkitRegistry or) – ToolkitWrapper object, optional, default=GLOBAL_TOOLKIT_REGISTRY ToolkitRegistry or ToolkitWrapper to use for I/O operations
- **allow_undefined_stereo** (bool, default=False) – If loaded from a file and False, raises an exception if undefined stereochemistry is detected during the molecule's construction.

Examples

Create an empty molecule:

```
>>> empty_molecule = FrozenMolecule()
```

Create a molecule from a file that can be used to construct a molecule, using either a filename or file-like object:

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = FrozenMolecule(sdf_filepath)
>>> molecule = FrozenMolecule(open(sdf_filepath, 'r'), file_format='sdf')
```

```
>>> import gzip
>>> mol2_gz_filepath = get_data_file_path('molecules/toluene.mol2.gz')
>>> molecule = FrozenMolecule(gzip.GzipFile(mol2_gz_filepath, 'r'), file_format=
↪'mol2')
```

Create a molecule from another molecule:

```
>>> molecule_copy = FrozenMolecule(molecule)
```

Convert to OpenEye OEMol object

```
>>> oemol = molecule.to_openeye()
```

Create a molecule from an OpenEye molecule:

```
>>> molecule = FrozenMolecule(oemol)
```

Convert to RDKit Mol object

```
>>> rdmol = molecule.to_rdkit()
```

Create a molecule from an RDKit molecule:

```
>>> molecule = FrozenMolecule(rdmol)
```

Convert the molecule into a dictionary and back again:

```
>>> serialized_molecule = molecule.to_dict()
>>> molecule_copy = Molecule(serialized_molecule)
```

Methods

`__init__`([other, file_format, ...])	Create a new FrozenMolecule object
`add_default_hierarchy_schemes`([...])	Adds `chain` and `residue` hierarchy schemes.
`add_hierarchy_scheme`(uniqueness_criteria, ...)	Use the molecule's metadata to facilitate iteration over its atoms.
`apply_elf_conformer_selection`([percentage, ...])	Select a set of diverse conformers from the molecule's conformers with ELF.

continues on next page

Table 1 – continued from previous page

are_isomorphic(mol1, mol2[, ...])	Determine if mol1 is isomorphic to mol2.
assign_fractional_bond_orders([...])	Update and store list of bond orders this molecule.
assign_partial_charges(partial_charge_method)	Calculate partial atomic charges and store them in the molecule.
atom(index)	Get the atom with the specified index.
atom_index(atom)	Returns the index of the given atom in this molecule
bond(index)	Get the bond with the specified index.
canonical_order_atoms([toolkit_registry])	Produce a copy of the molecule with the atoms reordered canonically.
chemical_environment_matches(query[, ...])	Find matches in the molecule for a SMARTS string or ChemicalEnvironment query
compute_partial_charges_am1bcc([...])	Deprecated since version 0.11.0.
delete_hierarchy_scheme(iter_name)	Remove an existing HierarchyScheme specified by its iterator name.
enumerate_protomers([max_states])	Enumerate the formal charges of a molecule to generate different protomoers.
enumerate_stereoisomers([undefined_only, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers([max_states, ...])	Enumerate the possible tautomers of the current molecule
find_rotatable_bonds([...])	Find all bonds classed as rotatable ignoring any matched to the ignore_functional_groups list.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(molecule_dict)	Create a new Molecule from a dictionary representation
from_file(file_path[, file_format, ...])	Create one or more molecules from a file
from_inchi(inchi[, allow_undefined_stereo, ...])	Construct a Molecule from a InChI representation
from_iupac(iupac_name[, toolkit_registry, ...])	Generate a molecule from IUPAC or common name
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mapped_smiles(mapped_smiles[, ...])	Create an Molecule from a mapped SMILES made with cmiles.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_openeye(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_pdb(file_path[, toolkit_registry])	Deprecated since version 0.11.0.
from_pdb_and_smiles(file_path, smiles[, ...])	Create a Molecule from a pdb file and a SMILES string using RDKit.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_polymer_pdb(file_path[, toolkit_registry])	Loads a polymer from a PDB file.

continues on next page

Table 1 – continued from previous page

from_qcschema(qca_record[, client, ...])	Create a Molecule from a QCArchive molecule record or dataset entry based on attached cmiles information.
from_rdkit(rdmol[, allow_undefined_stereo, ...])	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, ...])	Construct a Molecule from a SMILES representation
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_topology(topology)	Return a Molecule representation of an OpenFF Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit.
generate_unique_atom_names()	Generate unique atom names using element name and number of times that element has occurred e.g.
get_bond_between(i, j)	Returns the bond between two atoms
is_isomorphic_with(other, **kwargs)	Check if the molecule is isomorphic with the other molecule which can be an openff.toolkit.topology.Molecule or nx.Graph().
nth_degree_neighbors(n_degrees)	Return canonicalized pairs of atoms whose shortest separation is *exactly* n bonds.
ordered_connection_table_hash()	Compute an ordered hash of the atoms and bonds in the molecule
particle(index)	DEPRECATED: Use Molecule.atom instead.
particle_index(particle)	DEPRECATED: Use Molecule.atom_index instead.
remap(mapping_dict[, current_to_new])	Remap all of the indexes in the molecule to match the given mapping dict
strip_atom_stereochemistry(smarts[, ...])	Delete stereochemistry information for certain atoms, if it is present.
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(file_path, file_format[, ...])	Write the current molecule to a file or file-like object
to_hill_formula()	Generate the Hill formula of this molecule.
to_inchi([fixed_hydrogens, toolkit_registry])	Create an InChI string for the molecule using the requested toolkit backend.
to_inchikey([fixed_hydrogens, toolkit_registry])	Create an InChIKey for the molecule using the requested toolkit backend.
to_iupac([toolkit_registry])	Generate IUPAC name from Molecule
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_networkx()	Generate a NetworkX undirected graph from the molecule.

continues on next page

Table 1 – continued from previous page

to_openeye([toolkit_registry, aromaticity_model])	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Create a QCElemental Molecule.
to_rdkit([aromaticity_model, toolkit_registry])	Create an RDKit molecule
to_smiles([isomeric, explicit_hydrogens, ...])	Return a canonical isomeric SMILES representation of the current molecule.
to_toml()	Return a TOML serialized representation.
to_topology()	Return an OpenFF Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.
update_hierarchy_schemes([iter_names])	Infer a hierarchy from atom metadata according to the existing hierarchy schemes.

Attributes

amber_impropers	Iterate over all impropers with trivalent centers, reporting the central atom first.
angles	Get an iterator over all i-j-k angles.
atoms	Iterate over all Atom objects in the molecule.
bonds	Iterate over all Bond objects in the molecule.
conformers	Returns the list of conformers for this molecule.
has_unique_atom_names	True if the molecule has unique atom names, False otherwise.
hierarchy_schemes	The hierarchy schemes available on the molecule.
hill_formula	Get the Hill formula of the molecule
impropers	Iterate over all improper torsions in the molecule.
n_angles	Number of angles in the molecule.
n_atoms	The number of Atom objects.
n_bonds	The number of Bond objects in the molecule.
n_conformers	The number of conformers for this molecule.
n_impropers	Number of possible improper torsions in the molecule.
n_particles	Use Molecule.n_atoms instead.
n_propers	Number of proper torsions in the molecule.
name	The name (or title) of the molecule
partial_charges	Returns the partial charges (if present) on the molecule.
particles	Use Molecule.atoms instead.
propers	Iterate over all proper torsions in the molecule
properties	The properties dictionary of the molecule
smirnoff_impropers	Iterate over all impropers with trivalent centers, reporting the central atom second.
torsions	Get an iterator over all i-j-k-l torsions.
total_charge	Return the total charge on the molecule

property has_unique_atom_names: bool

True if the molecule has unique atom names, False otherwise.

generate_unique_atom_names()

Generate unique atom names using element name and number of times that element has occurred e.g. 'C1x', 'H1x', 'O1x', 'C2x', ...

The character 'x' is appended to these generated names to reduce the odds that they clash with an atom name or type imported from another source.

strip_atom_stereochemistry(*smarts*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Delete stereochemistry information for certain atoms, if it is present. This method can be used to "normalize" molecules imported from different cheminformatics toolkits, which differ in which atom centers are considered stereogenic.

Parameters

- **smarts** (`str` or `ChemicalEnvironment`) – Tagged SMARTS with a single atom with index 1. Any matches for this atom will have any assigned stereocheistry information removed.
- **toolkit_registry** (a `ToolkitRegistry` or `ToolkitWrapper` object, optional,) – default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use for I/O operations

to_dict()

Return a dictionary representation of the molecule.

Returns **molecule_dict** (*OrderedDict*) – A dictionary representation of the molecule.

ordered_connection_table_hash()

Compute an ordered hash of the atoms and bonds in the molecule

classmethod **from_dict**(*molecule_dict*)

Create a new Molecule from a dictionary representation

Parameters **molecule_dict** (`OrderedDict`) – A dictionary representation of the molecule.

Returns **molecule** (*Molecule*) – A Molecule created from the dictionary representation

add_default_hierarchy_schemes(*overwrite_existing=True*)

Adds `chain` and `residue` hierarchy schemes.

The Open Force Field Toolkit has no native understanding of hierarchical atom organisation schemes common to other biomolecular software, such as "residues" or "chains" (see *Hierarchy data (chains and residues)*). Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see `HierarchyScheme`.

If a `Molecule` with the default hierarchy schemes changes, `Molecule.update_hierarchy_schemes()` must be called before the residues or chains are iterated over again or else the iteration may be incorrect.

Parameters **overwrite_existing** (`bool`, `default=True`) – Whether to overwrite existing instances of the *residue* and *chain* hierarchy schemes. If this is `False` and either of the hierarchy schemes are already defined on this molecule, an exception will be raised.

Raises **HierarchySchemeWithIteratorNameAlreadyRegisteredException** – When `overwrite_existing=False` and either the `chains` or `residues` hierarchy scheme is already configured.

add_hierarchy_scheme(*uniqueness_criteria*, *iterator_name*)

Use the molecule's metadata to facilitate iteration over its atoms.

This method will add an attribute with the name given by the `iterator_name` argument that provides an iterator over groups of atoms. Atoms are grouped by the values in their `atom.metadata` dictionary; any atoms with the same values for the keys given in the `uniqueness_criteria` argument will be in the same group. These groups have the type `HierarchyElement`.

Hierarchy schemes are not updated dynamically; if a `Molecule` with hierarchy schemes changes, `Molecule.update_hierarchy_schemes()` must be called before the scheme is iterated over again or else the grouping may be incorrect.

Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see `HierarchyScheme`.

Parameters

- **uniqueness_criteria** (`tuple of str`) – The names of `Atom` metadata entries that define this scheme. An atom belongs to a `HierarchyElement` only if its metadata has the same values for these criteria as the other atoms in the `HierarchyElement`.
- **iterator_name** (`str`) – Name of the iterator that will be exposed to access the hierarchy elements generated by this scheme.

Returns **new_hier_scheme** (*openff.toolkit.topology.HierarchyScheme*) – The newly created HierarchyScheme

property hierarchy_schemes: Dict[str, HierarchyScheme]

The hierarchy schemes available on the molecule.

Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see `HierarchyScheme`.

Returns **A dict of the form {str** (*HierarchyScheme*}) – The HierarchySchemes associated with the molecule.

delete_hierarchy_scheme(*iter_name*)

Remove an existing `HierarchyScheme` specified by its iterator name.

Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see `HierarchyScheme`.

Parameters **iter_name** (`str`) –

update_hierarchy_schemes(*iter_names=None*)

Infer a hierarchy from atom metadata according to the existing hierarchy schemes.

Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see `HierarchyScheme`.

Parameters **iter_names** (`Iterable of str, Optional`) – Only perceive hierarchy for HierarchySchemes that expose these iterator names. If not provided, all known hierarchies will be perceived, overwriting previous results if applicable.

to_smiles(*isomeric=True*, *explicit_hydrogens=True*, *mapped=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Return a canonical isomeric SMILES representation of the current molecule. A partially mapped smiles can also be generated for atoms of interest by supplying an *atom_map* to the properties dictionary.

Note: RDKit and OpenEye versions will not necessarily return the same representation.

Parameters

- **isomeric** (bool optional, default= True) – return an isomeric smiles
- **explicit_hydrogens** (bool optional, default=True) – return a smiles string containing all hydrogens explicitly
- **mapped** (bool optional, default=False) – return a explicit hydrogen mapped smiles, the atoms to be mapped can be controlled by supplying an atom map into the properties dictionary. If no mapping is passed all atoms will be mapped in order, else an atom map dictionary from the current atom index to the map id should be supplied with no duplicates. The map ids (values) should start from 0 or 1.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for SMILES conversion

Returns **smiles** (*str*) – Canonical isomeric explicit-hydrogen SMILES

Examples

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> smiles = molecule.to_smiles()
```

classmethod **from_inchi**(*inchi*, *allow_undefined_stereo=False*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*)

Construct a Molecule from a InChI representation

Parameters

- **inchi** (str) – The InChI representation of the molecule.
- **allow_undefined_stereo** (bool, default=False) – Whether to accept InChI with undefined stereochemistry. If False, an exception will be raised if a InChI with undefined stereochemistry is passed into this function.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for InChI-to-molecule conversion

Returns **molecule** (*openff.toolkit.topology.Molecule*)

Examples

Make cis-1,2-Dichloroethene:

```
>>> molecule = Molecule.from_inchi('InChI=1S/C2H2Cl2/c3-1-2-4/h1-2H/b2-1-')
```

to_inchi(*fixed_hydrogens=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Create an InChI string for the molecule using the requested toolkit backend. InChI is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

- **fixed_hydrogens** (bool, default=False) – If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for molecule-to-InChI conversion

Returns **inchi** (*str*) – The InChI string of the molecule.

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

to_inchikey(*fixed_hydrogens=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Create an InChIKey for the molecule using the requested toolkit backend. InChIKey is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

- **fixed_hydrogens** (bool, default=False) – If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for molecule-to-InChIKey conversion

Returns **inchi_key** (*str*) – The InChIKey representation of the molecule.

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

classmethod from_smiles(*smiles*, *hydrogens_are_explicit=False*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*)

Construct a Molecule from a SMILES representation

Parameters

- **smiles** (str) – The SMILES representation of the molecule.

- **hydrogens_are_explicit** (bool, default = False) – If False, the cheminformatics toolkit will perform hydrogen addition
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** (bool, default=False) – Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

Returns **molecule** (*openff.toolkit.topology.Molecule*)

Examples

```
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
```

static are_isomorphic(*mol1*, *mol2*, *return_atom_map=False*, *aromatic_matching=True*, *formal_charge_matching=True*, *bond_order_matching=True*, *atom_stereochemistry_matching=True*, *bond_stereochemistry_matching=True*, *strip_pyrimidal_n_atom_stereo=True*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Determine if mol1 is isomorphic to mol2.

are_isomorphic() compares two molecule's graph representations and the chosen node/edge attributes. Connections and atomic numbers are always checked.

If nx.Graphs() are given they must at least have atomic_number attributes on nodes. Other attributes that are_isomorphic() can optionally check...

- ... in nodes are:
  - is_aromatic
  - formal_charge
  - stereochemistry
- ... in edges are:
  - is_aromatic
  - bond_order
  - stereochemistry

By default, all attributes are checked, but stereochemistry around pyrimidal nitrogen is ignored.

Warning: This API is experimental and subject to change.

Parameters

- **mol1** (an openff.toolkit.topology.molecule.FrozenMolecule or nx.Graph()) – The first molecule to test for isomorphism.
- **mol2** (an openff.toolkit.topology.molecule.FrozenMolecule or nx.Graph()) – The second molecule to test for isomorphism.
- **return_atom_map** (bool, default=False, optional) – Return a dict containing the atomic mapping instead of a bool.

- **aromatic_matching** (bool, default=True, optional) – If False, aromaticity of graph nodes and edges are ignored for the purpose of determining isomorphism.
- **formal_charge_matching** (bool, default=True, optional) – If False, formal charges of graph nodes are ignored for the purpose of determining isomorphism.
- **bond_order_matching** (bool, default=True, optional) – If False, bond orders of graph edges are ignored for the purpose of determining isomorphism.
- **atom_stereochemistry_matching** (bool, default=True, optional) – If False, atoms' stereochemistry is ignored for the purpose of determining isomorphism.
- **bond_stereochemistry_matching** (bool, default=True, optional) – If False, bonds' stereochemistry is ignored for the purpose of determining isomorphism.
- **strip_pyrimidal_n_atom_stereo** (bool, default=True, optional) – If True, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for removing stereochemistry from pyrimidal nitrogens.

Returns

- **molecules_are_isomorphic** (*bool*)
- **atom_map** (*default=None, Optional,*) – [Dict[int,int]] ordered by mol1 indexing {mol1_index: mol2_index} If molecules are not isomorphic given input arguments, will return None instead of dict.

is_isomorphic_with(*other*, ***kwargs*)

Check if the molecule is isomorphic with the other molecule which can be an openff.toolkit.topology.Molecule or nx.Graph(). Full matching is done using the options described bellow.

> **Warning:** This API is experimental and subject to change.

Parameters

- **other** (Molecule or nx.Graph()) –
- **aromatic_matching** (bool, default=True, optional) –
- **atoms.** (compare the formal charges attributes of the) –
- **formal_charge_matching** (bool, default=True, optional) –
- **atoms.** –
- **bond_order_matching** (bool, deafult=True, optional) –
- **bonds.** (compare the bond order on attributes of the) –
- **atom_stereochemistry_matching** (bool, default=True, optional) – If False, atoms' stereochemistry is ignored for the purpose of determining equality.
- **bond_stereochemistry_matching** (bool, default=True, optional) – If False, bonds' stereochemistry is ignored for the purpose of determining equality.

- **strip_pyrimidal_n_atom_stereo** (bool, default=True, optional) – If True, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for removing stereochemistry from pyrimidal nitrogens.

Returns **isomorphic** (*bool*)

generate_conformers(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *n_conformers=10*, *rms_cutoff=None*, *clear_existing=True*, *make_carboxylic_acids_cis=True*)

Generate conformers for this molecule using an underlying toolkit.

If n_conformers=0, no toolkit wrapper will be called. If n_conformers=0 and clear_existing=True, molecule.conformers will be set to None.

Parameters

- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion
- **n_conformers** (int, default=1) – The maximum number of conformers to produce
- **rms_cutoff** (openmm.unit.Quantity-wrapped float, in units of distance, optional, default=None) – The minimum RMS value at which two conformers are considered redundant and one is deleted. Precise implementation of this cutoff may be toolkit-dependent. If None, the cutoff is set to be the default value for each ToolkitWrapper (generally 1 Angstrom).
- **clear_existing** (bool, default=True) – Whether to overwrite existing conformers for the molecule
- **make_carboxylic_acids_cis** (bool, default=True) – Guarantee all conformers have exclusively cis carboxylic acid groups (COOH) by rotating the proton in any trans carboxylic acids 180 degrees around the C-O bond. Works around a bug in conformer generation by the OpenEye toolkit where trans COOH is much more common than it should be.

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers()
```

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

apply_elf_conformer_selection(*percentage: float = 2.0*, *limit: int = 10*, *toolkit_registry: Optional[Union[ToolkitRegistry, ToolkitWrapper]] = GLOBAL_TOOLKIT_REGISTRY*, ***kwargs*)

Select a set of diverse conformers from the molecule's conformers with ELF.

Applies the Electrostatically Least-interacting Functional groups method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from the molecule's conformers.

Parameters

- **toolkit_registry** – The underlying toolkit to use to select the ELF conformers.
- **percentage** – The percentage of conformers with the lowest electrostatic interaction energies to greedily select from.
- **limit** – The maximum number of conformers to select.

Notes

- The input molecule should have a large set of conformers already generated to select the ELF conformers from.
- The selected conformers will be retained in the *conformers* list while unselected conformers will be discarded.

See also:

openff.toolkit.utils.toolkits.OpenEyeToolkitWrapper.apply_elf_conformer_selection, openff.toolkit.utils.toolkits.RDKitToolkitWrapper.apply_elf_conformer_selection

compute_partial_charges_am1bcc(*use_conformers=None*, *strict_n_conformers=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0 and will soon be removed. Use `assign_partial_charges(partial_charge_method='am1bcc')` instead.

Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit and assign them to this molecule's `partial_charges` attribute.

Parameters

- **strict_n_conformers** (`bool`, `default=False`) – Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.
- **use_conformers** (`iterable of openmm.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3)`) – and dimension of distance. Optional, default=None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers for the given charge method will be generated.
- **toolkit_registry** (`ToolkitRegistry`) –
- **openff.toolkit.utils.toolkits.ToolkitWrapper** (`or`) – `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation
- **optional** – `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation
- **default=None** – `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers()
>>> molecule.compute_partial_charges_am1bcc()
```

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

assign_partial_charges(*partial_charge_method: str*, *strict_n_conformers=False*, *use_conformers=None*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *normalize_partial_charges=True*)

Calculate partial atomic charges and store them in the molecule.

`assign_partial_charges` computes charges using the specified toolkit and assigns the new values to the `partial_charges` attribute. Supported charge methods vary from toolkit to toolkit, but some supported methods are:

- `"am1bcc"`
- `"am1bccelf10"` (requires OpenEye Toolkits)
- `"am1-mulliken"`
- `"mmff94"`
- `"gasteiger"`

For more supported charge methods and details, see the corresponding methods in each toolkit wrapper:

- `OpenEyeToolkitWrapper.assign_partial_charges`
- `RDKitToolkitWrapper.assign_partial_charges`
- `AmberToolsToolkitWrapper.assign_partial_charges`
- `BuiltInToolkitWrapper.assign_partial_charges`

Parameters

- **partial_charge_method** (`string`) – The partial charge calculation method to use for partial charge calculation.
- **strict_n_conformers** (`bool`, `default=False`) – Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.
- **use_conformers** (`iterable of openmm.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and`) – dimension of distance. Optional, default=None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **toolkit_registry** (`openff.toolkit.utils.toolkits.ToolkitRegistry or`) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation.
- **normalize_partial_charges** (`bool`, `default=True`) – Whether to offset partial charges so that they sum to the total formal charge of the molecule. This is used to prevent accumulation of rounding errors when the partial charge assignment method returns values at limited precision.

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.assign_partial_charges('am1-mulliken')
```

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

See also:

openff.toolkit.utils.toolkits.OpenEyeToolkitWrapper.assign_partial_charges, openff.toolkit.utils.toolkits.RDKitToolkitWrapper.assign_partial_charges, openff.toolkit.utils.toolkits.AmberToolsToolkitWrapper.assign_partial_charges, openff.toolkit.utils.toolkits.BuiltInToolkitWrapper.assign_partial_charges

assign_fractional_bond_orders(*bond_order_model=None*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *use_conformers=None*)

Update and store list of bond orders this molecule.

Bond orders are stored on each bond, in the `bond.fractional_bond_order` attribute.

Warning: This API is experimental and subject to change.

Parameters

- **toolkit_registry** (`openff.toolkit.utils.toolkits.ToolkitRegistry` or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion
- **bond_order_model** (`string, optional. Default=None`) – The bond order model to use for fractional bond order calculation. If `None`, `"am1-wiberg"` is used.
- **use_conformers** (`iterable of openmm.unit.Quantity(np.array) with shape (n_atoms, 3) and dimension of distance,`) – optional, default=None The conformers to use for fractional bond order calculation. If `None`, an appropriate number of conformers will be generated by an available `ToolkitWrapper`.

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.assign_fractional_bond_orders()
```

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

to_networkx()

Generate a NetworkX undirected graph from the molecule.

Nodes are Atoms labeled with atom indices and atomic elements (via the `element` node atrribute). Edges denote chemical bonds between Atoms.

Returns **graph** (*networkx.Graph*) – The resulting graph, with nodes (atoms) labeled with atom indices, elements, stereochemistry and aromaticity flags and bonds with two atom indices, bond order, stereochemistry, and aromaticity flags

Examples

Retrieve the bond graph for imatinib (OpenEye toolkit required)

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> nxgraph = molecule.to_networkx()
```

find_rotatable_bonds(*ignore_functional_groups=None*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Find all bonds classed as rotatable ignoring any matched to the `ignore_functional_groups` list.

Parameters

- **ignore_functional_groups** (`optional, List[str], default=None,`) – A list of bond SMARTS patterns to be ignored when finding rotatable bonds.
- **toolkit_registry** (`ToolkitRegistry`) – or openff.toolkit.utils.toolkits.ToolkitWrapperl, optional, default=None `ToolkitRegistry` or `ToolkitWrapper` to use for SMARTS matching

Returns **bonds** (*List[openff.toolkit.topology.molecule.Bond]*) – The list of openff.toolkit.topology.molecule.Bond instances which are rotatable.

property partial_charges

Returns the partial charges (if present) on the molecule.

Returns **partial_charges** (*a openmm.unit.Quantity - wrapped numpy array [1 x n_atoms] or None*) – The partial charges on the molecule's atoms. Returns None if no charges have been specified.

property n_particles: int

Use Molecule.n_atoms instead.

Type DEPRECATED

property n_atoms: int

The number of Atom objects.

property n_bonds

The number of Bond objects in the molecule.

property n_angles: int

Number of angles in the molecule.

property n_propers: int

Number of proper torsions in the molecule.

property n_impropers: int

Number of possible improper torsions in the molecule.

property particles: List[Atom]

Use Molecule.atoms instead.

Type DEPRECATED

particle(*index: int*) → *Atom*

DEPRECATED: Use Molecule.atom instead.

particle_index(*particle:* Atom) → int

DEPRECATED: Use Molecule.atom_index instead.

property atoms

Iterate over all Atom objects in the molecule.

atom(*index: int*) → *Atom*

Get the atom with the specified index.

Parameters **index** (int) –

Returns **atom** (*openff.toolkit.topology.Atom*)

atom_index(*atom:* Atom) → int

Returns the index of the given atom in this molecule

Parameters **atom** (Atom) –

Returns **index** (*int*) – The index of the given atom in this molecule

property conformers

Returns the list of conformers for this molecule.

Conformers are presented as a list of `Quantity`-wrapped NumPy arrays, of shape (3 x n_atoms) and with dimensions of [Distance]. The return value is the actual list of conformers, and changes to the contents affect the original `FrozenMolecule`.

property n_conformers: int

The number of conformers for this molecule.

property bonds: List[Bond]

Iterate over all Bond objects in the molecule.

bond(*index: int*) → *Bond*

Get the bond with the specified index.

Parameters **index** (int) –

Returns **bond** (*openff.toolkit.topology.Bond*)

property angles: Set[Tuple[Atom, Atom, Atom]]

Get an iterator over all i-j-k angles.

property torsions: Set[Tuple[Atom, Atom, Atom, Atom]]

Get an iterator over all i-j-k-l torsions. Note that i-j-k-i torsions (cycles) are excluded.

Returns **torsions** (*iterable of 4-Atom tuples*)

property propers: Set[Tuple[Atom, Atom, Atom, Atom]]

Iterate over all proper torsions in the molecule

property impropers: Set[Tuple[Atom, Atom, Atom, Atom]]

Iterate over all improper torsions in the molecule.

Returns **impropers** (*set of tuple*) – An iterator of tuples, each containing the atoms making up a possible improper torsion.

See also:

smirnoff_impropers, amber_impropers

property smirnoff_impropers: Set[Tuple[Atom, Atom, Atom, Atom]]

Iterate over all impropers with trivalent centers, reporting the central atom second.

The central atom is reported second in each torsion. This method reports an improper for each trivalent atom in the molecule, whether or not any given force field would assign it improper torsion parameters.

Also note that this will return 6 possible atom orderings around each improper center. In current SMIRNOFF parameterization, three of these six orderings will be used for the actual assignment of the improper term and measurement of the angles. These three orderings capture the three unique angles that could be calculated around the improper center, therefore the sum of these three terms will always return a consistent energy.

The exact three orderings that will be applied during parameterization can not be determined in this method, since it requires sorting the atom indices, and those indices may change when this molecule is added to a Topology.

For more details on the use of three-fold ('trefoil') impropers, see https://openforcefield.github.io/standards/standards/smirnoff/#impropertorsions

Returns impropers (*set of tuple*) – An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed second in each tuple.

See also:

impropers, amber_impropers

property amber_impropers: Set[Tuple[Atom, Atom, Atom, Atom]]

Iterate over all impropers with trivalent centers, reporting the central atom first.

The central atom is reported first in each torsion. This method reports an improper for each trivalent atom in the molecule, whether or not any given force field would assign it improper torsion parameters.

Also note that this will return 6 possible atom orderings around each improper center. In current AMBER parameterization, one of these six orderings will be used for the actual assignment of the improper term and measurement of the angle. This method does not encode the logic to determine which of the six orderings AMBER would use.

Returns impropers (*set of tuple*) – An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed first in each tuple.

See also:

impropers, smirnoff_impropers

nth_degree_neighbors(*n_degrees*)

Return canonicalized pairs of atoms whose shortest separation is *exactly* n bonds. Only pairs with increasing atom indices are returned.

Parameters n (int) – The number of bonds separating atoms in each pair

Returns neighbors (*iterator of tuple of Atom*) – Tuples (len 2) of atom that are separated by n bonds.

Notes

The criteria used here relies on minimum distances; when there are multiple valid paths between atoms, such as atoms in rings, the shortest path is considered. For example, two atoms in "meta" positions with respect to each other in a benzene are separated by two paths, one length 2 bonds and the other length 4 bonds. This function would consider them to be 2 apart and would not include them if `n=4` was passed.

property total_charge

Return the total charge on the molecule

property name: str

The name (or title) of the molecule

property properties: Dict[str, Any]

The properties dictionary of the molecule

property hill_formula: str

Get the Hill formula of the molecule

to_hill_formula() → str

Generate the Hill formula of this molecule.

Returns formula (*the Hill formula of the molecule*)

:raises NotImplementedError : if the molecule is not of one of the specified types.:

chemical_environment_matches(*query*, *unique=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Find matches in the molecule for a SMARTS string or `ChemicalEnvironment` query

Parameters

- **query** (`str` or `ChemicalEnvironment`) – SMARTS string (with one or more tagged atoms) or `ChemicalEnvironment` query. Query will internally be resolved to SMIRKS using `query.asSMIRKS()` if it has an `.asSMIRKS` method.
- **toolkit_registry** (`ToolkitRegistry`) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use for chemical environment matches

Returns matches (*list of atom index tuples*) – A list of tuples, containing the indices of the matching atoms.

Examples

Retrieve all the carbon-carbon bond matches in a molecule

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> matches = molecule.chemical_environment_matches('[#6X3:1]~[#6X3:2]')
```

classmethod from_iupac(*iupac_name*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*, ***kwargs*)

Generate a molecule from IUPAC or common name

Note: This method requires the OpenEye toolkit to be installed.

Parameters

- **iupac_name** (str) – IUPAC name of molecule to be generated
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY ToolkitRegistry or ToolkitWrapper to use for chemical environment matches
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if molecule contains undefined stereochemistry.

Returns **molecule** (*Molecule*) – The resulting molecule with position

Examples

Create a molecule from an IUPAC name

```
>>> molecule = Molecule.from_iupac('4-[(4-methylpiperazin-1-yl)methyl]-N-(4-methyl-
↪3-{[4-(pyridin-3-yl)pyrimidin-2-yl]amino}phenyl)benzamide')  # noqa
```

Create a molecule from a common name

```
>>> molecule = Molecule.from_iupac('imatinib')
```

to_iupac(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Generate IUPAC name from Molecule

Returns

- **iupac_name** (*str*) – IUPAC name of the molecule
- *.. note :: This method requires the OpenEye toolkit to be installed.*

Examples

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> iupac_name = molecule.to_iupac()
```

classmethod from_topology(*topology*)

Return a Molecule representation of an OpenFF Topology containing a single Molecule object.

Parameters **topology** (Topology) – The Topology object containing a single Molecule object. Note that OpenMM and MDTraj Topology objects are not supported.

Returns **molecule** (*openff.toolkit.topology.Molecule*) – The Molecule object in the topology

Raises ValueError – If the topology does not contain exactly one molecule.

Examples

Create a molecule from a Topology object that contains exactly one molecule

```
>>> molecule = Molecule.from_topology(topology)
```

to_topology()

Return an OpenFF Topology representation containing one copy of this molecule

Returns **topology** (*openff.toolkit.topology.Topology*) – A Topology representation of this molecule

Examples

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> topology = molecule.to_topology()
```

classmethod **from_file**(*file_path*, *file_format=None*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*)

Create one or more molecules from a file

Parameters

- **file_path** (`str or file-like object`) – The path to the file or file-like object to stream one or more molecules from.
- **file_format** (`str, optional, default=None`) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for your loaded toolkits for details.
- **toolkit_registry** (`openff.toolkit.utils.toolkits.ToolkitRegistry` or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use for file loading. If a Toolkit is passed, only the highest-precedence toolkit is used
- **allow_undefined_stereo** (`bool, default=False`) – If false, raises an exception if oemol contains undefined stereochemistry.

Returns **molecules** (*Molecule or list of Molecules*) – If there is a single molecule in the file, a Molecule is returned; otherwise, a list of Molecule objects is returned.

Examples

```
>>> from openff.toolkit.tests.utils import get_monomer_mol2_file_path
>>> mol2_file_path = get_monomer_mol2_file_path('cyclohexane')
>>> molecule = Molecule.from_file(mol2_file_path)
```

classmethod **from_pdb**(*file_path*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*)

Deprecated since version 0.11.0: `from_pdb` is deprecated and will soon be removed. Use `from_polymer_pdb()` instead.

classmethod from_polymer_pdb(*file_path: ~typing.Union[str, ~typing.TextIO], toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*)

Loads a polymer from a PDB file.

Currently only supports proteins with canonical amino acids that are either uncapped or capped by ACE/NME groups, but may later be extended to handle other common polymers, or accept user-defined polymer templates.

Metadata such as residues, chains, and atom names are recorded in the `Atom.properties` attribute, which is a dictionary mapping from strings like "residue" to the appropriate value. `from_polymer_pdb` returns a molecule that can be iterated over with the `.residues` and `.chains` attributes, as well as the usual `.atoms`.

This method proceeds in the following order:

- Loads the polymer substructure template file (distributed with the OpenFF Toolkit)
- Loads the PDB into an OpenMM `openmm.app.PDBFile` object
- Turns OpenMM topology into a temporarily invalid rdkit Molecule
- **Adds chemical information to the molecule:**
  - **For each substructure loaded from the substructure template file**
    - Uses rdkit to find matches between the substructure and the molecule
    - For any matches, assigns the atom formal charge and bond order info from the substructure to the rdkit molecule, then marks the atoms and bonds as having been assigned so they can not be overwritten by subsequent isomorphisms
- Take coordinates from the OpenMM Topology and add them as a conformer
- Convert the rdkit Molecule to OpenFF

Parameters

- **file_path** (`str or file object`) – PDB information to be passed to OpenMM PDBFile object for loading
- **None** (`toolkit_registry = ToolkitWrapper or ToolkitRegistry. Default =`) – Either a ToolkitRegistry, ToolkitWrapper

Returns molecule (*openff.toolkit.topology.Molecule*)

to_file(*file_path*, *file_format*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Write the current molecule to a file or file-like object

Parameters

- **file_path** (`str or file-like object`) – A file-like object or the path to the file to be written.
- **file_format** (`str`) – Format specifier, one of ['MOL2', 'MOL2H', 'SDF', 'PDB', 'SMI', 'CAN', 'TDT'] Note that not all toolkits support all formats
- **toolkit_registry** (`ToolkitRegistry`) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use for file writing. If a Toolkit is passed, only the highest-precedence toolkit is used

Raises `ValueError` – If the requested file_format is not supported by one of the installed cheminformatics toolkits

Examples

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.to_file('imatinib.mol2', file_format='mol2')
>>> molecule.to_file('imatinib.sdf', file_format='sdf')
>>> molecule.to_file('imatinib.pdb', file_format='pdb')
```

enumerate_tautomers(*max_states=20*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Enumerate the possible tautomers of the current molecule

Parameters

- **max_states** (`int optional, default=20`) – The maximum amount of molecules that should be returned
- **toolkit_registry** (`ToolkitRegistry`) – or openff.toolkit.utils.toolkits.ToolkitWrapper, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use to enumerate the tautomers.

Returns molecules (*List[openff.toolkit.topology.Molecule]*) – A list of openff.toolkit.topology.Molecule instances not including the input molecule.

enumerate_stereoisomers(*undefined_only=False*, *max_isomers=20*, *rationalise=True*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Enumerate the stereocenters and bonds of the current molecule.

Parameters

- **undefined_only** (`bool optional, default=False`) – If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry
- **max_isomers** (`int optional, default=20`) – The maximum amount of molecules that should be returned
- **rationalise** (`bool optional, default=True`) – If we should try to build and rationalise the molecule to ensure it can exist
- **toolkit_registry** (`openff.toolkit.utils.toolkits.ToolkitRegistry` or) – lopenff.toolkit.utils.toolkits.ToolkitWrapper, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use to enumerate the stereoisomers.

Returns molecules (*List[openff.toolkit.topology.Molecule]*) – A list of `Molecule` instances not including the input molecule.

enumerate_protomers(*max_states=10*)

Enumerate the formal charges of a molecule to generate different protomoers.

Parameters **max_states** (`int optional, default=10,`) – The maximum number of protomer states to be returned.

Returns molecules (*List[openff.toolkit.topology.Molecule],*) – A list of the protomers of the input molecules not including the input.

classmethod from_rdkit(*rdmol*, *allow_undefined_stereo=False*, *hydrogens_are_explicit=False*)

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

Parameters

- **rdmol** (`rkit.RDMol`) – An RDKit molecule
- **allow_undefined_stereo** (`bool, default=False`) – If `False`, raises an exception if `rdmol` contains undefined stereochemistry.
- **hydrogens_are_explicit** (`bool, default=False`) – If `False`, RDKit will perform hydrogen addition using `Chem.AddHs`

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An OpenFF molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from rdkit import Chem
>>> from openff.toolkit.tests.utils import get_data_file_path
>>> rdmol = Chem.MolFromMolFile(get_data_file_path('systems/monomers/ethanol.sdf'))
>>> molecule = Molecule.from_rdkit(rdmol)
```

to_rdkit(*aromaticity_model=DEFAULT_AROMATICITY_MODEL*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Create an RDKit molecule

Requires the RDKit to be installed.

Parameters **aromaticity_model** (`str, optional, default=DEFAULT_AROMATICITY_MODEL`) – The aromaticity model to use

Returns **rdmol** (*rdkit.RDMol*) – An RDKit molecule

Examples

Convert a molecule to RDKit

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> rdmol = molecule.to_rdkit()
```

classmethod from_openeye(*oemol*, *allow_undefined_stereo=False*)

Create a `Molecule` from an OpenEye molecule.

Requires the OpenEye toolkit to be installed.

Parameters

- **oemol** (`openeye.oechem.OEMol`) – An OpenEye molecule
- **allow_undefined_stereo** (`bool, default=False`) – If `False`, raises an exception if oemol contains undefined stereochemistry.

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An OpenFF molecule

Examples

Create a `Molecule` from an OpenEye OEMol

```
>>> from openeye import oechem
>>> from openff.toolkit.tests.utils import get_data_file_path
>>> ifs = oechem.oemolistream(get_data_file_path('systems/monomers/ethanol.mol2'))
>>> oemols = list(ifs.GetOEGraphMols())
>>> molecule = Molecule.from_openeye(oemols[0])
```

to_qcschema(*multiplicity=1*, *conformer=0*, *extras=None*)

Create a QCElemental Molecule.

Warning: This API is experimental and subject to change.

Parameters

- **multiplicity** (`int`, `default=1,`) – The multiplicity of the molecule; sets `molecular_multiplicity` field for QCElemental Molecule.
- **conformer** (`int`, `default=0,`) – The index of the conformer to use for the QCElemental Molecule geometry.
- **extras** (`dict`, `default=None`) – A dictionary that should be included in the `extras` field on the QCElemental Molecule. This can be used to include extra information, such as a smiles representation.

Returns *qcelemental.models.Molecule* – A validated QCElemental Molecule.

Examples

Create a QCElemental Molecule:

```
>>> import qcelemental as qcel
>>> mol = Molecule.from_smiles('CC')
>>> mol.generate_conformers(n_conformers=1)
>>> qcemol = mol.to_qcschema()
```

Raises

- **MissingOptionalDependencyError** – If qcelemental is not installed, the qcschema can not be validated.
- **InvalidConformerError** – No conformer found at the given index.

classmethod from_mapped_smiles(*mapped_smiles*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*)

Create an `Molecule` from a mapped SMILES made with cmiles. The molecule will be in the order of the indexing in the mapped smiles string.

Warning: This API is experimental and subject to change.

Parameters

- **mapped_smiles** (str) – A CMILES-style mapped smiles string with explicit hydrogens.
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if oemol contains undefined stereochemistry.

Returns **offmol** (*openff.toolkit.topology.molecule.Molecule*) – An OpenFF molecule instance.

Raises **SmilesParsingError** – If the given SMILES had no indexing picked up by the toolkits.

classmethod **from_qcschema**(*qca_record*, *client=None*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*)

Create a Molecule from a QCArchive molecule record or dataset entry based on attached cmiles information.

For a molecule record, a conformer will be set from its geometry.

For a dataset entry, if a corresponding client instance is provided, the starting geometry for that entry will be used as a conformer.

A QCElemental Molecule produced from Molecule.to_qcschema can be round-tripped through this method to produce a new, valid Molecule.

Parameters

- **qca_record** (dict) – A QCArchive molecule record or dataset entry.
- **client** (optional, default=None,) – A qcportal.FractalClient instance to use for fetching an initial geometry. Only used if qca_record is a dataset entry.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if qca_record contains undefined stereochemistry.

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An OpenFF molecule instance.

Examples

Get Molecule from a QCArchive molecule record:

```
>>> from qcportal import FractalClient
>>> client = FractalClient()
>>> offmol = Molecule.from_qcschema(client.query_molecules(molecular_formula=
↪"C16H20N3O5")[0])
```

Get Molecule from a QCArchive optimization entry:

```
>>> from qcportal import FractalClient
>>> client = FractalClient()
>>> optds = client.get_collection("OptimizationDataset",
                                  "SMIRNOFF Coverage Set 1")
>>> offmol = Molecule.from_qcschema(optds.get_entry('coc(o)oc-0'))
```

Same as above, but with conformer(s) from initial molecule(s) by providing client to database:

```
>>> offmol = Molecule.from_qcschema(optds.get_entry('coc(o)oc-0'), client=client)
```

Raises

- `AttributeError` –
  - If the record dict can not be made from `qca_record`. - If a `client` is passed and it could not retrieve the initial molecule.
- `KeyError` – If the dict does not contain the `canonical_isomeric_explicit_hydrogen_mapped_smiles`.
- **`InvalidConformerError`** – Silent error, if the conformer could not be attached.

classmethod from_pdb_and_smiles(*file_path*, *smiles*, *allow_undefined_stereo=False*)

Create a Molecule from a pdb file and a SMILES string using RDKit.

Requires RDKit to be installed.

> **Warning:** This API is experimental and subject to change.

The molecule is created and sanitised based on the SMILES string, we then find a mapping between this molecule and one from the PDB based only on atomic number and connections. The SMILES molecule is then reindexed to match the PDB, the conformer is attached, and the molecule returned.

Note that any stereochemistry in the molecule is set by the SMILES, and not the coordinates of the PDB.

Parameters

- **`file_path`** (`str`) – PDB file path
- **`smiles`** (`str`) – a valid smiles string for the pdb, used for stereochemistry, formal charges, and bond order
- **`allow_undefined_stereo`** (`bool, default=False`) – If false, raises an exception if SMILES contains undefined stereochemistry.

Returns molecule (*openff.toolkit.Molecule*) – An OFFMol instance with ordering the same as used in the PDB file.

Raises `InvalidConformerError` – If the SMILES and PDB molecules are not isomorphic.

canonical_order_atoms(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Produce a copy of the molecule with the atoms reordered canonically.

Each toolkit defines its own canonical ordering of atoms. The canonical order may change from toolkit version to toolkit version or between toolkits.

> **Warning:** This API is experimental and subject to change.

Parameters **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An new OpenFF style molecule with atoms in the canonical order.

remap(*mapping_dict*, *current_to_new=True*)

Remap all of the indexes in the molecule to match the given mapping dict

> **Warning:** This API is experimental and subject to change.

Parameters

- **mapping_dict** (dict,) – A dictionary of the mapping between indexes, this should start from 0.
- **current_to_new** (bool, default=True) – If this is True, then mapping_dict is of the form {current_index: new_index}; otherwise, it is of the form {new_index: current_index}

Returns **new_molecule** (*openff.toolkit.topology.molecule.Molecule*) – An openff.toolkit.Molecule instance with all attributes transferred, in the PDB order.

to_openeye(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *aromaticity_model=DEFAULT_AROMATICITY_MODEL*)

Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

Parameters **aromaticity_model** (str, optional, default=DEFAULT_AROMATICITY_MODEL) – The aromaticity model to use

Returns **oemol** (*openeye.oechem.OEMol*) – An OpenEye molecule

Examples

Create an OpenEye molecule from a Molecule

```
>>> molecule = Molecule.from_smiles('CC')
>>> oemol = molecule.to_openeye()
```

get_bond_between(*i*, *j*)

Returns the bond between two atoms

Parameters

- **i** (int or Atom) – Atoms or atom indices to check
- **j** (int or Atom) – Atoms or atom indices to check

Returns **bond** (*Bond*) – The bond between i and j.

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters **serialized** (bytes) – A BSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters **serialized** (str) – A JSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters **serialized** (bytes) – A MessagePack-encoded bytes serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_pickle(*serialized*)

Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters **serialized** (str) – A pickled representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters **serlialized** (str) – A TOML serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters **serialized** (bytes) – An XML serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters **serialized** (str) – A YAML serialized representation of the object

Returns **instance** (*cls*) – Instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns **serialized** (*bytes*) – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters **indent** (`int, optional, default=None`) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*str*) – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation of the object

to_pickle()

Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns **serialized** (*str*) – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns **serialized** (*str*) – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters **indent** (`int, optional, default=2`) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns **serialized** (*str*) – A YAML serialized representation of the object

11.1.2 openff.toolkit.topology.Molecule

class openff.toolkit.topology.**Molecule**(**args*, ***kwargs*)

Mutable chemical representation of a molecule, such as a small molecule or biopolymer.

Examples

Create a molecule from an sdf file

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
```

Convert to OpenEye OEMol object

```
>>> oemol = molecule.to_openeye()
```

Create a molecule from an OpenEye molecule

```
>>> molecule = Molecule.from_openeye(oemol)
```

Convert to RDKit Mol object

```
>>> rdmol = molecule.to_rdkit()
```

Create a molecule from an RDKit molecule

```
>>> molecule = Molecule.from_rdkit(rdmol)
```

Create a molecule from IUPAC name (requires the OpenEye toolkit)

```
>>> molecule = Molecule.from_iupac('imatinib')
```

Create a molecule from SMILES

```
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
```

Warning: This API is experimental and subject to change.

__init__(**args*, ***kwargs*)

Create a new Molecule object

Parameters **other** (optional, default=None) – If specified, attempt to construct a copy of the molecule from the specified object. This can be any one of the following:

- a Molecule object
- a file that can be used to construct a Molecule object
- an openeye.oechem.OEMol
- an rdkit.Chem.rdchem.Mol
- a serialized Molecule object

Examples

Create an empty molecule:

```
>>> empty_molecule = Molecule()
```

Create a molecule from a file that can be used to construct a molecule, using either a filename or file-like object:

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> molecule = Molecule(open(sdf_filepath, 'r'), file_format='sdf')
```

```
>>> import gzip
>>> mol2_gz_filepath = get_data_file_path('molecules/toluene.mol2.gz')
>>> molecule = Molecule(gzip.GzipFile(mol2_gz_filepath, 'r'), file_format='mol2')
```

Create a molecule from another molecule:

```
>>> molecule_copy = Molecule(molecule)
```

Convert to OpenEye OEMol object

```
>>> oemol = molecule.to_openeye()
```

Create a molecule from an OpenEye molecule:

```
>>> molecule = Molecule(oemol)
```

Convert to RDKit Mol object

```
>>> rdmol = molecule.to_rdkit()
```

Create a molecule from an RDKit molecule:

```
>>> molecule = Molecule(rdmol)
```

Convert the molecule into a dictionary and back again:

```
>>> serialized_molecule = molecule.to_dict()
>>> molecule_copy = Molecule(serialized_molecule)
```

Methods

`__init__`(*args, **kwargs)	Create a new Molecule object
`add_atom`(atomic_number, formal_charge, ...)	Add an atom to the molecule.
`add_bond`(atom1, atom2, bond_order, is_aromatic)	Add a bond between two specified atom indices
`add_conformer`(coordinates)	Add a conformation of the molecule
`add_default_hierarchy_schemes`([...])	Adds `chain` and `residue` hierarchy schemes.

continues on next page

Table 2 – continued from previous page

add_hierarchy_scheme(uniqueness_criteria, ...)	Use the molecule's metadata to facilitate iteration over its atoms.
apply_elf_conformer_selection([percentage, ...])	Select a set of diverse conformers from the molecule's conformers with ELF.
are_isomorphic(mol1, mol2[, ...])	Determine if mol1 is isomorphic to mol2.
assign_fractional_bond_orders([...])	Update and store list of bond orders this molecule.
assign_partial_charges(partial_charge_method)	Calculate partial atomic charges and store them in the molecule.
atom(index)	Get the atom with the specified index.
atom_index(atom)	Returns the index of the given atom in this molecule
bond(index)	Get the bond with the specified index.
canonical_order_atoms([toolkit_registry])	Produce a copy of the molecule with the atoms reordered canonically.
chemical_environment_matches(query[, ...])	Find matches in the molecule for a SMARTS string or ChemicalEnvironment query
compute_partial_charges_am1bcc([...])	Deprecated since version 0.11.0.
delete_hierarchy_scheme(iter_name)	Remove an existing HierarchyScheme specified by its iterator name.
enumerate_protomers([max_states])	Enumerate the formal charges of a molecule to generate different protomoers.
enumerate_stereoisomers([undefined_only, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers([max_states, ...])	Enumerate the possible tautomers of the current molecule
find_rotatable_bonds([...])	Find all bonds classed as rotatable ignoring any matched to the ignore_functional_groups list.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(molecule_dict)	Create a new Molecule from a dictionary representation
from_file(file_path[, file_format, ...])	Create one or more molecules from a file
from_inchi(inchi[, allow_undefined_stereo, ...])	Construct a Molecule from a InChI representation
from_iupac(iupac_name[, toolkit_registry, ...])	Generate a molecule from IUPAC or common name
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mapped_smiles(mapped_smiles[, ...])	Create an Molecule from a mapped SMILES made with cmiles.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_openeye(oemol[, allow_undefined_stereo])	Create a Molecule from an OpenEye molecule.
from_pdb(file_path[, toolkit_registry])	Deprecated since version 0.11.0.

continues on next page

Table 2 – continued from previous page

from_pdb_and_smiles(file_path, smiles[, ...])	Create a Molecule from a pdb file and a SMILES string using RDKit.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_polymer_pdb(file_path[, toolkit_registry])	Loads a polymer from a PDB file.
from_qcschema(qca_record[, client, ...])	Create a Molecule from a QCArchive molecule record or dataset entry based on attached cmiles information.
from_rdkit(rdmol[, allow_undefined_stereo, ...])	Create a Molecule from an RDKit molecule.
from_smiles(smiles[, ...])	Construct a Molecule from a SMILES representation
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_topology(topology)	Return a Molecule representation of an OpenFF Topology containing a single Molecule object.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
generate_conformers([toolkit_registry, ...])	Generate conformers for this molecule using an underlying toolkit.
generate_unique_atom_names()	Generate unique atom names using element name and number of times that element has occurred e.g.
get_bond_between(i, j)	Returns the bond between two atoms
is_isomorphic_with(other, **kwargs)	Check if the molecule is isomorphic with the other molecule which can be an openff.toolkit.topology.Molecule or nx.Graph().
nth_degree_neighbors(n_degrees)	Return canonicalized pairs of atoms whose shortest separation is *exactly* n bonds.
ordered_connection_table_hash()	Compute an ordered hash of the atoms and bonds in the molecule
particle(index)	DEPRECATED: Use Molecule.atom instead.
particle_index(particle)	DEPRECATED: Use Molecule.atom_index instead.
perceive_residues([substructure_file_path, ...])	Perceive a polymer's residues and permit iterating over them.
remap(mapping_dict[, current_to_new])	Remap all of the indexes in the molecule to match the given mapping dict
strip_atom_stereochemistry(smarts[, ...])	Delete stereochemistry information for certain atoms, if it is present.
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dictionary representation of the molecule.
to_file(file_path, file_format[, ...])	Write the current molecule to a file or file-like object
to_hill_formula()	Generate the Hill formula of this molecule.
to_inchi([fixed_hydrogens, toolkit_registry])	Create an InChI string for the molecule using the requested toolkit backend.

continues on next page

Table 2 – continued from previous page

to_inchikey([fixed_hydrogens, toolkit_registry])	Create an InChIKey for the molecule using the requested toolkit backend.
to_iupac([toolkit_registry])	Generate IUPAC name from Molecule
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_networkx()	Generate a NetworkX undirected graph from the molecule.
to_openeye([toolkit_registry, aromaticity_model])	Create an OpenEye molecule
to_pickle()	Return a pickle serialized representation.
to_qcschema([multiplicity, conformer, extras])	Create a QCElemental Molecule.
to_rdkit([aromaticity_model, toolkit_registry])	Create an RDKit molecule
to_smiles([isomeric, explicit_hydrogens, ...])	Return a canonical isomeric SMILES representation of the current molecule.
to_toml()	Return a TOML serialized representation.
to_topology()	Return an OpenFF Topology representation containing one copy of this molecule
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.
update_hierarchy_schemes([iter_names])	Infer a hierarchy from atom metadata according to the existing hierarchy schemes.
visualize([backend, width, height, ...])	Render a visualization of the molecule in Jupyter

Attributes

amber_impropers	Iterate over all impropers with trivalent centers, reporting the central atom first.
angles	Get an iterator over all i-j-k angles.
atoms	Iterate over all Atom objects in the molecule.
bonds	Iterate over all Bond objects in the molecule.
conformers	Returns the list of conformers for this molecule.
has_unique_atom_names	True if the molecule has unique atom names, False otherwise.
hierarchy_schemes	The hierarchy schemes available on the molecule.
hill_formula	Get the Hill formula of the molecule
impropers	Iterate over all improper torsions in the molecule.
n_angles	Number of angles in the molecule.
n_atoms	The number of Atom objects.
n_bonds	The number of Bond objects in the molecule.
n_conformers	The number of conformers for this molecule.
n_impropers	Number of possible improper torsions in the molecule.
n_particles	Use Molecule.n_atoms instead.
n_propers	Number of proper torsions in the molecule.
name	The name (or title) of the molecule
partial_charges	Returns the partial charges (if present) on the molecule.
particles	Use Molecule.atoms instead.
propers	Iterate over all proper torsions in the molecule
properties	The properties dictionary of the molecule
smirnoff_impropers	Iterate over all impropers with trivalent centers, reporting the central atom second.
torsions	Get an iterator over all i-j-k-l torsions.
total_charge	Return the total charge on the molecule

add_atom(*atomic_number*, *formal_charge*, *is_aromatic*, *stereochemistry=None*, *name=None*, *metadata=None*)

Add an atom to the molecule.

Parameters

- **atomic_number** (int) – Atomic number of the atom
- **formal_charge** (int) – Formal charge of the atom
- **is_aromatic** (bool) – If True, atom is aromatic; if False, not aromatic
- **stereochemistry** (str, optional, default=None) – Either 'R' or 'S' for specified stereochemistry, or None if stereochemistry is irrelevant
- **name** (str, optional) – An optional name for the atom
- **metadata** (dict[str: (int, str)], default=None) – An optional dictionary where keys are strings and values are strings or ints. This is intended to record atom-level information used to inform hierarchy definition and iteration, such as grouping atom by residue and chain.

Returns **index** (*int*) – The index of the atom in the molecule

Examples

Define a methane molecule

```
>>> molecule = Molecule()
>>> molecule.name = 'methane'
>>> C = molecule.add_atom(6, 0, False)
>>> H1 = molecule.add_atom(1, 0, False)
>>> H2 = molecule.add_atom(1, 0, False)
>>> H3 = molecule.add_atom(1, 0, False)
>>> H4 = molecule.add_atom(1, 0, False)
>>> bond_idx = molecule.add_bond(C, H1, False, 1)
>>> bond_idx = molecule.add_bond(C, H2, False, 1)
>>> bond_idx = molecule.add_bond(C, H3, False, 1)
>>> bond_idx = molecule.add_bond(C, H4, False, 1)
```

add_bond(*atom1*, *atom2*, *bond_order*, *is_aromatic*, *stereochemistry=None*, *fractional_bond_order=None*)

Add a bond between two specified atom indices

Parameters

- **atom1** (`int or Atom`) – Index of first atom
- **atom2** (`int or Atom`) – Index of second atom
- **bond_order** (`int`) – Integral bond order of Kekulized form
- **is_aromatic** (`bool`) – True if this bond is aromatic, False otherwise
- **stereochemistry** (`str, optional, default=None`) – Either `'E'` or `'Z'` for specified stereochemistry, or `None` if stereochemistry is irrelevant
- **fractional_bond_order** (`float, optional, default=None`) – The fractional (eg. Wiberg) bond order

Returns **index** (*int*) – Index of the bond in this molecule

add_conformer(*coordinates*)

Add a conformation of the molecule

Parameters **coordinates** (`unit-wrapped np.array with shape (n_atoms, 3) and dimension of distance`) – Coordinates of the new conformer, with the first dimension of the array corresponding to the atom index in the molecule's indexing system.

Returns **index** (*int*) – The index of this conformer

visualize(*backend='rdkit'*, *width=None*, *height=None*, *show_all_hydrogens=True*)

Render a visualization of the molecule in Jupyter

Parameters

- **backend** (`str, optional, default='rdkit'`) – The visualization engine to use. Choose from:
  - `"rdkit"`
  - `"openeye"`

– "nglview" (requires conformers)

- **width** (int, optional, default=500) – Width of the generated representation (only applicable to backend=openeye or backend=rdkit)
- **height** (int, optional, default=300) – Width of the generated representation (only applicable to backend=openeye or backend=rdkit)
- **show_all_hydrogens** (bool, optional, default=True) – Whether to explicitly depict all hydrogen atoms. (only applicable to backend=openeye or backend=rdkit)

Returns

object – Depending on the backend chosen:

- rdkit → IPython.display.SVG
- openeye → IPython.display.Image
- nglview → nglview.NGLWidget

perceive_residues(*substructure_file_path=None*, *strict_chirality=True*)

Perceive a polymer's residues and permit iterating over them.

Perceives residues by matching substructures in the current molecule with a substructure dictionary file, using SMARTS, and assigns residue names and numbers to atom metadata. It then constructs a residue hierarchy scheme to allow iterating over residues.

Parameters

- **substructure_file_path** (str, optional, default=None) – Path to substructure library file in JSON format. Defaults to using built-in substructure file.
- **strict_chirality** (bool, optional, default=True) – Whether to use strict chirality symbols (stereomarks) for substructure matchings with SMARTS.

add_default_hierarchy_schemes(*overwrite_existing=True*)

Adds chain and residue hierarchy schemes.

The Open Force Field Toolkit has no native understanding of hierarchical atom organisation schemes common to other biomolecular software, such as "residues" or "chains" (see *Hierarchy data (chains and residues)*). Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see HierarchyScheme.

If a Molecule with the default hierarchy schemes changes, Molecule.update_hierarchy_schemes() must be called before the residues or chains are iterated over again or else the iteration may be incorrect.

Parameters **overwrite_existing** (bool, default=True) – Whether to overwrite existing instances of the *residue* and *chain* hierarchy schemes. If this is False and either of the hierarchy schemes are already defined on this molecule, an exception will be raised.

Raises **HierarchySchemeWithIteratorNameAlreadyRegisteredException** – When overwrite_existing=False and either the chains or residues hierarchy scheme is already configured.

add_hierarchy_scheme(*uniqueness_criteria*, *iterator_name*)

Use the molecule's metadata to facilitate iteration over its atoms.

This method will add an attribute with the name given by the iterator_name argument that provides an iterator over groups of atoms. Atoms are grouped by the values in their atom.metadata

dictionary; any atoms with the same values for the keys given in the `uniqueness_criteria` argument will be in the same group. These groups have the type `HierarchyElement`.

Hierarchy schemes are not updated dynamically; if a `Molecule` with hierarchy schemes changes, `Molecule.update_hierarchy_schemes()` must be called before the scheme is iterated over again or else the grouping may be incorrect.

Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see `HierarchyScheme`.

Parameters

- **uniqueness_criteria** (`tuple of str`) – The names of `Atom` metadata entries that define this scheme. An atom belongs to a `HierarchyElement` only if its metadata has the same values for these criteria as the other atoms in the `HierarchyElement`.
- **iterator_name** (`str`) – Name of the iterator that will be exposed to access the hierarchy elements generated by this scheme.

Returns **new_hier_scheme** (*openff.toolkit.topology.HierarchyScheme*) – The newly created HierarchyScheme

property amber_impropers: Set[Tuple[Atom, Atom, Atom, Atom]]

Iterate over all impropers with trivalent centers, reporting the central atom first.

The central atom is reported first in each torsion. This method reports an improper for each trivalent atom in the molecule, whether or not any given force field would assign it improper torsion parameters.

Also note that this will return 6 possible atom orderings around each improper center. In current AMBER parameterization, one of these six orderings will be used for the actual assignment of the improper term and measurement of the angle. This method does not encode the logic to determine which of the six orderings AMBER would use.

Returns **impropers** (*set of tuple*) – An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed first in each tuple.

See also:

`impropers`, `smirnoff_impropers`

property angles: Set[Tuple[Atom, Atom, Atom]]

Get an iterator over all i-j-k angles.

apply_elf_conformer_selection(*percentage: float = 2.0*, *limit: int = 10*, *toolkit_registry: Optional[Union[ToolkitRegistry, ToolkitWrapper]] = GLOBAL_TOOLKIT_REGISTRY*, ***kwargs*)

Select a set of diverse conformers from the molecule's conformers with ELF.

Applies the Electrostatically Least-interacting Functional groups method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from the molecule's conformers.

Parameters

- **toolkit_registry** – The underlying toolkit to use to select the ELF conformers.
- **percentage** – The percentage of conformers with the lowest electrostatic interaction energies to greedily select from.
- **limit** – The maximum number of conformers to select.

Notes

- The input molecule should have a large set of conformers already generated to select the ELF conformers from.
- The selected conformers will be retained in the *conformers* list while unselected conformers will be discarded.

See also:

`openff.toolkit.utils.toolkits.OpenEyeToolkitWrapper.apply_elf_conformer_selection`, `openff.toolkit.utils.toolkits.RDKitToolkitWrapper.apply_elf_conformer_selection`

static are_isomorphic(*mol1*, *mol2*, *return_atom_map=False*, *aromatic_matching=True*, *formal_charge_matching=True*, *bond_order_matching=True*, *atom_stereochemistry_matching=True*, *bond_stereochemistry_matching=True*, *strip_pyrimidal_n_atom_stereo=True*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Determine if `mol1` is isomorphic to `mol2`.

`are_isomorphic()` compares two molecule's graph representations and the chosen node/edge attributes. Connections and atomic numbers are always checked.

If nx.Graphs() are given they must at least have `atomic_number` attributes on nodes. Other attributes that `are_isomorphic()` can optionally check...

- ... in nodes are:
  - `is_aromatic`
  - `formal_charge`
  - `stereochemistry`
- ... in edges are:
  - `is_aromatic`
  - `bond_order`
  - `stereochemistry`

By default, all attributes are checked, but stereochemistry around pyrimidal nitrogen is ignored.

Warning: This API is experimental and subject to change.

Parameters

- **mol1** (an `openff.toolkit.topology.molecule.FrozenMolecule` or `nx.Graph()`) – The first molecule to test for isomorphism.
- **mol2** (an `openff.toolkit.topology.molecule.FrozenMolecule` or `nx.Graph()`) – The second molecule to test for isomorphism.
- **return_atom_map** (`bool, default=False, optional`) – Return a `dict` containing the atomic mapping instead of a `bool`.
- **aromatic_matching** (`bool, default=True, optional`) – If `False`, aromaticity of graph nodes and edges are ignored for the purpose of determining isomorphism.

- **formal_charge_matching** (bool, default=True, optional) – If False, formal charges of graph nodes are ignored for the purpose of determining isomorphism.
- **bond_order_matching** (bool, default=True, optional) – If False, bond orders of graph edges are ignored for the purpose of determining isomorphism.
- **atom_stereochemistry_matching** (bool, default=True, optional) – If False, atoms' stereochemistry is ignored for the purpose of determining isomorphism.
- **bond_stereochemistry_matching** (bool, default=True, optional) – If False, bonds' stereochemistry is ignored for the purpose of determining isomorphism.
- **strip_pyrimidal_n_atom_stereo** (bool, default=True, optional) – If True, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for removing stereochemistry from pyrimidal nitrogens.

Returns

- **molecules_are_isomorphic** (*bool*)
- **atom_map** (*default=None, Optional,*) – [Dict[int,int]] ordered by mol1 indexing {mol1_index: mol2_index} If molecules are not isomorphic given input arguments, will return None instead of dict.

assign_fractional_bond_orders(*bond_order_model=None*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *use_conformers=None*)

Update and store list of bond orders this molecule.

Bond orders are stored on each bond, in the bond.fractional_bond_order attribute.

Warning: This API is experimental and subject to change.

Parameters

- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion
- **bond_order_model** (string, optional. Default=None) – The bond order model to use for fractional bond order calculation. If None, "am1-wiberg" is used.
- **use_conformers** (iterable of openmm.unit.Quantity(np.array) with shape (n_atoms, 3) and dimension of distance,) – optional, default=None The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper.

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.assign_fractional_bond_orders()
```

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

assign_partial_charges(*partial_charge_method: str*, *strict_n_conformers=False*, *use_conformers=None*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *normalize_partial_charges=True*)

Calculate partial atomic charges and store them in the molecule.

`assign_partial_charges` computes charges using the specified toolkit and assigns the new values to the `partial_charges` attribute. Supported charge methods vary from toolkit to toolkit, but some supported methods are:

- `"am1bcc"`
- `"am1bccelf10"` (requires OpenEye Toolkits)
- `"am1-mulliken"`
- `"mmff94"`
- `"gasteiger"`

For more supported charge methods and details, see the corresponding methods in each toolkit wrapper:

- `OpenEyeToolkitWrapper.assign_partial_charges`
- `RDKitToolkitWrapper.assign_partial_charges`
- `AmberToolsToolkitWrapper.assign_partial_charges`
- `BuiltInToolkitWrapper.assign_partial_charges`

Parameters

- **partial_charge_method** (`string`) – The partial charge calculation method to use for partial charge calculation.
- **strict_n_conformers** (`bool, default=False`) – Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.
- **use_conformers** (`iterable of openmm.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3) and`) – dimension of distance. Optional, default=None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **toolkit_registry** (`openff.toolkit.utils.toolkits.ToolkitRegistry or`) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None `ToolkitRegistry` or `ToolkitWrapper` to use for the calculation.
- **normalize_partial_charges** (`bool, default=True`) – Whether to offset partial charges so that they sum to the total formal charge of the molecule. This is used to prevent accumulation of rounding errors when the partial charge assignment method returns values at limited precision.

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.assign_partial_charges('am1-mulliken')
```

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

See also:

openff.toolkit.utils.toolkits.OpenEyeToolkitWrapper.assign_partial_charges, openff.toolkit.utils.toolkits.RDKitToolkitWrapper.assign_partial_charges, openff.toolkit.utils.toolkits.AmberToolsToolkitWrapper.assign_partial_charges, openff.toolkit.utils.toolkits.BuiltInToolkitWrapper.assign_partial_charges

atom(*index: int*) → *Atom*

Get the atom with the specified index.

Parameters **index** (int) –

Returns **atom** (*openff.toolkit.topology.Atom*)

atom_index(*atom:* Atom) → int

Returns the index of the given atom in this molecule

Parameters **atom** (Atom) –

Returns **index** (*int*) – The index of the given atom in this molecule

property atoms

Iterate over all Atom objects in the molecule.

bond(*index: int*) → *Bond*

Get the bond with the specified index.

Parameters **index** (int) –

Returns **bond** (*openff.toolkit.topology.Bond*)

property bonds: List[Bond]

Iterate over all Bond objects in the molecule.

canonical_order_atoms(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Produce a copy of the molecule with the atoms reordered canonically.

Each toolkit defines its own canonical ordering of atoms. The canonical order may change from toolkit version to toolkit version or between toolkits.

Warning: This API is experimental and subject to change.

Parameters **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An new OpenFF style molecule with atoms in the canonical order.

chemical_environment_matches(*query*, *unique=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Find matches in the molecule for a SMARTS string or ChemicalEnvironment query

Parameters

- **query** (str or ChemicalEnvironment) – SMARTS string (with one or more tagged atoms) or ChemicalEnvironment query. Query will internally be resolved to SMIRKS using query.asSMIRKS() if it has an .asSMIRKS method.
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY ToolkitRegistry or ToolkitWrapper to use for chemical environment matches

Returns **matches** (*list of atom index tuples*) – A list of tuples, containing the indices of the matching atoms.

Examples

Retrieve all the carbon-carbon bond matches in a molecule

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> matches = molecule.chemical_environment_matches('[#6X3:1]~[#6X3:2]')
```

compute_partial_charges_am1bcc(*use_conformers=None*, *strict_n_conformers=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Deprecated since version 0.11.0: This method was deprecated in v0.11.0 and will soon be removed. Use assign_partial_charges(partial_charge_method='am1bcc') instead.

Calculate partial atomic charges for this molecule using AM1-BCC run by an underlying toolkit and assign them to this molecule's partial_charges attribute.

Parameters

- **strict_n_conformers** (bool, default=False) – Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.
- **use_conformers** (iterable of openmm.unit.Quantity-wrapped numpy arrays, each with shape (n_atoms, 3)) – and dimension of distance. Optional, default=None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers for the given charge method will be generated.
- **toolkit_registry** (ToolkitRegistry) –
- **openff.toolkit.utils.toolkits.ToolkitWrapper** (or) – ToolkitRegistry or ToolkitWrapper to use for the calculation
- **optional** – ToolkitRegistry or ToolkitWrapper to use for the calculation
- **default=None** – ToolkitRegistry or ToolkitWrapper to use for the calculation

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers()
>>> molecule.compute_partial_charges_am1bcc()
```

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

property conformers

Returns the list of conformers for this molecule.

Conformers are presented as a list of `Quantity`-wrapped NumPy arrays, of shape (3 x n_atoms) and with dimensions of [Distance]. The return value is the actual list of conformers, and changes to the contents affect the original `FrozenMolecule`.

delete_hierarchy_scheme(*iter_name*)

Remove an existing `HierarchyScheme` specified by its iterator name.

Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see `HierarchyScheme`.

Parameters **iter_name** (`str`) –

enumerate_protomers(*max_states=10*)

Enumerate the formal charges of a molecule to generate different protomoers.

Parameters **max_states** (`int optional, default=10,`) – The maximum number of protomer states to be returned.

Returns **molecules** (*List[openff.toolkit.topology.Molecule],*) – A list of the protomers of the input molecules not including the input.

enumerate_stereoisomers(*undefined_only=False*, *max_isomers=20*, *rationalise=True*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Enumerate the stereocenters and bonds of the current molecule.

Parameters

- **undefined_only** (`bool optional, default=False`) – If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry
- **max_isomers** (`int optional, default=20`) – The maximum amount of molecules that should be returned
- **rationalise** (`bool optional, default=True`) – If we should try to build and rationalise the molecule to ensure it can exist
- **toolkit_registry** (`openff.toolkit.utils.toolkits.ToolkitRegistry` or `openff.toolkit.utils.toolkits.ToolkitWrapper`, default=GLOBAL_TOOLKIT_REGISTRY `ToolkitRegistry` or `ToolkitWrapper` to use to enumerate the stereoisomers.

Returns **molecules** (*List[openff.toolkit.topology.Molecule]*) – A list of `Molecule` instances not including the input molecule.

enumerate_tautomers(*max_states=20*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Enumerate the possible tautomers of the current molecule

Parameters

- **max_states** (int optional, default=20) – The maximum amount of molecules that should be returned
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, default=GLOBAL_TOOLKIT_REGISTRY ToolkitRegistry or ToolkitWrapper to use to enumerate the tautomers.

Returns molecules (*List[openff.toolkit.topology.Molecule]*) – A list of openff.toolkit.topology.Molecule instances not including the input molecule.

find_rotatable_bonds(*ignore_functional_groups=None*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Find all bonds classed as rotatable ignoring any matched to the ignore_functional_groups list.

Parameters

- **ignore_functional_groups** (optional, List[str], default=None,) – A list of bond SMARTS patterns to be ignored when finding rotatable bonds.
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapperl, optional, default=None ToolkitRegistry or ToolkitWrapper to use for SMARTS matching

Returns bonds (*List[openff.toolkit.topology.molecule.Bond]*) – The list of openff.toolkit.topology.molecule.Bond instances which are rotatable.

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters serialized (bytes) – A BSON serialized representation of the object

Returns instance (*cls*) – An instantiated object

classmethod from_dict(*molecule_dict*)

Create a new Molecule from a dictionary representation

Parameters molecule_dict (OrderedDict) – A dictionary representation of the molecule.

Returns molecule (*Molecule*) – A Molecule created from the dictionary representation

classmethod from_file(*file_path*, *file_format=None*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*)

Create one or more molecules from a file

Parameters

- **file_path** (str or file-like object) – The path to the file or file-like object to stream one or more molecules from.
- **file_format** (str, optional, default=None) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for your loaded toolkits for details.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY ToolkitRegistry or ToolkitWrapper to use for file loading. If a Toolkit is passed, only the highest-precedence toolkit is used

- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if oemol contains undefined stereochemistry.

Returns **molecules** (*Molecule or list of Molecules*) – If there is a single molecule in the file, a Molecule is returned; otherwise, a list of Molecule objects is returned.

Examples

```
>>> from openff.toolkit.tests.utils import get_monomer_mol2_file_path
>>> mol2_file_path = get_monomer_mol2_file_path('cyclohexane')
>>> molecule = Molecule.from_file(mol2_file_path)
```

classmethod **from_inchi**(*inchi*, *allow_undefined_stereo=False*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*)

Construct a Molecule from a InChI representation

Parameters

- **inchi** (str) – The InChI representation of the molecule.
- **allow_undefined_stereo** (bool, default=False) – Whether to accept InChI with undefined stereochemistry. If False, an exception will be raised if a InChI with undefined stereochemistry is passed into this function.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for InChI-to-molecule conversion

Returns **molecule** (*openff.toolkit.topology.Molecule*)

Examples

Make cis-1,2-Dichloroethene:

```
>>> molecule = Molecule.from_inchi('InChI=1S/C2H2Cl2/c3-1-2-4/h1-2H/b2-1-')
```

classmethod **from_iupac**(*iupac_name*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*, ***kwargs*)

Generate a molecule from IUPAC or common name

Note: This method requires the OpenEye toolkit to be installed.

Parameters

- **iupac_name** (str) – IUPAC name of molecule to be generated
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY ToolkitRegistry or ToolkitWrapper to use for chemical environment matches
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if molecule contains undefined stereochemistry.

Returns **molecule** (*Molecule*) – The resulting molecule with position

Examples

Create a molecule from an IUPAC name

```
>>> molecule = Molecule.from_iupac('4-[(4-methylpiperazin-1-yl)methyl]-N-(4-methyl-
↪3-{[4-(pyridin-3-yl)pyrimidin-2-yl]amino}phenyl)benzamide')  # noqa
```

Create a molecule from a common name

```
>>> molecule = Molecule.from_iupac('imatinib')
```

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters **serialized** (str) – A JSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_mapped_smiles(*mapped_smiles*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*)

Create an Molecule from a mapped SMILES made with cmiles. The molecule will be in the order of the indexing in the mapped smiles string.

Warning: This API is experimental and subject to change.

Parameters

- **mapped_smiles** (str) – A CMILES-style mapped smiles string with explicit hydrogens.
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if oemol contains undefined stereochemistry.

Returns **offmol** (*openff.toolkit.topology.molecule.Molecule*) – An OpenFF molecule instance.

Raises **SmilesParsingError** – If the given SMILES had no indexing picked up by the toolkits.

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters **serialized** (bytes) – A MessagePack-encoded bytes serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_openeye(*oemol*, *allow_undefined_stereo=False*)

Create a `Molecule` from an OpenEye molecule.

Requires the OpenEye toolkit to be installed.

Parameters

- **oemol** (`openeye.oechem.OEMol`) – An OpenEye molecule
- **allow_undefined_stereo** (`bool`, `default=False`) – If `False`, raises an exception if oemol contains undefined stereochemistry.

Returns molecule (*openff.toolkit.topology.Molecule*) – An OpenFF molecule

Examples

Create a `Molecule` from an OpenEye OEMol

```
>>> from openeye import oechem
>>> from openff.toolkit.tests.utils import get_data_file_path
>>> ifs = oechem.oemolistream(get_data_file_path('systems/monomers/ethanol.mol2'))
>>> oemols = list(ifs.GetOEGraphMols())
>>> molecule = Molecule.from_openeye(oemols[0])
```

classmethod from_pdb(*file_path*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*)

Deprecated since version 0.11.0: `from_pdb` is deprecated and will soon be removed. Use `from_polymer_pdb()` instead.

classmethod from_pdb_and_smiles(*file_path*, *smiles*, *allow_undefined_stereo=False*)

Create a Molecule from a pdb file and a SMILES string using RDKit.

Requires RDKit to be installed.

> **Warning:** This API is experimental and subject to change.

The molecule is created and sanitised based on the SMILES string, we then find a mapping between this molecule and one from the PDB based only on atomic number and connections. The SMILES molecule is then reindexed to match the PDB, the conformer is attached, and the molecule returned.

Note that any stereochemistry in the molecule is set by the SMILES, and not the coordinates of the PDB.

Parameters

- **file_path** (`str`) – PDB file path
- **smiles** (`str`) – a valid smiles string for the pdb, used for stereochemistry, formal charges, and bond order
- **allow_undefined_stereo** (`bool`, `default=False`) – If false, raises an exception if SMILES contains undefined stereochemistry.

Returns molecule (*openff.toolkit.Molecule*) – An OFFMol instance with ordering the same as used in the PDB file.

Raises **InvalidConformerError** – If the SMILES and PDB molecules are not isomorphic.

classmethod from_pickle(*serialized*)

Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters **serialized** (str) – A pickled representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_polymer_pdb(*file_path: ~typing.Union[str, ~typing.TextIO]*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*)

Loads a polymer from a PDB file.

Currently only supports proteins with canonical amino acids that are either uncapped or capped by ACE/NME groups, but may later be extended to handle other common polymers, or accept user-defined polymer templates.

Metadata such as residues, chains, and atom names are recorded in the `Atom.properties` attribute, which is a dictionary mapping from strings like "residue" to the appropriate value. `from_polymer_pdb` returns a molecule that can be iterated over with the `.residues` and `.chains` attributes, as well as the usual `.atoms`.

This method proceeds in the following order:

- Loads the polymer substructure template file (distributed with the OpenFF Toolkit)
- Loads the PDB into an OpenMM `openmm.app.PDBFile` object
- Turns OpenMM topology into a temporarily invalid rdkit Molecule
- **Adds chemical information to the molecule:**
  - **For each substructure loaded from the substructure template file**
    - Uses rdkit to find matches between the substructure and the molecule
    - For any matches, assigns the atom formal charge and bond order info from the substructure to the rdkit molecule, then marks the atoms and bonds as having been assigned so they can not be overwritten by subsequent isomorphisms
- Take coordinates from the OpenMM Topology and add them as a conformer
- Convert the rdkit Molecule to OpenFF

Parameters

- **file_path** (`str or file object`) – PDB information to be passed to OpenMM PDBFile object for loading
- **None** (`toolkit_registry = ToolkitWrapper or ToolkitRegistry. Default =`) – Either a ToolkitRegistry, ToolkitWrapper

Returns **molecule** (*openff.toolkit.topology.Molecule*)

classmethod from_qcschema(*qca_record*, *client=None*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*)

Create a Molecule from a QCArchive molecule record or dataset entry based on attached cmiles information.

For a molecule record, a conformer will be set from its geometry.

For a dataset entry, if a corresponding client instance is provided, the starting geometry for that entry will be used as a conformer.

A QCElemental Molecule produced from `Molecule.to_qcschema` can be round-tripped through this method to produce a new, valid Molecule.

Parameters

- **qca_record** (`dict`) – A QCArchive molecule record or dataset entry.
- **client** (`optional, default=None,`) – A qcportal.FractalClient instance to use for fetching an initial geometry. Only used if `qca_record` is a dataset entry.
- **toolkit_registry** (`openff.toolkit.utils.toolkits.ToolkitRegistry` or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** (`bool, default=False`) – If false, raises an exception if qca_record contains undefined stereochemistry.

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An OpenFF molecule instance.

Examples

Get Molecule from a QCArchive molecule record:

```
>>> from qcportal import FractalClient
>>> client = FractalClient()
>>> offmol = Molecule.from_qcschema(client.query_molecules(molecular_formula=
↪"C16H20N3O5")[0])
```

Get Molecule from a QCArchive optimization entry:

```
>>> from qcportal import FractalClient
>>> client = FractalClient()
>>> optds = client.get_collection("OptimizationDataset",
                                  "SMIRNOFF Coverage Set 1")
>>> offmol = Molecule.from_qcschema(optds.get_entry('coc(o)oc-0'))
```

Same as above, but with conformer(s) from initial molecule(s) by providing client to database:

```
>>> offmol = Molecule.from_qcschema(optds.get_entry('coc(o)oc-0'), client=client)
```

Raises

- `AttributeError` –
  - If the record dict can not be made from `qca_record`. - If a `client` is passed and it could not retrieve the initial molecule.
- `KeyError` – If the dict does not contain the `canonical_isomeric_explicit_hydrogen_mapped_smiles`.

- **InvalidConformerError** – Silent error, if the conformer could not be attached.

classmethod from_rdkit(*rdmol*, *allow_undefined_stereo=False*, *hydrogens_are_explicit=False*)

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

Parameters

- **rdmol** (`rkit.RDMol`) – An RDKit molecule
- **allow_undefined_stereo** (`bool, default=False`) – If `False`, raises an exception if `rdmol` contains undefined stereochemistry.
- **hydrogens_are_explicit** (`bool, default=False`) – If `False`, RDKit will perform hydrogen addition using `Chem.AddHs`

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An OpenFF molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from rdkit import Chem
>>> from openff.toolkit.tests.utils import get_data_file_path
>>> rdmol = Chem.MolFromMolFile(get_data_file_path('systems/monomers/ethanol.sdf'))
>>> molecule = Molecule.from_rdkit(rdmol)
```

classmethod from_smiles(*smiles*, *hydrogens_are_explicit=False*, *toolkit_registry=<ToolkitRegistry containing The RDKit, AmberTools, Built-in Toolkit>*, *allow_undefined_stereo=False*)

Construct a Molecule from a SMILES representation

Parameters

- **smiles** (`str`) – The SMILES representation of the molecule.
- **hydrogens_are_explicit** (`bool, default = False`) – If False, the cheminformatics toolkit will perform hydrogen addition
- **toolkit_registry** (`ToolkitRegistry`) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES-to-molecule conversion
- **allow_undefined_stereo** (`bool, default=False`) – Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.

Returns **molecule** (*openff.toolkit.topology.Molecule*)

Examples

```
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
```

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters **serlialized** (str) – A TOML serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_topology(*topology*)

Return a Molecule representation of an OpenFF Topology containing a single Molecule object.

Parameters **topology** (Topology) – The Topology object containing a single Molecule object. Note that OpenMM and MDTraj Topology objects are not supported.

Returns **molecule** (*openff.toolkit.topology.Molecule*) – The Molecule object in the topology

Raises ValueError – If the topology does not contain exactly one molecule.

Examples

Create a molecule from a Topology object that contains exactly one molecule

```
>>> molecule = Molecule.from_topology(topology)
```

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters **serialized** (bytes) – An XML serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters **serialized** (str) – A YAML serialized representation of the object

Returns **instance** (*cls*) – Instantiated object

generate_conformers(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *n_conformers=10*, *rms_cutoff=None*, *clear_existing=True*, *make_carboxylic_acids_cis=True*)

Generate conformers for this molecule using an underlying toolkit.

If n_conformers=0, no toolkit wrapper will be called. If n_conformers=0 and clear_existing=True, molecule.conformers will be set to None.

Parameters

- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry or) – openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for SMILES-to-molecule conversion

- **n_conformers** (int, default=1) – The maximum number of conformers to produce
- **rms_cutoff** (openmm.unit.Quantity-wrapped float, in units of distance, optional, default=None) – The minimum RMS value at which two conformers are considered redundant and one is deleted. Precise implementation of this cutoff may be toolkit-dependent. If None, the cutoff is set to be the default value for each ToolkitWrapper (generally 1 Angstrom).
- **clear_existing** (bool, default=True) – Whether to overwrite existing conformers for the molecule
- **make_carboxylic_acids_cis** (bool, default=True) – Guarantee all conformers have exclusively cis carboxylic acid groups (COOH) by rotating the proton in any trans carboxylic acids 180 degrees around the C-O bond. Works around a bug in conformer generation by the OpenEye toolkit where trans COOH is much more common than it should be.

Examples

```
>>> molecule = Molecule.from_smiles('CCCCCC')
>>> molecule.generate_conformers()
```

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

generate_unique_atom_names()

Generate unique atom names using element name and number of times that element has occurred e.g. 'C1x', 'H1x', 'O1x', 'C2x', ...

The character 'x' is appended to these generated names to reduce the odds that they clash with an atom name or type imported from another source.

get_bond_between(*i*, *j*)

Returns the bond between two atoms

Parameters

- **i** (int or Atom) – Atoms or atom indices to check
- **j** (int or Atom) – Atoms or atom indices to check

Returns **bond** (*Bond*) – The bond between i and j.

property has_unique_atom_names: bool

True if the molecule has unique atom names, False otherwise.

property hierarchy_schemes: Dict[str, HierarchyScheme]

The hierarchy schemes available on the molecule.

Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see HierarchyScheme.

Returns **A dict of the form {str** (*HierarchyScheme*}) – The HierarchySchemes associated with the molecule.

property hill_formula: str

Get the Hill formula of the molecule

property impropers: Set[Tuple[Atom, Atom, Atom, Atom]]

Iterate over all improper torsions in the molecule.

Returns impropers (*set of tuple*) – An iterator of tuples, each containing the atoms making up a possible improper torsion.

See also:

smirnoff_impropers, amber_impropers

is_isomorphic_with(*other*, ***kwargs*)

Check if the molecule is isomorphic with the other molecule which can be an openff.toolkit.topology.Molecule or nx.Graph(). Full matching is done using the options described bellow.

Warning: This API is experimental and subject to change.

Parameters

- **other** (Molecule or nx.Graph()) –
- **aromatic_matching** (bool, default=True, optional) –
- **atoms.** (compare the formal charges attributes of the) –
- **formal_charge_matching** (bool, default=True, optional) –
- **atoms.** –
- **bond_order_matching** (bool, deafult=True, optional) –
- **bonds.** (compare the bond order on attributes of the) –
- **atom_stereochemistry_matching** (bool, default=True, optional) – If False, atoms' stereochemistry is ignored for the purpose of determining equality.
- **bond_stereochemistry_matching** (bool, default=True, optional) – If False, bonds' stereochemistry is ignored for the purpose of determining equality.
- **strip_pyrimidal_n_atom_stereo** (bool, default=True, optional) – If True, any stereochemistry defined around pyrimidal nitrogen stereocenters will be disregarded in the isomorphism check.
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for removing stereochemistry from pyrimidal nitrogens.

Returns isomorphic (*bool*)

property n_angles: int

Number of angles in the molecule.

property n_atoms: int

The number of Atom objects.

property n_bonds

The number of Bond objects in the molecule.

property n_conformers: int

The number of conformers for this molecule.

property n_impropers: int

Number of possible improper torsions in the molecule.

property n_particles: int

Use Molecule.n_atoms instead.

Type DEPRECATED

property n_propers: int

Number of proper torsions in the molecule.

property name: str

The name (or title) of the molecule

nth_degree_neighbors(*n_degrees*)

Return canonicalized pairs of atoms whose shortest separation is *exactly* n bonds. Only pairs with increasing atom indices are returned.

Parameters **n** (int) – The number of bonds separating atoms in each pair

Returns **neighbors** (*iterator of tuple of Atom*) – Tuples (len 2) of atom that are separated by n bonds.

Notes

The criteria used here relies on minimum distances; when there are multiple valid paths between atoms, such as atoms in rings, the shortest path is considered. For example, two atoms in "meta" positions with respect to each other in a benzene are separated by two paths, one length 2 bonds and the other length 4 bonds. This function would consider them to be 2 apart and would not include them if n=4 was passed.

ordered_connection_table_hash()

Compute an ordered hash of the atoms and bonds in the molecule

property partial_charges

Returns the partial charges (if present) on the molecule.

Returns **partial_charges** (*a openmm.unit.Quantity - wrapped numpy array [1 x n_atoms] or None*) – The partial charges on the molecule's atoms. Returns None if no charges have been specified.

particle(*index: int*) → *Atom*

DEPRECATED: Use Molecule.atom instead.

particle_index(*particle:* Atom) → int

DEPRECATED: Use Molecule.atom_index instead.

property particles: List[Atom]

Use Molecule.atoms instead.

Type DEPRECATED

property propers: Set[Tuple[Atom, Atom, Atom, Atom]]

Iterate over all proper torsions in the molecule

property properties: Dict[str, Any]

The properties dictionary of the molecule

remap(*mapping_dict*, *current_to_new=True*)

Remap all of the indexes in the molecule to match the given mapping dict

> **Warning:** This API is experimental and subject to change.

Parameters

- **mapping_dict** (dict,) – A dictionary of the mapping between indexes, this should start from 0.
- **current_to_new** (bool, default=True) – If this is True, then mapping_dict is of the form {current_index: new_index}; otherwise, it is of the form {new_index: current_index}

Returns new_molecule (*openff.toolkit.topology.molecule.Molecule*) – An openff.toolkit.Molecule instance with all attributes transferred, in the PDB order.

property smirnoff_impropers: Set[Tuple[Atom, Atom, Atom, Atom]]

Iterate over all impropers with trivalent centers, reporting the central atom second.

The central atom is reported second in each torsion. This method reports an improper for each trivalent atom in the molecule, whether or not any given force field would assign it improper torsion parameters.

Also note that this will return 6 possible atom orderings around each improper center. In current SMIRNOFF parameterization, three of these six orderings will be used for the actual assignment of the improper term and measurement of the angles. These three orderings capture the three unique angles that could be calculated around the improper center, therefore the sum of these three terms will always return a consistent energy.

The exact three orderings that will be applied during parameterization can not be determined in this method, since it requires sorting the atom indices, and those indices may change when this molecule is added to a Topology.

For more details on the use of three-fold ('trefoil') impropers, see https://openforcefield.github.io/standards/standards/smirnoff/#impropertorsions

Returns impropers (*set of tuple*) – An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed second in each tuple.

See also:

impropers, amber_impropers

strip_atom_stereochemistry(*smarts*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Delete stereochemistry information for certain atoms, if it is present. This method can be used to "normalize" molecules imported from different cheminformatics toolkits, which differ in which atom centers are considered stereogenic.

Parameters

- **smarts** (str or ChemicalEnvironment) – Tagged SMARTS with a single atom with index 1. Any matches for this atom will have any assigned stereocheistry information removed.

- **toolkit_registry** (a ToolkitRegistry or ToolkitWrapper object, optional,) – default=GLOBAL_TOOLKIT_REGISTRY ToolkitRegistry or ToolkitWrapper to use for I/O operations

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns **serialized** (*bytes*) – A BSON serialized representation of the objecft

to_dict()

Return a dictionary representation of the molecule.

Returns **molecule_dict** (*OrderedDict*) – A dictionary representation of the molecule.

to_file(*file_path*, *file_format*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Write the current molecule to a file or file-like object

Parameters

- **file_path** (str or file-like object) – A file-like object or the path to the file to be written.
- **file_format** (str) – Format specifier, one of ['MOL2', 'MOL2H', 'SDF', 'PDB', 'SMI', 'CAN', 'TDT'] Note that not all toolkits support all formats
- **toolkit_registry** (ToolkitRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=GLOBAL_TOOLKIT_REGISTRY ToolkitRegistry or ToolkitWrapper to use for file writing. If a Toolkit is passed, only the highest-precedence toolkit is used

Raises **ValueError** – If the requested file_format is not supported by one of the installed cheminformatics toolkits

Examples

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> molecule.to_file('imatinib.mol2', file_format='mol2')
>>> molecule.to_file('imatinib.sdf', file_format='sdf')
>>> molecule.to_file('imatinib.pdb', file_format='pdb')
```

to_hill_formula() → str

Generate the Hill formula of this molecule.

Returns **formula** (*the Hill formula of the molecule*)

:raises NotImplementedError : if the molecule is not of one of the specified types.:

to_inchi(*fixed_hydrogens=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Create an InChI string for the molecule using the requested toolkit backend. InChI is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

- **fixed_hydrogens** (bool, default=False) – If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for molecule-to-InChI conversion

Returns **inchi** (*str*) – The InChI string of the molecule.

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

to_inchikey(*fixed_hydrogens=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Create an InChIKey for the molecule using the requested toolkit backend. InChIKey is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

- **fixed_hydrogens** (bool, default=False) – If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolRegistry) – or openff.toolkit.utils.toolkits.ToolkitWrapper, optional, default=None ToolkitRegistry or ToolkitWrapper to use for molecule-to-InChIKey conversion

Returns **inchi_key** (*str*) – The InChIKey representation of the molecule.

Raises **InvalidToolkitRegistryError** – If an invalid object is passed as the toolkit_registry parameter

to_iupac(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Generate IUPAC name from Molecule

Returns

- **iupac_name** (*str*) – IUPAC name of the molecule
- *.. note :: This method requires the OpenEye toolkit to be installed.*

Examples

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> iupac_name = molecule.to_iupac()
```

to_json(*indent=None*) → str

Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters **indent** (int, optional, default=None) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*str*) – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation of the object

to_networkx()

Generate a NetworkX undirected graph from the molecule.

Nodes are Atoms labeled with atom indices and atomic elements (via the `element` node atrribute). Edges denote chemical bonds between Atoms.

Returns **graph** (*networkx.Graph*) – The resulting graph, with nodes (atoms) labeled with atom indices, elements, stereochemistry and aromaticity flags and bonds with two atom indices, bond order, stereochemistry, and aromaticity flags

Examples

Retrieve the bond graph for imatinib (OpenEye toolkit required)

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> nxgraph = molecule.to_networkx()
```

to_openeye(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*, *aromaticity_model=DEFAULT_AROMATICITY_MODEL*)

Create an OpenEye molecule

Requires the OpenEye toolkit to be installed.

Parameters **aromaticity_model** (`str, optional, default=DEFAULT_AROMATICITY_MODEL`) – The aromaticity model to use

Returns **oemol** (*openeye.oechem.OEMol*) – An OpenEye molecule

Examples

Create an OpenEye molecule from a Molecule

```
>>> molecule = Molecule.from_smiles('CC')
>>> oemol = molecule.to_openeye()
```

to_pickle()

Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns **serialized** (*str*) – A pickled representation of the object

to_qcschema(*multiplicity=1*, *conformer=0*, *extras=None*)

Create a QCElemental Molecule.

> **Warning:** This API is experimental and subject to change.

Parameters

- **multiplicity** (int, default=1,) – The multiplicity of the molecule; sets molecular_multiplicity field for QCElemental Molecule.
- **conformer** (int, default=0,) – The index of the conformer to use for the QCElemental Molecule geometry.
- **extras** (dict, default=None) – A dictionary that should be included in the extras field on the QCElemental Molecule. This can be used to include extra information, such as a smiles representation.

Returns *qcelemental.models.Molecule* – A validated QCElemental Molecule.

Examples

Create a QCElemental Molecule:

```
>>> import qcelemental as qcel
>>> mol = Molecule.from_smiles('CC')
>>> mol.generate_conformers(n_conformers=1)
>>> qcemol = mol.to_qcschema()
```

Raises

- **MissingOptionalDependencyError** – If qcelemental is not installed, the qcschema can not be validated.
- **InvalidConformerError** – No conformer found at the given index.

to_rdkit(*aromaticity_model=DEFAULT_AROMATICITY_MODEL*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Create an RDKit molecule

Requires the RDKit to be installed.

Parameters **aromaticity_model** (str, optional, default=DEFAULT_AROMATICITY_MODEL) – The aromaticity model to use

Returns **rdmol** (*rdkit.RDMol*) – An RDKit molecule

Examples

Convert a molecule to RDKit

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> rdmol = molecule.to_rdkit()
```

to_smiles(*isomeric=True*, *explicit_hydrogens=True*, *mapped=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Return a canonical isomeric SMILES representation of the current molecule. A partially mapped smiles can also be generated for atoms of interest by supplying an *atom_map* to the properties dictionary.

Note: RDKit and OpenEye versions will not necessarily return the same representation.

Parameters

- **isomeric** (`bool optional, default= True`) – return an isomeric smiles
- **explicit_hydrogens** (`bool optional, default=True`) – return a smiles string containing all hydrogens explicitly
- **mapped** (`bool optional, default=False`) – return a explicit hydrogen mapped smiles, the atoms to be mapped can be controlled by supplying an atom map into the properties dictionary. If no mapping is passed all atoms will be mapped in order, else an atom map dictionary from the current atom index to the map id should be supplied with no duplicates. The map ids (values) should start from 0 or 1.
- **toolkit_registry** (`openff.toolkit.utils.toolkits.ToolkitRegistry` or `openff.toolkit.utils.toolkits.ToolkitWrapper`, optional, default=None `ToolkitRegistry` or `ToolkitWrapper` to use for SMILES conversion

Returns **smiles** (*str*) – Canonical isomeric explicit-hydrogen SMILES

Examples

```
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> smiles = molecule.to_smiles()
```

to_toml()

Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns **serialized** (*str*) – A TOML serialized representation of the object

to_topology()

Return an OpenFF Topology representation containing one copy of this molecule

Returns **topology** (*openff.toolkit.topology.Topology*) – A Topology representation of this molecule

Examples

```
>>> molecule = Molecule.from_iupac('imatinib')
>>> topology = molecule.to_topology()
```

to_xml(*indent=2*)

Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters **indent** (int, optional, default=2) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns **serialized** (*str*) – A YAML serialized representation of the object

property torsions: Set[Tuple[Atom, Atom, Atom, Atom]]

Get an iterator over all i-j-k-l torsions. Note that i-j-k-i torsions (cycles) are excluded.

Returns **torsions** (*iterable of 4-Atom tuples*)

property total_charge

Return the total charge on the molecule

update_hierarchy_schemes(*iter_names=None*)

Infer a hierarchy from atom metadata according to the existing hierarchy schemes.

Hierarchy schemes allow iteration over groups of atoms according to their metadata. For more information, see HierarchyScheme.

Parameters **iter_names** (Iterable of str, Optional) – Only perceive hierarchy for HierarchySchemes that expose these iterator names. If not provided, all known hierarchies will be perceived, overwriting previous results if applicable.

11.1.3 openff.toolkit.topology.Topology

class openff.toolkit.topology.**Topology**(*other=None*)

A Topology is a chemical representation of a system containing one or more molecules appearing in a specified order.

> **Warning:** This API is experimental and subject to change.

Examples

Import some utilities

```
>>> from openmm import app
>>> from openff.toolkit.tests.utils import get_data_file_path, get_packmol_pdb_file_path
>>> pdb_filepath = get_packmol_pdb_file_path('cyclohexane_ethanol_0.4_0.6')
>>> monomer_names = ('cyclohexane', 'ethanol')
```

Create a Topology object from a PDB file and sdf files defining the molecular contents

```
>>> from openff.toolkit import Molecule, Topology
>>> pdbfile = app.PDBFile(pdb_filepath)
>>> sdf_filepaths = [get_data_file_path(f'systems/monomers/{name}.sdf') for name in␣
↪monomer_names]
>>> unique_molecules = [Molecule.from_file(sdf_filepath) for sdf_filepath in sdf_
↪filepaths]
>>> topology = Topology.from_openmm(pdbfile.topology, unique_molecules=unique_molecules)
```

Create a Topology object from a PDB file and IUPAC names of the molecular contents

```
>>> pdbfile = app.PDBFile(pdb_filepath)
>>> unique_molecules = [Molecule.from_iupac(name) for name in monomer_names]
>>> topology = Topology.from_openmm(pdbfile.topology, unique_molecules=unique_molecules)
```

Create an empty Topology object and add a few copies of a single benzene molecule

```
>>> topology = Topology()
>>> molecule = Molecule.from_iupac('benzene')
>>> molecule_topology_indices = [topology.add_molecule(molecule) for index in range(10)]
```

__init__(*other=None*)

Create a new Topology.

Parameters **other** (optional, default=None) – If specified, attempt to construct a copy of the Topology from the specified object. This might be a Topology object, or a file that can be used to construct a Topology object or serialized Topology object.

Methods

__init__([other])	Create a new Topology.
add_constraint(iatom, jatom[, distance])	Mark a pair of atoms as constrained.
add_molecule(molecule)	
assert_bonded(atom1, atom2)	Raise an exception if the specified atoms are not bonded in the topology.
atom(atom_topology_index)	Get the TopologyAtom at a given Topology atom index.
atom_index(atom)	Returns the index of a given atom in this topology
bond(bond_topology_index)	Get the TopologyBond at a given Topology bond index.

continues on next page

Table 3 – continued from previous page

chemical_environment_matches(query[, ...])	Retrieve all matches for a given chemical environment query.
copy_initializer(other)	
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(topology_dict)	Create a new Topology from a dictionary representation
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_mdtraj(mdtraj_topology[, unique_molecules])	Construct an OpenFF Topology object from an MDTraj Topology object.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_molecules(molecules)	Create a new Topology object containing one copy of each of the specified molecule(s).
from_openmm(openmm_topology[, unique_molecules])	Construct an OpenFF Topology object from an OpenMM Topology object.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
get_bond_between(i, j)	Returns the bond between two atoms
hierarchy_iterator(iter_name)	Get a HierarchyElement iterator from all of the molecules in this topology that provide the appropriately named iterator.
is_bonded(i, j)	Returns True if the two atoms are bonded
is_constrained(iatom, jatom)	Check if a pair of atoms are marked as constrained.
molecule(index)	Returns the molecule with a given index in this Topology.
molecule_atom_start_index(molecule)	Returns the index of a molecule's first atom in this topology
molecule_index(molecule)	Returns the index of a given molecule in this topology
nth_degree_neighbors(n_degrees)	Return canonicalized pairs of atoms whose shortest separation is *exactly* n bonds.
particle_index(particle)	DEPRECATED: Use Topology.atom_index instead.
to_bson()	Return a BSON serialized representation.
to_dict()	
to_file(filename, positions[, file_format, ...])	Save coordinates and topology to a PDB file.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_openmm([ensure_unique_atom_names])	Create an OpenMM Topology object.
to_pickle()	Return a pickle serialized representation.

continues on next page

Table 3 – continued from previous page

to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

amber_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.
angles	iterator over the angles in this Topology.
aromaticity_model	Get the aromaticity model applied to all molecules in the topology.
atoms	Returns an iterator over the atoms in this Topology.
bonds	Returns an iterator over the bonds in this Topology
box_vectors	Return the box vectors of the topology, if specified
constrained_atom_pairs	Returns the constrained atom pairs of the Topology
identical_molecule_groups	Returns groups of chemically identical molecules, identified by index and atom map.
impropers	iterator over the possible improper torsions in this Topology.
is_periodic	Return whether or not this Topology is intended to be described with periodic boundary conditions.
molecules	Returns an iterator over all the Molecules in this Topology
n_angles	number of angles in this Topology.
n_atoms	Returns the number of atoms in in this Topology.
n_bonds	Returns the number of Bonds in in this Topology.
n_impropers	number of possible improper torsions in this Topology.
n_molecules	Returns the number of molecules in this Topology
n_particles	Use Topology.n_atoms instead.
n_propers	number of proper torsions in this Topology.
n_reference_molecules	Use Topology.n_molecules instead.
n_topology_atoms	Use Topology.n_atoms instead.
n_topology_bonds	Use Topology.n_bonds instead.
n_topology_molecules	Use Topology.n_molecules instead.
n_topology_particles	Use Topology.n_particles instead.
particles	Use Topology.molecules instead.
propers	iterator over the proper torsions in this Topology.
reference_molecules	Get a list of reference molecules in this Topology.

continues on next page

Table 4 – continued from previous page

smirnoff_impropers	Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.
topology_atoms	Use Topology.atoms instead.
topology_bonds	Use Topology.bonds instead.
topology_molecules	Use Topology.molecules instead.
topology_particles	Use Topology.particles instead.

property reference_molecules: List[Molecule]

Get a list of reference molecules in this Topology.

Returns *iterable of openff.toolkit.topology.Molecule*

classmethod from_molecules(*molecules: Union[*Molecule*, List[*Molecule*]]*)

Create a new Topology object containing one copy of each of the specified molecule(s).

Parameters **molecules** (Molecule or iterable of Molecules) – One or more molecules to be added to the Topology

Returns **topology** (*Topology*) – The Topology created from the specified molecule(s)

assert_bonded(*atom1*, *atom2*)

Raise an exception if the specified atoms are not bonded in the topology.

Parameters

- **atom1** (Atom or int) – The atoms or atom topology indices to check to ensure they are bonded
- **atom2** (Atom or int) – The atoms or atom topology indices to check to ensure they are bonded

property aromaticity_model

Get the aromaticity model applied to all molecules in the topology.

Returns **aromaticity_model** (*str*) – Aromaticity model in use.

property box_vectors

Return the box vectors of the topology, if specified

Returns **box_vectors** (*unit-wrapped numpy array of shape (3, 3)*) – The unit-wrapped box vectors of this topology

property is_periodic

Return whether or not this Topology is intended to be described with periodic boundary conditions.

property constrained_atom_pairs: Dict[Tuple[int], Union[openff.units.units.Quantity, bool]]

Returns the constrained atom pairs of the Topology

Returns **constrained_atom_pairs** (*dict of Tuple[int]: Union[unit.Quantity, bool]*) – dictionary of the form {(atom1_topology_index, atom2_topology_index): distance}

property n_molecules

Returns the number of molecules in this Topology

Returns **n_molecules** (*Iterable of Molecule*)

property molecules: Generator[Union[Molecule, openff.toolkit.topology._mm_molecule._SimpleMolecule], None, None]

Returns an iterator over all the Molecules in this Topology

Returns molecules (*Iterable of Molecule*)

molecule(*index*)

Returns the molecule with a given index in this Topology.

Returns molecule (*openff.toolkit.topology.Molecule*)

property n_atoms: int

Returns the number of atoms in in this Topology.

Returns n_atoms (*int*)

property atoms: Generator[Atom, None, None]

Returns an iterator over the atoms in this Topology. These will be in ascending order of topology index.

Returns atoms (*Generator of TopologyAtom*)

atom_index(*atom*)

Returns the index of a given atom in this topology

Parameters atom (Atom) –

Returns index (*int*) – The index of the given atom in this topology

:raises AtomNotInTopologyError : If the given atom is not in this topology:

molecule_index(*molecule*)

Returns the index of a given molecule in this topology

Parameters molecule (FrozenMolecule) –

Returns index (*int*) – The index of the given molecule in this topology

:raises MoleculeNotInTopologyError : If the given atom is not in this topology:

molecule_atom_start_index(*molecule*)

Returns the index of a molecule's first atom in this topology

Parameters molecule (FrozenMolecule) –

Returns index (*int*)

property n_bonds

Returns the number of Bonds in in this Topology.

Returns n_bonds (*int*)

property bonds

Returns an iterator over the bonds in this Topology

Returns bonds (*Iterable of Bond*)

property n_angles

number of angles in this Topology.

Type *int*

property angles

iterator over the angles in this Topology.

Type Iterable of Tuple[Atom]

property n_propers

number of proper torsions in this Topology.

Type *int*

property propers

iterator over the proper torsions in this Topology.

Type Iterable of Tuple[TopologyAtom]

property n_impropers

number of possible improper torsions in this Topology.

Type *int*

property impropers

iterator over the possible improper torsions in this Topology.

Type Iterable of Tuple[TopologyAtom]

property smirnoff_impropers

Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom second in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current SMIRNOFF parameterization, three of these six orderings will be used for the actual assignment of the improper term and measurement of the angles. These three orderings capture the three unique angles that could be calculated around the improper center, therefore the sum of these three terms will always return a consistent energy.

For more details on the use of three-fold ('trefoil') impropers, see https://openforcefield.github.io/standards/standards/smirnoff/#impropertorsions

Returns impropers (*set of tuple*) – An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed second in each tuple.

See also:

`impropers`, `amber_impropers`

property amber_impropers

Iterate over improper torsions in the molecule, but only those with trivalent centers, reporting the central atom first in each improper.

Note that it's possible that a trivalent center will not have an improper assigned. This will depend on the force field that is used.

Also note that this will return 6 possible atom orderings around each improper center. In current AMBER parameterization, one of these six orderings will be used for the actual assignment of the improper term and measurement of the angle. This method does not encode the logic to determine which of the six orderings AMBER would use.

Returns impropers (*set of tuple*) – An iterator of tuples, each containing the indices of atoms making up a possible improper torsion. The central atom is listed first in each tuple.

See also:

`impropers`, `smirnoff_impropers`

nth_degree_neighbors(*n_degrees: int*)

Return canonicalized pairs of atoms whose shortest separation is *exactly* n bonds. Only pairs with increasing atom indices are returned.

Parameters n (`int`) – The number of bonds separating atoms in each pair

Returns neighbors (*iterator of tuple of Atom*) – Tuples (len 2) of atom that are separated by n bonds.

Notes

The criteria used here relies on minimum distances; when there are multiple valid paths between atoms, such as atoms in rings, the shortest path is considered. For example, two atoms in "meta" positions with respect to each other in a benzene are separated by two paths, one length 2 bonds and the other length 4 bonds. This function would consider them to be 2 apart and would not include them if `n=4` was passed.

chemical_environment_matches(*query*, *aromaticity_model='MDL'*, *unique=False*, *toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*)

Retrieve all matches for a given chemical environment query.

TODO:

- Do we want to generalize this to other kinds of queries too, like mdtraj DSL, pymol selections, atom index slices, etc? We could just call it topology.matches(query)

Parameters

- **query** (`str` or `ChemicalEnvironment`) – SMARTS string (with one or more tagged atoms) or `ChemicalEnvironment` query Query will internally be resolved to SMARTS using `query.as_smarts()` if it has an `.as_smarts` method.
- **aromaticity_model** (`str`) – Override the default aromaticity model for this topology and use the specified aromaticity model instead. Allowed values: ['MDL']

Returns matches (*list of Topology._ChemicalEnvironmentMatch*) – A list of tuples, containing the topology indices of the matching atoms.

property identical_molecule_groups

Returns groups of chemically identical molecules, identified by index and atom map.

Returns

- **identical_molecule_groups** ({*int:[[int: {int: int}]]*}) – A dict of the form {unique_mol_idx : [[topology_mol_idx, atom_map],...]. A dict where each key is the topology molecule index of a unique chemical species, and each value is a list describing all of the instances of that chemical species in the topology. Each instance is a two-membered list where the first element is the topology molecule index, and the second element is a dict describing the atom map from the unique molecule to the instance of it in the topology.

- *>>> from openff.toolkit import Molecule, Topology*
- *>>> # Create a water ordered as OHH*
- *>>> water1 = Molecule()*
- *>>> water1.add_atom(8, 0, False)*
- *>>> water1.add_atom(1, 0, False)*
- *>>> water1.add_atom(1, 0, False)*
- *>>> water1.add_bond(0, 1, 1, False)*
- *>>> water1.add_bond(0, 2, 1, False)*
- *>>> # Create a different water ordered as HOH*
- *>>> water2 = Molecule()*
- *>>> water2.add_atom(1, 0, False)*
- *>>> water2.add_atom(8, 0, False)*
- *>>> water2.add_atom(1, 0, False)*
- *>>> water2.add_bond(0, 1, 1, False)*
- *>>> water2.add_bond(1, 2, 1, False)*
- *>>> top = Topology.from_molecules([water1, water2])*
- *>>> top.identical_molecule_groups*
- **{0** (*[[0, {0: 0, 1: 1, 2: 2}], [1, {0: 1, 1: 0, 2: 2}]]*})

classmethod from_dict(*topology_dict*)

Create a new Topology from a dictionary representation

Parameters **topology_dict** (OrderedDict) – A dictionary representation of the topology.

Returns **topology** (*Topology*) – A Topology created from the dictionary representation

classmethod from_openmm(*openmm_topology*, *unique_molecules=None*)

Construct an OpenFF Topology object from an OpenMM Topology object.

This method guarantees that the order of atoms in the input OpenMM Topology will be the same as the ordering of atoms in the output OpenFF Topology. However it does not guarantee the order of the bonds will be the same.

Parameters

- **openmm_topology** (openmm.app.Topology) – An OpenMM Topology object
- **unique_molecules** (iterable of objects that can be used to construct unique Molecule objects) – All unique molecules must be provided, in any order, though multiple copies of each molecule are allowed. The atomic elements and bond connectivity will be used to match the reference molecules to molecule graphs appearing in the OpenMM Topology. If bond orders are present in the OpenMM Topology, these will be used in matching as well.

Returns **topology** (*openff.toolkit.topology.Topology*) – An OpenFF Topology object

to_openmm(*ensure_unique_atom_names=True*)

Create an OpenMM Topology object.

The atom metadata fields *residue_name*, *residue_number*, and *chain_id* are used to group atoms into OpenMM residues and chains.

Contiguously-indexed atoms with the same *residue_name*, *residue_number*, and *chain_id* will be put into the same OpenMM residue.

Contiguously-indexed residues with with the same *chain_id* will be put into the same OpenMM chain.

This method will never make an OpenMM chain or residue larger than the OpenFF Molecule that it came from. In other words, no chain or residue will span two OpenFF Molecules.

This method will **not** populate the OpenMM Topology with virtual sites.

Parameters **ensure_unique_atom_names** (`bool, optional. Default=True`) – Whether to check that the molecules in each molecule have unique atom names, and regenerate them if not. Note that this looks only at molecules, and does not guarantee uniqueness in the entire Topology.

Returns **openmm_topology** (*openmm.app.Topology*) – An OpenMM Topology object

to_file(*filename: str*, *positions*, *file_format: str = 'PDB'*, *keepIds=False*)

Save coordinates and topology to a PDB file.

Reference: https://github.com/openforcefield/openff-toolkit/issues/502

Notes:

1. Atom numbering may not remain same, for example if the atoms in water are numbered as 1001, 1002, 1003, they would change to 1, 2, 3. This doesn't affect the topology or coordinates or atom-ordering in any way.
2. Same issue with the amino acid names in the pdb file, they are not returned.

Parameters

- **filename** (`str`) – name of the pdb file to write to
- **positions** (`n_atoms x 3 numpy array or openmm.unit.Quantity-wrapped n_atoms x 3 iterable`) –

  Can be a

  - *openmm.unit.Quantity' object which has atomic positions as a list of unit-tagged `Vec3* objects
  - *openff.units.unit.Quantity* object which wraps a *numpy.ndarray* with units
  - (unitless) 2D *numpy.ndarray*, in which it is assumed that the positions are in units of Angstroms.

  For all data types, must have shape (n_atoms, 3) where n_atoms is the number of atoms in this topology.
- **file_format** (`str`) – Output file format. Case insensitive. Currently only supported value is "pdb".

classmethod from_mdtraj(*mdtraj_topology*, *unique_molecules=None*)

Construct an OpenFF Topology object from an MDTraj Topology object.

Parameters

- **mdtraj_topology** (Topology) – An MDTraj Topology object
- **unique_molecules** (iterable of objects that can be used to construct unique Molecule objects) – All unique molecules must be provided, in any order, though multiple copies of each molecule are allowed. The atomic elements and bond connectivity will be used to match the reference molecules to molecule graphs appearing in the MDTraj Topology. If bond orders are present in the MDTraj Topology, these will be used in matching as well.

Returns **topology** (*openff.toolkit.topology.Topology*) – An OpenFF Topology object

get_bond_between(*i*, *j*)

Returns the bond between two atoms

Parameters

- **i** (int or TopologyAtom) – Atoms or atom indices to check
- **j** (int or TopologyAtom) – Atoms or atom indices to check

Returns **bond** (*TopologyBond*) – The bond between i and j.

is_bonded(*i*, *j*)

Returns True if the two atoms are bonded

Parameters

- **i** (int or TopologyAtom) – Atoms or atom indices to check
- **j** (int or TopologyAtom) – Atoms or atom indices to check

Returns **is_bonded** (*bool*) – True if atoms are bonded, False otherwise.

atom(*atom_topology_index*)

Get the TopologyAtom at a given Topology atom index.

Parameters **atom_topology_index** (int) – The index of the TopologyAtom in this Topology

Returns *An openff.toolkit.topology.TopologyAtom*

bond(*bond_topology_index*)

Get the TopologyBond at a given Topology bond index.

Parameters **bond_topology_index** (int) – The index of the TopologyBond in this Topology

Returns *An openff.toolkit.topology.TopologyBond*

add_constraint(*iatom*, *jatom*, *distance=True*)

Mark a pair of atoms as constrained.

Constraints between atoms that are not bonded (e.g., rigid waters) are permissible.

Parameters

- **iatom** (Atom) – Atoms to mark as constrained These atoms may be bonded or not in the Topology
- **jatom** (Atom) – Atoms to mark as constrained These atoms may be bonded or not in the Topology
- **distance** (unit-wrapped float, optional, default=True) – Constraint distance True if distance has yet to be determined False if constraint is to be removed

is_constrained(*iatom*, *jatom*)

Check if a pair of atoms are marked as constrained.

Parameters

- **iatom** (int) – Indices of atoms to mark as constrained.
- **jatom** (int) – Indices of atoms to mark as constrained.

Returns distance (*unit-wrapped float or bool*) – True if constrained but constraints have not yet been applied Distance if constraint has already been added to Topology

hierarchy_iterator(*iter_name*)

Get a HierarchyElement iterator from all of the molecules in this topology that provide the appropriately named iterator. This iterator will yield HierarchyElements sorted first by the order that molecules are listed in the Topology, and second by the specific sorting of HierarchyElements defined in each molecule.

Parameters iter_name (string) – The iterator name associated with the HierarchyScheme to retrieve (for example 'residues' or 'chains')

Returns *iterator of HierarchyElement*

property n_topology_atoms: int

Use Topology.n_atoms instead.

Type DEPRECATED

property topology_atoms

Use Topology.atoms instead.

Type DEPRECATED

property n_topology_bonds: int

Use Topology.n_bonds instead.

Type DEPRECATED

property topology_bonds

Use Topology.bonds instead.

Type DEPRECATED

property n_topology_particles: int

Use Topology.n_particles instead.

Type DEPRECATED

property topology_particles

Use Topology.particles instead.

Type DEPRECATED

property n_reference_molecules: int

Use Topology.n_molecules instead.

Type DEPRECATED

property n_topology_molecules: int

Use Topology.n_molecules instead.

Type DEPRECATED

property topology_molecules

Use Topology.molecules instead.

Type DEPRECATED

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters **serialized** (bytes) – A BSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters **serialized** (str) – A JSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters **serialized** (bytes) – A MessagePack-encoded bytes serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_pickle(*serialized*)

Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters **serialized** (str) – A pickled representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters **serlialized** (str) – A TOML serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters **serialized** (bytes) – An XML serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters **serialized** (str) – A YAML serialized representation of the object

Returns **instance** (*cls*) – Instantiated object

property n_particles: int

Use Topology.n_atoms instead.

Type DEPRECATED

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns **serialized** (*bytes*) – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters **indent** (int, optional, default=None) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*str*) – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation of the object

to_pickle()

Return a pickle serialized representation.

Warning: This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns **serialized** (*str*) – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns **serialized** (*str*) – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters **indent** (int, optional, default=2) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns **serialized** (*str*) – A YAML serialized representation of the object

property particles

Use Topology.molecules instead.

Type DEPRECATED

particle_index(*particle*) → int

DEPRECATED: Use Topology.atom_index instead.

11.2 Secondary objects

Particle	Base class for all particles in a molecule.
Atom	A chemical atom.
Bond	Chemical bond representation.
ValenceDict	Enforce uniqueness in atom indices.
ImproperDict	Symmetrize improper torsions.
HierarchyScheme	Perceives hierarchy elements from the metadata of atoms in a Molecule.
HierarchyElement	An element in a metadata hierarchy scheme, such as a residue or chain.

11.2.1 openff.toolkit.topology.Particle

class openff.toolkit.topology.**Particle**

Base class for all particles in a molecule.

A particle object could be an Atom or similar.

> **Warning:** This API is experimental and subject to change.

__init__()

Methods

`__init__`()	
`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(d)	Static constructor from dictionary representation.
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`from_messagepack`(serialized)	Instantiate an object from a MessagePack serialized representation.
`from_pickle`(serialized)	Instantiate an object from a pickle serialized representation.
`from_toml`(serialized)	Instantiate an object from a TOML serialized representation.
`from_xml`(serialized)	Instantiate an object from an XML serialized representation.
`from_yaml`(serialized)	Instantiate from a YAML serialized representation.
`to_bson`()	Return a BSON serialized representation.
`to_dict`()	Convert to dictionary representation.
`to_json`([indent])	Return a JSON serialized representation.
`to_messagepack`()	Return a MessagePack representation.
`to_pickle`()	Return a pickle serialized representation.
`to_toml`()	Return a TOML serialized representation.
`to_xml`([indent])	Return an XML representation.
`to_yaml`()	Return a YAML serialized representation.

Attributes

`molecule`	The `Molecule` this particle is part of.
`molecule_particle_index`	Returns the index of this particle in its molecule
`name`	The name of the particle

property molecule

The `Molecule` this particle is part of.

property molecule_particle_index

Returns the index of this particle in its molecule

property name

The name of the particle

to_dict()

Convert to dictionary representation.

classmethod from_dict(*d*)

Static constructor from dictionary representation.

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters **serialized** (bytes) – A BSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters **serialized** (str) – A JSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters **serialized** (bytes) – A MessagePack-encoded bytes serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_pickle(*serialized*)

Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters **serialized** (str) – A pickled representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters **serlialized** (str) – A TOML serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters **serialized** (bytes) – An XML serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters **serialized** (str) – A YAML serialized representation of the object

Returns **instance** (*cls*) – Instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns **serialized** (*bytes*) – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters **indent** (`int, optional, default=None`) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*str*) – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation of the object

to_pickle()

Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns **serialized** (*str*) – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns **serialized** (*str*) – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters **indent** (`int, optional, default=2`) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns **serialized** (*str*) – A YAML serialized representation of the object

11.2.2 openff.toolkit.topology.Atom

class openff.toolkit.topology.**Atom**(*atomic_number*, *formal_charge*, *is_aromatic*, *name=None*, *molecule=None*, *stereochemistry=None*, *metadata=None*)

A chemical atom.

> **Warning:** This API is experimental and subject to change.

__init__(*atomic_number*, *formal_charge*, *is_aromatic*, *name=None*, *molecule=None*, *stereochemistry=None*, *metadata=None*)

Create an immutable Atom object.

Object is serializable and immutable.

Parameters

- **atomic_number** (int) – Atomic number of the atom
- **formal_charge** (int or openff.units.unit.Quantity-wrapped int with dimension "charge") – Formal charge of the atom
- **is_aromatic** (bool) – If True, atom is aromatic; if False, not aromatic
- **stereochemistry** (str, optional, default=None) – Either 'R' or 'S' for specified stereochemistry, or None for ambiguous stereochemistry
- **name** (str, optional, default=None) – An optional name to be associated with the atom
- **metadata** (dict[str: (int, str)], default=None) – An optional dictionary where keys are strings and values are strings or ints. This is intended to record atom-level information used to inform hierarchy definition and iteration, such as grouping atom by residue and chain.

Examples

Create a non-aromatic carbon atom

```
>>> atom = Atom(6, 0, False)
```

Create a chiral carbon atom

```
>>> atom = Atom(6, 0, False, stereochemistry='R', name='CT')
```

Methods

`__init__`(atomic_number, formal_charge, ...)	Create an immutable Atom object.
`add_bond`(bond)	Adds a bond that this atom is involved in .
`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(atom_dict)	Create an Atom from a dict representation.
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`from_messagepack`(serialized)	Instantiate an object from a MessagePack serialized representation.
`from_pickle`(serialized)	Instantiate an object from a pickle serialized representation.
`from_toml`(serialized)	Instantiate an object from a TOML serialized representation.
`from_xml`(serialized)	Instantiate an object from an XML serialized representation.
`from_yaml`(serialized)	Instantiate from a YAML serialized representation.
`is_bonded_to`(atom2)	Determine whether this atom is bound to another atom
`is_in_ring`([toolkit_registry])	Return whether or not this atom is in a ring(s) (of any size)
`to_bson`()	Return a BSON serialized representation.
`to_dict`()	Return a dict representation of the atom.
`to_json`([indent])	Return a JSON serialized representation.
`to_messagepack`()	Return a MessagePack representation.
`to_pickle`()	Return a pickle serialized representation.
`to_toml`()	Return a TOML serialized representation.
`to_xml`([indent])	Return an XML representation.
`to_yaml`()	Return a YAML serialized representation.

Attributes

atomic_number	The integer atomic number of the atom.
bonded_atoms	The list of Atom objects this atom is involved in bonds with
bonds	The list of Bond objects this atom is involved in.
formal_charge	The atom's formal charge
is_aromatic	The atom's is_aromatic flag
mass	The standard atomic weight (abundance-weighted isotopic mass) of the atomic site.
metadata	The atom's metadata dictionary
molecule	The Molecule this particle is part of.
molecule_atom_index	The index of this Atom within the the list of atoms in the parent Molecule.
molecule_particle_index	Returns the index of this particle in its molecule
name	The name of this atom, if any
partial_charge	The partial charge of the atom, if any.
stereochemistry	The atom's stereochemistry (if defined, otherwise None)
symbol	Return the symbol implied by the atomic number of this atom

add_bond(*bond*)

Adds a bond that this atom is involved in .. todo :: Is this how we want to keep records?

Parameters **bond** (an openff.toolkit.topology.molecule.Bond) – A bond involving this atom

to_dict()

Return a dict representation of the atom.

classmethod from_dict(*atom_dict*)

Create an Atom from a dict representation.

property metadata

The atom's metadata dictionary

property formal_charge

The atom's formal charge

property partial_charge

The partial charge of the atom, if any.

Returns *unit-wrapped float with dimension of atomic charge, or None if no charge has been specified*

property is_aromatic

The atom's is_aromatic flag

property stereochemistry

The atom's stereochemistry (if defined, otherwise None)

property atomic_number: int

The integer atomic number of the atom.

property symbol: str

Return the symbol implied by the atomic number of this atom

property mass: Quantity

The standard atomic weight (abundance-weighted isotopic mass) of the atomic site.

The mass is reported in units of Dalton.

property name

The name of this atom, if any

property bonds

The list of Bond objects this atom is involved in.

property bonded_atoms

The list of Atom objects this atom is involved in bonds with

is_bonded_to(*atom2*)

Determine whether this atom is bound to another atom

Parameters **atom2** (Atom) – a different atom in the same molecule

Returns *bool* – Whether this atom is bound to atom2

is_in_ring(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*) → bool

Return whether or not this atom is in a ring(s) (of any size)

This Atom is expected to be attached to a molecule (*Atom.molecule*).

Parameters **toolkit_registry** (ToolkitRegistry, default=GLOBAL_TOOLKIT_REGISTRY) – ToolkitRegistry to use to enumerate the tautomers.

property molecule_atom_index

The index of this Atom within the the list of atoms in the parent Molecule.

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters **serialized** (bytes) – A BSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters **serialized** (str) – A JSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters **serialized** (bytes) – A MessagePack-encoded bytes serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_pickle(*serialized*)

Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters **serialized** (str) – A pickled representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters **serlialized** (str) – A TOML serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters **serialized** (bytes) – An XML serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters **serialized** (str) – A YAML serialized representation of the object

Returns **instance** (*cls*) – Instantiated object

property molecule

The Molecule this particle is part of.

property molecule_particle_index

Returns the index of this particle in its molecule

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns **serialized** (*bytes*) – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters **indent** (int, optional, default=None) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*str*) – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation of the object

to_pickle()

Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns **serialized** (*str*) – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns **serialized** (*str*) – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters **indent** (`int, optional, default=2`) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns **serialized** (*str*) – A YAML serialized representation of the object

11.2.3 openff.toolkit.topology.Bond

class openff.toolkit.topology.**Bond**(*atom1*, *atom2*, *bond_order*, *is_aromatic*, *fractional_bond_order=None*, *stereochemistry=None*)

Chemical bond representation.

> **Warning:** This API is experimental and subject to change.

Attributes

- **atom1, atom2** (*openff.toolkit.topology.Atom*) – Atoms involved in the bond
- **bond_order** (*int*) – The (integer) bond order of this bond.
- **is_aromatic** (*bool*) – Whether or not this bond is aromatic.

- **fractional_bond_order** (*float, optional*) – The fractional bond order, or partial bond order of this bond.
- **stereochemstry** (*str, optional, default=None*) – A string representing this stereochemistry of this bond.
- **.. warning :: This API is experimental and subject to change.**

__init__(*atom1*, *atom2*, *bond_order*, *is_aromatic*, *fractional_bond_order=None*, *stereochemistry=None*)

Create a new chemical bond.

Methods

__init__(atom1, atom2, bond_order, is_aromatic)	Create a new chemical bond.
from_bson(serialized)	Instantiate an object from a BSON serialized representation.
from_dict(molecule, d)	Create a Bond from a dict representation.
from_json(serialized)	Instantiate an object from a JSON serialized representation.
from_messagepack(serialized)	Instantiate an object from a MessagePack serialized representation.
from_pickle(serialized)	Instantiate an object from a pickle serialized representation.
from_toml(serialized)	Instantiate an object from a TOML serialized representation.
from_xml(serialized)	Instantiate an object from an XML serialized representation.
from_yaml(serialized)	Instantiate from a YAML serialized representation.
is_in_ring([toolkit_registry])	Return whether or not this bond is in a ring(s) (of any size)
to_bson()	Return a BSON serialized representation.
to_dict()	Return a dict representation of the bond.
to_json([indent])	Return a JSON serialized representation.
to_messagepack()	Return a MessagePack representation.
to_pickle()	Return a pickle serialized representation.
to_toml()	Return a TOML serialized representation.
to_xml([indent])	Return an XML representation.
to_yaml()	Return a YAML serialized representation.

Attributes

atom1	
atom1_index	
atom2	
atom2_index	
atoms	
bond_order	
fractional_bond_order	
is_aromatic	
molecule	
molecule_bond_index	The index of this Bond within the the list of bonds in Molecules.
stereochemistry	

to_dict()

Return a dict representation of the bond.

classmethod from_dict(*molecule*, *d*)

Create a Bond from a dict representation.

property molecule_bond_index

The index of this Bond within the the list of bonds in Molecules.

is_in_ring(*toolkit_registry=GLOBAL_TOOLKIT_REGISTRY*) → bool

Return whether or not this bond is in a ring(s) (of any size)

This Bond is expected to be attached to a molecule (*Bond.molecule*).

Note: Bonds containing atoms that are only in separate rings, i.e. the central bond in a biphenyl, are not considered to be bonded by this criteria.

Parameters **toolkit_registry** (ToolkitRegistry, default=GLOBAL_TOOLKIT_REGISTRY) – ToolkitRegistry to use to enumerate the tautomers.

Returns **is_in_ring** (*bool*) – Whether or not this bond is in a ring.

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters **serialized** (bytes) – A BSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters **serialized** (str) – A JSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters **serialized** (bytes) – A MessagePack-encoded bytes serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_pickle(*serialized*)

Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters **serialized** (str) – A pickled representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters **serlialized** (str) – A TOML serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

classmethod from_xml(*serialized*)

Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters **serialized** (bytes) – An XML serialized representation

Returns **instance** (*cls*) – Instantiated object.

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters **serialized** (str) – A YAML serialized representation of the object

Returns **instance** (*cls*) – Instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns **serialized** (*bytes*) – A BSON serialized representation of the objecft

to_json(*indent=None*) → str

Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters **indent** (int, optional, default=None) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*str*) – A JSON serialized representation of the object

to_messagepack()

Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation of the object

to_pickle()

Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns **serialized** (*str*) – A pickled representation of the object

to_toml()

Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns **serialized** (*str*) – A TOML serialized representation of the object

to_xml(*indent=2*)

Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters **indent** (int, optional, default=2) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation.

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns **serialized** (*str*) – A YAML serialized representation of the object

11.2.4 openff.toolkit.topology.ValenceDict

class openff.toolkit.topology.**ValenceDict**(**args*, ***kwargs*)

Enforce uniqueness in atom indices.

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
get(k[,d])	
index_of(key[, possible])	Generates a canonical ordering of the equivalent permutations of key (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is.
items()	
key_transform(key)	Reverse tuple if first element is larger than last element.
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

static key_transform(*key*)

Reverse tuple if first element is larger than last element.

static index_of(*key*, *possible=None*)

Generates a canonical ordering of the equivalent permutations of key (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is. This method is useful when multiple SMARTS patterns might match the same atoms, but local molecular symmetry or the use of wildcards in the SMARTS could make the matches occur in arbitrary order.

This method can be restricted to a subset of the canonical orderings, by providing the optional possible keyword argument. If provided, the index returned by this method will be the index of the element in possible after undergoing the same canonical sorting as above.

Parameters

- **key** (iterable of int) – A valid key for ValenceDict
- **possible** (iterable of iterable of int, optional. default=``None``) – A subset of the possible orderings that this match might take.

Returns **index** (*int*)

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

11.2.5 openff.toolkit.topology.ImproperDict

class openff.toolkit.topology.**ImproperDict**(**args*, ***kwargs*)
Symmetrize improper torsions.

__init__(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`clear`()	
`get`(k[,d])	
`index_of`(key[, possible])	Generates a canonical ordering of the equivalent permutations of `key` (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is.
`items`()	
`key_transform`(key)	Reorder tuple in numerical order except for element[1] which is the central atom; it retains its position.
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
`popitem`()	as a 2-tuple; but raise KeyError if D is empty.
`setdefault`(k[,d])	
`update`([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
`values`()	

static key_transform(*key*)

Reorder tuple in numerical order except for element[1] which is the central atom; it retains its position.

static index_of(*key*, *possible=None*)

Generates a canonical ordering of the equivalent permutations of `key` (equivalent rearrangements of indices) and identifies which of those possible orderings this particular ordering is. This method is useful when multiple SMARTS patterns might match the same atoms, but local molecular symmetry or the use of wildcards in the SMARTS could make the matches occur in arbitrary order.

This method can be restricted to a subset of the canonical orderings, by providing the optional `possible` keyword argument. If provided, the index returned by this method will be the index of the element in `possible` after undergoing the same canonical sorting as above.

Parameters

- **key** (`iterable of int`) – A valid key for ValenceDict
- **possible** (`iterable of iterable of int, optional. default=``None```) – A subset of the possible orderings that this match might take.

Returns **index** (*int*)

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

11.2.6 openff.toolkit.topology.HierarchyScheme

class openff.toolkit.topology.**HierarchyScheme**(*parent:* FrozenMolecule, *uniqueness_criteria: Union[Tuple[str], List[str]]*, *iterator_name: str*)

Perceives hierarchy elements from the metadata of atoms in a `Molecule`.

The Open Force Field Toolkit has no native understanding of hierarchical atom organisation schemes common to other biomolecular software, such as "residues" or "chains" (see *Hierarchy data (chains and residues)*). To facilitate iterating over groups of atoms, a `HierarchyScheme` can be used to collect atoms into `HierarchyElements`, groups of atoms that share the same values for certain metadata elements. Metadata elements are stored in the `Atom.properties` attribute.

Hierarchy schemes are not updated dynamically; if a `Molecule` with hierarchy schemes changes, `Molecule.update_hierarchy_schemes()` must be called before the scheme is iterated over again or else the grouping may be incorrect.

A `HierarchyScheme` contains the information needed to perceive `HierarchyElement` objects from a `Molecule` containing atoms with metadata.

__init__(*parent:* FrozenMolecule, *uniqueness_criteria: Union[Tuple[str], List[str]]*, *iterator_name: str*)

Create a new hierarchy scheme for iterating over groups of atoms.

Parameters

- **parent** – The `Molecule` to which this scheme belongs.
- **uniqueness_criteria** – The names of `Atom` metadata entries that define this scheme. An atom belongs to a `HierarchyElement` only if its metadata has the same values for these criteria as the other atoms in the `HierarchyElement`.
- **iterator_name** – The name of the iterator that will be exposed to access the hierarchy elements generated by this scheme

Methods

`__init__`(parent, uniqueness_criteria, ...)	Create a new hierarchy scheme for iterating over groups of atoms.
`add_hierarchy_element`(identifier, atom_indices)	Instantiate a new HierarchyElement belonging to this HierarchyScheme.
`perceive_hierarchy`()	Prepare the parent `Molecule` for iteration according to this scheme.
`sort_hierarchy_elements`()	Semantically sort the HierarchyElements belonging to this object, according to their identifiers.
`to_dict`()	Serialize this object to a basic dict of strings, ints, and floats

to_dict()

Serialize this object to a basic dict of strings, ints, and floats

perceive_hierarchy()

Prepare the parent `Molecule` for iteration according to this scheme.

Groups the atoms of the parent of this `HierarchyScheme` according to their metadata, and creates `HierarchyElement` objects suitable for iteration over the parent. Atoms missing the metadata fields in this object's `uniqueness_criteria` tuple will have those spots populated with the string `'None'`.

This method overwrites the scheme's `hierarchy_elements` attribute in place. Each `HierarchyElement` in the scheme's *hierarchy_elements* attribute is *static* — that is, it is updated only when *perceive_hierarchy()* is called, and *not* on-the-fly when atom metadata is modified.

add_hierarchy_element(*identifier*, *atom_indices*)

Instantiate a new HierarchyElement belonging to this HierarchyScheme.

This is the main way to instantiate new HierarchyElements.

Parameters

- **identifier** (`tuple of str and int`) – Tuple of metadata values (not keys) that define the uniqueness criteria for this element
- **atom_indices** (`iterable int`) – The indices of atoms in `scheme.parent` that are in this element

sort_hierarchy_elements()

Semantically sort the HierarchyElements belonging to this object, according to their identifiers.

11.2.7 openff.toolkit.topology.HierarchyElement

class openff.toolkit.topology.**HierarchyElement**(*scheme*, *identifier*, *atom_indices*)

An element in a metadata hierarchy scheme, such as a residue or chain.

__init__(*scheme*, *identifier*, *atom_indices*)

Create a new hierarchy element.

Parameters

- **scheme** (`HierarchyScheme`) – The scheme to which this `HierarchyElement` belongs

- **id** (tuple of str and int) – Tuple of metadata values (not keys) that define the uniqueness criteria for this element
- **atom_indices** (iterable int) – The indices of particles in scheme.parent that are in this element

Methods

__init__(scheme, identifier, atom_indices)	Create a new hierarchy element.
atom(index)	Get atom with a specified index.
to_dict()	Serialize this object to a basic dict of strings, ints, and floats

Attributes

atoms
parent

to_dict()

Serialize this object to a basic dict of strings, ints, and floats

atom(*index: int*)

Get atom with a specified index.

Parameters **index** (int) –

Returns **atom** (*openff.toolkit.topology.molecule.Atom*)

CHAPTER

TWELVE

FORCE FIELD TYPING TOOLS

12.1 Chemical environments

Tools for representing and operating on chemical environments

ChemicalEnvironment	Chemical environment abstract base class used for validating SMIRKS

12.1.1 openff.toolkit.typing.chemistry.ChemicalEnvironment

class openff.toolkit.typing.chemistry.**ChemicalEnvironment**(*smirks=None*, *label=None*, *validate_parsable=True*, *validate_valence_type=True*, *toolkit_registry=None*)

Chemical environment abstract base class used for validating SMIRKS

__init__(*smirks=None*, *label=None*, *validate_parsable=True*, *validate_valence_type=True*, *toolkit_registry=None*)

Initialize a chemical environment abstract base class.

smirks = string, optional if smirks is not None, a chemical environment is built from the provided SMIRKS string

label = anything, optional intended to be used to label this chemical environment could be a string, int, or float, or anything

validate_parsable: bool, optional, default=True If specified, ensure the provided smirks is parsable

validate_valence_type [bool, optional, default=True] If specified, ensure the tagged atoms are appropriate to the specified valence type

toolkit_registry = string or ToolkitWrapper or ToolkitRegistry. Default = None Either a ToolkitRegistry, ToolkitWrapper, or the strings 'openeye' or 'rdkit', indicating the backend to use for validating the correct connectivity of the SMIRKS during initialization. If None, this function will use the GLOBAL_TOOLKIT_REGISTRY

Raises

- **SMIRKSParsingError** – if smirks was unparsable
- **SMIRKSMismatchError** – if smirks did not have expected connectivity between tagged atoms and validate_valence_type=True

Methods

`__init__`([smirks, label, validate_parsable, ...])	Initialize a chemical environment abstract base class.
`get_type`([toolkit_registry])	Return the valence type implied by the connectivity of the bound atoms in this ChemicalEnvironment.
`validate`([validate_valence_type, ...])	Returns True if the underlying smirks is the correct valence type, False otherwise.
`validate_smirks`(smirks[, validate_parsable, ...])	Check the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

validate(*validate_valence_type=True*, *toolkit_registry=None*)

Returns True if the underlying smirks is the correct valence type, False otherwise. If the expected type is None, this method always returns True.

validate_valence_type [bool, optional, default=True] If specified, ensure the tagged atoms are appropriate to the specified valence type

toolkit_registry = ToolkitWrapper or ToolkitRegistry. Default = None Either a ToolkitRegistry or ToolkitWrapper, indicating the backend to use for validating the correct connectivity of the SMIRKS during initialization. If None, this function will use the GLOBAL_TOOLKIT_REGISTRY

Raises

- **SMIRKSParsingError** – if smirks was unparsable
- **SMIRKSMismatchError** – if smirks did not have expected connectivity between tagged atoms and validate_valence_type=True

classmethod validate_smirks(*smirks*, *validate_parsable=True*, *validate_valence_type=True*, *toolkit_registry=None*)

Check the provided SMIRKS string is valid, and if requested, tags atoms appropriate to the specified valence type.

Parameters

- **smirks** (`str`) – The SMIRKS expression to validate
- **validate_parsable** (`bool, optional, default=True`) – If specified, ensure the provided smirks is parsable
- **validate_valence_type** (`bool, optional, default=True`) – If specified, ensure the tagged atoms are appropriate to the specified valence type
- **None** (`toolkit_registry = string or ToolkitWrapper or ToolkitRegistry. Default =`) – Either a ToolkitRegistry, ToolkitWrapper, or the strings 'openeye' or 'rdkit', indicating the backend to use for validating the correct connectivity of the SMIRKS during initialization. If None, this function will use the GLOBAL_TOOLKIT_REGISTRY

Raises

- **SMIRKSParsingError** – if smirks was unparsable

- **SMIRKSMismatchError** – if smirks did not have expected connectivity between tagged atoms and validate_valence_type=True

get_type(*toolkit_registry=None*)

Return the valence type implied by the connectivity of the bound atoms in this ChemicalEnvironment.

Parameters **toolkit_registry** (openff.toolkit.utils.ToolkitRegistry or openff.toolkit.utils.ToolkitWrapper) – The cheminformatics toolkit to use for parsing the smirks

Returns **valence_type** (*str*) – One of "Atom", "Bond", "Angle", "ProperTorsion", "ImproperTorsion", or None. If tagged atoms are not connected in a known pattern this method will return None.

Raises **SMIRKSParsingError** – if smirks was unparsable

12.2 Force field typing engines

Engines for applying parameters to chemical systems

12.2.1 The SMIRks-Native Open Force Field (SMIRNOFF)

A reference implementation of the SMIRNOFF specification for parameterizing biomolecular systems

ForceField

The ForceField class is a primary part of the top-level toolkit API. ForceField objects are initialized from SMIRNOFF data sources (e.g. an OFFXML file). For a basic example of OpenMM System creation using a ForceField, see examples/SMIRNOFF_simulation.

ForceField	A factory that assigns SMIRNOFF parameters to a molecular system
get_available_force_fields	Get the filenames of all available .offxml force field files.

openff.toolkit.typing.engines.smirnoff.ForceField

class openff.toolkit.typing.engines.smirnoff.**ForceField**(**sources*, *aromaticity_model=DEFAULT_AROMATICITY_MODEL*, *parameter_handler_classes=None*, *parameter_io_handler_classes=None*, *disable_version_check=False*, *allow_cosmetic_attributes=False*, *load_plugins=False*)

A factory that assigns SMIRNOFF parameters to a molecular system

ForceField is a factory that constructs an OpenMM openmm.System object from a openff.toolkit.topology.Topology object defining a (bio)molecular system containing one or more molecules.

When a `ForceField` object is created from one or more specified SMIRNOFF serialized representations, all `ParameterHandler` subclasses currently imported are identified and registered to handle different sections of the SMIRNOFF force field definition file(s).

All `ParameterIOHandler` subclasses currently imported are identified and registered to handle different serialization formats (such as XML).

The force field definition is processed by these handlers to populate the `ForceField` object model data structures that can easily be manipulated via the API:

Processing a `Topology` object defining a chemical system will then call all `ParameterHandler` objects in an order guaranteed to satisfy the declared processing order constraints of each `ParameterHandler`.

Examples

Create a new ForceField containing the smirnoff99Frosst parameter set:

```
>>> from openff.toolkit import ForceField
>>> forcefield = ForceField('test_forcefields/test_forcefield.offxml')
```

Create an OpenMM system from a `openff.toolkit.topology.Topology` object:

```
>>> from openff.toolkit import Molecule, Topology
>>> ethanol = Molecule.from_smiles('CCO')
>>> topology = Topology.from_molecules(molecules=[ethanol])
>>> system = forcefield.create_openmm_system(topology)
```

Modify the long-range electrostatics method:

```
>>> forcefield.get_parameter_handler('Electrostatics').periodic_potential = 'PME'
```

Inspect the first few vdW parameters:

```
>>> low_precedence_parameters = forcefield.get_parameter_handler('vdW').parameters[0:3]
```

Retrieve the vdW parameters by SMIRKS string and manipulate it:

```
>>> parameter = forcefield.get_parameter_handler('vdW').parameters['[#1:1]-[#7]']
>>> parameter.rmin_half += 0.1 * unit.angstroms
>>> parameter.epsilon *= 1.02
```

Make a child vdW type more specific (checking modified SMIRKS for validity):

```
>>> forcefield.get_parameter_handler('vdW').parameters[-1].smirks += '~[#53]'
```

Warning: While we check whether the modified SMIRKS is still valid and has the appropriate valence type, we currently don't check whether the typing remains hierarchical, which could result in some types no longer being assignable because more general types now come *below* them and preferentially match.

Delete a parameter:

```
>>> del forcefield.get_parameter_handler('vdW').parameters['[#1:1]-[#6X4]']
```

Insert a parameter at a specific point in the parameter tree:

```
>>> from openff.toolkit.typing.engines.smirnoff import vdWHandler
>>> new_parameter = vdWHandler.vdWType(
>>>     smirks='[*:1]',
>>>     epsilon=0.0157*unit.kilocalories_per_mole,
>>>     rmin_half=0.6000*unit.angstroms,
>>> )
>>> forcefield.get_parameter_handler('vdW').parameters.insert(0, new_parameter)
```

Warning: We currently don't check whether removing a parameter could accidentally remove the root type, so it's possible to no longer type all molecules this way.

__init__(**sources*, *aromaticity_model=DEFAULT_AROMATICITY_MODEL*, *parameter_handler_classes=None*, *parameter_io_handler_classes=None*, *disable_version_check=False*, *allow_cosmetic_attributes=False*, *load_plugins=False*)

Create a new `ForceField` object from one or more SMIRNOFF parameter definition files.

Parameters

- **sources** (`string or file-like object or open file handle or URL (or iterable of these)`) – A list of files defining the SMIRNOFF force field to be loaded. Currently, only the SMIRNOFF XML format is supported. Each entry may be an absolute file path, a path relative to the current working directory, a path relative to this module's data subdirectory (for built in force fields), or an open file-like object with a `read()` method from which the force field XML data can be loaded. If multiple files are specified, any top-level tags that are repeated will be merged if they are compatible, with files appearing later in the sequence resulting in parameters that have higher precedence. Support for multiple files is primarily intended to allow solvent parameters to be specified by listing them last in the sequence.
- **aromaticity_model** (`string, default='OEAroModel_MDL'`) – The aromaticity model used by the force field. Currently, only 'OEAroModel_MDL' is supported
- **parameter_handler_classes** (`iterable of ParameterHandler classes, optional, default=None`) – If not None, the specified set of ParameterHandler classes will be instantiated to create the parameter object model. By default, all imported subclasses of ParameterHandler are automatically registered.
- **parameter_io_handler_classes** (`iterable of ParameterIOHandler classes`) – If not None, the specified set of ParameterIOHandler classes will be used to parse/generate serialized parameter sets. By default, all imported subclasses of ParameterIOHandler are automatically registered.
- **disable_version_check** (`bool, optional, default=False`) – If True, will disable checks against the current highest supported force field version. This option is primarily intended for force field development.
- **allow_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to retain non-spec kwargs from data sources.
- **load_plugins** (`bool, optional. Default = False`) – Whether to load `ParameterHandler` classes which have been registered by installed plugins.

Examples

Load one SMIRNOFF parameter set in XML format (searching the package data directory by default, which includes some standard parameter sets):

```
>>> forcefield = ForceField('test_forcefields/test_forcefield.offxml')
```

Load multiple SMIRNOFF parameter sets:

```
>>> forcefield = ForceField('test_forcefields/test_forcefield.offxml', 'test_
↪forcefields/tip3p.offxml')
```

Load a parameter set from a string:

```
>>> offxml = '<SMIRNOFF version="0.2" aromaticity_model="OEAroModel_MDL"/>'
>>> forcefield = ForceField(offxml)
```

Methods

`__init__`(*sources[, aromaticity_model, ...])	Create a new `ForceField` object from one or more SMIRNOFF parameter definition files.
`create_interchange`(topology[, ...])	Create an Interchange object from a ForceField, Topology, and (optionally) box vectors.
`create_openmm_system`(topology, **kwargs)	Create an OpenMM System from this ForceField and a Topology.
`deregister_parameter_handler`(handler)	Deregister a parameter handler specified by tag name, class, or instance.
`get_parameter_handler`(tagname[, ...])	Retrieve the parameter handlers associated with the provided tagname.
`get_parameter_io_handler`(io_format)	Retrieve the parameter handlers associated with the provided tagname.
`get_partial_charges`(molecule, **kwargs)	Generate the partial charges for the given molecule in this force field.
`label_molecules`(topology)	Return labels for a list of molecules corresponding to parameters from this force field.
`parse_smirnoff_from_source`(source)	Reads a SMIRNOFF data structure from a source, which can be one of many types.
`parse_sources`(sources[, ...])	Parse a SMIRNOFF force field definition.
`register_parameter_handler`(parameter_handler)	Register a new ParameterHandler for a specific tag, making it available for lookup in the ForceField.
`register_parameter_io_handler`(...)	Register a new ParameterIOHandler, making it available for lookup in the ForceField.
`to_file`(filename[, io_format, ...])	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.
`to_string`([io_format, ...])	Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

Attributes

aromaticity_model	Returns the aromaticity model for this ForceField object.
author	Returns the author data for this ForceField object.
date	Returns the date data for this ForceField object.
registered_parameter_handlers	Return the list of registered parameter handlers by name

property aromaticity_model

Returns the aromaticity model for this ForceField object.

Returns *aromaticity_model* – The aromaticity model for this force field.

property author

Returns the author data for this ForceField object. If not defined in any loaded files, this will be None.

Returns **author** (*str*) – The author data for this force field.

property date

Returns the date data for this ForceField object. If not defined in any loaded files, this will be None.

Returns **date** (*str*) – The date data for this force field.

register_parameter_handler(*parameter_handler*)

Register a new ParameterHandler for a specific tag, making it available for lookup in the ForceField.

> **Warning:** This API is experimental and subject to change.

Parameters **parameter_handler** (`A ParameterHandler object`) – The ParameterHandler to register. The TAGNAME attribute of this object will be used as the key for registration.

register_parameter_io_handler(*parameter_io_handler*)

Register a new ParameterIOHandler, making it available for lookup in the ForceField.

> **Warning:** This API is experimental and subject to change.

Parameters **parameter_io_handler** (`A ParameterIOHandler object`) – The ParameterIOHandler to register. The FORMAT attribute of this object will be used to associate it to a file format/suffix.

property registered_parameter_handlers

Return the list of registered parameter handlers by name

> **Warning:** This API is experimental and subject to change.

Returns **registered_parameter_handlers** (*iterable of names of ParameterHandler objects in this ForceField*)

get_parameter_handler(*tagname*, *handler_kwargs=None*, *allow_cosmetic_attributes=False*)

Retrieve the parameter handlers associated with the provided tagname.

If the parameter handler has not yet been instantiated, it will be created and returned. If a parameter handler object already exists, it will be checked for compatibility and an Exception raised if it is incompatible with the provided kwargs. If compatible, the existing ParameterHandler will be returned.

Parameters

- **tagname** (`str`) – The name of the parameter to be handled.
- **handler_kwargs** (`dict, optional. Default = None`) – Dict to be passed to the handler for construction or checking compatibility. If this is None and no existing ParameterHandler exists for the desired tag, a handler will be initialized with all default values. If this is None and a handler for the desired tag exists, the existing ParameterHandler will be returned.
- **allow_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to permit non-spec kwargs in smirnoff_data.

Returns **handler** (*An openff.toolkit.engines.typing.smirnoff.ParameterHandler*)

Raises `KeyError` – If there is no ParameterHandler for the given tagname

get_parameter_io_handler(*io_format*)

Retrieve the parameter handlers associated with the provided tagname. If the parameter IO handler has not yet been instantiated, it will be created.

Parameters **io_format** (`str`) – The name of the io format to be handled.

Returns **io_handler** (*An openff.toolkit.engines.typing.smirnoff.ParameterIOHandler*)

Raises `KeyError` – If there is no ParameterIOHandler for the given tagname

deregister_parameter_handler(*handler*)

Deregister a parameter handler specified by tag name, class, or instance.

Parameters **handler** (`str, openff.toolkit.typing.engines.smirnoff.ParameterHandler-derived type or object`) – The handler to deregister.

parse_sources(*sources*, *allow_cosmetic_attributes=True*)

Parse a SMIRNOFF force field definition.

Parameters

- **sources** (`string or file-like object or open file handle or URL (or iterable of these)`) – A list of files defining the SMIRNOFF force field to be loaded. Currently, only the SMIRNOFF XML format is supported. Each entry may be an absolute file path, a path relative to the current working directory, a path relative to this module's data subdirectory (for built in force fields), or an open file-like object with a `read()` method from which the force field XML data can be loaded. If multiple files are specified, any top-level tags that are repeated will be merged if they are compatible, with files appearing later in the sequence resulting in parameters that have higher precedence. Support for multiple files is primarily intended to allow solvent parameters to be specified by listing them last in the sequence.

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs present in the source.

Notes

- New SMIRNOFF sections are handled independently, as if they were specified in the same file.
- **If a SMIRNOFF section that has already been read appears again, its definitions are appended to the e** of the previously-read definitions if the sections are configured with compatible attributes; otherwise, an IncompatibleTagException is raised.

parse_smirnoff_from_source(*source*)

Reads a SMIRNOFF data structure from a source, which can be one of many types.

Parameters **source** (str or bytes or file-like object) – File defining the SMIRNOFF force field to be loaded Currently, only the SMIRNOFF XML format is supported. The file may be an absolute file path, a path relative to the current working directory, a path relative to this module's data subdirectory (for built in force fields), or an open file-like object with a read() method from which the force field XML data can be loaded.

Returns **smirnoff_data** (*OrderedDict*) – A representation of a SMIRNOFF-format data structure. Begins at top-level 'SMIRNOFF' key.

to_string(*io_format='XML'*, *discard_cosmetic_attributes=False*)

Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

Parameters

- **io_format** (str or ParameterIOHandler, optional. Default='XML') – The serialization format to write to
- **discard_cosmetic_attributes** (bool, default=False) – Whether to discard any non-spec attributes stored in the ForceField.

Returns **forcefield_string** (*str*) – The string representation of the serialized force field

to_file(*filename*, *io_format=None*, *discard_cosmetic_attributes=False*)

Write this Forcefield and all its associated parameters to a string in a given format which complies with the SMIRNOFF spec.

Parameters

- **filename** (str) – The filename to write to
- **io_format** (str or ParameterIOHandler, optional. Default=None) – The serialization format to write out. If None, will attempt to be inferred from the filename.
- **discard_cosmetic_attributes** (bool, default=False) – Whether to discard any non-spec attributes stored in the ForceField.

Returns **forcefield_string** (*str*) – The string representation of the serialized force field

create_openmm_system(*topology:* Topology, ***kwargs*) → Union[openmm.System, Tuple[openmm.System, *Topology*]]

Create an OpenMM System from this ForceField and a Topology.

Parameters **topology** (openforcefield.topology.Topology) – The Topology which is to be parameterized with this ForceField.

create_interchange(*topology: Topology*, *toolkit_registry: Optional[Union[ToolkitRegistry, ToolkitWrapper]] = None*, *charge_from_molecules: Optional[List[Molecule]] = None*, *partial_bond_orders_from_molecules: Optional[List[Molecule]] = None*, *allow_nonintegral_charges: bool = False*)

Create an Interchange object from a ForceField, Topology, and (optionally) box vectors.

WARNING: This API and functionality are experimental and not suitable for production.

Parameters

- **topology** (Topology) – The topology to create this *Interchange* object from.
- **toolkit_registry** (ToolkitRegistry, optional) – A *ToolkitRegistry* containing the toolkit wrappers to be used during parametriation, i.e. for assigning partial charges and fractional bond orders.

Returns **interchange** (*openff.interchange.Interchange*) – An *Interchange* object resulting from applying this *ForceField* to a *Topology*.

label_molecules(*topology*)

Return labels for a list of molecules corresponding to parameters from this force field. For each molecule, a dictionary of force types is returned, and for each force type, each force term is provided with the atoms involved, the parameter id assigned, and the corresponding SMIRKS.

Parameters **topology** (Topology) – A Topology object containing one or more unique molecules to be labeled

Returns

- **molecule_labels** (*list*) – List of labels for unique molecules. Each entry in the list corresponds to one unique molecule in the Topology and is a dictionary keyed by force type, i.e., molecule_labels[0]['HarmonicBondForce'] gives details for the harmonic bond parameters for the first molecule. Each element is a list of the form: [([atom1, ..., atomN], parameter_id, SMIRKS), ...].
- .. *todo ::* – What is the most useful API for this method? Should we instead accept Molecule objects as input and individually return labels? Should we attach the labels to the Molecule object? Or should we label all interactions in a Topology instead of just labeling its

  unique_molecules?

get_partial_charges(*molecule: Molecule*, ***kwargs*) → unit.Quantity

Generate the partial charges for the given molecule in this force field.

Parameters

- **molecule** (openff.toolkit.topology.Molecule) – The Molecule corresponding to the system to be parameterized
- **toolkit_registry** (openff.toolkit.utils.toolkits.ToolkitRegistry, default=GLOBAL_TOOLKIT_REGISTRY) – The toolkit registry to use for operations like conformer generation and partial charge assignment.

Returns **charges** (openmm.unit.Quantity with shape (n_atoms,) and dimensions of charge) – The partial charges of the provided molecule in this force field.

Raises

- **PartialChargeVirtualSitesError** – If the ForceField applies virtual sites to the Molecule. get_partial_charges cannot identify which virtual site charges may belong to which atoms in this case.
- **Other exceptions** – As any ParameterHandler may in principle modify charges, the entire force field must be applied to the molecule to produce the charges. Calls to this method from incorrectly or incompletely specified ForceField objects thus may raise an exception.

Examples

```
>>> from openff.toolkit import ForceField, Molecule
>>> ethanol = Molecule.from_smiles('CCO')
>>> force_field = ForceField('test_forcefields/test_forcefield.offxml')
```

Assign partial charges to the molecule according to the force field:

```
>>> ethanol.partial_charges = force_field.get_partial_charges(ethanol)
```

Use the assigned partial charges when creating an OpenMM System:

```
>>> topology = ethanol.to_topology()
>>> system = forcefield.create_openmm_system(
...     topology,
...     charge_from_molecules=[ethanol]
... )
```

This is especially useful when you want to create multiple systems with the same molecule or molecules, as it allows the expensive charge calculation to be cached.

openff.toolkit.typing.engines.smirnoff.get_available_force_fields

openff.toolkit.typing.engines.smirnoff.**get_available_force_fields**(*full_paths=False*)

Get the filenames of all available .offxml force field files.

Availability is determined by what is discovered through the openforcefield.smirnoff_forcefield_directory entry point. If the openff-forcefields package is installed, this should include several .offxml files such as openff-1.0.0.offxml.

Parameters **full_paths** (bool, default=False) – If False, return the name of each available *.offxml file. If True, return the full path to each available *.offxml file.

Returns **available_force_fields** (*List[str]*) – List of available force field files

Parameter Type

`ParameterType` objects are representations of individual SMIRKS-based SMIRNOFF parameters. These are usually initialized during `ForceField` creation, and can be inspected and modified by users via the Python API. For more information, see `examples/forcefield_modification`.

`ParameterType`	Base class for SMIRNOFF parameter types.
`ConstraintType`	A SMIRNOFF constraint type
`BondType`	A SMIRNOFF bond type
`AngleType`	A SMIRNOFF angle type.
`ProperTorsionType`	A SMIRNOFF torsion type for proper torsions.
`ImproperTorsionType`	A SMIRNOFF torsion type for improper torsions.
`vdWType`	A SMIRNOFF vdWForce type.
`LibraryChargeType`	A SMIRNOFF Library Charge type.
`GBSAType`	A SMIRNOFF GBSA type.
`ChargeIncrementType`	A SMIRNOFF bond charge correction type.
`VirtualSiteType`	

openff.toolkit.typing.engines.smirnoff.ParameterType

class openff.toolkit.typing.engines.smirnoff.**ParameterType**(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

Base class for SMIRNOFF parameter types.

This base class provides utilities to create new parameter types. See the below for examples of how to do this.

Warning: This API is experimental and subject to change.

Attributes

- **smirks** (*str*) – The SMIRKS pattern that this parameter matches.
- **id** (*str or None*) – An optional identifier for the parameter.
- **parent_id** (*str or None*) – Optionally, the identifier of the parameter of which this parameter is a specialization.

See also:

`ParameterAttribute`, `IndexedParameterAttribute`

Examples

This class allows to define new parameter types by just listing its attributes. In the example below, `_VALENCE_TYPE` AND `_ELEMENT_NAME` are used for the validation of the SMIRKS pattern associated to the parameter and the automatic serialization/deserialization into a `dict`.

```
>>> class MyBondParameter(ParameterType):
...     _VALENCE_TYPE = 'Bond'
...     _ELEMENT_NAME = 'Bond'
...     length = ParameterAttribute(unit=unit.angstrom)
...     k = ParameterAttribute(unit=unit.kilocalorie / unit.mole / unit.angstrom**2)
...
```

The parameter automatically inherits the required smirks attribute from `ParameterType`. Associating a `unit` to a `ParameterAttribute` cause the attribute to accept only values in compatible units and to parse string expressions.

```
>>> my_par = MyBondParameter(
...     smirks='[*:1]-[*:2]',
...     length='1.01 * angstrom',
...     k=5 * unit.kilocalorie / unit.mole / unit.angstrom**2
... )
>>> my_par.length
Quantity(value=1.01, unit=angstrom)
>>> my_par.k = 3.0 * unit.gram
Traceback (most recent call last):
...
openff.toolkit.utils.utils.IncompatibleUnitError: k=3.0 g should have units of␣
↪kilocalorie/(angstrom**2*mole)
```

Each attribute can be made optional by specifying a default value, and you can attach a converter function by passing a callable as an argument or through the decorator syntax.

```
>>> class MyParameterType(ParameterType):
...     _VALENCE_TYPE = 'Atom'
...     _ELEMENT_NAME = 'Atom'
...
...     attr_optional = ParameterAttribute(default=2)
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to floats
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameterType(smirks='[*:1]', attr_all_to_float='3.0', attr_int_to_
↪float=1)
>>> my_par.attr_optional
```

(continues on next page)

(continued from previous page)

```
2
>>> my_par.attr_all_to_float
3.0
>>> my_par.attr_int_to_float
1.0
```

The float() function can convert strings to integers, but our custom converter forbids it

```
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_int_to_float = '4.0'
Traceback (most recent call last):
...
TypeError: Cannot convert '4.0' to float
```

Parameter attributes that can be indexed can be handled with the `IndexedParameterAttribute`. These support unit validation and converters exactly as `ParameterAttributes`, but the validation/conversion is performed for each indexed attribute.

```
>>> class MyTorsionType(ParameterType):
...     _VALENCE_TYPE = 'ProperTorsion'
...     _ELEMENT_NAME = 'Proper'
...     periodicity = IndexedParameterAttribute(converter=int)
...     k = IndexedParameterAttribute(unit=unit.kilocalorie / unit.mole)
...
>>> my_par = MyTorsionType(
...     smirks='[*:1]-[*:2]-[*:3]-[*:4]',
...     periodicity1=2,
...     k1=5 * unit.kilocalorie / unit.mole,
...     periodicity2='3',
...     k2=6 * unit.kilocalorie / unit.mole,
... )
>>> my_par.periodicity
[2, 3]
```

Indexed attributes, can be accessed both as a list or as their indexed parameter name.

```
>>> my_par.periodicity2 = 6
>>> my_par.periodicity[0] = 1
>>> my_par.periodicity
[1, 6]
```

__init__(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (`str`) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (`bool optional. Default = False`) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(smirks[, allow_cosmetic_attributes])	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`id`
`parent_id`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (`str`) – Name of the attribute to define for this object.
- **attr_value** (`str`) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.ConstraintType

class openff.toolkit.typing.engines.smirnoff.**ConstraintType**(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF constraint type

> **Warning:** This API is experimental and subject to change.

__init__(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (str) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (bool optional. Default = False) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(smirks[, allow_cosmetic_attributes])	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`distance`
`id`
`parent_id`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.BondType

class openff.toolkit.typing.engines.smirnoff.**BondType**(***kwargs*)

A SMIRNOFF bond type

> **Warning:** This API is experimental and subject to change.

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (str) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (bool optional. Default = False) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(**kwargs)	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`id`
`k`
`k_bondorder`
`length`
`length_bondorder`
`parent_id`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (`list of string, optional. Default = None`) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.AngleType

class openff.toolkit.typing.engines.smirnoff.**AngleType**(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF angle type.

> **Warning:** This API is experimental and subject to change.

__init__(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (`str`) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (`bool optional. Default = False`) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(smirks[, allow_cosmetic_attributes])	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`angle`
`id`
`k`
`parent_id`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.ProperTorsionType

class openff.toolkit.typing.engines.smirnoff.**ProperTorsionType**(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF torsion type for proper torsions.

> **Warning:** This API is experimental and subject to change.

__init__(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (str) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (bool optional. Default = False) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(smirks[, allow_cosmetic_attributes])	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`id`
`idivf`
`k`
`k_bondorder`
`parent_id`
`periodicity`
`phase`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.ImproperTorsionType

class openff.toolkit.typing.engines.smirnoff.**ImproperTorsionType**(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF torsion type for improper torsions.

> **Warning:** This API is experimental and subject to change.

__init__(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (str) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (bool optional. Default = False) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(smirks[, allow_cosmetic_attributes])	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`id`
`idivf`
`k`
`parent_id`
`periodicity`
`phase`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (`list of string, optional. Default = None`) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.vdWType

class openff.toolkit.typing.engines.smirnoff.**vdWType**(***kwargs*)

A SMIRNOFF vdWForce type.

> **Warning:** This API is experimental and subject to change.

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (`str`) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (`bool optional. Default = False`) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(**kwargs)	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`epsilon`
`id`
`parent_id`
`rmin_half`
`sigma`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (`str`) – Name of the attribute to define for this object.
- **attr_value** (`str`) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.LibraryChargeType

class openff.toolkit.typing.engines.smirnoff.**LibraryChargeType**(***kwargs*)

A SMIRNOFF Library Charge type.

> **Warning:** This API is experimental and subject to change.

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (str) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (bool optional. Default = False) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(**kwargs)	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`from_molecule`(molecule)	Construct a LibraryChargeType from a molecule with existing partial charges.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`charge`
`id`
`name`
`parent_id`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (`str`) – Name of the attribute to define for this object.
- **attr_value** (`str`) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

classmethod from_molecule(*molecule:* Molecule)

Construct a LibraryChargeType from a molecule with existing partial charges.

Parameters **molecule** (Molecule) – The molecule to create the LibraryChargeType from. The molecule must have partial charges.

Returns **library_charge_type** (*LibraryChargeType*) – A LibraryChargeType that is expected to match this molecule and its partial charges.

:raises MissingPartialChargesError : If the input molecule does not have partial charges.:

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.GBSAType

class openff.toolkit.typing.engines.smirnoff.**GBSAType**(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF GBSA type.

> **Warning:** This API is experimental and subject to change.

__init__(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (str) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (bool optional. Default = False) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will

be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(smirks[, allow_cosmetic_attributes])	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`id`
`parent_id`
`radius`
`scale`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (`str`) – Name of the attribute to define for this object.
- **attr_value** (`str`) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (`list of string, optional. Default = None`) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.ChargeIncrementType

class openff.toolkit.typing.engines.smirnoff.**ChargeIncrementType**(***kwargs*)

A SMIRNOFF bond charge correction type.

> **Warning:** This API is experimental and subject to change.

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (`str`) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (`bool optional. Default = False`) – Whether to permit non-spec kwargs (“cosmetic attributes”). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

`__init__`(**kwargs)	Create a ParameterType.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`to_dict`([discard_cosmetic_attributes, ...])	Convert this object to dict format.

Attributes

`charge_increment`
`id`
`parent_id`
`smirks`

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

openff.toolkit.typing.engines.smirnoff.VirtualSiteType

class openff.toolkit.typing.engines.smirnoff.**VirtualSiteType**(***kwargs*)

__init__(***kwargs*)

Create a ParameterType.

Parameters

- **smirks** (str) – The SMIRKS match for the provided parameter type.
- **allow_cosmetic_attributes** (bool optional. Default = False) – Whether to permit non-spec kwargs ("cosmetic attributes"). If True, non-spec kwargs will be stored as an attribute of this parameter which can be accessed and written out. Otherwise an exception will be raised.

Methods

__init__(**kwargs)	Create a ParameterType.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
to_dict([discard_cosmetic_attributes, ...])	Convert this object to dict format.
type_to_parent_index(type_)	Returns the index of the atom matched by the SMIRKS pattern that should be considered the 'parent' to a given type of virtual site.

Attributes

charge_increment	
distance	
epsilon	
id	
inPlaneAngle	
match	
name	
outOfPlaneAngle	
parent_id	
parent_index	Returns the index of the atom matched by the SMIRKS pattern that should be considered the 'parent' to the virtual site.
rmin_half	
sigma	
smirks	
type	

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

property parent_index: int

Returns the index of the atom matched by the SMIRKS pattern that should be considered the 'parent' to the virtual site. A value of `0` corresponds to the atom matched by the `:1` selector in the SMIRKS pattern, a value `2` the atom matched by `:2` and so on.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

classmethod type_to_parent_index(*type_: Literal['BondCharge', 'MonovalentLonePair', 'DivalentLonePair', 'TrivalentLonePair']*) → int

Returns the index of the atom matched by the SMIRKS pattern that should be considered the 'parent' to a given type of virtual site. A value of `0` corresponds to the atom matched by the `:1` selector in the SMIRKS pattern, a value `2` the atom matched by `:2` and so on.

Parameter Handlers

Each `ForceField` primarily consists of several `ParameterHandler` objects, which each contain the machinery to add one energy component to an OpenMM `System`. During `System` creation, each `ParameterHandler` registered to a `ForceField` has its `assign_parameters()` function called.

`ParameterList`	Parameter list that also supports accessing items by SMARTS string.
`ParameterHandler`	Base class for parameter handlers.
`ConstraintHandler`	Handle SMIRNOFF `<Constraints>` tags
`BondHandler`	Handle SMIRNOFF `<Bonds>` tags
`AngleHandler`	Handle SMIRNOFF `<AngleForce>` tags
`ProperTorsionHandler`	Handle SMIRNOFF `<ProperTorsionForce>` tags
`ImproperTorsionHandler`	Handle SMIRNOFF `<ImproperTorsionForce>` tags
`vdWHandler`	Handle SMIRNOFF `<vdW>` tags
`ElectrostaticsHandler`	Handles SMIRNOFF `<Electrostatics>` tags.
`LibraryChargeHandler`	Handle SMIRNOFF `<LibraryCharges>` tags
`ToolkitAM1BCCHandler`	Handle SMIRNOFF `<ToolkitAM1BCC>` tags
`GBSAHandler`	Handle SMIRNOFF `<GBSA>` tags
`ChargeIncrementModelHandler`	Handle SMIRNOFF `<ChargeIncrementModel>` tags
`VirtualSiteHandler`	Handle SMIRNOFF `<VirtualSites>` tags TODO: Add example usage/documentation .

openff.toolkit.typing.engines.smirnoff.parameters.ParameterList

class openff.toolkit.typing.engines.smirnoff.parameters.**ParameterList**(*input_parameter_list=None*)

Parameter list that also supports accessing items by SMARTS string.

> **Warning:** This API is experimental and subject to change.

__init__(*input_parameter_list=None*)

Initialize a new ParameterList, optionally providing a list of ParameterType objects to initially populate it.

Parameters **input_parameter_list** (`list[ParameterType]`, `default=None`) – A pre-existing list of ParameterType-based objects. If None, this ParameterList will be initialized empty.

Methods

`__init__`([input_parameter_list])	Initialize a new ParameterList, optionally providing a list of ParameterType objects to initially populate it.
`append`(parameter)	Add a ParameterType object to the end of the ParameterList
`clear`()	Remove all items from list.
`copy`()	Return a shallow copy of the list.
`count`(value, /)	Return number of occurrences of value.
`extend`(other)	Add a ParameterList object to the end of the ParameterList
`index`(item)	Get the numerical index of a ParameterType object or SMIRKS in this ParameterList.
`insert`(index, parameter)	Add a ParameterType object as if this were a list
`pop`([index])	Remove and return item at index (default last).
`remove`(value, /)	Remove first occurrence of value.
`reverse`()	Reverse *IN PLACE*.
`sort`(*[, key, reverse])	Sort the list in ascending order and return None.
`to_list`([discard_cosmetic_attributes])	Render this ParameterList to a normal list, serializing each ParameterType object in it to dict.

append(*parameter*)

Add a ParameterType object to the end of the ParameterList

Parameters **parameter** (a ParameterType object) –

extend(*other*)

Add a ParameterList object to the end of the ParameterList

Parameters **other** (a ParameterList) –

index(*item*)

Get the numerical index of a ParameterType object or SMIRKS in this ParameterList. Raises ParameterLookupError if the item is not found.

Parameters **item** (ParameterType object or str) – The parameter or SMIRKS to look up in this ParameterList

Returns **index** (*int*) – The index of the found item

Raises **ParameterLookupError if SMIRKS pattern is passed in but not found** –

insert(*index*, *parameter*)

Add a ParameterType object as if this were a list

Parameters

- **index** (int) – The numerical position to insert the parameter at
- **parameter** (a ParameterType object) – The parameter to insert

to_list(*discard_cosmetic_attributes=True*)

Render this ParameterList to a normal list, serializing each ParameterType object in it to dict.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = True) – Whether to discard non-spec attributes of each ParameterType object.

Returns **parameter_list** (*List[dict]*) – A serialized representation of a ParameterList, with each ParameterType it contains converted to dict.

clear()

Remove all items from list.

copy()

Return a shallow copy of the list.

count(*value*, /)

Return number of occurrences of value.

pop(*index=- 1*, /)

Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(*value*, /)

Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()

Reverse *IN PLACE*.

sort(**, key=None, reverse=False*)

Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

openff.toolkit.typing.engines.smirnoff.parameters.ParameterHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**ParameterHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Base class for parameter handlers.

Parameter handlers are configured with some global parameters for a given section. They may also contain a `ParameterList` populated with `ParameterType` objects if they are responsible for assigning SMIRKS-based parameters.

> **Warning:** This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes

of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
version	

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

property known_kwargs

List of kwargs that can be parsed by the function.

check_handler_compatibility(*handler_kwargs*)

Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **handler_kwargs** (dict) – The kwargs that would be used to construct

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns **matches** (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

openff.toolkit.typing.engines.smirnoff.parameters.ConstraintHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**ConstraintHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <Constraints> tags

ConstraintHandler must be applied before BondHandler and AngleHandler, since those classes add constraints for which equilibrium geometries are needed from those tags.

Warning: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
version	

class ConstraintType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF constraint type

> **Warning:** This API is experimental and subject to change.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (`str`) – Name of the attribute to define for this object.
- **attr_value** (`str`) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to discard non-spec attributes of this object

- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns smirnoff_dict (*dict*) – The SMIRNOFF-compliant dict representation of this object.

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns handler_name (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

check_handler_compatibility(*handler_kwargs*)

Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **handler_kwargs** (dict) – The kwargs that would be used to construct

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns matches (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns params (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.BondHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**BondHandler**(***kwargs*)

Handle SMIRNOFF <Bonds> tags

Warning: This API is experimental and subject to change.

__init__(***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__(**kwargs)	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
fractional_bondorder_interpolation	
fractional_bondorder_method	
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	
version	

class BondType(***kwargs*)

A SMIRNOFF bond type

> **Warning:** This API is experimental and subject to change.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns **matches** (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.AngleHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**AngleHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <AngleForce> tags

Warning: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	
version	

class AngleType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF angle type.

> **Warning:** This API is experimental and subject to change.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.

- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns matches (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters parameter_attrs (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns params (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters discard_cosmetic_attributes (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns smirnoff_data (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.ProperTorsionHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**ProperTorsionHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <ProperTorsionForce> tags

Warning: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
default_idivf	
fractional_bondorder_interpolation	
fractional_bondorder_method	
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	
version	

class ProperTorsionType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF torsion type for proper torsions.

> **Warning:** This API is experimental and subject to change.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns **matches** (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.ImproperTorsionHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**ImproperTorsionHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <ImproperTorsionForce> tags

Warning: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the improper torsions in the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
default_idivf	
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	
version	

class ImproperTorsionType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF torsion type for improper torsions.

> **Warning:** This API is experimental and subject to change.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (`str`) – Name of the attribute to define for this object.
- **attr_value** (`str`) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (`list of string, optional. Default = None`) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

find_matches(*entity*, *unique=False*)

Find the improper torsions in the topology/molecule matched by a parameter type.

Parameters **entity** (Topology) – Topology to search.

Returns **matches** (*ImproperDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the 4-tuple of atom indices in entity.

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.

- When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
- When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
- The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.vdWHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**vdWHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <vdW> tags

Warning: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
combining_rules	
cutoff	
known_kwargs	List of kwargs that can be parsed by the function.
method	
parameters	The ParameterList that holds this ParameterHandler's parameter objects
potential	
scale12	
scale13	
scale14	
scale15	
switch_width	
version	

class vdWType(***kwargs*)

A SMIRNOFF vdWForce type.

> **Warning:** This API is experimental and subject to change.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
```

(continues on next page)

(continued from previous page)

```
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns **matches** (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[:1]~[#16:2]=,:[#6:3]~[:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.ElectrostaticsHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**ElectrostaticsHandler**(***kwargs*)

Handles SMIRNOFF <Electrostatics> tags.

> **Warning:** This API is experimental and subject to change.

__init__(***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__(**kwargs)	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
cutoff	
exception_potential	
known_kwargs	List of kwargs that can be parsed by the function.
nonperiodic_potential	
parameters	The ParameterList that holds this ParameterHandler's parameter objects
periodic_potential	
scale12	
scale13	
scale14	
scale15	
solvent_dielectric	
switch_width	
version	

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns **matches** (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.LibraryChargeHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**LibraryChargeHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <LibraryCharges> tags

Warning: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.

- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(handler_kwargs)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
version	

class LibraryChargeType(***kwargs*)

A SMIRNOFF Library Charge type.

> **Warning:** This API is experimental and subject to change.

classmethod from_molecule(*molecule:* Molecule)

Construct a LibraryChargeType from a molecule with existing partial charges.

Parameters molecule (Molecule) – The molecule to create the LibraryChargeType from. The molecule must have partial charges.

Returns **library_charge_type** (*LibraryChargeType*) – A LibraryChargeType that is expected to match this molecule and its partial charges.

:raises MissingPartialChargesError : If the input molecule does not have partial charges.:

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (`str`) – Name of the attribute to define for this object.
- **attr_value** (`str`) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (`list of string, optional. Default = None`) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters **entity** (Topology) – Topology to search.

Returns **matches** (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – `matches[atom_indices]` is the `ParameterType` object matching the tuple of atom indices in `entity`.

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

- The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

check_handler_compatibility(*handler_kwargs*)

Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **handler_kwargs** (dict) – The kwargs that would be used to construct

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – Name of the cosmetic attribute to delete.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (`dict of {attr: value}`) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (`bool, optional. Default = False.`) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.ToolkitAM1BCCHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**ToolkitAM1BCCHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <ToolkitAM1BCC> tags

Warning: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

__init__([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
add_cosmetic_attribute(attr_name, attr_value)	Add a cosmetic attribute to this object.
add_parameter([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
attribute_is_cosmetic(attr_name)	Determine whether an attribute of this object is cosmetic.
check_handler_compatibility(other_handler[, ...])	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
create_force(*args, **kwarsg)	
delete_cosmetic_attribute(attr_name)	Delete a cosmetic attribute from this object.
find_matches(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
get_parameter(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
to_dict([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
known_kwargs	List of kwargs that can be parsed by the function.
parameters	The ParameterList that holds this ParameterHandler's parameter objects
version	

check_handler_compatibility(*other_handler*, *assume_missing_is_default=True*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added

- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns matches (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns params (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.GBSAHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**GBSAHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <GBSA> tags

> **Warning:** This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_handler_compatibility`(other_handler)	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
`create_force`(*args, **kwarsg)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

`TAGNAME`	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
`gb_model`	
`known_kwargs`	List of kwargs that can be parsed by the function.
`parameters`	The ParameterList that holds this ParameterHandler's parameter objects
`sa_model`	
`solute_dielectric`	
`solvent_dielectric`	
`solvent_radius`	
`surface_area_penalty`	
`version`	

class GBSAType(*smirks*, *allow_cosmetic_attributes=False*, ***kwargs*)

A SMIRNOFF GBSA type.

> **Warning:** This API is experimental and subject to change.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (`str`) – Name of the attribute to define for this object.
- **attr_value** (`str`) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (`str`) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the `ParameterAttribute` and `IndexedParameterAttribute` as well as cosmetic attributes if `discard_cosmetic_attributes` is `False`.

Parameters

- **discard_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (`list of string, optional. Default = None`) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.

- The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

Warning: The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns matches (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters parameter_attrs (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[*:1]~[#16:2]=,:[#6:3]~[*:4]", "id": "t105"})

Returns params (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters discard_cosmetic_attributes (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.ChargeIncrementModelHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**ChargeIncrementModelHandler**(*allow_cosmetic_attributes=F*
skip_version_check=False,
***kwargs*)

Handle SMIRNOFF `<ChargeIncrementModel>` tags

Warning: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (`bool, optional. Default = False`) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (`bool, optional. Default = False`) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (`dict`) – The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_handler_compatibility`(other_handler[, ...])	Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler.
`create_force`(*args, **kwarsg)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

TAGNAME	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
known_kwargs	List of kwargs that can be parsed by the function.
number_of_conformers	
parameters	The ParameterList that holds this ParameterHandler's parameter objects
partial_charge_method	
version	

class ChargeIncrementType(***kwargs*)

A SMIRNOFF bond charge correction type.

> **Warning:** This API is experimental and subject to change.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

check_handler_compatibility(*other_handler*, *assume_missing_is_default=True*)

Checks whether this ParameterHandler encodes compatible physics as another ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **other_handler** (a ParameterHandler object) – The handler to compare to.

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters **entity** (Topology) – Topology to search.

Returns **matches** (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[:1]=[:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[:1]~[#16:2]=,:[#6:3]~[:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

openff.toolkit.typing.engines.smirnoff.parameters.VirtualSiteHandler

class openff.toolkit.typing.engines.smirnoff.parameters.**VirtualSiteHandler**(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Handle SMIRNOFF <VirtualSites> tags TODO: Add example usage/documentation .. warning :: This API is experimental and subject to change.

__init__(*allow_cosmetic_attributes=False*, *skip_version_check=False*, ***kwargs*)

Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.

Parameters

- **allow_cosmetic_attributes** (bool, optional. Default = False) – Whether to permit non-spec kwargs. If True, non-spec kwargs will be stored as attributes of this object and can be accessed and modified. Otherwise an exception will be raised if a non-spec kwarg is encountered.
- **skip_version_check** (bool, optional. Default = False) – If False, the SMIRNOFF section version will not be checked, and the ParameterHandler will be initialized with version set to _MAX_SUPPORTED_SECTION_VERSION.
- ****kwargs** (dict) – The dict representation of the SMIRNOFF data source

Methods

`__init__`([allow_cosmetic_attributes, ...])	Initialize a ParameterHandler, optionally with a list of parameters and other kwargs.
`add_cosmetic_attribute`(attr_name, attr_value)	Add a cosmetic attribute to this object.
`add_parameter`([parameter_kwargs, parameter, ...])	Add a parameter to the force field, ensuring all parameters are valid.
`attribute_is_cosmetic`(attr_name)	Determine whether an attribute of this object is cosmetic.
`check_handler_compatibility`(other_handler)	Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler.
`create_force`(*args, **kwarsg)	
`delete_cosmetic_attribute`(attr_name)	Delete a cosmetic attribute from this object.
`find_matches`(entity[, unique])	Find the elements of the topology/molecule matched by a parameter type.
`get_parameter`(parameter_attrs)	Return the parameters in this ParameterHandler that match the parameter_attrs argument.
`to_dict`([discard_cosmetic_attributes])	Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Attributes

`TAGNAME`	The name of this ParameterHandler corresponding to the SMIRNOFF tag name
`exclusion_policy`	
`known_kwargs`	List of kwargs that can be parsed by the function.
`parameters`	The ParameterList that holds this ParameterHandler's parameter objects
`version`	

class VirtualSiteType(***kwargs*)

property parent_index: int

Returns the index of the atom matched by the SMIRKS pattern that should be considered the 'parent' to the virtual site. A value of `0` corresponds to the atom matched by the `:1` selector in the SMIRKS pattern, a value `2` the atom matched by `:2` and so on.

classmethod type_to_parent_index(*type_: Literal['BondCharge', 'MonovalentLonePair', 'DivalentLonePair', 'TrivalentLonePair']*) → int

Returns the index of the atom matched by the SMIRKS pattern that should be considered the 'parent' to a given type of virtual site. A value of `0` corresponds to the atom matched by the `:1` selector in the SMIRKS pattern, a value `2` the atom matched by `:2` and so on.

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

to_dict(*discard_cosmetic_attributes=False*, *duplicate_attributes=None*)

Convert this object to dict format.

The returning dictionary contains all the ParameterAttribute and IndexedParameterAttribute as well as cosmetic attributes if discard_cosmetic_attributes is False.

Parameters

- **discard_cosmetic_attributes** (bool, optional. Default = False) – Whether to discard non-spec attributes of this object
- **duplicate_attributes** (list of string, optional. Default = None) – A list of names of attributes that redundantly decsribe data and should be discarded during serializaiton

Returns **smirnoff_dict** (*dict*) – The SMIRNOFF-compliant dict representation of this object.

property TAGNAME

The name of this ParameterHandler corresponding to the SMIRNOFF tag name

Returns **handler_name** (*str*) – The name of this parameter handler

add_cosmetic_attribute(*attr_name*, *attr_value*)

Add a cosmetic attribute to this object.

This attribute will not have a functional effect on the object in the OpenFF Toolkit, but can be written out during output.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters

- **attr_name** (str) – Name of the attribute to define for this object.
- **attr_value** (str) – The value of the attribute to define for this object.

add_parameter(*parameter_kwargs=None*, *parameter=None*, *after=None*, *before=None*)

Add a parameter to the force field, ensuring all parameters are valid.

Parameters

- **parameter_kwargs** (dict, optional) – The kwargs to pass to the ParameterHandler.INFOTYPE (a ParameterType) constructor
- **parameter** (ParameterType, optional) – A ParameterType to add to the ParameterHandler
- **after** (str or int, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly before where the new parameter will be added
- **before** (str, optional) – The SMIRKS pattern (if str) or index (if int) of the parameter directly after where the new parameter will be added
- **behavior** (Note the following) –
  - Either *parameter_kwargs* or *parameter* must be specified.
  - When *before* and *after* are both *None*, the new parameter will be appended to the **END** of the parameter list.
  - When *before* and *after* are both specified, the new parameter will be added immediately after the parameter matching the *after* pattern or index.
  - The order of parameters in a parameter list can have significant impacts on parameter assignment. For details, see the SMIRNOFF specification: https://openforcefield.github.io/standards/standards/smirnoff/#smirnoff-parameter-specification-is-hierarchical

Examples

Add a ParameterType to an existing ParameterList at a specified position.

Given an existing parameter handler and a new parameter to add to it:

```
>>> from openmm import unit
>>> bh = BondHandler(skip_version_check=True)
>>> length = 1.5 * unit.angstrom
>>> k = 100 * unit.kilocalorie / unit.mole / unit.angstrom ** 2
>>> bh.add_parameter({'smirks': '[*:1]-[*:2]', 'length': length, 'k': k, 'id': 'b1
↪'})
```

(continues on next page)

(continued from previous page)

```
>>> bh.add_parameter({'smirks': '[*:1]=[*:2]', 'length': length, 'k': k, 'id': 'b2
↪'})
>>> bh.add_parameter({'smirks': '[*:1]#[*:2]', 'length': length, 'k': k, 'id': 'b3
↪'})
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b3']
```

```
>>> param = {'smirks': '[#1:1]-[#6:2]', 'length': length, 'k': k, 'id': 'b4'}
```

Add a new parameter immediately after the parameter with the smirks '[*:1]=[*:2]'

```
>>> bh.add_parameter(param, after='[*:1]=[*:2]')
>>> [p.id for p in bh.parameters]
['b1', 'b2', 'b4', 'b3']
```

attribute_is_cosmetic(*attr_name*)

Determine whether an attribute of this object is cosmetic.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – The attribute name to check

Returns **is_cosmetic** (*bool*) – Returns True if the attribute is defined and is cosmetic. Returns False otherwise.

delete_cosmetic_attribute(*attr_name*)

Delete a cosmetic attribute from this object.

> **Warning:** The API for modifying cosmetic attributes is experimental and may change in the future (see issue #338).

Parameters **attr_name** (str) – Name of the cosmetic attribute to delete.

find_matches(*entity*, *unique=False*)

Find the elements of the topology/molecule matched by a parameter type.

Parameters

- **entity** (Topology) – Topology to search.
- **unique** (bool, default=False) – If False, SMARTS matching will enumerate every valid permutation of matching atoms. If True, only one order of each unique match will be returned.

Returns **matches** (*ValenceDict[Tuple[int], ParameterHandler._Match]*) – matches[atom_indices] is the ParameterType object matching the tuple of atom indices in entity.

get_parameter(*parameter_attrs*)

Return the parameters in this ParameterHandler that match the parameter_attrs argument. When multiple attrs are passed, parameters that have any (not all) matching attributes are returned.

Parameters **parameter_attrs** (dict of {attr: value}) – The attrs mapped to desired values (for example {"smirks": "[:1]~[#16:2]=,:[#6:3]~[:4]", "id": "t105"})

Returns **params** (*list of ParameterType objects*) – A list of matching ParameterType objects

Examples

Create a parameter handler and populate it with some data.

```
>>> from openmm import unit
>>> handler = BondHandler(skip_version_check=True)
>>> handler.add_parameter(
...     {
...         'smirks': '[*:1]-[*:2]',
...         'length': 1*unit.angstrom,
...         'k': 10*unit.kilocalorie / unit.mole/unit.angstrom**2,
...     }
... )
```

Look up, from this handler, all parameters matching some SMIRKS pattern

```
>>> handler.get_parameter({'smirks': '[*:1]-[*:2]'})
[<BondType with smirks: [*:1]-[*:2]  length: 1 A  k: 10 kcal/(A**2 mol)  >]
```

property known_kwargs

List of kwargs that can be parsed by the function.

property parameters

The ParameterList that holds this ParameterHandler's parameter objects

to_dict(*discard_cosmetic_attributes=False*)

Convert this ParameterHandler to an OrderedDict, compliant with the SMIRNOFF data spec.

Parameters **discard_cosmetic_attributes** (bool, optional. Default = False.) – Whether to discard non-spec parameter and header attributes in this ParameterHandler.

Returns **smirnoff_data** (*OrderedDict*) – SMIRNOFF-spec compliant representation of this ParameterHandler and its internal ParameterList.

check_handler_compatibility(*other_handler*)

Checks if a set of kwargs used to create a ParameterHandler are compatible with this ParameterHandler. This is called if a second handler is attempted to be initialized for the same tag.

Parameters **handler_kwargs** (dict) – The kwargs that would be used to construct

Raises **IncompatibleParameterError if handler_kwargs are incompatible with existing parameters.** –

Parameter I/O Handlers

`ParameterIOHandler` objects handle reading and writing of serialzied SMIRNOFF data sources.

ParameterIOHandler	Base class for handling serialization/deserialization of SMIRNOFF ForceField objects
XMLParameterIOHandler	Handles serialization/deserialization of SMIRNOFF ForceField objects from OFFXML format.

openff.toolkit.typing.engines.smirnoff.io.ParameterIOHandler

class openff.toolkit.typing.engines.smirnoff.io.**ParameterIOHandler**

Base class for handling serialization/deserialization of SMIRNOFF ForceField objects

__init__()

Create a new ParameterIOHandler.

Methods

__init__()	Create a new ParameterIOHandler.
parse_file(file_path)	**param file_path**
parse_string(data)	Parse a SMIRNOFF force field definition in a seriaized format
to_file(file_path, smirnoff_data)	Write the current force field parameter set to a file.
to_string(smirnoff_data)	Render the force field parameter set to a string

parse_file(*file_path*)

Parameters **file_path** –

parse_string(*data*)

Parse a SMIRNOFF force field definition in a seriaized format

Parameters **data** –

to_file(*file_path*, *smirnoff_data*)

Write the current force field parameter set to a file.

Parameters

- **file_path** (str) – The path to the file to write to.
- **smirnoff_data** (dict) – A dictionary structured in compliance with the SMIRNOFF spec

to_string(*smirnoff_data*)

Render the force field parameter set to a string

Parameters **smirnoff_data** (dict) – A dictionary structured in compliance with the SMIRNOFF spec

Returns *str*

openff.toolkit.typing.engines.smirnoff.io.XMLParameterIOHandler

class openff.toolkit.typing.engines.smirnoff.io.**XMLParameterIOHandler**

Handles serialization/deserialization of SMIRNOFF ForceField objects from OFFXML format.

__init__()

Create a new ParameterIOHandler.

Methods

__init__()	Create a new ParameterIOHandler.
parse_file(source)	Parse a SMIRNOFF force field definition in XML format, read from a file.
parse_string(data)	Parse a SMIRNOFF force field definition in XML format.
to_file(file_path, smirnoff_data)	Write the current force field parameter set to a file.
to_string(smirnoff_data)	Write the current force field parameter set to an XML string.

parse_file(*source*)

Parse a SMIRNOFF force field definition in XML format, read from a file.

Parameters **source** (str or RawIOBase) – File path of file-like object implementing a read() method specifying a SMIRNOFF force field definition in the SMIRNOFF XML format.

Raises

- **SMIRNOFFParseError** – If the XML cannot be processed.
- **FileNotFoundError** – If the file could not found.

parse_string(*data*)

Parse a SMIRNOFF force field definition in XML format.

A SMIRNOFFParseError is raised if the XML cannot be processed.

Parameters **data** (str) – A SMIRNOFF force field definition in the SMIRNOFF XML format.

to_file(*file_path*, *smirnoff_data*)

Write the current force field parameter set to a file.

Parameters

- **file_path** (str) – The path to the file to be written. The *.offxml* or *.xml* file extension must be present.
- **smirnoff_data** (dict) – A dict structured in compliance with the SMIRNOFF data spec.

to_string(*smirnoff_data*)

Write the current force field parameter set to an XML string.

Parameters **smirnoff_data** (dict) – A dictionary structured in compliance with the SMIRNOFF spec

Returns **serialized_forcefield** (*str*) – XML String representation of this force field.

Parameter Attributes

`ParameterAttribute` and `IndexedParameterAttribute` provide a standard backend for ParameterHandler and Parameter attributes, while also enforcing validation of types and units.

`ParameterAttribute`	A descriptor for `ParameterType` attributes.
`IndexedParameterAttribute`	The attribute of a parameter with an unspecified number of terms.
`MappedParameterAttribute`	The attribute of a parameter in which each term is a mapping.
`IndexedMappedParameterAttribute`	The attribute of a parameter with an unspecified number of terms, where each term is a mapping.

openff.toolkit.typing.engines.smirnoff.parameters.ParameterAttribute

class openff.toolkit.typing.engines.smirnoff.parameters.**ParameterAttribute**(*default=UNDEFINED*, *unit=None*, *converter=None*, *docstring=''*)

A descriptor for `ParameterType` attributes.

The descriptors allows associating to the parameter a default value, which makes the attribute optional, a unit, and a custom converter.

Because we may want to have `None` as a default value, required attributes have the `default` set to the special type `UNDEFINED`.

Converters can be both static or instance functions/methods with respective signatures:

```
converter(value): -> converted_value
converter(instance, parameter_attribute, value): -> converted_value
```

A decorator syntax is available (see example below).

Parameters

- **default** (`object, optional`) – When specified, the descriptor makes this attribute optional by attaching a default value to it.
- **unit** (`openmm.unit.Quantity, optional`) – When specified, only quantities with compatible units are allowed to be set, and string expressions are automatically parsed into a `Quantity`.
- **converter** (`callable, optional`) – An optional function that can be used to convert values before setting the attribute.

See also:

`IndexedParameterAttribute` A parameter attribute with multiple terms.

Examples

Create a parameter type with an optional and a required attribute.

```
>>> class MyParameter:
...     attr_required = ParameterAttribute()
...     attr_optional = ParameterAttribute(default=2)
...
>>> my_par = MyParameter()
```

Even without explicit assignment, the default value is returned.

```
>>> my_par.attr_optional
2
```

If you try to access an attribute without setting it first, an exception is raised.

```
>>> my_par.attr_required
Traceback (most recent call last):
...
AttributeError: 'MyParameter' object has no attribute '_attr_required'
```

The attribute allow automatic conversion and validation of units.

```
>>> from openmm import unit
>>> class MyParameter:
...     attr_quantity = ParameterAttribute(unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.attr_quantity = '1.0 * nanometer'
>>> my_par.attr_quantity
Quantity(value=1.0, unit=nanometer)
>>> my_par.attr_quantity = 3.0
Traceback (most recent call last):
...
openff.toolkit.utils.utils.IncompatibleUnitError: attr_quantity=3.0 dimensionless␣
↪should have units of angstrom
```

You can attach a custom converter to an attribute.

```
>>> class MyParameter:
...     # Both strings and integers convert nicely to floats with float().
...     attr_all_to_float = ParameterAttribute(converter=float)
...     attr_int_to_float = ParameterAttribute()
...     @attr_int_to_float.converter
...     def attr_int_to_float(self, attr, value):
...         # This converter converts only integers to float
...         # and raise an exception for the other types.
...         if isinstance(value, int):
...             return float(value)
...         elif not isinstance(value, float):
...             raise TypeError(f"Cannot convert '{value}' to float")
...         return value
...
>>> my_par = MyParameter()
```

attr_all_to_float accepts and convert to float both strings and integers

```
>>> my_par.attr_all_to_float = 1
>>> my_par.attr_all_to_float
1.0
>>> my_par.attr_all_to_float = '2.0'
>>> my_par.attr_all_to_float
2.0
```

The custom converter associated to attr_int_to_float converts only integers instead.

```
>>> my_par.attr_int_to_float = 3
>>> my_par.attr_int_to_float
3.0
>>> my_par.attr_int_to_float = '4.0'
Traceback (most recent call last):
...
TypeError: Cannot convert '4.0' to float
```

__init__(*default=UNDEFINED*, *unit=None*, *converter=None*, *docstring=''*)

Methods

`__init__`([default, unit, converter, docstring])	
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

class UNDEFINED

Custom type used by `ParameterAttribute` to differentiate between `None` and undeclared default.

converter(*converter*)

Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

openff.toolkit.typing.engines.smirnoff.parameters.IndexedParameterAttribute

class openff.toolkit.typing.engines.smirnoff.parameters.**IndexedParameterAttribute**(*default=UNDEFINED*, *unit=None*, *converter=None*, *docstring=''*)

The attribute of a parameter with an unspecified number of terms.

Some parameters can be associated to multiple terms, For example, torsions have parameters such as k1, k2, ..., and `IndexedParameterAttribute` can be used to encapsulate the sequence of terms.

The only substantial difference with `ParameterAttribute` is that only sequences are supported as values and converters and units are checked on each element of the sequence.

Currently, the descriptor makes the sequence immutable. This is to avoid that an element of the sequence could be set without being properly validated. In the future, the data could be wrapped in a safe list that would safely allow mutability.

Parameters

- **default** (`object, optional`) – When specified, the descriptor makes this attribute optional by attaching a default value to it.
- **unit** (`openmm.unit.Quantity, optional`) – When specified, only sequences of quantities with compatible units are allowed to be set.
- **converter** (`callable, optional`) – An optional function that can be used to validate and cast each element of the sequence before setting the attribute.

See also:

`ParameterAttribute` A simple parameter attribute.

`MappedParameterAttribute` A parameter attribute representing a mapping.

`IndexedMappedParameterAttribute` A parameter attribute representing a sequence, each term of which is a mapping.

Examples

Create an optional indexed attribute with unit of angstrom.

```
>>> from openmm import unit
>>> class MyParameter:
...     length = IndexedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = ['1 * angstrom', 0.5 * unit.nanometer]
>>> my_par.length[0]
Quantity(value=1, unit=angstrom)
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [1, '1.0', '1e-2', 4.0]
>>> my_par.attr_indexed
[1.0, 1.0, 0.01, 4.0]
```

__init__(*default=UNDEFINED*, *unit=None*, *converter=None*, *docstring=''*)

Methods

`__init__`([default, unit, converter, docstring])	
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

class UNDEFINED

Custom type used by `ParameterAttribute` to differentiate between `None` and undeclared default.

converter(*converter*)

Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

openff.toolkit.typing.engines.smirnoff.parameters.MappedParameterAttribute

class openff.toolkit.typing.engines.smirnoff.parameters.**MappedParameterAttribute**(*default=UNDEFINED*, *unit=None*, *converter=None*, *docstring=''*)

The attribute of a parameter in which each term is a mapping.

The substantial difference with `IndexedParameterAttribute` is that, unlike indexing, the mapping can be based on artbitrary references, like indices but can starting at non-zero values and include non-adjacent keys.

Parameters

- **default** (`object, optional`) – When specified, the descriptor makes this attribute optional by attaching a default value to it.
- **unit** (`openmm.unit.Quantity, optional`) – When specified, only sequences of mappings where values are quantities with compatible units are allowed to be set.
- **converter** (`callable, optional`) – An optional function that can be used to validate and cast each component of each element of the sequence before setting the attribute.

See also:

`IndexedParameterAttribute` A parameter attribute representing a sequence.

IndexedMappedParameterAttribute A parameter attribute representing a sequence, each term of which is a mapping.

Examples

Create an optional indexed attribute with unit of angstrom.

```
>>> from openmm import unit
>>> class MyParameter:
...     length = MappedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Like other ParameterAttribute objects, strings are parsed into Quantity objects.

```
>>> my_par.length = {1:'1.5 * angstrom', 2: '1.4 * angstrom'}
>>> my_par.length[1]
Quantity(value=1.5, unit=angstrom)
```

Unlike other ParameterAttribute objects, the reference points can do not need ot be zero-indexed, non-adjancent, such as interpolating defining a bond parameter for interpolation by defining references values and bond orders 2 and 3:

```
>>> my_par.length = {2:'1.42 * angstrom', 3: '1.35 * angstrom'}
>>> my_par.length[2]
Quantity(value=1.42, unit=angstrom)
```

__init__(*default=UNDEFINED*, *unit=None*, *converter=None*, *docstring=''*)

Methods

`__init__`([default, unit, converter, docstring])	
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

class UNDEFINED

Custom type used by `ParameterAttribute` to differentiate between `None` and undeclared default.

converter(*converter*)

Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

openff.toolkit.typing.engines.smirnoff.parameters.IndexedMappedParameterAttribute

class openff.toolkit.typing.engines.smirnoff.parameters.**IndexedMappedParameterAttribute**(*default=UNDEFINED*, *unit=None*, *converter=None*, *docstring=''*)

The attribute of a parameter with an unspecified number of terms, where each term is a mapping.

Some parameters can be associated to multiple terms, where those terms have multiple components. For example, torsions with fractional bond orders have parameters such as k1_bondorder1, k1_bondorder2, k2_bondorder1, k2_bondorder2, ..., and `IndexedMappedParameterAttribute` can be used to encapsulate the sequence of terms as mappings (typically, `dicts`) of their components.

The only substantial difference with `IndexedParameterAttribute` is that only sequences of mappings are supported as values and converters and units are checked on each component of each element in the sequence.

Currently, the descriptor makes the sequence immutable. This is to avoid that an element of the sequence could be set without being properly validated. In the future, the data could be wrapped in a safe list that would safely allow mutability.

Parameters

- **default** (`object, optional`) – When specified, the descriptor makes this attribute optional by attaching a default value to it.
- **unit** (`openmm.unit.Quantity, optional`) – When specified, only sequences of mappings where values are quantities with compatible units are allowed to be set.
- **converter** (`callable, optional`) – An optional function that can be used to validate and cast each component of each element of the sequence before setting the attribute.

See also:

`IndexedParameterAttribute` A parameter attribute representing a sequence.

`MappedParameterAttribute` A parameter attribute representing a mapping.

Examples

Create an optional indexed attribute with unit of angstrom.

```
>>> from openmm import unit
>>> class MyParameter:
...     length = IndexedMappedParameterAttribute(default=None, unit=unit.angstrom)
...
>>> my_par = MyParameter()
>>> my_par.length is None
True
```

Strings are parsed into Quantity objects.

```
>>> my_par.length = [{1:'1 * angstrom'}, {1: 0.5 * unit.nanometer}]
>>> my_par.length[0]
{1: Quantity(value=1, unit=angstrom)}
```

Similarly, custom converters work as with `ParameterAttribute`, but they are used to validate each value in the sequence.

```
>>> class MyParameter:
...     attr_indexed = IndexedMappedParameterAttribute(converter=float)
...
>>> my_par = MyParameter()
>>> my_par.attr_indexed = [{1: 1}, {2: '1.0', 3: '1e-2'}, {4: 4.0}]
>>> my_par.attr_indexed
[{1: 1.0}, {2: 1.0, 3: 0.01}, {4: 4.0}]
```

__init__(*default=UNDEFINED*, *unit=None*, *converter=None*, *docstring=''*)

Methods

`__init__`([default, unit, converter, docstring])	
`converter`(converter)	Create a new ParameterAttribute with an associated converter.

Attributes

`name`

class UNDEFINED

Custom type used by `ParameterAttribute` to differentiate between `None` and undeclared default.

converter(*converter*)

Create a new ParameterAttribute with an associated converter.

This is meant to be used as a decorator (see main examples).

CHAPTER

THIRTEEN

UTILITIES

13.1 Toolkit wrappers

The toolkit wrappers provide a simple uniform API for accessing minimal functionality of cheminformatics toolkits.

These toolkit wrappers are generally used through a `ToolkitRegistry`, which can be constructed with a desired precedence of toolkits:

```
>>> from openff.toolkit.utils.toolkits import ToolkitRegistry, OpenEyeToolkitWrapper,␣
↪RDKitToolkitWrapper, AmberToolsToolkitWrapper
>>> toolkit_registry = ToolkitRegistry()
>>> toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper,␣
↪AmberToolsToolkitWrapper]
>>> [ toolkit_registry.register_toolkit(toolkit) for toolkit in toolkit_precedence if␣
↪toolkit.is_available() ]
[None, None, None]
```

The toolkit wrappers can then be accessed through the registry:

```
>>> from openff.toolkit.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
>>> from openff.toolkit import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> smiles = toolkit_registry.call('to_smiles', molecule)
```

The order of toolkits, as specified in `toolkit_precedence` above, determines the order in which the called method is resolved, i.e. if the toolkit with highest precedence has a `to_smiles` method, that is the toolkit that will be called. If the toolkit with highest precedence does not have such a method, it is attempted with other toolkits until one is found. By default, if a toolkit with an appropriately-named method raises an exception of any type, then iteration over the registered toolkits stops and that exception is raised. To continue iteration if specific exceptions are encountered, customize this behavior using the optional `raise_exception_types` keyword argument to `ToolkitRegistry.call`. If no registered toolkits have the method, a ValueError is raised, containing a message listing the registered toolkits and exceptions (if any) that were ignored.

Alternatively, the global toolkit registry (which will attempt to register any available toolkits) can be used:

```
>>> from openff.toolkit.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
>>> len(toolkit_registry.registered_toolkits)
4
```

Individual toolkits can be registered or deregistered to control the backend that ToolkitRegistry calls resolve to. This can be useful for debugging and exploring subtley different behavior between toolkit wrappers.

```
from openff.toolkit.utils.toolkits import OpenEyeToolkitWrapper, BuiltInToolkitWrapper
from openff.toolkit.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_registry
toolkit_registry.deregister_toolkit(OpenEyeToolkitWrapper)
toolkit_registry.register_toolkit(BuiltInToolkitWrapper)
toolkit_registry.registered_toolkits
```

For example, differences in `to_smiles` functionality between OpenEye toolkits and The RDKit can be explored by selecting which toolkit(s) are and are not registered.

```
>>> from openff.toolkit.utils.toolkits import OpenEyeToolkitWrapper, GLOBAL_TOOLKIT_REGISTRY␣
↪as toolkit_registry
>>> from openff.toolkit import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> smiles_via_openeye = toolkit_registry.call('to_smiles', molecule)
>>> print(smiles_via_openeye)
[H]c1c(c(c(c(c1[H])[H])C([H])([H])[H])[H])[H]

>>> toolkit_registry.deregister_toolkit(OpenEyeToolkitWrapper)
>>> smiles_via_rdkit = toolkit_registry.call('to_smiles', molecule)
>>> print(smiles_via_rdkit)
[H][c]1[c]([H])[c]([H])[c]([C]([H])([H])[H])[c]([H])[c]1[H]
```

ToolkitRegistry	Registry for ToolkitWrapper objects
ToolkitWrapper	Toolkit wrapper base class.
OpenEyeToolkitWrapper	OpenEye toolkit wrapper
RDKitToolkitWrapper	RDKit toolkit wrapper
AmberToolsToolkitWrapper	AmberTools toolkit wrapper
BuiltInToolkitWrapper	Built-in ToolkitWrapper for very basic functionality.

13.1.1 openff.toolkit.utils.toolkits.ToolkitRegistry

class openff.toolkit.utils.toolkits.**ToolkitRegistry**(*toolkit_precedence=None*, *exception_if_unavailable=True*, *_register_imported_toolkit_wrappers=False*)

Registry for ToolkitWrapper objects

Examples

Register toolkits in a specified order, skipping if unavailable

```
>>> from openff.toolkit.utils.toolkits import ToolkitRegistry
>>> toolkit_precedence = [OpenEyeToolkitWrapper, RDKitToolkitWrapper,␣
↪AmberToolsToolkitWrapper]
>>> toolkit_registry = ToolkitRegistry(toolkit_precedence)
>>> toolkit_registry
ToolkitRegistry containing OpenEye Toolkit, The RDKit, AmberTools
```

Register all available toolkits (in the order OpenEye, RDKit, AmberTools, built-in)

```
>>> toolkits = [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper,␣
↪BuiltInToolkitWrapper]
>>> toolkit_registry = ToolkitRegistry(toolkit_precedence=toolkits)
>>> toolkit_registry
ToolkitRegistry containing OpenEye Toolkit, The RDKit, AmberTools, Built-in Toolkit
```

Retrieve the global singleton toolkit registry, which is created when this module is imported from all available toolkits:

```
>>> from openff.toolkit.utils.toolkits import GLOBAL_TOOLKIT_REGISTRY as toolkit_
↪registry
>>> toolkit_registry
ToolkitRegistry containing OpenEye Toolkit, The RDKit, AmberTools, Built-in Toolkit
```

Note that this will contain different ToolkitWrapper objects based on what toolkits are currently installed.

Warning: This API is experimental and subject to change.

__init__(*toolkit_precedence=None*, *exception_if_unavailable=True*, *_register_imported_toolkit_wrappers=False*)

Create an empty toolkit registry.

Parameters

- **toolkit_precedence** (list, optional, default=None) – List of toolkit wrapper classes, in order of desired precedence when performing molecule operations. If None, no toolkits will be registered.
- **exception_if_unavailable** (bool, optional, default=True) – If True, an exception will be raised if the toolkit is unavailable
- **_register_imported_toolkit_wrappers** (bool, optional, default=False) – If True, will attempt to register all imported ToolkitWrapper subclasses that can be found in the order of toolkit_precedence, if specified. If toolkit_precedence is not specified, the default order is [OpenEyeToolkitWrapper, RDKitToolkitWrapper, AmberToolsToolkitWrapper, BuiltInToolkitWrapper].

Methods

__init__([toolkit_precedence, ...])	Create an empty toolkit registry.
add_toolkit(toolkit_wrapper)	Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry
call(method_name, *args[, raise_exception_types])	Execute the requested method by attempting to use all registered toolkits in order of precedence.
deregister_toolkit(toolkit_wrapper)	Remove a ToolkitWrapper from the list of toolkits in this ToolkitRegistry
register_toolkit(toolkit_wrapper[, ...])	Register the provided toolkit wrapper class, instantiating an object of it.
resolve(method_name)	Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

Attributes

registered_toolkit_versions	Return a dict containing the version of each registered toolkit.
registered_toolkits	List registered toolkits.

property registered_toolkits

List registered toolkits.

> **Warning:** This API is experimental and subject to change.

Returns **toolkits** (*iterable of toolkit objects*)

property registered_toolkit_versions

Return a dict containing the version of each registered toolkit.

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_versions** (*dict[str, str]*) – A dictionary mapping names and versions of wrapped toolkits

register_toolkit(*toolkit_wrapper*, *exception_if_unavailable=True*)

Register the provided toolkit wrapper class, instantiating an object of it.

> **Warning:** This API is experimental and subject to change.

Parameters

- **toolkit_wrapper** (instance or subclass of ToolkitWrapper) – The toolkit wrapper to register or its class.
- **exception_if_unavailable** (bool, optional, default=True) – If True, an exception will be raised if the toolkit is unavailable

deregister_toolkit(*toolkit_wrapper*)

Remove a ToolkitWrapper from the list of toolkits in this ToolkitRegistry

> **Warning:** This API is experimental and subject to change.

Parameters **toolkit_wrapper** (instance or subclass of ToolkitWrapper) – The toolkit wrapper to remove from the registry

Raises

- **InvalidToolkitError** – If toolkit_wrapper is not a ToolkitWrapper or subclass
- **ToolkitUnavailableException** – If toolkit_wrapper is not found in the registry

add_toolkit(*toolkit_wrapper*)

Append a ToolkitWrapper onto the list of toolkits in this ToolkitRegistry

> **Warning:** This API is experimental and subject to change.

Parameters **toolkit_wrapper** (openff.toolkit.utils.ToolkitWrapper) – The ToolkitWrapper object to add to the list of registered toolkits

Raises **InvalidToolkitError** – If toolkit_wrapper is not a ToolkitWrapper or subclass

resolve(*method_name*)

Resolve the requested method name by checking all registered toolkits in order of precedence for one that provides the requested method.

Parameters **method_name** (str) – The name of the method to resolve

Returns *method* – The method of the first registered toolkit that provides the requested method name

Raises **NotImplementedError if the requested method cannot be found among the registered toolkits** –

Examples

Create a molecule, and call the toolkit `to_smiles()` method directly

```
>>> from openff.toolkit import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> toolkit_registry = ToolkitRegistry([OpenEyeToolkitWrapper, RDKitToolkitWrapper,
↪ AmberToolsToolkitWrapper])
>>> method = toolkit_registry.resolve('to_smiles')
>>> smiles = method(molecule)
```

call(*method_name*, **args*, *raise_exception_types=None*, ***kwargs*)

Execute the requested method by attempting to use all registered toolkits in order of precedence.

`*args` and `**kwargs` are passed to the desired method, and return values of the method are returned

This is a convenient shorthand for `toolkit_registry.resolve_method(method_name)(*args, **kwargs)`

Parameters

- **method_name** (str) – The name of the method to execute
- **raise_exception_types** (list of Exception subclasses, default=None) – A list of exception-derived types to catch and raise immediately. If None, this will be set to [Exception], which will raise an error immediately if the first ToolkitWrapper in the registry fails. To try each ToolkitWrapper that provides a suitably-named method, set this to the empty list ([]). If all ToolkitWrappers run without raising any exceptions in this list, a single ValueError will be raised containing the each ToolkitWrapper that was tried and the exception it raised.

Raises

- **NotImplementedError if the requested method cannot be found among the registered toolkits** –
- **ValueError if no exceptions in the raise_exception_types list were raised by ToolkitWrappers, and** –
- **all ToolkitWrappers in the ToolkitRegistry were tried.** –
- **Other forms of exceptions are possible if raise_exception_types is specified.** –
- **These are defined by the ToolkitWrapper method being called.** –

Examples

Create a molecule, and call the toolkit `to_smiles()` method directly

```
>>> from openff.toolkit import Molecule
>>> molecule = Molecule.from_smiles('Cc1ccccc1')
>>> toolkit_registry = ToolkitRegistry([OpenEyeToolkitWrapper,␣
↪RDKitToolkitWrapper])
>>> smiles = toolkit_registry.call('to_smiles', molecule)
```

13.1.2 openff.toolkit.utils.toolkits.ToolkitWrapper

class openff.toolkit.utils.toolkits.**ToolkitWrapper**

Toolkit wrapper base class.

> **Warning:** This API is experimental and subject to change.

__init__()

Methods

`__init__`()	
`from_file`(file_path, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file using this toolkit.
`from_file_obj`(file_obj, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.
`is_available`()	Check whether the corresponding toolkit can be imported
`requires_toolkit`()	

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	Instructions on how to install the wrapped toolkit.
toolkit_name	Return the name of the toolkit wrapped by this class as a str
toolkit_version	Return the version of the wrapped toolkit as a str

property toolkit_name

Return the name of the toolkit wrapped by this class as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_name** (*str*) – The name of the wrapped toolkit

property toolkit_installation_instructions

Instructions on how to install the wrapped toolkit.

property toolkit_file_read_formats

List of file formats that this toolkit can read.

property toolkit_file_write_formats

List of file formats that this toolkit can write.

classmethod is_available()

Check whether the corresponding toolkit can be imported

Returns **is_installed** (*bool*) – True if corresponding toolkit is installed, False otherwise.

property toolkit_version

Return the version of the wrapped toolkit as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_version** (*str*) – The version of the wrapped toolkit

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*)

Return an openff.toolkit.topology.Molecule from a file using this toolkit.

Parameters

- **file_path** (str) – The file to read the molecule from
- **file_format** (str) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if any molecules contain undefined stereochemistry.

- **_cls** (class) – Molecule constructor

Returns molecules (*Molecule or list of Molecules*) – a list of Molecule objects is returned.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.

Parameters

- **file_obj** (file-like object) – The file-like object to read the molecule from
- **file_format** (str) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.
- **_cls** (class) – Molecule constructor

Returns molecules (*Molecule or list of Molecules*) – a list of Molecule objects is returned.

13.1.3 openff.toolkit.utils.toolkits.OpenEyeToolkitWrapper

class openff.toolkit.utils.toolkits.**OpenEyeToolkitWrapper**

OpenEye toolkit wrapper

> **Warning:** This API is experimental and subject to change.

__init__()

Methods

__init__()	
apply_elf_conformer_selection(molecule[, ...])	Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.
assign_fractional_bond_orders(molecule[, ...])	Update and store list of bond orders this molecule.
assign_partial_charges(molecule[, ...])	Compute partial charges with OpenEye quacpac, and assign the new values to the partial_charges attribute.
atom_is_in_ring(atom)	Return whether or not an atom is in a ring.
bond_is_in_ring(bond)	Return whether or not a bond is in a ring.
canonical_order_atoms(molecule)	Canonical order the atoms in the molecule using the OpenEye toolkit.
compute_partial_charges_am1bcc(molecule[, ...])	Compute AM1BCC partial charges with OpenEye quacpac.
enumerate_protomers(molecule[, max_states])	Enumerate the formal charges of a molecule to generate different protomoers.
enumerate_stereoisomers(molecule[, ...])	Enumerate the stereocenters and bonds of the current molecule.
enumerate_tautomers(molecule[, max_states])	Enumerate the possible tautomers of the current molecule
find_smarts_matches(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
from_file(file_path, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file using this toolkit.
from_file_obj(file_obj, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.
from_inchi(inchi[, allow_undefined_stereo, _cls])	Construct a Molecule from a InChI representation
from_iupac(iupac_name[, ...])	Construct a Molecule from an IUPAC name
from_object(obj[, allow_undefined_stereo, _cls])	Convert an OEMol (or OEMol-derived object) into an openff.toolkit.topology.molecule
from_openeye(oemol[, ...])	Create a Molecule from an OpenEye molecule.
from_smiles(smiles[, ...])	Create a Molecule from a SMILES string using the OpenEye toolkit.
generate_conformers(molecule[, ...])	Generate molecule conformers using OpenEye Omega.
get_tagged_smarts_connectivity(smarts)	Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string.
is_available()	Check if the given OpenEye toolkit components are available.
requires_toolkit()	
to_file(molecule, file_path, file_format)	Writes an OpenFF Molecule to a file-like object
to_file_obj(molecule, file_obj, file_format)	Writes an OpenFF Molecule to a file-like object
to_inchi(molecule[, fixed_hydrogens])	Create an InChI string for the molecule using the RDKit Toolkit.
to_inchikey(molecule[, fixed_hydrogens])	Create an InChIKey for the molecule using the RDKit Toolkit.
to_iupac(molecule)	Generate IUPAC name from Molecule
to_openeye(molecule[, aromaticity_model])	Create an OpenEye molecule using the specified aromaticity model
to_smiles(molecule[, isomeric, ...])	Uses the OpenEye toolkit to convert a Molecule into a SMILES string.

Attributes

to_openeye_cache	
toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	Instructions on how to install the wrapped toolkit.
toolkit_name	Return the name of the toolkit wrapped by this class as a str
toolkit_version	Return the version of the wrapped toolkit as a str

classmethod is_available()

Check if the given OpenEye toolkit components are available.

If the OpenEye toolkit is not installed or no license is found for at least one the required toolkits , `False` is returned.

Returns **all_installed** (*bool*) – `True` if all required OpenEye tools are installed and licensed, `False` otherwise

from_object(*obj*, *allow_undefined_stereo=False*, *_cls=None*)

Convert an OEMol (or OEMol-derived object) into an openff.toolkit.topology.molecule

Parameters

- **obj** (`A molecule-like object`) – An object to by type-checked.
- **allow_undefined_stereo** (`bool`, `default=False`) – Whether to accept molecules with undefined stereocenters. If False, an exception will be raised if a molecule with undefined stereochemistry is passed into this function.
- **_cls** (`class`) – Molecule constructor

Returns *Molecule* – An openff.toolkit.topology.molecule Molecule.

Raises `NotImplementedError` – If the object could not be converted into a Molecule.

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openff.toolkit.topology.Molecule from a file using this toolkit.

Parameters

- **file_path** (`str`) – The file to read the molecule from
- **file_format** (`str`) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (`bool`, `default=False`) – If false, raises an exception if oemol contains undefined stereochemistry.
- **_cls** (`class`) – Molecule constructor

Returns **molecules** (*List[Molecule]*) – The list of `Molecule` objects in the file.

Raises `GAFFAtomTypeWarning` – If the loaded mol2 file possibly uses GAFF atom types, which are not supported.

Examples

Load a mol2 file into an OpenFF `Molecule` object.

```
>>> from openff.toolkit.utils import get_data_file_path
>>> mol2_file_path = get_data_file_path('molecules/cyclohexane.mol2')
>>> toolkit = OpenEyeToolkitWrapper()
>>> molecule = toolkit.from_file(mol2_file_path, file_format='mol2')
```

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.

Parameters

- **file_obj** (`file-like object`) – The file-like object to read the molecule from
- **file_format** (`str`) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (`bool, default=False`) – If false, raises an exception if oemol contains undefined stereochemistry.
- **_cls** (`class`) – Molecule constructor

Returns **molecules** (*List[Molecule]*) – The list of Molecule objects in the file object.

Raises **GAFFAtomTypeWarning** – If the loaded mol2 file possibly uses GAFF atom types, which are not supported.

to_file_obj(*molecule*, *file_obj*, *file_format*)

Writes an OpenFF Molecule to a file-like object

Parameters

- **molecule** (`an OpenFF Molecule`) – The molecule to write
- **file_obj** – The file-like object to write to
- **file_format** – The format for writing the molecule data

to_file(*molecule*, *file_path*, *file_format*)

Writes an OpenFF Molecule to a file-like object

Parameters

- **molecule** (`an OpenFF Molecule`) – The molecule to write
- **file_path** – The file path to write to.
- **file_format** – The format for writing the molecule data

enumerate_protomers(*molecule*, *max_states=10*)

Enumerate the formal charges of a molecule to generate different protomoers.

Parameters

- **molecule** (`Molecule`) – The molecule whose state we should enumerate
- **max_states** (`int optional, default=10,`) – The maximum number of protomer states to be returned.

Returns molecules (*List[openff.toolkit.topology.Molecule],*) – A list of the protomers of the input molecules not including the input.

enumerate_stereoisomers(*molecule*, *undefined_only=False*, *max_isomers=20*, *rationalise=True*)

Enumerate the stereocenters and bonds of the current molecule.

Parameters

- **molecule** (Molecule) – The molecule whose state we should enumerate
- **undefined_only** (bool optional, default=False) – If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry
- **max_isomers** (int optional, default=20) – The maximum amount of molecules that should be returned
- **rationalise** (bool optional, default=True) – If we should try to build and rationalise the molecule to ensure it can exist

Returns molecules (*List[openff.toolkit.topology.Molecule]*) – A list of openff.toolkit.topology.Molecule instances

enumerate_tautomers(*molecule*, *max_states=20*)

Enumerate the possible tautomers of the current molecule

Parameters

- **molecule** (Molecule) – The molecule whose state we should enumerate
- **max_states** (int optional, default=20) – The maximum amount of molecules that should be returned

Returns molecules (*List[openff.toolkit.topology.Molecule]*) – A list of openff.toolkit.topology.Molecule instances excluding the input molecule.

static from_openeye(*oemol*, *allow_undefined_stereo=False*, *_cls=None*)

Create a Molecule from an OpenEye molecule. If the OpenEye molecule has implicit hydrogens, this function will make them explicit.

OEAtom s have a different set of allowed value for partial charges than openff.toolkit.topology.Molecule s. In the OpenEye toolkits, partial charges are stored on individual OEAtom s, and their values are initialized to 0.0. In the Open Force Field Toolkit, an openff.toolkit.topology.Molecule's partial_charges attribute is initialized to None and can be set to a unit-wrapped numpy array with units of elementary charge. The Open Force Field Toolkit considers an OEMol where every OEAtom has a partial charge of float('nan') to be equivalent to an Open Force Field Toolkit *Molecule*'s partial_charges = None. This assumption is made in both to_openeye and from_openeye.

Warning: This API is experimental and subject to change.

Parameters

- **oemol** (openeye.oechem.OEMol) – An OpenEye molecule
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if oemol contains undefined stereochemistry.
- **_cls** (class) – Molecule constructor

Returns molecule (*openff.toolkit.topology.Molecule*) – An OpenFF molecule

Examples

Create a Molecule from an OpenEye OEMol

```
>>> from openeye import oechem
>>> from openff.toolkit.tests.utils import get_data_file_path
>>> ifs = oechem.oemolistream(get_data_file_path('systems/monomers/ethanol.mol2'))
>>> oemols = list(ifs.GetOEGraphMols())
```

```
>>> toolkit_wrapper = OpenEyeToolkitWrapper()
>>> molecule = toolkit_wrapper.from_openeye(oemols[0])
```

to_openeye(*molecule*, *aromaticity_model=DEFAULT_AROMATICITY_MODEL*)

Create an OpenEye molecule using the specified aromaticity model

`OEAtom` s have a different set of allowed value for partial charges than `openff.toolkit.topology.Molecules`. In the OpenEye toolkits, partial charges are stored on individual `OEAtoms`, and their values are initialized to `0.0`. In the Open Force Field Toolkit, an``openff.toolkit.topology.Molecule``'s `partial_charges` attribute is initialized to `None` and can be set to a unit-wrapped numpy array with units of elementary charge. The Open Force Field Toolkit considers an `OEMol` where every `OEAtom` has a partial charge of `float('nan')` to be equivalent to an Open Force Field Toolkit `Molecule`'s `partial_charges = None`. This assumption is made in both `to_openeye` and `from_openeye`.

Warning: This API is experimental and subject to change.

Parameters

- **molecule** (`openff.toolkit.topology.molecule.Molecule object`) – The molecule to convert to an OEMol
- **aromaticity_model** (`str, optional, default=DEFAULT_AROMATICITY_MODEL`) – The aromaticity model to use

Returns **oemol** (*openeye.oechem.OEMol*) – An OpenEye molecule

Examples

Create an OpenEye molecule from a Molecule

```
>>> from openff.toolkit import Molecule
>>> toolkit_wrapper = OpenEyeToolkitWrapper()
>>> molecule = Molecule.from_smiles('CC')
>>> oemol = toolkit_wrapper.to_openeye(molecule)
```

atom_is_in_ring(*atom:* Atom) → bool

Return whether or not an atom is in a ring.

It is assumed that this atom is in molecule.

Parameters **atom** (Atom) – The molecule containing the atom of interest

Returns **is_in_ring** (*bool*) – Whether or not the atom is in a ring.

Raises **NotAttachedToMoleculeError** –

bond_is_in_ring(*bond:* Bond) → bool

Return whether or not a bond is in a ring.

It is assumed that this atom is in molecule.

Parameters **bond** (Bond) – The molecule containing the atom of interest

Returns **is_in_ring** (*bool*) – Whether or not the bond of index *bond_index* is in a ring

Raises **NotAttachedToMoleculeError** –

to_smiles(*molecule*, *isomeric=True*, *explicit_hydrogens=True*, *mapped=False*)

Uses the OpenEye toolkit to convert a Molecule into a SMILES string. A partially mapped smiles can also be generated for atoms of interest by supplying an *atom_map* to the properties dictionary.

Parameters

- **molecule** (An openff.toolkit.topology.Molecule) – The molecule to convert into a SMILES.
- **isomeric** (bool optional, default= True) – return an isomeric smiles
- **explicit_hydrogens** (bool optional, default=True) – return a smiles string containing all hydrogens explicitly
- **mapped** (bool optional, default=False) – return a explicit hydrogen mapped smiles, the atoms to be mapped can be controlled by supplying an atom map into the properties dictionary. If no mapping is passed all atoms will be mapped in order, else an atom map dictionary from the current atom index to the map id should be supplied with no duplicates. The map ids (values) should start from 0 or 1.

Returns **smiles** (*str*) – The SMILES of the input molecule.

to_inchi(*molecule*, *fixed_hydrogens=False*)

Create an InChI string for the molecule using the RDKit Toolkit. InChI is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

- **molecule** (An openff.toolkit.topology.Molecule) – The molecule to convert into a SMILES.
- **fixed_hydrogens** (bool, default=False) – If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

Returns **inchi** (*str*) – The InChI string of the molecule.

to_inchikey(*molecule*, *fixed_hydrogens=False*)

Create an InChIKey for the molecule using the RDKit Toolkit. InChIKey is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

- **molecule** (An openff.toolkit.topology.Molecule) – The molecule to convert into a SMILES.

- **fixed_hydrogens** (bool, default=False) – If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

Returns **inchi_key** (*str*) – The InChIKey representation of the molecule.

to_iupac(*molecule*)

Generate IUPAC name from Molecule

Parameters **molecule** (An openff.toolkit.topology.Molecule) – The molecule to convert into a SMILES.

Returns **iupac_name** (*str*) – IUPAC name of the molecule

Examples

```
>>> from openff.toolkit import Molecule
>>> from openff.toolkit.utils import get_data_file_path
>>> sdf_filepath = get_data_file_path('molecules/ethanol.sdf')
>>> molecule = Molecule(sdf_filepath)
>>> toolkit = OpenEyeToolkitWrapper()
>>> iupac_name = toolkit.to_iupac(molecule)
```

canonical_order_atoms(*molecule*)

Canonical order the atoms in the molecule using the OpenEye toolkit.

Parameters

- **molecule** (Molecule) – The input molecule

  Returns

- **------** –

- **molecule** – The input molecule, with canonically-indexed atoms and bonds.

from_smiles(*smiles*, *hydrogens_are_explicit=False*, *allow_undefined_stereo=False*, *_cls=None*)

Create a Molecule from a SMILES string using the OpenEye toolkit.

> **Warning:** This API is experimental and subject to change.

Parameters

- **smiles** (str) – The SMILES string to turn into a molecule
- **hydrogens_are_explicit** (bool, default = False) – If False, OE will perform hydrogen addition using OEAddExplicitHydrogens
- **allow_undefined_stereo** (bool, default=False) – Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.
- **_cls** (class) – Molecule constructor

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An OpenFF style molecule.

from_inchi(*inchi*, *allow_undefined_stereo=False*, *_cls=None*)

Construct a Molecule from a InChI representation

Parameters

- **inchi** (str) – The InChI representation of the molecule.
- **allow_undefined_stereo** (bool, default=False) – Whether to accept InChI with undefined stereochemistry. If False, an exception will be raised if a InChI with undefined stereochemistry is passed into this function.
- **_cls** (class) – Molecule constructor

Returns **molecule** (*openff.toolkit.topology.Molecule*)

from_iupac(*iupac_name*, *allow_undefined_stereo=False*, *_cls=None*, ***kwargs*)

Construct a Molecule from an IUPAC name

Parameters

- **iupac_name** (str) – The IUPAC or common name of the molecule.
- **allow_undefined_stereo** (bool, default=False) – Whether to accept a molecule name with undefined stereochemistry. If False, an exception will be raised if a molecule name with undefined stereochemistry is passed into this function.
- **_cls** (class) – Molecule constructor

Returns **molecule** (*openff.toolkit.topology.Molecule*)

generate_conformers(*molecule*, *n_conformers=1*, *rms_cutoff=None*, *clear_existing=True*, *make_carboxylic_acids_cis=False*)

Generate molecule conformers using OpenEye Omega.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (a Molecule) – The molecule to generate conformers for.
- **n_conformers** (int, default=1) – The maximum number of conformers to generate.
- **rms_cutoff** (unit-wrapped float, in units of distance, optional, default=None) – The minimum RMS value at which two conformers are considered redundant and one is deleted. If None, the cutoff is set to 1 Angstrom
- **clear_existing** (bool, default=True) – Whether to overwrite existing conformers for the molecule
- **make_carboxylic_acids_cis** (bool, default=False) – Guarantee all conformers have exclusively cis carboxylic acid groups (COOH) by rotating the proton in any trans carboxylic acids 180 degrees around the C-O bond.

apply_elf_conformer_selection(*molecule: Molecule*, *percentage: float = 2.0*, *limit: int = 10*)

Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.

Notes

- The input molecule should have a large set of conformers already generated to select the ELF conformers from.
- The selected conformers will be retained in the `molecule.conformers` list while unselected conformers will be discarded.
- Conformers generated with the OpenEye toolkit often include trans carboxylic acids (COOH). These are unphysical and will be rejected by `apply_elf_conformer_selection`. If no conformers are selected, try re-running `generate_conformers` with the `make_carboxylic_acids_cis` argument set to `True`

See also:

`RDKitToolkitWrapper.apply_elf_conformer_selection`

Parameters

- **molecule** – The molecule which contains the set of conformers to select from.
- **percentage** – The percentage of conformers with the lowest electrostatic interaction energies to greedily select from.
- **limit** – The maximum number of conformers to select.

assign_partial_charges(*molecule*, *partial_charge_method=None*, *use_conformers=None*, *strict_n_conformers=False*, *normalize_partial_charges=True*, *_cls=None*)

Compute partial charges with OpenEye quacpac, and assign the new values to the partial_charges attribute.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (`Molecule`) – Molecule for which partial charges are to be computed
- **partial_charge_method** (`str, optional, default=None`) – The charge model to use. One of ['amberff94', 'mmff', 'mmff94', 'am1-mulliken', 'am1bcc', 'am1bccnosymspt', 'am1bccelf10'] If None, 'am1-mulliken' will be used.
- **use_conformers** (`iterable of unit-wrapped numpy arrays, each with`) – shape (n_atoms, 3) and dimension of distance. Optional, default = None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **strict_n_conformers** (`bool, default=False`) – Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.
- **normalize_partial_charges** (`bool, default=True`) – Whether to offset partial charges so that they sum to the total formal charge of the molecule. This is used to prevent accumulation of rounding errors when the partial charge generation method has low precision.
- **_cls** (`class`) – Molecule constructor

Raises

- **ChargeMethodUnavailableError if the requested charge method can not be handled by this toolkit** –
- **ChargeCalculationError if the charge method is supported by this toolkit, but fails** –

compute_partial_charges_am1bcc(*molecule*, *use_conformers=None*, *strict_n_conformers=False*)

Compute AM1BCC partial charges with OpenEye quacpac. This function will attempt to use the OEAM1BCCELF10 charge generation method, but may print a warning and fall back to normal OEAM1BCC if an error is encountered. This error is known to occur with some carboxylic acids, and is under investigation by OpenEye.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (Molecule) – Molecule for which partial charges are to be computed
- **use_conformers** (iterable of unit-wrapped numpy arrays, each with) – shape (n_atoms, 3) and dimension of distance. Optional, default = None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **strict_n_conformers** (bool, default=False) – Whether to raise an exception if an invalid number of conformers is provided. If this is False and an invalid number of conformers is found, a warning will be raised instead of an Exception.

Returns charges (*numpy.array of shape (natoms) of type float*) – The partial charges

assign_fractional_bond_orders(*molecule*, *bond_order_model=None*, *use_conformers=None*, *_cls=None*)

Update and store list of bond orders this molecule. Bond orders are stored on each bond, in the *bond.fractional_bond_order* attribute.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (openff.toolkit.topology.molecule Molecule) – The molecule to assign wiberg bond orders to
- **bond_order_model** (str, optional, default=None) – The charge model to use. One of ['am1-wiberg', 'am1-wiberg-elf10', 'pm3-wiberg', 'pm3-wiberg-elf10']. If None, 'am1-wiberg' will be used.
- **use_conformers** (iterable of unit-wrapped np.array with shape (n_atoms, 3) and) – dimension of distance, optional, default=None The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper. If the chosen bond_order_model is an ELF variant, the ELF conformer selection method will be applied to the provided conformers.
- **_cls** (class) – Molecule constructor

get_tagged_smarts_connectivity(*smarts*)

Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string. Does not return bond order.

Parameters **smarts** (str) – The tagged SMARTS to analyze

Returns

- **unique_tags** (*tuple of int*) – A sorted tuple of all unique tagged atom map indices.
- **tagged_atom_connectivity** (*tuple of tuples of int, shape n_tagged_bonds x 2*) – A tuple of tuples, where each inner tuple is a pair of tagged atoms (tag_idx_1, tag_idx_2) which are bonded. The inner tuples are ordered smallest-to-largest, and the tuple of tuples is ordered lexically. The return value for an improper torsion would be ((1, 2), (2, 3), (2, 4)).

Raises **SMIRKSParsingError** – If OpenEye toolkit was unable to parse the provided smirks/tagged smarts

find_smarts_matches(*molecule*, *smarts*, *aromaticity_model='OEAroModel_MDL'*, *unique=False*)

Find all SMARTS matches for the specified molecule, using the specified aromaticity model.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (Molecule) – The molecule for which all specified SMARTS matches are to be located
- **smarts** (str) – SMARTS string with optional SMIRKS-style atom tagging
- **aromaticity_model** (str, optional, default='OEAroModel_MDL') – Molecule is prepared with this aromaticity model prior to querying.
- : (.. note) – Currently:
- **OEAroModel_MDL** (the only supported aromaticity_model is) –

property toolkit_file_read_formats

List of file formats that this toolkit can read.

property toolkit_file_write_formats

List of file formats that this toolkit can write.

property toolkit_installation_instructions

Instructions on how to install the wrapped toolkit.

property toolkit_name

Return the name of the toolkit wrapped by this class as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_name** (*str*) – The name of the wrapped toolkit

property toolkit_version

Return the version of the wrapped toolkit as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_version** (*str*) – The version of the wrapped toolkit

13.1.4 openff.toolkit.utils.toolkits.RDKitToolkitWrapper

class openff.toolkit.utils.toolkits.**RDKitToolkitWrapper**

RDKit toolkit wrapper

> **Warning:** This API is experimental and subject to change.

__init__()

Methods

`__init__`()	
`apply_elf_conformer_selection`(molecule[, ...])	Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.
`assign_partial_charges`(molecule[, ...])	Compute partial charges with RDKit, and assign the new values to the partial_charges attribute.
`atom_is_in_ring`(atom)	Return whether or not an atom is in a ring.
`bond_is_in_ring`(bond)	Return whether or not a bond is in a ring.
`canonical_order_atoms`(molecule)	Canonical order the atoms in the molecule using the RDKit.
`enumerate_stereoisomers`(molecule[, ...])	Enumerate the stereocenters and bonds of the current molecule.
`enumerate_tautomers`(molecule[, max_states])	Enumerate the possible tautomers of the current molecule.
`find_smarts_matches`(molecule, smarts[, ...])	Find all SMARTS matches for the specified molecule, using the specified aromaticity model.
`from_file`(file_path, file_format[, ...])	Create an openff.toolkit.topology.Molecule from a file using this toolkit.
`from_file_obj`(file_obj, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file-like object using this toolkit.
`from_inchi`(inchi[, allow_undefined_stereo, _cls])	Construct a Molecule from a InChI representation
`from_object`(obj[, allow_undefined_stereo, _cls])	If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openff.toolkit.topology.molecule.
`from_pdb_and_smiles`(file_path, smiles[, ...])	Create a Molecule from a pdb file and a SMILES string using RDKit.
`from_rdkit`(rdmol[, allow_undefined_stereo, ...])	Create a Molecule from an RDKit molecule.
`from_smiles`(smiles[, ...])	Create a Molecule from a SMILES string using the RDKit toolkit.
`generate_conformers`(molecule[, ...])	Generate molecule conformers using RDKit.
`get_tagged_smarts_connectivity`(smarts)	Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string.
`is_available`()	Check whether the RDKit toolkit can be imported
`requires_toolkit`()	
`to_file`(molecule, file_path, file_format)	Writes an OpenFF Molecule to a file-like object
`to_file_obj`(molecule, file_obj, file_format)	Writes an OpenFF Molecule to a file-like object
`to_inchi`(molecule[, fixed_hydrogens])	Create an InChI string for the molecule using the RDKit Toolkit.
`to_inchikey`(molecule[, fixed_hydrogens])	Create an InChIKey for the molecule using the RDKit Toolkit.
`to_rdkit`(molecule[, aromaticity_model])	Create an RDKit molecule Requires the RDKit to be installed.
`to_smiles`(molecule[, isomeric, ...])	Uses the RDKit toolkit to convert a Molecule into a SMILES string.

Attributes

to_rdkit_cache

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	Instructions on how to install the wrapped toolkit.
toolkit_name	Return the name of the toolkit wrapped by this class as a str
toolkit_version	Return the version of the wrapped toolkit as a str

property toolkit_file_write_formats

List of file formats that this toolkit can write.

classmethod is_available()

Check whether the RDKit toolkit can be imported

Returns **is_installed** (*bool*) – True if RDKit is installed, False otherwise.

from_object(*obj*, *allow_undefined_stereo=False*, *_cls=None*)

If given an rdchem.Mol (or rdchem.Mol-derived object), this function will load it into an openff.toolkit.topology.molecule. Otherwise, it will return False.

Parameters

- **obj** (A rdchem.Mol-derived object) – An object to be type-checked and converted into a Molecule, if possible.
- **allow_undefined_stereo** (bool, default=False) – Whether to accept molecules with undefined stereocenters. If False, an exception will be raised if a molecule with undefined stereochemistry is passed into this function.
- **_cls** (class) – Molecule constructor

Returns *Molecule or False* – An openff.toolkit.topology.molecule Molecule.

Raises NotImplementedError – If the object could not be converted into a Molecule.

from_pdb_and_smiles(*file_path*, *smiles*, *allow_undefined_stereo=False*, *_cls=None*)

Create a Molecule from a pdb file and a SMILES string using RDKit.

Requires RDKit to be installed.

The molecule is created and sanitised based on the SMILES string, we then find a mapping between this molecule and one from the PDB based only on atomic number and connections. The SMILES molecule is then reindexed to match the PDB, the conformer is attached, and the molecule returned.

Note that any stereochemistry in the molecule is set by the SMILES, and not the coordinates of the PDB.

Parameters

- **file_path** (str) – PDB file path
- **smiles** (str) – a valid smiles string for the pdb, used for stereochemistry, formal charges, and bond order

- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if SMILES contains undefined stereochemistry.
- **_cls** (class) – Molecule constructor

Returns molecule (*openff.toolkit.Molecule (or _cls() type)*) – An OFFMol instance with ordering the same as used in the PDB file.

:raises InvalidConformerError : if the SMILES and PDB molecules are not isomorphic.:

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Create an openff.toolkit.topology.Molecule from a file using this toolkit.

Parameters

- **file_path** (str) – The file to read the molecule from
- **file_format** (str) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if oemol contains undefined stereochemistry.
- **_cls** (class) – Molecule constructor

Returns molecules (*iterable of Molecules*) – a list of Molecule objects is returned.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openff.toolkit.topology.Molecule from a file-like object using this toolkit.

A file-like object is an object with a ".read()" method.

> **Warning:** This API is experimental and subject to change.

Parameters

- **file_obj** (file-like object) – The file-like object to read the molecule from
- **file_format** (str) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if oemol contains undefined stereochemistry.
- **_cls** (class) – Molecule constructor

Returns molecules (*Molecule or list of Molecules*) – a list of Molecule objects is returned.

to_file_obj(*molecule*, *file_obj*, *file_format*)

Writes an OpenFF Molecule to a file-like object

Parameters

- **molecule** (an OpenFF Molecule) – The molecule to write
- **file_obj** – The file-like object to write to
- **file_format** – The format for writing the molecule data

to_file(*molecule*, *file_path*, *file_format*)

Writes an OpenFF Molecule to a file-like object

Parameters

- **molecule** (an OpenFF Molecule) – The molecule to write
- **file_path** – The file path to write to
- **file_format** – The format for writing the molecule data

enumerate_stereoisomers(*molecule*, *undefined_only=False*, *max_isomers=20*, *rationalise=True*)

Enumerate the stereocenters and bonds of the current molecule.

Parameters

- **molecule** (Molecule) – The molecule whose state we should enumerate
- **undefined_only** (bool optional, default=False) – If we should enumerate all stereocenters and bonds or only those with undefined stereochemistry
- **max_isomers** (int optional, default=20) – The maximum amount of molecules that should be returned
- **rationalise** (bool optional, default=True) – If we should try to build and rationalise the molecule to ensure it can exist

Returns molecules (*List[openff.toolkit.topology.Molecule]*) – A list of openff.toolkit.topology.Molecule instances

enumerate_tautomers(*molecule*, *max_states=20*)

Enumerate the possible tautomers of the current molecule.

Parameters

- **molecule** (Molecule) – The molecule whose state we should enumerate
- **max_states** (int optional, default=20) – The maximum amount of molecules that should be returned

Returns molecules (*List[openff.toolkit.topology.Molecule]*) – A list of openff.toolkit.topology.Molecule instances not including the input molecule.

canonical_order_atoms(*molecule*)

Canonical order the atoms in the molecule using the RDKit.

Parameters

- **molecule** (Molecule) – The input molecule

  Returns

- **-------** –
- **molecule** – The input molecule, with canonically-indexed atoms and bonds.

to_smiles(*molecule*, *isomeric=True*, *explicit_hydrogens=True*, *mapped=False*)

Uses the RDKit toolkit to convert a Molecule into a SMILES string. A partially mapped smiles can also be generated for atoms of interest by supplying an *atom_map* to the properties dictionary.

Parameters

- **molecule** (An openff.toolkit.topology.Molecule) – The molecule to convert into a SMILES.
- **isomeric** (bool optional, default= True) – return an isomeric smiles

- **explicit_hydrogens** (bool optional, default=True) – return a smiles string containing all hydrogens explicitly
- **mapped** (bool optional, default=False) – return a explicit hydrogen mapped smiles, the atoms to be mapped can be controlled by supplying an atom map into the properties dictionary. If no mapping is passed all atoms will be mapped in order, else an atom map dictionary from the current atom index to the map id should be supplied with no duplicates. The map ids (values) should start from 0 or 1.

Returns **smiles** (*str*) – The SMILES of the input molecule.

from_smiles(*smiles*, *hydrogens_are_explicit=False*, *allow_undefined_stereo=False*, *_cls=None*)

Create a Molecule from a SMILES string using the RDKit toolkit.

> **Warning:** This API is experimental and subject to change.

Parameters

- **smiles** (str) – The SMILES string to turn into a molecule
- **hydrogens_are_explicit** (bool, default=False) – If False, RDKit will perform hydrogen addition using Chem.AddHs
- **allow_undefined_stereo** (bool, default=False) – Whether to accept SMILES with undefined stereochemistry. If False, an exception will be raised if a SMILES with undefined stereochemistry is passed into this function.
- **_cls** (class) – Molecule constructor

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An OpenFF style molecule.

from_inchi(*inchi*, *allow_undefined_stereo=False*, *_cls=None*)

Construct a Molecule from a InChI representation

Parameters

- **inchi** (str) – The InChI representation of the molecule.
- **allow_undefined_stereo** (bool, default=False) – Whether to accept InChI with undefined stereochemistry. If False, an exception will be raised if a InChI with undefined stereochemistry is passed into this function.
- **_cls** (class) – Molecule constructor

Returns **molecule** (*openff.toolkit.topology.Molecule*)

generate_conformers(*molecule*, *n_conformers=1*, *rms_cutoff=None*, *clear_existing=True*, *_cls=None*, *make_carboxylic_acids_cis=False*)

Generate molecule conformers using RDKit.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (a Molecule) – The molecule to generate conformers for.

- **n_conformers** (int, default=1) – Maximum number of conformers to generate.
- **rms_cutoff** (unit-wrapped float, in units of distance, optional, default=None) – The minimum RMS value at which two conformers are considered redundant and one is deleted. If None, the cutoff is set to 1 Angstrom
- **clear_existing** (bool, default=True) – Whether to overwrite existing conformers for the molecule.
- **_cls** (class) – Molecule constructor
- **make_carboxylic_acids_cis** (bool, default=False) – Guarantee all conformers have exclusively cis carboxylic acid groups (COOH) by rotating the proton in any trans carboxylic acids 180 degrees around the C-O bond.

assign_partial_charges(*molecule*, *partial_charge_method=None*, *use_conformers=None*, *strict_n_conformers=False*, *normalize_partial_charges=True*, *_cls=None*)

Compute partial charges with RDKit, and assign the new values to the partial_charges attribute.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (Molecule) – Molecule for which partial charges are to be computed
- **partial_charge_method** (str, optional, default=None) – The charge model to use. One of ['mmff94']. If None, 'mmff94' will be used.
  - **'mmff94': Applies partial charges using the Merck Molecular Force Field** (MMFF). This method does not make use of conformers, and hence use_conformers and strict_n_conformers will not impact the partial charges produced.
- **use_conformers** (iterable of unit-wrapped numpy arrays, each with) – shape (n_atoms, 3) and dimension of distance. Optional, default = None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **strict_n_conformers** (bool, default=False) – Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.
- **normalize_partial_charges** (bool, default=True) – Whether to offset partial charges so that they sum to the total formal charge of the molecule. This is used to prevent accumulation of rounding errors when the partial charge generation method has low precision.
- **_cls** (class) – Molecule constructor

Raises

- **ChargeMethodUnavailableError if the requested charge method can not be handled by this toolkit** –
- **ChargeCalculationError if the charge method is supported by this toolkit, but fails** –

apply_elf_conformer_selection(*molecule:* Molecule, *percentage: float = 2.0*, *limit: int = 10*, *rms_tolerance: Quantity = 0.05 * unit.angstrom*)

Applies the ELF method to select a set of diverse conformers which have minimal electrostatically strongly interacting functional groups from a molecules conformers.

The diverse conformer selection is performed by the `_elf_select_diverse_conformers` function, which attempts to greedily select conformers which are most distinct according to their RMS.

Warning:

- Although this function is inspired by the OpenEye ELF10 method, this implementation may yield slightly different conformers due to potential differences in this and the OE closed source implementation.

Notes

- The input molecule should have a large set of conformers already generated to select the ELF10 conformers from.
- The selected conformers will be retained in the *molecule.conformers* list while unselected conformers will be discarded.
- Only heavy atoms are included when using the RMS to select diverse conformers.

See also:

`RDKitToolkitWrapper._elf_select_diverse_conformers`

Parameters

- **molecule** – The molecule which contains the set of conformers to select from.
- **percentage** – The percentage of conformers with the lowest electrostatic interaction energies to greedily select from.
- **limit** – The maximum number of conformers to select.
- **rms_tolerance** – Conformers whose RMS is within this amount will be treated as identical and the duplicate discarded.

from_rdkit(*rdmol*, *allow_undefined_stereo=False*, *hydrogens_are_explicit=False*, *_cls=None*)

Create a Molecule from an RDKit molecule.

Requires the RDKit to be installed.

Warning: This API is experimental and subject to change.

Parameters

- **rdmol** (`rkit.RDMol`) – An RDKit molecule
- **allow_undefined_stereo** (`bool, default=False`) – If false, raises an exception if rdmol contains undefined stereochemistry.
- **hydrogens_are_explicit** (`bool, default=False`) – If False, RDKit will perform hydrogen addition using Chem.AddHs

- **_cls** (class) – Molecule constructor

Returns **molecule** (*openff.toolkit.topology.Molecule*) – An OpenFF molecule

Examples

Create a molecule from an RDKit molecule

```
>>> from rdkit import Chem
>>> from openff.toolkit.tests.utils import get_data_file_path
>>> rdmol = Chem.MolFromMolFile(get_data_file_path('systems/monomers/ethanol.sdf'))
```

```
>>> toolkit_wrapper = RDKitToolkitWrapper()
>>> molecule = toolkit_wrapper.from_rdkit(rdmol)
```

to_rdkit(*molecule*, *aromaticity_model=DEFAULT_AROMATICITY_MODEL*)

Create an RDKit molecule Requires the RDKit to be installed.

Warning: This API is experimental and subject to change.

Parameters **aromaticity_model** (str, optional, default=DEFAULT_AROMATICITY_MODEL) – The aromaticity model to use

Returns **rdmol** (*rkit.RDMol*) – An RDKit molecule

Examples

Convert a molecule to RDKit >>> from openff.toolkit import Molecule >>> ethanol = Molecule.from_smiles('CCO') >>> rdmol = ethanol.to_rdkit()

to_inchi(*molecule*, *fixed_hydrogens=False*)

Create an InChI string for the molecule using the RDKit Toolkit. InChI is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

- **molecule** (An openff.toolkit.topology.Molecule) – The molecule to convert into a SMILES.
- **fixed_hydrogens** (bool, default=False) – If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

Returns **inchi** (*str*) – The InChI string of the molecule.

to_inchikey(*molecule*, *fixed_hydrogens=False*)

Create an InChIKey for the molecule using the RDKit Toolkit. InChIKey is a standardised representation that does not capture tautomers unless specified using the fixed hydrogen layer.

For information on InChi see here https://iupac.org/who-we-are/divisions/division-details/inchi/

Parameters

- **molecule** (An `openff.toolkit.topology.Molecule`) – The molecule to convert into a SMILES.
- **fixed_hydrogens** (`bool, default=False`) – If a fixed hydrogen layer should be added to the InChI, if *True* this will produce a non standard specific InChI string of the molecule.

Returns **inchi_key** (*str*) – The InChIKey representation of the molecule.

get_tagged_smarts_connectivity(*smarts*)

Returns a tuple of tuples indicating connectivity between tagged atoms in a SMARTS string. Does not return bond order.

Parameters **smarts** (`str`) – The tagged SMARTS to analyze

Returns

- **unique_tags** (*tuple of int*) – A sorted tuple of all unique tagged atom map indices.
- **tagged_atom_connectivity** (*tuple of tuples of int, shape n_tagged_bonds x 2*) –

  A tuple of tuples, where each inner tuple is a pair of tagged atoms (tag_idx_1, tag_idx_2) which are bonded. The inner tuples are ordered smallest-to-largest, and the tuple of tuples is ordered lexically. So the return value for an improper torsion would be ((1, 2), (2, 3), (2, 4)).

Raises **SMIRKSParsingError** – If RDKit was unable to parse the provided smirks/tagged smarts

find_smarts_matches(*molecule*, *smarts*, *aromaticity_model='OEAroModel_MDL'*, *unique=False*)

Find all SMARTS matches for the specified molecule, using the specified aromaticity model.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (`Molecule`) – The molecule for which all specified SMARTS matches are to be located
- **smarts** (`str`) – SMARTS string with optional SMIRKS-style atom tagging
- **aromaticity_model** (`str, optional, default='OEAroModel_MDL'`) – Molecule is prepared with this aromaticity model prior to querying.
- **:** (`.. note`) – Currently:
- **OEAroModel_MDL** (`the only supported aromaticity_model is`) –

atom_is_in_ring(*atom:* Atom) → bool

Return whether or not an atom is in a ring.

It is assumed that this atom is in molecule.

Parameters **atom** (`Atom`) – The molecule containing the atom of interest

Returns **is_in_ring** (*bool*) – Whether or not the atom is in a ring.

Raises **NotAttachedToMoleculeError** –

bond_is_in_ring(*bond:* Bond) → bool

Return whether or not a bond is in a ring.

It is assumed that this atom is in molecule.

Parameters **bond** (Bond) – The molecule containing the atom of interest

Returns **is_in_ring** (*bool*) – Whether or not the bond of index *bond_index* is in a ring

Raises **NotAttachedToMoleculeError** –

property toolkit_file_read_formats

List of file formats that this toolkit can read.

property toolkit_installation_instructions

Instructions on how to install the wrapped toolkit.

property toolkit_name

Return the name of the toolkit wrapped by this class as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_name** (*str*) – The name of the wrapped toolkit

property toolkit_version

Return the version of the wrapped toolkit as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_version** (*str*) – The version of the wrapped toolkit

13.1.5 openff.toolkit.utils.toolkits.AmberToolsToolkitWrapper

class openff.toolkit.utils.toolkits.**AmberToolsToolkitWrapper**

AmberTools toolkit wrapper

> **Warning:** This API is experimental and subject to change.

__init__()

Methods

`__init__`()	
`assign_fractional_bond_orders`(molecule[, ...])	Update and store list of bond orders this molecule.
`assign_partial_charges`(molecule[, ...])	Compute partial charges with AmberTools using antechamber/sqm, and assign the new values to the partial_charges attribute.
`compute_partial_charges_am1bcc`(molecule[, ...])	Compute partial charges with AmberTools using antechamber/sqm.
`from_file`(file_path, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file using this toolkit.
`from_file_obj`(file_obj, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.
`is_available`()	Check whether the AmberTools toolkit is installed
`requires_toolkit`()	

Attributes

`toolkit_file_read_formats`	List of file formats that this toolkit can read.
`toolkit_file_write_formats`	List of file formats that this toolkit can write.
`toolkit_installation_instructions`	Instructions on how to install the wrapped toolkit.
`toolkit_name`	Return the name of the toolkit wrapped by this class as a str
`toolkit_version`	Return the version of the wrapped toolkit as a str

static is_available()

Check whether the AmberTools toolkit is installed

Returns **is_installed** (*bool*) – True if AmberTools is installed, False otherwise.

assign_partial_charges(*molecule*, *partial_charge_method=None*, *use_conformers=None*, *strict_n_conformers=False*, *normalize_partial_charges=True*, *_cls=None*)

Compute partial charges with AmberTools using antechamber/sqm, and assign the new values to the partial_charges attribute.

> **Warning:** This API experimental and subject to change.

Parameters

- **molecule** (`Molecule`) – Molecule for which partial charges are to be computed
- **partial_charge_method** (`str, optional, default=None`) – The charge model to use. One of ['gasteiger', 'am1bcc', 'am1-mulliken']. If None, 'am1-mulliken' will be used.

- **use_conformers** (iterable of unit-wrapped numpy arrays, each) – with shape (n_atoms, 3) and dimension of distance. Optional, default = None List of unit-wrapped numpy arrays to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **strict_n_conformers** (bool, default=False) – Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised.
- **normalize_partial_charges** (bool, default=True) – Whether to offset partial charges so that they sum to the total formal charge of the molecule. This is used to prevent accumulation of rounding errors when the partial charge generation method has low precision.
- **_cls** (class) – Molecule constructor

Raises

- **ChargeMethodUnavailableError if the requested charge method can not be handled by this toolkit** –
- **ChargeCalculationError if the charge method is supported by this toolkit, but fails** –

compute_partial_charges_am1bcc(*molecule*, *use_conformers=None*, *strict_n_conformers=False*)

Compute partial charges with AmberTools using antechamber/sqm. This will calculate AM1-BCC charges on the first conformer only.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (Molecule) – Molecule for which partial charges are to be computed
- **use_conformers** (iterable of unit-wrapped numpy arrays,) – each with shape (n_atoms, 3) and dimension of distance. Optional, default = None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **strict_n_conformers** (bool, default=False) – Whether to raise an exception if an invalid number of conformers is provided. If this is False and an invalid number of conformers is found, a warning will be raised instead of an Exception.

Returns charges (*numpy.array of shape (natoms) of type float*) – The partial charges

assign_fractional_bond_orders(*molecule*, *bond_order_model=None*, *use_conformers=None*, *_cls=None*)

Update and store list of bond orders this molecule. Bond orders are stored on each bond, in the *bond.fractional_bond_order* attribute.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (openff.toolkit.topology.molecule Molecule) – The molecule to assign wiberg bond orders to
- **bond_order_model** (str, optional, default=None) – The charge model to use. Only allowed value is 'am1-wiberg'. If None, 'am1-wiberg' will be used.
- **use_conformers** (iterable of unit-wraapped np.array with shape (n_atoms, 3)) – and dimension of distance, optional, default=None The conformers to use for fractional bond order calculation. If None, an appropriate number of conformers will be generated by an available ToolkitWrapper.
- **_cls** (class) – Molecule constructor

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*)

Return an openff.toolkit.topology.Molecule from a file using this toolkit.

Parameters

- **file_path** (str) – The file to read the molecule from
- **file_format** (str) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if any molecules contain undefined stereochemistry.
- **_cls** (class) – Molecule constructor

Returns molecules (*Molecule or list of Molecules*) – a list of Molecule objects is returned.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.

Parameters

- **file_obj** (file-like object) – The file-like object to read the molecule from
- **file_format** (str) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.
- **_cls** (class) – Molecule constructor

Returns molecules (*Molecule or list of Molecules*) – a list of Molecule objects is returned.

property toolkit_file_read_formats

List of file formats that this toolkit can read.

property toolkit_file_write_formats

List of file formats that this toolkit can write.

property toolkit_installation_instructions

Instructions on how to install the wrapped toolkit.

property toolkit_name

Return the name of the toolkit wrapped by this class as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_name** (*str*) – The name of the wrapped toolkit

property toolkit_version

Return the version of the wrapped toolkit as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_version** (*str*) – The version of the wrapped toolkit

13.1.6 openff.toolkit.utils.toolkits.BuiltInToolkitWrapper

class openff.toolkit.utils.toolkits.**BuiltInToolkitWrapper**

Built-in ToolkitWrapper for very basic functionality. Intended for testing and not much more.

> **Warning:** This API is experimental and subject to change.

__init__()

Methods

__init__()	
assign_partial_charges(molecule[, ...])	Compute partial charges with the built-in toolkit using simple arithmetic operations, and assign the new values to the partial_charges attribute.
from_file(file_path, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file using this toolkit.
from_file_obj(file_obj, file_format[, ...])	Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.
is_available()	Check whether the corresponding toolkit can be imported
requires_toolkit()	

Attributes

toolkit_file_read_formats	List of file formats that this toolkit can read.
toolkit_file_write_formats	List of file formats that this toolkit can write.
toolkit_installation_instructions	Instructions on how to install the wrapped toolkit.
toolkit_name	Return the name of the toolkit wrapped by this class as a str
toolkit_version	Return the version of the wrapped toolkit as a str

assign_partial_charges(*molecule*, *partial_charge_method=None*, *use_conformers=None*, *strict_n_conformers=False*, *normalize_partial_charges=True*, *_cls=None*)

Compute partial charges with the built-in toolkit using simple arithmetic operations, and assign the new values to the partial_charges attribute.

> **Warning:** This API is experimental and subject to change.

Parameters

- **molecule** (Molecule) – Molecule for which partial charges are to be computed
- **partial_charge_method** (str, optional, default=None) – The charge model to use. One of ['zeros', 'formal_charge']. If None, 'formal_charge' will be used.
- **use_conformers** (iterable of unit-wrapped numpy arrays, each with shape) – (n_atoms, 3) and dimension of distance. Optional, default = None Coordinates to use for partial charge calculation. If None, an appropriate number of conformers will be generated.
- **strict_n_conformers** (bool, default=False) – Whether to raise an exception if an invalid number of conformers is provided for the given charge method. If this is False and an invalid number of conformers is found, a warning will be raised instead of an Exception.
- **normalize_partial_charges** (bool, default=True) – Whether to offset partial charges so that they sum to the total formal charge of the molecule. This is used to prevent accumulation of rounding errors when the partial charge generation method has low precision.
- **_cls** (class) – Molecule constructor

Raises

- **ChargeMethodUnavailableError if this toolkit cannot handle the requested charge method** –
- **IncorrectNumConformersError if strict_n_conformers is True and use_conformers is provided** –
- **and specifies an invalid number of conformers for the requested method** –
- **ChargeCalculationError if the charge calculation is supported by this toolkit, but fails** –

from_file(*file_path*, *file_format*, *allow_undefined_stereo=False*)

Return an openff.toolkit.topology.Molecule from a file using this toolkit.

Parameters

- **file_path** (str) – The file to read the molecule from
- **file_format** (str) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if any molecules contain undefined stereochemistry.
- **_cls** (class) – Molecule constructor

Returns molecules (*Molecule or list of Molecules*) – a list of Molecule objects is returned.

from_file_obj(*file_obj*, *file_format*, *allow_undefined_stereo=False*, *_cls=None*)

Return an openff.toolkit.topology.Molecule from a file-like object (an object with a ".read()" method using this toolkit.

Parameters

- **file_obj** (file-like object) – The file-like object to read the molecule from
- **file_format** (str) – Format specifier, usually file suffix (eg. 'MOL2', 'SMI') Note that not all toolkits support all formats. Check ToolkitWrapper.toolkit_file_read_formats for details.
- **allow_undefined_stereo** (bool, default=False) – If false, raises an exception if any molecules contain undefined stereochemistry. If false, the function skips loading the molecule.
- **_cls** (class) – Molecule constructor

Returns molecules (*Molecule or list of Molecules*) – a list of Molecule objects is returned.

classmethod is_available()

Check whether the corresponding toolkit can be imported

Returns is_installed (*bool*) – True if corresponding toolkit is installed, False otherwise.

property toolkit_file_read_formats

List of file formats that this toolkit can read.

property toolkit_file_write_formats

List of file formats that this toolkit can write.

property toolkit_installation_instructions

Instructions on how to install the wrapped toolkit.

property toolkit_name

Return the name of the toolkit wrapped by this class as a str

> **Warning:** This API is experimental and subject to change.

Returns toolkit_name (*str*) – The name of the wrapped toolkit

property toolkit_version

Return the version of the wrapped toolkit as a str

> **Warning:** This API is experimental and subject to change.

Returns **toolkit_version** (*str*) – The version of the wrapped toolkit

13.2 Serialization support

Serializable	Mix-in to add serialization and deserialization support via JSON, YAML, BSON, TOML, MessagePack, and XML.

13.2.1 openff.toolkit.utils.serialization.Serializable

class openff.toolkit.utils.serialization.**Serializable**

Mix-in to add serialization and deserialization support via JSON, YAML, BSON, TOML, MessagePack, and XML.

For more information on these formats, see: JSON, BSON, YAML, TOML, MessagePack, and XML.

To use this mix-in, the class inheriting from this class must have implemented `to_dict()` and `from_dict()` methods that utilize dictionaries containing only serialiable Python objects.

> **Warning:** The serialization/deserialiation schemes used here place some strict constraints on what kinds of `dict` objects can be serialized. No effort is made to add further protection to ensure serialization is possible. Use with caution.

Examples

Example class using `Serializable` mix-in:

```
>>> from openff.toolkit.utils.serialization import Serializable
>>> class Thing(Serializable):
...     def __init__(self, description):
...         self.description = description
...
...     def to_dict(self):
...         return { 'description' : self.description }
...
...     @classmethod
...     def from_dict(cls, d):
...         return cls(d['description'])
...
>>> # Create an example object
>>> thing = Thing('blorb')
```

Get JSON representation:

```
>>> json_thing = thing.to_json()
```

Reconstruct an object from its JSON representation:

```
>>> thing_from_json = Thing.from_json(json_thing)
```

Get BSON representation:

```
>>> bson_thing = thing.to_bson()
```

Reconstruct an object from its BSON representation:

```
>>> thing_from_bson = Thing.from_bson(bson_thing)
```

Get YAML representation:

```
>>> yaml_thing = thing.to_yaml()
```

Reconstruct an object from its YAML representation:

```
>>> thing_from_yaml = Thing.from_yaml(yaml_thing)
```

Get MessagePack representation:

```
>>> messagepack_thing = thing.to_messagepack()
```

Reconstruct an object from its MessagePack representation:

```
>>> thing_from_messagepack = Thing.from_messagepack(messagepack_thing)
```

Get XML representation:

```
>>> xml_thing = thing.to_xml()
```

__init__()

Methods

`__init__`()	
`from_bson`(serialized)	Instantiate an object from a BSON serialized representation.
`from_dict`(d)	
`from_json`(serialized)	Instantiate an object from a JSON serialized representation.
`from_messagepack`(serialized)	Instantiate an object from a MessagePack serialized representation.
`from_pickle`(serialized)	Instantiate an object from a pickle serialized representation.
`from_toml`(serialized)	Instantiate an object from a TOML serialized representation.
`from_xml`(serialized)	Instantiate an object from an XML serialized representation.
`from_yaml`(serialized)	Instantiate from a YAML serialized representation.
`to_bson`()	Return a BSON serialized representation.
`to_dict`()	
`to_json`([indent])	Return a JSON serialized representation.
`to_messagepack`()	Return a MessagePack representation.
`to_pickle`()	Return a pickle serialized representation.
`to_toml`()	Return a TOML serialized representation.
`to_xml`([indent])	Return an XML representation.
`to_yaml`()	Return a YAML serialized representation.

to_json(*indent=None*) → str

Return a JSON serialized representation.

Specification: https://www.json.org/

Parameters **indent** (`int, optional, default=None`) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*str*) – A JSON serialized representation of the object

classmethod from_json(*serialized: str*)

Instantiate an object from a JSON serialized representation.

Specification: https://www.json.org/

Parameters **serialized** (`str`) – A JSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

to_bson()

Return a BSON serialized representation.

Specification: http://bsonspec.org/

Returns **serialized** (*bytes*) – A BSON serialized representation of the objecft

classmethod from_bson(*serialized*)

Instantiate an object from a BSON serialized representation.

Specification: http://bsonspec.org/

Parameters **serialized** (bytes) – A BSON serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

to_toml()

Return a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Returns **serialized** (*str*) – A TOML serialized representation of the object

classmethod from_toml(*serialized*)

Instantiate an object from a TOML serialized representation.

Specification: https://github.com/toml-lang/toml

Parameters **serlialized** (str) – A TOML serialized representation of the object

Returns **instance** (*cls*) – An instantiated object

to_yaml()

Return a YAML serialized representation.

Specification: http://yaml.org/

Returns **serialized** (*str*) – A YAML serialized representation of the object

classmethod from_yaml(*serialized*)

Instantiate from a YAML serialized representation.

Specification: http://yaml.org/

Parameters **serialized** (str) – A YAML serialized representation of the object

Returns **instance** (*cls*) – Instantiated object

to_messagepack()

Return a MessagePack representation.

Specification: https://msgpack.org/index.html

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation of the object

classmethod from_messagepack(*serialized*)

Instantiate an object from a MessagePack serialized representation.

Specification: https://msgpack.org/index.html

Parameters **serialized** (bytes) – A MessagePack-encoded bytes serialized representation

Returns **instance** (*cls*) – Instantiated object.

to_xml(*indent=2*)

Return an XML representation.

Specification: https://www.w3.org/XML/

Parameters **indent** (int, optional, default=2) – If not None, will pretty-print with specified number of spaces for indentation

Returns **serialized** (*bytes*) – A MessagePack-encoded bytes serialized representation.

classmethod **from_xml**(*serialized*)

Instantiate an object from an XML serialized representation.

Specification: https://www.w3.org/XML/

Parameters **serialized** (bytes) – An XML serialized representation

Returns **instance** (*cls*) – Instantiated object.

to_pickle()

Return a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Returns **serialized** (*str*) – A pickled representation of the object

classmethod **from_pickle**(*serialized*)

Instantiate an object from a pickle serialized representation.

> **Warning:** This is not recommended for safe, stable storage since the pickle specification may change between Python versions.

Parameters **serialized** (str) – A pickled representation of the object

Returns **instance** (*cls*) – An instantiated object

13.3 Collections

Custom collections for the toolkit

ValidatedList	A list that runs custom converter and validators when new elements are added.
ValidatedDict	A dict that runs custom converter and validators when new elements are added.

13.3.1 openff.toolkit.utils.collections.ValidatedList

class openff.toolkit.utils.collections.**ValidatedList**(*seq=()*, *converter=None*, *validator=None*)

A list that runs custom converter and validators when new elements are added.

Multiple converters and validators can be assigned to the list. These are executed in the given order with converters run before validators.

Validators must take the new element as the first argument and raise an exception if validation fails.

validator(new_element) -> None

Converters must also take the new element as the first argument, but they have to return the converted value.

converter(new_element) -> converted_value

Examples

We can define validator and converter functions that are run on each element of the list.

```
>>> def is_positive_validator(value):
...     if value <= 0:
...         raise TypeError('value must be positive')
...
>>> vl = ValidatedList([1, -1], validator=is_positive_validator)
Traceback (most recent call last):
...
TypeError: value must be positive
```

Multiple converters that are run before the validators can be specified.

```
>>> vl = ValidatedList([-1, '2', 3.0], converter=[float, abs],
...                    validator=is_positive_validator)
>>> vl
[1.0, 2.0, 3.0]
```

__init__(*seq=()*, *converter=None*, *validator=None*)

Initialize the list.

Parameters

- **seq** (Iterable) – A sequence of elements.
- **converter** (callable or List[callable]) – Functions that will be used to convert each new element of the list.
- **validator** (callable or List[callable]) – Functions that will be used to convert each new element of the list.

Methods

__init__([seq, converter, validator])	Initialize the list.
append(p_object)	Append object to the end of the list.
clear()	Remove all items from list.
copy()	Return a shallow copy of the list.
count(value, /)	Return number of occurrences of value.
extend(iterable)	Extend list by appending elements from the iterable.
index(value[, start, stop])	Return first index of value.
insert(index, p_object)	Insert object before index.
pop([index])	Remove and return item at index (default last).
remove(value, /)	Remove first occurrence of value.
reverse()	Reverse *IN PLACE*.
sort(*[, key, reverse])	Sort the list in ascending order and return None.

extend(*iterable*)

Extend list by appending elements from the iterable.

append(*p_object*)
Append object to the end of the list.

insert(*index*, *p_object*)
Insert object before index.

copy()
Return a shallow copy of the list.

clear()
Remove all items from list.

count(*value*, */*)
Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

pop(*index=- 1*, */*)
Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(*value*, */*)
Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()
Reverse *IN PLACE*.

sort(***, *key=None*, *reverse=False*)
Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

13.3.2 openff.toolkit.utils.collections.ValidatedDict

class openff.toolkit.utils.collections.**ValidatedDict**(*mapping*, *converter=None*, *validator=None*)
A dict that runs custom converter and validators when new elements are added.

Multiple converters and validators can be assigned to the dict. These are executed in the given order with converters run before validators.

Validators must take the new element as the first argument and raise an exception if validation fails.

validator(new_element) -> None

Converters must also take the new element as the first argument, but they have to return the converted value.

converter(new_element) -> converted_value

Examples

We can define validator and converter functions that are run on each value of the dict.

```
>>> def is_positive_validator(value):
...     if value <= 0:
...         raise TypeError('value must be positive')
...
>>> vl = ValidatedDict({'a': 1, 'b': -1}, validator=is_positive_validator)
Traceback (most recent call last):
...
TypeError: value must be positive
```

Multiple converters that are run before the validators can be specified.

```
>>> vl = ValidatedDict({'c': -1, 'd': '2', 'e': 3.0}, converter=[float, abs],
...                    validator=is_positive_validator)
>>> vl
{'c': 1.0, 'd': 2.0, 'e': 3.0}
```

__init__(*mapping*, *converter=None*, *validator=None*)

Initialize the dict.

Parameters

- **mapping** (Mapping) – A mapping of elements, probably a dict.
- **converter** (callable or List[callable]) – Functions that will be used to convert each new element of the dict.
- **validator** (callable or List[callable]) – Functions that will be used to convert each new element of the dict.

Methods

`__init__`(mapping[, converter, validator])	Initialize the dict.
`clear`()	
`copy`()	
`fromkeys`([value])	Create a new dictionary with keys from iterable and values set to value.
`get`(key[, default])	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised
`popitem`()	Remove and return a (key, value) pair as a 2-tuple.
`setdefault`(key[, default])	Insert key with a value of default if key is not in the dictionary.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

copy() → a shallow copy of D

clear() → None. Remove all items from D.

fromkeys(*value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

values() → an object providing a view on D's values

13.4 Miscellaneous utilities

Miscellaneous utility functions.

inherit_docstrings	Inherit docstrings from parent class
all_subclasses	Recursively retrieve all subclasses of the specified class
temporary_cd	Context to temporary change the working directory.
get_data_file_path	Get the full path to one of the reference files in test-systems.
convert_0_1_smirnoff_to_0_2	Convert an 0.1-compliant SMIRNOFF dict to an 0.2-compliant one.
convert_0_2_smirnoff_to_0_3	Convert an 0.2-compliant SMIRNOFF dict to an 0.3-compliant one.
get_molecule_parameterIDs	Process a list of molecules with a specified SMIRNOFF ffxml file and determine which parameters are used by which molecules, returning collated results.
unit_to_string	

13.4.1 openff.toolkit.utils.utils.inherit_docstrings

openff.toolkit.utils.utils.**inherit_docstrings**(*cls*)

Inherit docstrings from parent class

13.4.2 openff.toolkit.utils.utils.all_subclasses

openff.toolkit.utils.utils.**all_subclasses**(*cls*)

Recursively retrieve all subclasses of the specified class

13.4.3 openff.toolkit.utils.utils.temporary_cd

openff.toolkit.utils.utils.**temporary_cd**(*dir_path*)

Context to temporary change the working directory.

Parameters **dir_path** (str) – The directory path to enter within the context

Examples

```
>>> dir_path = '/tmp'
>>> with temporary_cd(dir_path):
...     pass  # do something in dir_path
```

13.4.4 openff.toolkit.utils.utils.get_data_file_path

openff.toolkit.utils.utils.**get_data_file_path**(*relative_path*)

Get the full path to one of the reference files in testsystems. In the source distribution, these files are in `openff/toolkit/data/`, but on installation, they're moved to somewhere in the user's python site-packages directory.

Parameters **name** (str) – Name of the file to load (with respect to the repex folder).

13.4.5 openff.toolkit.utils.utils.convert_0_1_smirnoff_to_0_2

openff.toolkit.utils.utils.**convert_0_1_smirnoff_to_0_2**(*smirnoff_data_0_1*)

Convert an 0.1-compliant SMIRNOFF dict to an 0.2-compliant one. This involves renaming several tags, adding Electrostatics and ToolkitAM1BCC tags, and separating improper torsions into their own section.

Parameters **smirnoff_data_0_1** (dict) – Hierarchical dict representing a SMIRNOFF data structure according the the 0.1 spec

Returns *smirnoff_data_0_2* – Hierarchical dict representing a SMIRNOFF data structure according the the 0.2 spec

13.4.6 openff.toolkit.utils.utils.convert_0_2_smirnoff_to_0_3

openff.toolkit.utils.utils.**convert_0_2_smirnoff_to_0_3**(*smirnoff_data_0_2*)

Convert an 0.2-compliant SMIRNOFF dict to an 0.3-compliant one. This involves removing units from header tags and adding them to attributes of child elements. It also requires converting ProperTorsions and ImproperTorsions potentials from "charmm" to "fourier".

Parameters **smirnoff_data_0_2** (dict) – Hierarchical dict representing a SMIRNOFF data structure according the the 0.2 spec

Returns *smirnoff_data_0_3* – Hierarchical dict representing a SMIRNOFF data structure according the the 0.3 spec

13.4.7 openff.toolkit.utils.utils.get_molecule_parameterIDs

openff.toolkit.utils.utils.**get_molecule_parameterIDs**(*molecules*, *forcefield*)

Process a list of molecules with a specified SMIRNOFF ffxml file and determine which parameters are used by which molecules, returning collated results.

Parameters

- **molecules** (list of openff.toolkit.topology.Molecule) – List of molecules (with explicit hydrogens) to parse

- **forcefield** (ForceField) – The ForceField to apply

Returns

- **parameters_by_molecule** (*dict*) – Parameter IDs used in each molecule, keyed by isomeric SMILES generated from provided OEMols. Each entry in the dict is a list which does not necessarily have unique entries; i.e. parameter IDs which are used more than once will occur multiple times.
- **parameters_by_ID** (*dict*) – Molecules in which each parameter ID occur, keyed by parameter ID. Each entry in the dict is a set of isomeric SMILES for molecules in which that parameter occurs. No frequency information is stored.

13.4.8 openff.toolkit.utils.utils.unit_to_string

openff.toolkit.utils.utils.**unit_to_string**(*input_unit: Unit*) → str

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X